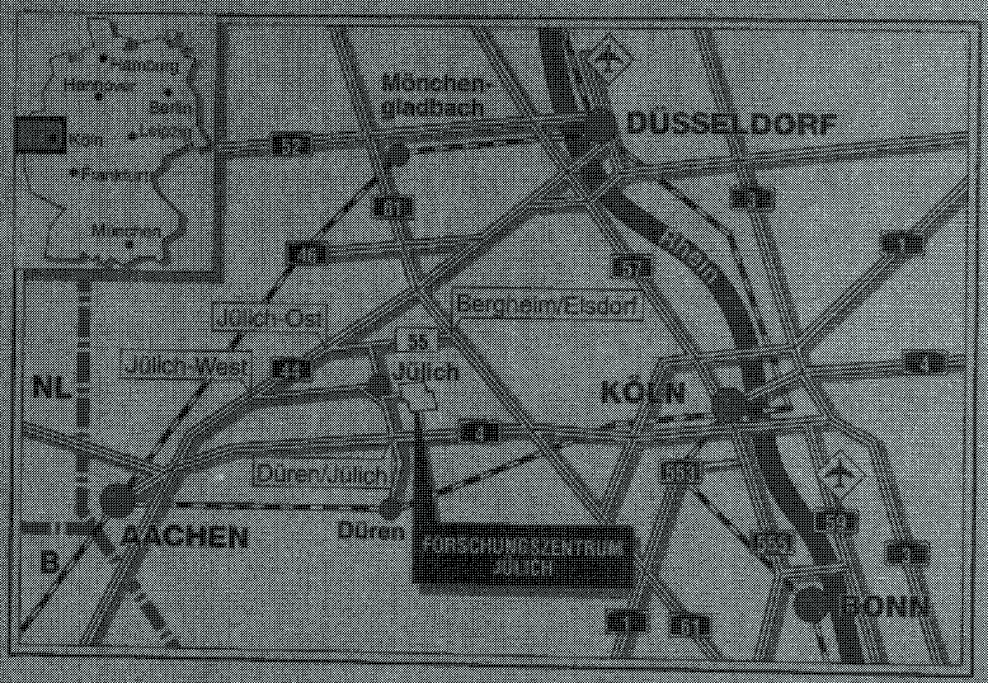
Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Einbindung von Multi-Threading und effizienter Synchronisation in den SVM-Fortran-Compiler

Joachim Worringen

Jül-3743



Berichte des Forschungszentrums Jülich ; 3443
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3443

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

Einbindung von Multi-Threading und effizienter Synchronisation in den SVM-Fortran-Compiler

Joachim Worringen

22. Oktober 1997

Einbindung von Multi-Threading und effizienter Synchronisation in den SVM-Fortran-Compiler

Die Verwendung von gemeinsamem Speicher auf Parallelrechnern vereinfacht die Programmierung dieser Systeme. Auf Systemen mit physikalisch getrenntem Speicher kann durch Software virtuell gemeinsamer Speicher zur Verfügung gestellt werden. Dieses Prinzip verwirklicht ASVM auf dem Parallelrechner Intel Paragon XP/S. Der virtuell gemeinsame Speicher ist seitenweise organisiert: Ein Zugriff auf ein nicht lokal vorhandenes Datum führt zu einem Seitenfehler, der die Übermittlung der referenzierten Seite auslöst. Die Dauer der Behandlung dieses Seitenfehlers liegt im Bereich von mehreren Millisekunden.
Der Fortrandialekt SVM-Fortran führt parallele Erweiterungen von Fortran77 auf Systemen mit virtuell gemeinsamem Speicher ein. Die Erweiterungen sind mittels eines Precompilers und zugehöriger Laufzeitbibliothek implementiert. Zur Verdeckung der auftretenden Latenzzeiten bei Seitenfehlern wurde Multithreading in SVM-Fortran integriert, um bei Auftreten eines Seitenfehlers einen anderen Thread einen anderen Teil der Daten bearbeiten zu lassen.
Die Arbeit beschreibt die Grundlagen des Multithreadings und von effizienten Synchronisationsalgorithmen für verschiedene Architekturklasssen paralleler Systeme (UMA, NUMA, NORMA). Sie erläutert die Einbindung des Multithreadings in den Precompiler und die Laufzeitbibliothek und evaluiert die Effekte des Multithreadings anhand synthetischer und praktischer Benchmarks. Die Ergebnisse belegen die grundsätzliche Wirksamkeit des vorgestellten Prinzips.

Integration of Multithreading and Efficient Synchronization in the SVM-Fortran-Compiler

Programming parallel systems using shared memory comes easier and more natural. On systems lacking physical shared memory, software solutions offering virtual shared memory are available. An example of such a solution is ASVM running on the Intel Paragon XP/S. The virtual shared memory is organized in pages: accessing an item which is not stored locally causes a page fault which triggers the transmission of the referred page. The latency of ASVM for processing such a page fault amounts to several milliseconds.
SVM-Fortran is an language extension to Fortran77 for parallel systems with virtual shared memory. The extensions are implemented using a precompiler and a run-time library. To hide the mentioned latencies in conjunction with page faults, multithreading was introduced into SVM-Fortran. If a thread causes a page fault, another thread will work on a different part of the data.
This paper describes the principles of multithreading and efficient synchronization algorithms for different classes of parallel systems (UMA, NUMA, NORMA). It illustrates the implementation of multithreading into the precompiler and the run-time library and evaluates the effects of multithreading by performing synthetical and practical benchmarks. The results demonstrate the fundamental efficiency of the principle.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Bedarf an immer höherer Rechenleistung sowohl im Bereich der Supercomputer als auch bei Workstations läßt sich nur noch durch den Einsatz von Parallelrechnern befriedigen. Bei den Supercomputern werden zum einen die klassischen Vektorrechner durch die Verwendung mehrerer, aber noch relativ weniger Prozessoren beschleunigt, zum anderen werden hier auch *massiv-parallele Systeme* eingesetzt.

Als massiv-parallel gelten im allgemeinen Rechner mit mehr als 100 Prozessoren. In solchen Systemen kann den Prozessoren kein gemeinsamer Speicher mehr zur Verfügung gestellt werden, da ein solcher Speicher für den gleichartigen Zugriff aller Prozessoren mit den erforderlichen Bandbreiten wirtschaftlich nicht sinnvoll zu realisieren ist. Darüberhinaus wäre ein solches System sehr aufwendig zu skalieren. Skalierbarkeit ist jedoch eine Vorraussetzung, um dem Anwender die von ihm gewünschte Rechenleistung mit der Option auf Erweiterbarkeit offerieren zu können. Daher sind Systeme verbreitet, in denen die Prozessoren auf ihrem lokalen Speicher arbeiten und durch den Austausch von Nachrichten über ein Netzwerk kommunizieren. Dies ist ein völlig anderes Modell als auf Multiprozessor-Systemen mit wenigen Prozessoren, auf denen die Kommunikation durch einfache Speicherzugriffe der Prozessoren in den gemeinsamen Speicher erfolgt.

Um die Programmierung paralleler Applikationen auf massiv-parallelen Systemen zu erleichtern, wurden Software-Lösungen entwickelt, die der Applikation den herkömmlichen, gemeinsam Adreßraum vorspiegeln. Diese Virtualisierung bezeichnet man als *Shared Virtual Memory* (SVM) [17]. Alternativ zu den Software-Lösungen werden zunehmend Hardware-Lösungen (cc-NUMA) wie SGI Origin 2000 [27] entwickelt, die mit erhöhtem Hardwareaufwand wesentlich geringere Latenzzeiten beim Zugriff auf entfernten Speicher erreichen als dies Software-Implementierungen von SVM leisten können.

Die Abstraktion zu einem prozessorübergreifenden Adreßraum, ob in Hardware oder Software realisiert, erleichtert die Portierbarkeit von parallelen Applikationen, da SVM den physikalischen Aufbau eines parallelen Systems hinter seinen Schnittstellen verbirgt. Neben der Verfügbarkeit von Shared Virtual Memory ist auch die Bereitstellung einer Programmiersprache nötig, die dem Programmierer die speziellen Fähigkeiten eines parallelen Systems mit möglichst geringem Aufwand nutzbar macht und die Komplexität des Systems, soweit effizient möglich, hinreichend verbirgt. Dazu zählt die Parallelität, die Daten- und Lastverteilung und die Synchronisation. Eine solche Programmiersprache für

den technisch-wissenschaftlichen Bereich ist *SVM-Fortran*[4]. Der Compiler für SVM-Fortran übersetzt das mit Direktiven versetzte Fortran-Programm in Standard-Fortran mit Aufrufen von Funktionen einer Laufzeitbibliothek. Dieses Programm kann anschließend von dem systemspezifischen Fortran-Compiler übersetzt werden.

Zur Minimierung der Auswirkungen der angesprochenen Latenzzeiten in SVM-Systemen, wie sie beim Zugriff auf den Speicher anderer Prozessoren entstehen, wurden verschiedene Techniken entwickelt. In dieser Arbeit wird der Ansatz untersucht, durch die Verwendung von Software-basiertem Multithreading die auftretende Kommunikation mit sinnvoller Berechnung zu überlappen und somit die Auslastung des Prozessors zu verbessern.

Im folgenden Kapitel wird detailliert auf die Problemstellung sowie die in dieser Arbeit verwendete Klassifikation von massiv-parallelen Systemarchitekturen eingegangen. In Kapitel 3 wird die Entwicklung und die im Rahmen des Entwicklungsprozeßes entstandenen verschiedenen Typen von Threads dargestellt, bevor eine verbreitete Schnittstellendefinition für die Programmierung von Threads vorgestellt wird, das POSIX API[1]. Es wird ein analytisches Modell zum Verstecken von kommunikationsbedingten Latenzen (hier die Bedienzeit von Seitenfehlern) durch Multithreading vorgestellt, um anschließend auf die verschiedenen Techniken einzugehen, die diesen Zweck verfolgen.

Kapitel 4 behandelt die Grundlagen der Synchronisation, die im Zusammenhang mit dem Ziel dieser Arbeit eine wichtige Rolle spielt, da die Ergebnisse von Teilaufgaben immer wieder zum Gesamtergebnis zusammengeführt werden müssen. Darauf aufbauend werden eine Anzahl von Locks (Methoden zum gegenseitigen Ausschluß) und Barrieren vorgestellt, die jeweils zur Verwendung auf einer bestimmten Rechnerarchitektur besonders geeignet sind und entsprechend unterschiedliche Eigenschaften haben.

Die Konzeptionierung der Software und Einbindung in den SVM-Fortran-Compiler und die Laufzeitbibliothek wird in Kapitel 5 beschrieben, die Evaluierung und Bewertung der durch die Implementierung auf verschiedenen Systemen erzielten Leistungsänderungen erfolgt in Kapitel 6 anhand von synthetischen und praktischen Benchmarks. Kapitel 7 gibt eine Zusammenfassung der erzielten Ergebnisse und legt die möglichen Weiterentwicklungen dar.

[1] API = Application Programming Interface

Kapitel 2

Motivation

In diesem Kapitel wird zunächst ein Überblick über Ziel und Aufbau dieser Arbeit gegeben. Anschließend wird die verwendete Klassifizierung von parallelen Rechnerarchitekturen festgelegt.

2.1 Virtuell gemeinsamer Speicher

Die virtuelle Bereitstellung eines gemeinsamen Adreßraums über eine Softwareschicht unterhalb der Applikation ist stark verwandt mit der Bereitstellung eines Adreßraums auf einem Einzelprozessorsystem, der größer ist als der physikalisch vorhandene Speicher (*virtueller Speicher*, siehe auch [39]). Der virtuelle Adreßraum wird in Seiten von gleicher Größe unterteilt. Diese Seite migrieren im Falle des virtuellen Speichers zwischen Vorder- und Hintergrundspeicher, im Falle von virtuell gemeinsamen Speicher zwischen den lokalen Speichern der Prozessoren des Systems. Die Organisation des virtuell gemeinsamen Speichers wird von einem Seitenverwalter übernommen, der oft Teil des Betriebssystems ist. Seine Hauptaufgabe ist es, die von einem Prozessor benötigten Seiten im System zu lokalisieren und sie über das Netzwerk in den lokalen Speicher des Prozessors zu übertragen. Um mehreren Prozessoren den gleichzeitigen Zugriff auf eine Speicherseite zu ermöglichen, können mehrere Kopien einer Seite an mehrere Prozessoren verteilt werden. Bei Schreibzugriffen auf diese Seiten muß sichergestellt werden, daß die derart replizierten Seiten jeweils konsistent bleiben, wozu die geänderte Seite vor einem Lesezugriff aktualisiert werden muß. Grundsätzlich kann man eine Operation auf den virtuell gemeinsamen Speicher, die eine Übertragung der zugeordneten Seite in den lokalen Speicher des betreffenden Prozessors auslöst, als *Seitenfehler* bezeichnen. Somit löst ein Seitenfehler, der durch einen lesenden Zugriff verursacht wurde (*Lese-Seitenfehler*), nur die einfache Übertragung der Seite in den lokalen Speicher aus; ein *Schreib-Seitenfehler* bedeutet jedoch, bei strenger Kohärenz, die Aktualisierung aller zu diesem Zeitpunkt erstellten Kopien der Seite.

2.2 Problemstellung

Die Einführung von virtuell gemeinsamen Speicher führt zu einem einfacheren Programmiermodell, bringt aber auch Probleme mit sich. Da der Programmierer in einem einfachen Programmiermodell keinen direkten Einfluß auf die Plazierung von Daten auf den Prozessormoduelen hat, kommt es leicht zu einer großen Anzahl von *Seitenfehlern*. Das dadurch ausgelöste Verschicken der Seiten kostet im Verhältnis zu den darauf ausgeführten Operationen eine weitaus größere Zahl von Taktzyklen (siehe Kapitel 6.1.1).

Eine Lösung dieses Problems ist es, wenn dieses sogenannte *Thrashing* durch eine Änderung des Algorithmus beseitigt werden kann. Dazu muß jedoch zuerst die Ursache der schlechten Leistungs in einem Programm lokalisiert werden. Hilfreich sind dabei Software-Werkzeuge wie OPAL[8] oder PARvis [2]. Die Beseitigung einer erkannten Quelle des Thrashing ist sodann oftmals nicht trivial, da der gesamte Algorithmus betroffen sein kann. Aufwendige Konstrukte zur besseren Verteilung der Daten können die Folge sein. Zudem macht dieser Aufwand das parallele Programmiermodell nicht attraktiver.

Ein anderer Ansatz ist es, Seitenfehler zu *tolerieren*. Damit ist hier aber nicht gemeint, den Leistungsrückgang zu akzeptieren, sondern Techniken in den Compiler und die Laufzeitbibliothek einzuführen, die die Auswirkungen auftretender Seitenfehler verstecken. Eine solche Technik ist die Verwendung von *Multithreading*, um die es in dieser Arbeit geht. Der Grundgedanke ist dabei: „Solange ein Prozessor auf das Eintreffen einer angeforderten Seite wartet, soll mit einem anderen Teil der Aufgabe fortgefahren werden.“

2.3 Klassifizierung von Rechnerarchitekturen

Die Klassifizierung der Architektur von Parallelrechnern in Bezug auf die Verbindung der Prozessoren untereinander und die Anbindung des Speichersystems ist in der Literatur nicht einheitlich definiert. Um zumindest in dieser Arbeit die Bezeichnungen konsistent zu halten, wird hier eine Übersicht der gängigen Architekturen gegeben. Dazu werden jeweils Beispiele für Implementierungen angeführt. In Tabelle 2.1 sind die verwendeten Klassifizierungen mit ihren wichtigsten Merkmalen aufgeführt, die zudem im nächsten Abschnitt kurz beschrieben werden. Die Spalte *Adressen* gibt dabei an, ob Speicherseiten migrierbar sind (Beispiel KSR) oder auf einem Speichermodul alloziert werden und dort verbleiben. Die Spalte *R/W-Zugriff* gibt an, ob Lese- und Schreibzugriffe unterschieden werden müssen: bei Cache-kohärenten Systemen bedeutet ein Schreibzugriff im allgemeinen die Invalidierung der entsprechenden Einträge in den anderen Caches. Neben den klassischen Typen UMA, COMA und NORMA wurden die Typen *cc-UMA* und *cc-NUMA* als Weiterentwicklungen von UMA bzw. NUMA eingeführt. Die Abbildungen 2.1 bis 2.4 skizzieren das Grundprinzip des jeweiligen Ansatzes. Dabei steht P für einen Prozessor, C für (optionalen) Cache und M für Speicher.

- **UMA** *(Uniform Memory Access)*: Dieses Konzept gewährleistet allen Prozessoren eines Systems gleichschnellen Zugang auf den globalen Adreßraum, solange keine

Zugriffskonflikte auftreten. Letzteres ist bei einem System nach Abbildung 2.1(a) durch die Verwendung eines einzelnen Busses für alle Prozessoren jedoch sehr häufig der Fall. Daher hat z.B. Cray eine Speicherarchitektur gemäß Abbildung 2.1(c) eingeführt, wo der Speicher in unabhängige Bänke eingeteilt wird, auf die über ein Schaltnetzwerk mehrere Prozessoren gleichzeitig zugreifen können. Konflikte sind hierbei deutlich seltener, besonders wenn der Entwurf der verwendeten Datenstrukturen auf diese Architektur abgestimmt ist.

- **cc-UMA** *(cache coherent Uniform Memory Access*, oft auch SMP genannt): Der Unterschied zwischen UMA und cc-UMA liegt in der Verwendung von Caches. Dies können lokale Caches jedes Prozessors, ein globaler Cache für alle Prozessoren oder beides zugleich sein. Diese Caches entlasten das globale Speichersystem, erfordern jedoch hohen Aufwand zur Aufrechterhaltung der Kohärenz der Daten in den Caches. Dies begrenzt die Anzahl der Prozessoren und Caches, für die eine solche Architektur noch effizient ist.

- **cc-NUMA** *(cache coherent NonUniform Memory Access)*: Systeme, die der ursprünglichen NUMA-Architektur entsprechen (z.B. BBN Butterfly), sind heute nicht mehr üblich, daher wurde diese Kategorie nicht mehr aufgeführt. Stattdessen sind System entsprechnend dem cc-NUMA-Konzept verbreitet. Hierbei steht cc für „Cache coherent“. Alle Prozessoren verfügen über den gleichen globalen Adreßraum, wobei Zugriffe auf den lokalen Speicher schneller ablaufen als auf den entfernten Speicher eines anderen Prozessors. Entfernte Zugriffe werden über Caches beschleunigt.

- **COMA** *(Cache Only Memory Access)*: Ähnlich wie reine NUMA-Systeme sind auch COMA-Systeme kaum mehr anzutreffen. Jedoch wird das COMA-Prinzip, wo kein Prozessor einen eigenen festen Adreßraum besitzt, sondern die jeweils benötigten Seiten in seinen Cache transferiert, in Abwandlungen auch in cc-NUMA-Systemen verwendet, so z.B. in der Origin-Familie von SGI. In einem Cache-Verzeichnis werden Informationen über den Status von Speicherbereichen auf diesem Prozessorknoten gehalten.

- **NORMA** *(No Remote Memory Access)*: Massiv-parallele Rechner werden häufig als NORMA-Systeme entworfen, da bei der großen Zahl von Prozessoren (> 100) ein gemeinsamer Bus oder ein Schaltnetzwerk nicht mehr effizient und wirtschaftlich realisiert werden kann und zudem schlecht skalierbar ist. Die Prozessoren sind

Klassif.	Abb.	Beispiel	Adreßraum	Adressen	R/W-Zugriff
UMA	2.1 (c)	Cray PVP	gemeinsam	fest	gleich
cc-UMA	2.1 (a),(b)	SUN SMP	gemeinsam	fest	ungleich
cc-NUMA	2.2	SGI Origin	gemeinsam	fest	ungleich
COMA	2.3	KSR	gemeinsam	nicht fest	ungleich
NORMA	2.4	Intel Paragon	verteilt	fest	—

Tabelle 2.1: Klassifizierung der Architekturkonzepte von Parallelrechnern

über ein Netzwerk unterschiedlicher Topologie verbunden und kommunizieren nur über den Austausch von Nachrichten. Ein direkter Zugriff auf den Speicher eines anderen Prozessors ist so nicht möglich. Dies kann softwaremäßig über das Betriebssystem oder eine Laufzeitumgebung möglich gemacht werden, wie es im Entwurf zum MPI-2 Standard über sogenannte *single-sided operations* vorgesehen ist. Dies ist jedoch mit hohem Aufwand verbunden.

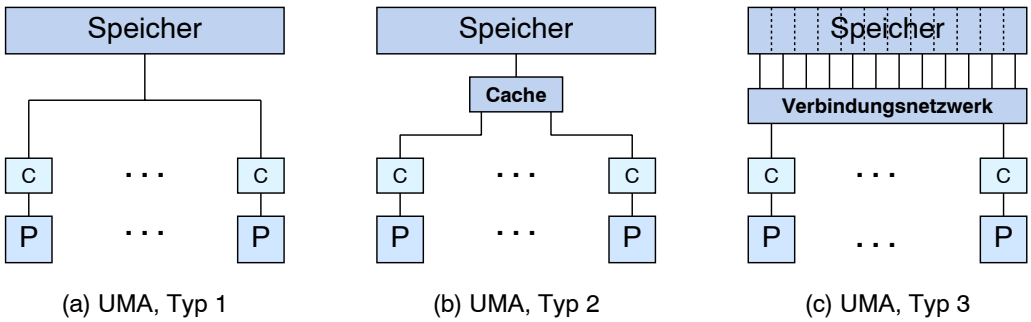


Abbildung 2.1: Verschiedene Typen von UMA-Architekturen

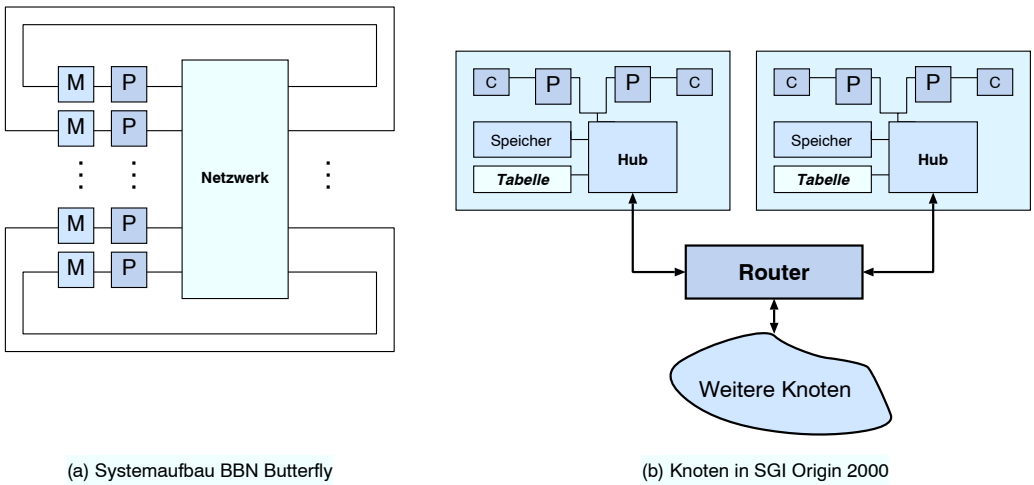


Abbildung 2.2: Klassische (a) und modifizierte (b) NUMA-Architektur

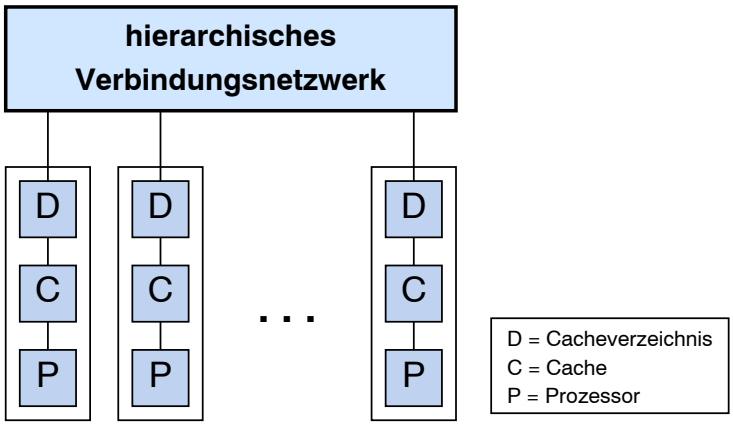


Abbildung 2.3: COMA-Architektur (KSR-1)

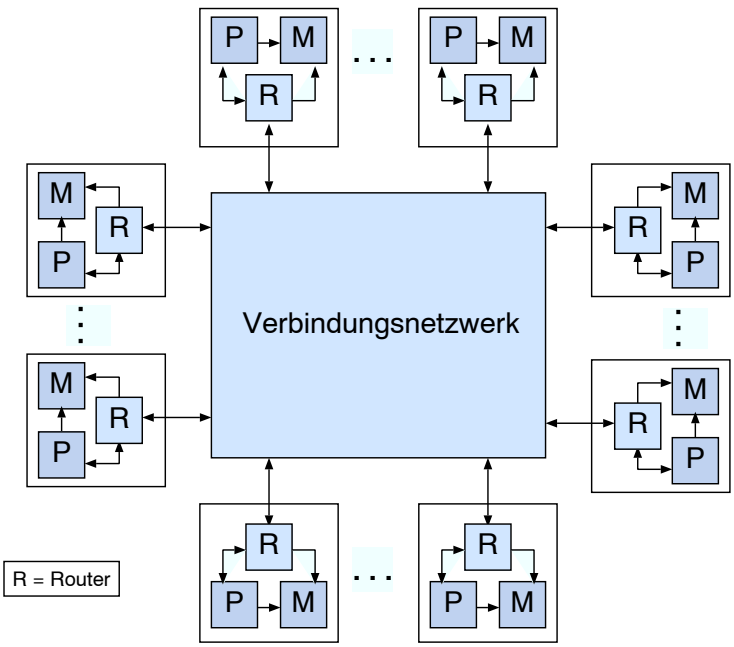


Abbildung 2.4: Universelles Schema der NORMA-Architektur

Kapitel 3

Threads

Zunächst werden in diesem Kapitel die klassischen Prozesse unter UNIX vorgestellt, wobei rasch die Unzulänglichkeiten eines Prozesses für unsere Ziele deutlich werden. Dies bietet den Einstieg in die Threads, die in ihrem Konzept und im folgenden Kapitel in den unterschiedlichen Implementierungen erläutert werden.

3.1 Der Prozeß unter UNIX

Die Grundfunktion eines jeden Betriebssystems ist die Bereitstellung einer Umgebung, um Benutzerprogramme (Applikationen) ablaufen zu lassen. Diese Umgebung beinhaltet die Bereitstellung von Diensten etwa zur Dateiverwaltung oder Ein-/Ausgabe und der zugehörigen Schnittstelle. Dazu enthält das UNIX- Betriebssystem die grundlegende Abstraktion des Prozesses, der in herkömmlichen UNIX-Systemen eine sequentielle Folge von Befehlen innerhalb seines Adressraumes ausführt. Der Adressraum eines Prozesses ist ein Satz von realen oder virtuellen Speicherbereichen, auf den dieser Prozeß exklusiven Zugriff hat. Weiterhin kennzeichnen einen Prozeß sein Kontrollpunkt, also der Programmzähler, der angibt, welcher Befehl als nächstes zu bearbeiten ist, sowie der aktuelle Zustand der Register des Prozessors, sein Stack und eine Datenstruktur mit Informationen über den Prozeß (üblicherweise in der sogenannten `proc`-Struktur zusammengefaßt), die das Betriebssystem zur Steuerung und Verwaltung aller Prozesse auf dem System unterhält.

Unter UNIX können eine große Zahl von Prozessen gestartet und abgearbeitet werden, auch wenn der Rechner selbst viel weniger Prozessoren, häufig nur einen einzigen, besitzt. Zwangsläufig muss dann in der Abarbeitung zwischen den Prozessen gewechselt werden, um so eine Quasi-Parallelität zu erreichen. Die Kriterien für einen Prozeßwechsel, die der Scheduler (dafür zuständiger Teil des Betriebssystems) anwendet, sind sehr vielfältig. Entscheidend für unsere Problematik ist aber, was bei einem Prozeßwechsel vorgeht. Darüberhinaus ist von Interesse, was bei einer besonderen Art von Prozeßwechsel, nämlich der Erzeugung bzw. Vernichtung eines Prozesses geschieht.

Die Erzeugung eines neuen Prozesses erfolgt immer durch einen bereits laufenden Prozeß, so daß eine Vater-Sohn-Beziehung entsteht. Unter UNIX wird ein neuer Prozeß im

allgemeinen durch den Aufruf der Systemfunktion `fork` erzeugt. Dazu sind eine Reihe von Maßnahmen erforderlich, die das Betriebssystem treffen muß, von denen hier nur die zeitintensivsten berücksichtigt werden:

- Auslagerungsspeicher für den Datenbereich und Stack reservieren
- Reservieren und initialisieren einer `proc`-Struktur für den Sohn-Prozeß
- Anlegen des Benutzeradressraums und der zugehörigen Seitentabellen zum Schutz des Adressraums
- Referenzen auf vom Vater-Prozeß ererbte Resourcen anlegen, so z.B. geöffnete Dateien oder das aktuelle Verzeichnis
- Einfügen des neuen Prozesses in die Warteschlange des Schedulers

Bei Beendigung eines Prozesss müssen die belegten Resourcen wieder freigegeben werden sowie einige andere Aufräumarbeiten durchgeführt werden, um das System stabil zu halten:

- Offene Dateien schließen, Einträge in die Protokolldatei schreiben
- Alle Sohnprozesse an den `init` Prozeß übergeben
- Freigeben des Adreßraums, der Tabellen und des Auslagerungsspeichers
- Einfügen in die Warteschlange der sog. Zombie-Prozesse, die auf die restlose Beseitigung durch den Vaterprozeß warten

Der Wechsel von einem Prozeß zum nächsten durch den Scheduler erfordert ebenfalls eine Reihe von Maßnahmen:

- Sichern des Hardware-Kontextes auf dem Stack
- Laden des neuen Hardware-Kontextes
- Kennzeichnung des alten Prozesses als *rechenbereit*
- Vorbereiten des Stacks für den nächsten Prozess
- Auswahl des nächsten Prozesses aus der Warteschlange
- Einrichten des neuen virtuellen Adeßraums

Es ist offensichtlich, daß diese Vorgänge, besonders das Anlegen des Stacks und die Einrichtung des prozeßeigenen Adreßraums, eine größere Anzahl von Taktzyklen benötigen und daher häufiges Erzeugen und Beenden von Prozessen oder rascher Wechsel zwischen den laufenden Prozessen im Abstand von wenigen Taktzyklen stark leistungsminderd wirkt.

3.2 Threads unter Unix

Das schnelle Erzeugen von nebenläufigen Programmabschnitten und der schnelle Wechsel zwischen diesen Abschnitten wird jedoch, wie in Kapitel 3.5 dargelegt wird, in bestimmten Fällen benötigt. Zu diesem Zweck bieten fast alle UNIX-Derivate *Threads* an. Damit können innerhalb einer Applikationen mehrere voneinander möglichst unabhängige Aufgaben parallel[1] bearbeitet werden, ohne dabei den beschriebenen Überhang von mehreren UNIX-Prozessen tragen zu müssen. Ein Beispiel für eine solche Applikation ist ein Client-Server System: jede Anfrage eines Clients an den Server kann unabhängig von anderen Anfragen entweder duch einen eigenen Prozeß oder aber durch einen Thread bearbeitet werden.

Was ist aber nun der Unterschied zwischen einem Thread und einem Prozeß? Das Grundkonzept der Threads ist die Einführung zusätzlicher *Kontrollpunkte* innerhalb eines Prozesses, zwischen denen gewechselt werden kann. Im Grunde ist ein Thread also nur durch einen weiteren Programmzähler innerhalb eines Prozesses definiert. Er benötigt z.B. keine eigene `proc`-Struktur, diese Informationen verwaltet der Prozeß. Dadurch ergibt sich, daß alle Threads eines Prozesses im gleichen Adreßraum arbeiten – eben in dem des Prozesses, der sie erzeugt hat. Auch die anderen Resourcen eines Prozesses (offene Dateien, Benutzerinformationen usw.) sind für alle Threads eines Prozesses erreichbar. Ein Thread hat aber dennoch notwendige private Objekte wie eben seinen Programmzähler, den Inhalt der Register und implementationsabhängig einen Stack und weitere Datenstrukturen, auf die er exklusiven Zugriff hat. Daraus ergeben sich bei bestimmten Aspekten grundlegende Unterschiede zwischen Prozessen und Threads, die nachfolgend erläutert werden. Die quantitative Ausprägung dieser Unterschiede hängt jedoch stark von der Hardwarearchitektur des Rechners (insbesondere des Prozessors), den Möglichkeiten des Betriebssystems und der Art der Implementierung der Threads ab.

- **Geschwindigkeit:** die Zeit, die zur Erzeugung und Vernichtung sowie zum Wechsel benötig wird, ist bei Threads sehr viel geringer als bei Prozessen. Dies wird deutlich wenn man bedenkt, das bei Threads fast alle Vorgänge wegfallen, die im vorhergehenden Kapitel bei den Prozessen aufgeführt waren: es muß kein Adreßraum angelegt werden, Threads sind untereinander nicht verkettet, haben keine eigenen E/A-Schnittstellen oder Stack. Häufig muß für einen Wechsel zwischen Threads noch nicht einmal der Benutzeradressraum verlassen werden. Dies erspart eine Menge Zeit.

- **Speicherverbrauch:** Threads haben keinen eigenen Adreßraum und können mit dem gleichen Code auf denselben Daten arbeiten, was bei parallel gestarteten Prozessen nicht ohne erheblichen Aufwand möglich ist. Daher können auf einem gegebenen System weitaus mehr Threads gestartet und betrieben werden als es mit Prozessen möglich ist. Diese verstärkte Nebenläufigkeit kann leistungssteigernd eingesetzt werden, wie es auch in dieser Arbeit versucht wird.

[1] Damit kann echte Parallelität auf mehreren Prozessoren oder pseudoparallele Verarbeitung (Nebenläufigkeit) auf einem einzigen Prozessor gemeint sein; entscheinend ist hier die logische Parallelität im Programmablauf.

- **Kommunikationsaufwand:** Die Kommunikation zwischen Prozessen ist selbst auf einem Uniprozessor vergleichsweise aufwendig, da sie keinen gemeinsamen Adreßraum haben. Die verwendeten Verfahren reichen von der Kommunikation mittels des Dateisystems (z.B. Lockdateien) über *named pipes* bis hin zu Nachrichtenversand, um die Adreßraumgrenzen zu überwinden. Diese Mechanismen sind für viele Anwendungsfälle ausreichend, aber zur Parallelverarbeitung gemeinsamer Daten deutlich zu langsam. Hierzu braucht man vor allem zur Synchronisation eine Kommunikation mit minimaler Latenz, wie sie durch den Zugriff auf gemeinsame Speicherstellen einfach realisiert werden kann.

3.3 Implementierung von Threads

Die Verwendung mehrerer Kontrollpunkte innerhalb eines Prozesses ist das Konzept der Threads[40]. Es gibt jedoch verschiedene Modelle, wie ein Thread im System verwaltet und ausgeführt wird und wie der Wechsel zwischen den Kontrollpunkten gesteuert wird. Dabei unterscheidet man drei wesentliche Typen: *Kernelthreads*, *Userthreads* und sogenannte *leichtgewichtige Prozesse*. Daneben gibt es noch Variationen dieser Klassen.

3.3.1 Kernelthreads

Wie der Name schon vermuten läßt, ist ein *Kernelthread* nicht mit einem Benutzerprozeß verbunden, sondern wird unabhängig innerhalb des Kerneladreßraums verwaltet und ausgeführt. Er entspricht also nicht ganz der oben gegebenen Vorstellung im Sinne der inneren Nebenläufigkeit von Applikationen, sondern wird nach Bedarf für interne Operationen des Betriebssystems wie der Bereitstellung von asynchoner E/A verwendet. Auch die vielfältigen Dämonen, die in UNIX Systemen im Hintergrund ihre Aufgaben versehen, sind in in der Regel als Kernelthreads implementiert.

Kernelthreads werden unabhänig verwaltet und verwenden die Standardmechanismen zur Synchronisation wie `sleep()` und `wakeup()`. Die einzigen Resourcen, die sie belegen, sind der Kernelstack und ein Bereich, im dem der Registerinhalt zwischengespeichert werden kann. Dazu kommt eine Datenstruktur für Informationen zum Scheduling und zur Synchronisation. Kontextwechsel zwischen Kernelthreads gehen, wie erwartet, schnell vonstatten, da sie alle den gleichen Adressraum, den des Kernels, haben und daher die Speicherzuordnung nicht geändert werden muß.

3.3.2 Leichtgewichtige Prozesse

Leichtgewichtige Prozesse[2] (LWP) sind im Gegensatz zu Kernelthreads einem Prozeß zugeordnet und arbeiten in seinem Adreßraum. Jedoch ist jedem LWP ein Kernelthread zugeordnet, so daß die Kontrolle über den LWP beim Kernel liegt. Dadurch vereinfacht

[2]Mitunter werden auch Userthreads als leichtgewichtige Prozesse bezeichnet. Hier wird jedoch zwischen Userthreads und LWPs unterschieden.

	Erzeugung [μs]	Synchronisation [μs]
User thread	52	66
LWP	350	390
Process	1700	200

Tabelle 3.1: Latenzzeiten für Userthread, LWP und Prozeßoperationen auf einer SPARCstation2 (nach [40])

sich die Implementierung der LWPs, die somit ggf. vom Kernel (für die Applikation automatisch) verwaltet oder auch auf mehrere Prozessoren verteilt werden können. Jedoch bringt es Leistungseinbußen mit sich, da jede Operation auf einen LWP, etwa Erzeugung, Beseitigung oder Synchronisation, einen Systemaufruf erfordert. Dies erzwingt jedesmal einen Wechsel vom Benutzermodus in den Kernelmodus und zurück, der wegen der nötigen Gültigkeitsüberprüfungen des Kernels zeitaufwendig ist. In vielen Fällen sind reine Userthreads, wie sie im folgenden Kapitel beschrieben werden, für die Applikation vorteilhafter, wenn nicht das Kriterium der Verwaltung und Verteilbarkeit der Threads durch den Kernel entscheidend ist.

3.3.3 Userthreads

Im Gegensatz zu den Kernelthreads und den darauf aufbauenden leichtgewichtigen Prozessen sind *Userthreads* völlig unabhänig vom Kernel; sie werden von diesem auch gar nicht wahrgenommen. Ihre Verwaltung obliegt einer Bibliothek, die für Verwaltung, Kontextwechsel und Synchronisation ausschließlich im Benutzeradreßraum operiert. Sie stellt für die Threads sozusagen einen Minikernel dar. Die Userthreads wiederum können direkt einem Prozeß zugeordnet werden. Das hat aber zur Folge, daß ein blockierender Systemaufruf diesen Prozeß und alle seine anderen Threads blockiert, was natürlich nicht erwünscht ist. Daher ist es üblich, die Userthreads auf eine Anzahl von LWPs abzubilden. Diese Abbildung muß nicht zwangsläufig eine 1:1-Abbildung sein. Effizienter ist es in der Regel, eine Anzahl von Userthreads auf eine kleinere Zahl von LWPs abzubilden, so daß bei einem blockierenden Aufruf eines Threads die anderen Threads auf andere LWPs abgebildet werden können.

Der große Vorteil von Userthreads liegt in ihrem sehr geringen Resourcenverbrauch: Resourcen des Kernel werden nur belegt, wenn die Threads an LWPs gebunden sind, und auch dann ist der Verbrauch pro Thread durch die individuelle Abbildung der Threads auf die LWPs reduzierbar. Ansonsten belegt jeder Userthread nur seinen Stack und einen Bereich zur Sicherung des Registerkontextes und anderer Statusinformationen. Diese Daten werden im Benutzeradreßraum abgelegt. Dadurch sind zur Verwaltung der Threads keine Aufrufe von Systemfunktionen nötig, was sich positiv auf das Zeitverhalten der Threads auswirkt. Tabelle 3.1 gibt eine Vorstellung davon, welche Leistungsunterschiede möglich sind.

Andererseits werfen Userthreads auch Probleme auf. Ein Round-Robin-Scheduling ist schwierig zu realisieren, denn dazu benötigt der Prozeß der Threadbibliothek ein Uh-

rensignal. Dies wiederum kann auch von den Threads benötigt werden, was zu Konflikten führt. Daher sind rein Userthreads häufig kooperativ verwaltet, d.h. ein Thread muß freiwillig den Prozessor abgeben, damit ein anderer Thread bearbeitet werden kann. Ein weiteres Problem ist der Aufruf von potentiell blockierenden Systemfunktionen: im Fall der Blockierung sind alles Userthreads blockiert. Somit wird durch Userthreads zwar in jedem Fall die Nebenläufigkeit einer Applikation erreicht, aber nicht unbedingt die Parallelität auf einem Multiprozessor, wenn mehrere Threads auf den gleichen LWP bzw. Prozeß abgebildet werden. Auch auf Ereignisse innerhalb des Kernels könne Userthreads nicht bewußt reagieren.

3.3.4 Variante Implementierungen

Very Ligthweighted Threads

Die besondere Eigenschaft von *Very Lightweighted Threads* (VLWT, [28]) ist es, daß die Spezifikation des Threads von seiner Ausführung getrennt ist. Dadurch ist es möglich, die Belegung des Ausführungskontextes bis zum letzten Moment (bis zur tatsächlichen Ausführung des Threads) zu verzögern. Da dieser Ausführungskontext unter anderem den Stack und den Heap des Threads beinhaltet, stellt er eine recht große Datenstruktur dar, wohingegen die Datenstruktur zur Spezifikation des Threads ausgesprochen klein ist. Dadurch können mehr Threads spezifiziert werden (z.B. in rekursiven Algorithmen), ohne das System zu überlasten. Zur tatsächlichen Ausführung können die VLWT-Spezifikationen dann etwa mit geringem Aufwand auf die Prozessoren verteilt werden. Die Migration eines Threads von einem Prozessor auf einen anderen ist sehr viel schneller, solange der Thread nocht nicht ausgeführt wird.

Run-to-completion Threads

Eine spezielle Variante von Threads sind die *run-to-completion Threads* (RTC, [6]). Sie sind durch den Verzicht auf preemptives Scheduling, blockierendes Warten und E/A-Kanäle wiederum deutlich leichtgewichtiger als Userthreads, aber aufgrund dieser Einschränkungen auch nicht universell einsetzbar. Zur Ausführung ihrer Aufgabe benötigen sie nur einen Zeiger auf den Code, einen Zeiger auf die Argumente sowie Zugriff auf den Speicher im Adreßraum, um globale Variablen zu erreichen. Sie kommen vorwiegend in parallelisierenden Compilern zum Einsatz.

Filaments

Filaments [19] sind eine Implementierung von Userthreads mit speziellen Eigenschaften. Die wichtigste Eigenschaft der Filaments ist wie bei den run-to-completion Threads die Verwendung von zustandslosen Threads. Durch diese Einschränkung benötigen Filament-Threads keinen privaten Stack, was zu extrem kurzen Latenzzeiten für den Kontextwechseln führt. Natürlich macht diese Einschränkung Filament-Threads nur für spezielle Zwecke, etwa die Abarbeitung einer Schleife ohne Aufruf eines Unterprogramms oder

einer Systemfunktion, nutzbar, dann aber mit höherem Leistungsgewinn als bei vollwertigen Threads.

Weitere Eigenschaften der Filaments sind unter anderem die Möglichkeit, die Threads gezielt zu plazieren, um eine höhere Datenlokalität zu erreichen (siehe auch [38]) und Einsatz von *continuations* (siehe Kapitel 3.3.5).

3.3.5 Continuations

Eine weitere Methode zur Verringerung der von einem Thread belegten Ressourcen (und damit zur Steigerung der Systemleistung) ist der Einsatz von *Continuations*, wie es im Mach Kernel geschieht. Die Idee hinter den Continuations ist, daß nicht jeder Thread im Kernel einen eigenen Ausführungskontext, vor allem der Stack und der Heap als relativ große Datenstruktur, haben muß. Stattdessen gibt der Thread vor dem Übergang in den blockierten Status eine Funktion an, mit der er nach der Blockade fortfahren will[3]. Diese Funktion, die die Continuation darstellt, kann jedoch später nicht wie eine normale Funktion in den alten Ausführungskontext zurückkehren, da dieser ja nicht mehr existiert. Sie kann nur wiederum andere Funktionen bzw. Continuations aufrufen.

Es treten zwei Arten von Kontrollübergängen auf, bei denen Continuations eingesetzt werden können: ein Thread überschreitet aufgrund eines Traps oder Fehlers die Grenze vom Benutzer- in den Kerneladreßraum oder die Kontrolle geht innerhalb des Kernels von einem Thread auf einen anderen über. Dabei können aufgrund der Eigenschaften der Continuations verschiedene Optimierungen zum Einsatz kommen: *Verwerfen des Stacks, Stackübergabe* und *Continuation-Erkennung*. Das Verwerfen des Stacks ist das normale Verhalten, was der Grundidee der Continuations entspricht, indem statt des gesamten Ausführungskontextes nur die Referenz auf die Continuation-Funktion gesichert wird. Wenn nun aber nach dem Thread, der gerade unter Angabe einer Continuation blockiert, ein Thread, der zuvor ebenfalls unter Verwendung einer Continuation blockierte, ausgeführt werden soll, so kann dieser direkt den Stack übernehmen, den der vorhergehende Thread verwendet hat. Dies reduziert den Speicherbedarf des Kernels sowie die Zahl der Transfers im Speicher, wodurch die Effizienz der Prozessor-Caches und der Seitendeskriptortabellen gesteigert wird. Optimierung durch Continuation-Erkennung basiert darauf, daß der Kernel zur Laufzeit die vom Thread angegebene Continuation-Funktion als bekannt erkennt und stattdessen zur Wiederaufnahme der Ausführung des Threads eine spezielle und schnellere Routine verwendet.

Der Einsatz von Continuations beim Übergang vom Benutzer- in den Kerneladreßraum kann auch zur Beschleunigung von *Remote Procedure Calls* (RPC), der Behandlung von Ausnahmezuständen (Exception Handling), dem preemptiven Scheduling von Threads sowie der Behandlung von Seitenfehlern im Benutzeradreßraum dienen.

[3] Da die Angabe einer solchen Funktion nicht in jedem Fall möglich oder mitunter zu kompliziert ist, werden Continuations in der Regel als Alternative zum herkömmlichen Vorgehen bei nebenläufiger Programmierung angeboten.

3.3.6 Zusammenfassung

Um eine Gesamtansicht der verschiedenen Typen von Threads zu geben, werden die vorgestellten Implementierungen mit ihren wichtigsten Eigenschaften in Tabelle 3.2 dargestellt. In der Tabelle finden sich Verweise auf Bemerkungen, die Eigenschaften beschreiben, welche sich nicht in die Tabelle einordnen lassen. Diese Bermerkungen sind direkt im Anschluß zu finden.

Typ	Adreß-raum	Benutzer-prozeß	eigener Stack	Scheduling durch	Bemerkung (s.u.)
Kernel	Kernel	nein	ja	Kernel	
LWP	User	ja	ja	Kernel	1.
User	User	ja	ja	User	
VLWP	User	ja	ja	User	2.
RTC	User	ja	nein	entfällt	

Tabelle 3.2: Verschiedene Typen von Threads

Bemerkungen zu Tabelle 3.2:

1. Ein oder mehrere Userthreads sind an einen Kernelthread gebunden
2. Trennung der Spezifikation des Threads von seinem Ausführungskontext

3.4 Ein Standard: POSIX-Threads

Die Entwicklung und Implementierung von Threads in verschiedenen Betriebssystemen auf unterschiedlichen Plattformen brachte auch eine entsprechende Anzahl von Programmiermodellen und -schnittstellen mit sich. Diese nutzen zwar unter Umständen die speziellen Vorraussetzung des Betriebssystems und der Hardware besonders gut aus, erschweren aber das Portieren von Applikationen, die Threads verwenden. Im Zuge der Gestaltung des POSIX-Standards für offene Systeme wurde im Juli 1995 von der IEEE auch ein POSIX-Standard für Threads verabschiedet. Die Implementierungen dieser Programmierschnittstelle (Application Programming Interface, API) werden im Allgemeinen als *pThreads* bezeichnet. Sie sind mittlerweile auf allen wesentlichen UNIX-Plattformen verfügbar. Daher wird dieses API hier ausführlicher vorgestellt. Dem POSIX-API sehr ähnlich ist das ältere DCE[4]-API, das z.B. unter dem OSF/1 der Intel Paragon zur Verfügung steht.

Bis zur Verabschiedung des aktuellen Standards POSIX 1003.4a *Threads Extension* (Draft 10) gab es eine entsprechende Anzahl von Entwürfen (Drafts), die von einigen Herstellern bereits implementiert wurden, dann aber mitunter nicht mehr aktualisiert worden sind. Zum anderen wurde häufig nicht der komplette Standard, sondern nur Teile davon implementiert. Dies muß berücksichtigt werden, wenn portable Anwendungen entworfen werden.

[4] DCE = Distributed Computing Environment

Typ	Beschreibung
pthread_attr_t	Threadattribute
pthread_mutexattr_t	Attribute eines Mutex Locks
pthread_condattr_t	Attribute einer Bedingungsvariablen
pthread_mutex_t	Mutex Lock
pthread_cond_t	Bedingungsvariable
pthread_t	Threadkennung
pthread_once_t	Einmal-Ausführung zur Initialisierung
pthread_key_t	Schlüssel für private Daten eines Threads

Tabelle 3.3: POSIX.1c Typen

Namensgebung, Typen und Konventionen

Die Namensgebung für Typen und Funktionen basiert auf einem einheitlichen Schema. Für Typen ist dies:

`pthread_`*object*`_t`

In Tabelle 3.3 sind die acht Typen des Standards angegeben. Die Funktionen werden nach dem Schema

`pthread_`*object_operation*`_NP`

benannt. Dabei weist *object* jeweils auf den Typ hin (kann weggelassen werden, falls es sich um einen Thread handelt), *operation* bezeichnet die Funktion, die ausgeführt wird und *NP* ist für Implementierungsspezifische, nicht-portable Erweiterungen reserviert.

Alle POSIX.1c-Funktionen liefern bei erfolgreicher Ausführung Null (0) zurück, andenfalls einen `errno` Wert.

Threads

Dieser Abschnitt behandelt die Funktionen, die zur Erzeugung und Verwaltung von Threads benötigt werden.

- `pthread_create()`
  Erzeugt einen neuen Thread, wobei ihm Attribute übergeben werden können, die sein Scheduling oder andere Parameter festlegen. Ein Pflichtparameter ist die initale Funktion des Threads, mit der dieser seine Ausführung beginnt. Es ist nicht möglich, eine Anzahl von Threads auf einmal zu erzeugen, dies muß durch wiederholte Aufrufe erfolgen. Mittels der hierbei generierten Thread-Kennung kann der erzeugende Thread auf das Ende des erzeugten Threads warten (siehe `pthread_join()`).
- `pthread_detach()`
  Startet einen abgekoppelten Thread. Dies wird bei manchen Implementierungen jedoch über die normale Funktion `pthread_create()` erreicht, wenn sie mit

dem entsprechenden Attribut als Parameter aufgerufen wird. Wenn ein Thread mittels dieser Funktion entkoppelt wurde, kann auf ihn kein `pthread_join()` mehr ausgeführt werden, da die zugehörigen Datenstrukturen freigegeben wurden.

- `pthread_self()`
  Liefert die numerische Kennung des aufrufenden Threads zurück.

- `pthread_equal()`
  Vergleicht zwei Thread-Kennungen auf Identizität. Üblicherweise sind die Thread-Kennungen ganze Zahlen und könnten somit auch mit einer einfachen Vergleichsoperation behandelt werden; über `pthread_equal()` erfolgt jedoch eine Abstraktion, die völlige Unabhängigkeit von der unterliegenden Datenstruktur bietet.

- `pthread_exit()`
  Beendet den Thread an dieser Stelle (und nicht erst am Ende der initialen Funktion).

- `pthread_join()`
  Blockiert den aufrufenden Thread solange, bis das der Thread, dessen Kennung der Funktion übergeben wurde, an das Ende seiner Ausführung gelangt (entweder durch Erreichen des Endes der initialen Funktion oder durch Aufruf von `pthread_exit()`).

- `pthread_setschedparam()` und `pthread_getschedparam()`
  Setzt bzw. gibt die Parameter für das Scheduling wie die verwendete Strategie zum Wechsel zwischen den Threads (FIFO, Round Robin oder andere) oder die Priorität. Viele Implementierungen bieten jedoch nur ein festes Schedulingverfahren. In dem Fall sind beide Funktionen obsolet. Die Parameter werden in einem Attribut gespeichert (siehe folgendes Kapitel).

Attribute

Attribute eröffnen die Möglichkeit, spezielle Eigenschaften der zu startenden Threads zu bestimmen. In den meisten Implementierungen ist so die Größe des Stacks bestimmbar; selten kann auch die Art des Scheduling und die Priorität eines Threads individuell bestimmt werden. Auch die Eigenschaften der Synchronisationsmittel (siehe folgende Abschnitte) wie Locks und Bedingungsvariablen werden über zugehörige Attribute gesteuert.

- `pthread_attr_init()`
  Jedes Attribut muß vor der Verwendung initialisiert werden, um keinen unsinnigen Werte zu enthalten. Ein derart initialisiertes Attribut kann danach an die eigenen Bedürfnisse angepaßt werden. Ein Attribut ist jedoch nicht fest an einen Thread gebunden, da es einen prozeßweiten Sichtbarkeitsbereich hat. Daher kann es immer wieder bei der Erzeugung neuer Threads verwendet werden.

- `pthread_attr_destroy()`
  Beseitigt ein Attribut nach der Verwendung. Gegebenenfalls kann es danach wieder neu initialisiert werden.

Eigenschaft	Beschreibung
Threads	
`detachstate`	Automatisches entkoppeln eines neuen Threads?
`inheritsched`	Werden die Scheduling-Attribute vom Vater-Thread geerbt oder vom Attribut-Objekt übernommen?
`schedparam`	Setzen der initialen Priorität
`schedpolicy`	Setzen der Scheduling-Regeln
`stacksize`	Setzen der Stackgröße
`stackaddr`	Angabe der Adresse des Stackspeichers
Mutexes	
`protocol`	Welche Priorität erhält der Thread, der diese Mutex belegt (die der Mutex oder die eines Threads, der die Mutex bereits hält)?
`prioceiling`	Setzen der minimalen Priorität

Tabelle 3.4: POSIX.1c Attributeigenschaften

- `pthread_attr_set`*eigenschaft*`()`
  Setzt eine bestimmte Eigenschaft in einem Attribut. In Tabelle 3.4 sind einige der zulässigen Attributeigenschaften aufgeführt[5]. Um einem Objekt diese Eigenschaften zu verleihen, muß bei der Erzeugung des Objekts auf dieses Attribut Bezug genommen werden.

- `pthread_attr_get`*eigenschaft*`()`
  Legt die aktuellen Parameter für eine bestimmte Eigenschaft des Objekts, die durch *eigenschaft* spezifiziert wird, in der Parameterstruktur ab. Objekte können in diesem Zusammenhang Threads, Mutex Locks oder Bedingungsvariablen sein. Dadurch kann man z.B. die Priorität eines Threads erfahren, nötigenfalls ändern und an den Thread zurückgeben. Bestimmte Eigenschaften eines Threads können jedoch nur von dem Thread selber erfahren werden; einzig die Schedulingparameter können von außen beeinflußt werden.

Mutex Locks

Mutex Locks sind eine grundlegende Einrichtung zur Synchronisation. Sie dienen dazu, den Zugriff von nebenläufigen Programmteilen auf eine gemeinsame Ressource (etwa eine globale Variable) zu serialisieren, um die Konsistenz der Daten zu bewahren. In Kapitel 4.2 finden sich weitere Informationen über die Verwendung von Mutex Locks.

- `pthread_mutex_init()`
  Initialisiert einen Mutex Lock. Nach diesem Aufruf kann er benutzt werden. Es ist zu beachten, daß der Mutex Lock nur einmal initialisiert werden darf und nicht

[5] Auf dem primären Zielsystem Intel Paragon sind jedoch diese Attribute nicht implementiert

etwa von jedem Thread, da ihr Sichtbarkeitsbereich prozeßweit ist. Daher sollten die Mutex Locks, die verwendet werden, vom Prozeß initialisiert werden.

- `pthread_mutex_destroy()`
  Beseitigt einen Mutex Lock und gibt die belegten Ressourcen frei. Ein Mutex Lock sollte nur dann beseitigt werden, wenn sichergestellt ist, daß er nicht mehr belegt ist.

- `pthread_mutex_lock()`
  Mit diesem Aufruf versucht ein Thread einen Programmabschnitt zu betreten, der durch einen Mutex Lock gesichert ist. Wenn sich bereits ein anderer Thread in diesem Abschnitt befindet, hat dieser den Lock belegt. Folgende Aufrufe blockieren dann, bis der Lock wieder freigegeben wird. Wenn mehrere Threads auf den Lock warten, kann nicht davon ausgegangen werden, daß die Threads den Lock in der Reihenfolge erhalten, in der die Aufrufe von `pthread_mutex_lock()` erfolgt sind.

- `pthread_mutex_trylock()`
  Versucht ebenfalls, einen Mutex Lock zu belegen. Im Gegensatz zu `pthread_mutex_lock()` blockiert dieser Aufruf jedoch nicht, wenn der Lock nicht verfügbar ist, sondern kehrt mit einer Fehlermeldung unmittelbar zurück.

- `pthread_mutex_unlock()`
  Gibt einen belegten Mutex Lock wieder frei. Wenn in der Kombination von `lock()` und `unlock()` nicht sorgfältig darauf geachtet wird, daß jeder belegt Lock auch garantiert nach endlicher, möglichst kurzer Zeit wieder freigegeben wird, kann es zu Deadlocks kommen.

Bedingungsvariablen

Bedingungsvariablen werden dazu benutzt, um ohne den Verbrauch von Rechenzeit auf ein bestimmtes Ereignisses (d.h. einen bestimmten Zustand der Bedingungsvariablen) zu warten. Über den Eintritt dieses Ereignisses werden die wartenden Threads durch den Thread, der das Ereignis auslöst, informiert (signalisiert). Dabei muß beachtet werden, daß diese Signale nicht gesammelt werden, um von einem später an der Bedingungsvariablen eintreffenden Thread ausgewertet zu werden. Vielmehr geht ein solches Signal verloren, wenn nicht zur *gleichen Zeit* ein Thread auf dieses Signal wartet. Dies kann zu Deadlocks führen, wenn nicht wie in Abbildung 3.1 neben dem obligatorischen Mutex Lock mit einer zusätzlichen boolschen Variablen gearbeitet wird. Damit wird verhindert, daß ein Thread auf ein Ereignis wartet, das schon längst eingetreten ist und u.U. nie wieder eintreten wird.

- `pthread_cond_init()`
  Initialisierung einer Bedingungsvariablen.

Algorithmus 3.1 Deadlocksichere Verwendung von Bedingungsvariablen

```
int event_occured = FALSE;
pthread_condvar_t *event_cv;
pthread_mutex_t *event_mtx;

void signaling_thread()
{
  pthread_mutex_lock (event_mtx);
  event_occured = TRUE;
  pthread_cond_signal (event_cv);
  pthread_mutex_unlock (event_mtx);
}

void waiting_thread()
{
  pthread_mutex_lock (event_mtx);
  while (!event_occured)
    /* gibt Lock ab und wartet an Bedingungsvariablen */
    pthread_cond_wait (event_cv, event_mtx);

  /* Lock endgueltig abgeben */
  pthread_mutex_unlock (event_mtx);
}
```

- `pthread_cond_destroy()`
  Freigeben der durch die Bedingungsvariablen belegten Resourcen. Vor der Freigabe muß sichergestellt sein, daß keine Threads mehr an dieser Bedingungsvariablen warten, um Deadlocks zu vermeiden.

- `pthread_cond_wait()`
  Sobald ein Thread diese Funktion aufruft, wartet er auf das Eintreten des zu der Bedingungsvariablen gehörenden Ereignisses, was ihm mittels der Signalisierung durch einen anderen Thread mitgeteilt wird. Er gibt also den Prozessor an dieser Stelle ab und kann ihn nur noch durch einen anderen Thread zurückerhalten.

- `pthread_cond_timedwait()`
  Mittels dieses Aufrufs wartet der Thread nur eine maximale Zeit auf die Signalisierung durch einen anderen Thread, bevor er wieder als ausführbereit eingestuft wird. Dies kann zur Vermeidung von Deadlocks eingesetzt werden, was üblicherweise aber auch anders sichergestellt werden kann. Daher ist diese Funktion nicht auf allen Systemen implementiert.

- `pthread_cond_signal()`
  Der aufrufende Thread signalisiert *einem* der (möglicherweise) an der Bedingungsvariablen wartenden Threads, daß die entsprechende Bedingung eingetreten ist. Dieser erhält aber zu diesem Zeitpunkt nicht notwendigerweise sofort den Prozessor. Es ist auch nicht bestimmbar, welchem Thread signalisiert wird, wenn mehrere Threads an der betreffenden Bedingungsvariablen warten sollten.

- `pthread_cond_broadcast()`
  Alle an der betreffenden Bedingungsvariablen wartenden Threads werden durch diesen Broadcast signalisiert. Sie kommen jedoch nicht sofort zur Ausführung (d.h. der aufrufende Thread behält möglicherweise den Prozessor), und die Reihenfolge, in der die Threads schließlich zur Ausführung kommen, ist ebenfalls nicht bestimmbar und unabhängig von der Reihenfolge, in der sie an der Bedingungsvariablen angelangt sind.

3.5 Überlappung von Latenzzeiten durch Multithreading

Ein Einsatzgebiet von Threads, das auch den Schwerpunkt dieser Arbeit darstellt, ist das Verbergen von kommunikationsbedingten Latenzzeiten in Parallelrechnern durch den Einsatz von Multithreading. In jeder realen parallelisierten Anwendung tritt mehr oder weniger Kommunikation zwischen den einzelnen Prozessoren auf. Diese Kommunikation findet auf NORMA-Systemen durch den Austausch von Nachrichten (*message passing*) oder den darüber softwaremäßig realisierten Zugriff auf entfernten Speicher statt; auf Systemen mit gemeinsamen Speicher führt der Zugriff auf diesen Speicher ebenfalls zu Latenzzeiten.
Die Auslastung der Prozessoren kann jedoch verbessert werden, wenn während dieser Latenzzeiten nicht nur auf das Eintreffen des Ergebnisses der Kommunikation gewartet wird, sondern andere Aufgaben erledigt werden, die mit der laufenden Kommunikation in keinem Zusammenhang stehen und selber nicht wiederum Kommunikation auslösen. Diese Überlappung von Kommunikation und Ausführung kann zum einen explizit programmiert werden, indem ein asynchroner Befehl zur Kommunikation vor einer Anzahl von Befehlen auf lokal vorhandene Daten abgesetzt wird, bevor nach erfolgter Abarbeitung dieser Befehle mit dem Ergebnis der Kommunikation weitergearbeitet wird (sieh auch Kapitel 3.5.2). Diese Vorgehensweise ist aber nur möglich, wenn in dem verwendeten Programmiermodell Kommunikation explizit erfolgt. Gerade in modernen Programmiermodellen (z.B. HPF oder SVM-Fortran mit virtuell gemeinsamen Speicher) wird jedoch versucht, das Erstellen von parallelen Anwendungen dadurch zu vereinfachen, daß sich der Programmierer nicht mehr selber um die Abwicklung der Kommunikation zu kümmern braucht. Diese wird implizit abgewickelt. Der Programmierer kann daher gar nicht mehr genau wissen, wo denn Kommunikation beim Ablauf seines Programmes stattfinden wird, es sei denn, er benutzt Werkzeuge, die zur Laufzeit des Programmes sein Kommunikationsverhalten protokollieren. Dadurch können die Stellen, an denen Leistungsverlust durch Kommunikationslatenz auftritt, ermittelt werden. Sodann muß ein Ansatz gefunden werden, diese Leistungseinbußen durch Überlappung von Kommunikation und Ausführung zu minimieren.

Ein solcher Ansatz ist der Einsatz von Threads. Das Prinzip ist einfach: obwohl die Kommunikation implizit abgewickelt wird, weiß der Programmierer entweder aus dem Code, seiner Erfahrung oder durch den Einsatz der angesprochenen Werkzeuge, in welchen Abschnitten seines Programms Kommunikation auftreten wird. Eine Schleife etwa, in der Zugriffe auf ein größeres Feld erfolgen, birgt potentielle Kommunikation. Daher unterteilt der Programmierer oder idealerweise der Compiler diese Schleife in mehrere unabhängig voneinander zu verarbeitende Abschnitte und verteilt diese auf eine gleiche Anzahl von Threads. Wenn nun ein Thread aufgrund von Kommunikation warten muß, schaltet das Betriebssystem oder gegebenenfalls das Laufzeitsystem zum nächsten ausführbereiten Thread um. Damit diese Strategie tatsächlich zu einer Leistungssteigerung führt, sind einige Rahmenbedingungen zu beachten, um eine geeignete Parametrisierung zu erreichen. Diese werden im folgenden Kapitel ermittelt.

3.5.1 Analytisches Modell zur Effizienz

In [33] findet sich ein einfaches analytisches Modell für eine Architektur, die Kommunikation und Berechnung durch Multithreading überlappt. Das Modell beruht auf vier Parametern:

- **L** ist die Latenz, die durch Kommunikation auftritt. In der einfachsten Version des vorgestellten Modells ist L konstant. Ein erweitertes, realistischeres Modell nimmt L als eine geometrisch verteilte Größe an, was zu einer Markovkette führt. In der Realität ist diese Größe von der System-Hardware, aber auch vom Verhalten des eigenen Programm und anderer auf dem System laufender Programme abhängig und daher stochastisch.
- **N** stellt die zur Laufzeit konstante Anzahl der Threads auf einem Prozessor dar. Sie kann entweder bereits zur Compilezeit oder erst zur Laufzeit vom Programmierer bzw. der Laufzeitumgebung bestimmt werden.
- **C** repräsentiert die Anzahl an Zyklen, die für einen Kontextwechsel zwischen zwei Threads verloren gehen. Dies ist eine zeitlich konstante Größe, die von der Hardware und dem Laufzeitsystem, also der verwendeten Thread-Bibliothek, abhängt.
- **R** steht für die zeitliche Distanz zwischen zwei Kontextwechseln aufgrund von Kommunikation. R setzt sich aus Programmverhalten und Speichermodell des Systems zusammen und ist wie L eine stochastische Größe.

Der Wirkungsgrad eines Prozessors in diesem Modell definiert sich allgemein wie folgt:

$$\epsilon_P = \frac{Busy}{Busy + Switching + Idle}$$

Hierbei ist *Busy* die Zeit, in der der Prozessor an seiner originären Aufgabe arbeitet, *Switching* die Zeit, die er bei Kontextwechseln zwischen Threads verbraucht sowie *Idle* die beim Warten verstrichene Zeit.

Einige grundlegende Zusammenhänge lassen sich daraus schnell herleiten, wenn man R und C konstant annimmt. Für einen Prozessor mit nur einem Thread ergibt sich dann eine Effizienz von

$$\epsilon_1 = \frac{R}{R + L} = \frac{1}{1 + \frac{L}{R}}$$

Der starke Leistungsabfall eines solchen Systems bei großer Kommunikationslatenz wird aus dieser Formel deutlich. Die maximal erreichbare Effizienz eines Prozessor mit mehreren Threads ergibt sich analog zu

$$\epsilon_{max} = \frac{R}{R + C} = \frac{1}{1 + \frac{C}{R}}$$

da hierbei alle Latenzzeiten wegfallen, jedoch Verluste durch die erforderlichen Kontextwechsel anfallen. Diese sollten demnach möglichst deutlich kürzer sein als die Kommunikationslatenz, um die Effizienz zu verbessern. Die maximale Effizienz kann dann

erreicht werden, wenn jederzeit ein ausführbereiter Thread zur Verfügung steht, um einen blockierten Thread abzulösen. Diesen Zustand bezeichnet man als *Sättigung*, die sich dann einstellt, wenn der Prozessor mehr Zeit damit verbringt, andere Threads zu bearbeiten als zur Abwicklung einer Kommunikationsanforderung benötigt wird, wenn also $(N-1)(R+C) > L$ gilt. Die zur Sättigung notwendige Anzahl von Threads ergibt sich daraus als

$$N_{Sättigung} = \frac{L}{R+C} + 1 \tag{3.1}$$

Für $N < N_{Sättigung}$, wenn also der Prozessor nicht bei jedem fälligen Kontextwechsel einen ausführbereiten Thread vorfindet, führt dieses Modell zu einer linear verlaufenden Effizienz:

$$\epsilon_{linear} = \frac{NR}{R+C+L}$$

Eine bessere Modellierung des realen Systemverhaltens ist möglich, wenn R als stochastische Größe mit geometrischer Verteilung angenommen wird. Dann kann die Wahrscheinlichkeit, als nächtes eine Anweisung auszuführen, die einen Kontextwechsel auslöst, zu $p = \frac{1}{R}$ gewählt werden. Dies führt zu einer Markov-Kette, die auch in einem Petrinetz (Abbildung 3.1) dargestellt werden kann. In dieser Darstellung ist E ein sofortiger Übergang, C und L sind deterministische Übergänge und R ist, wie schon erwähnt, ein stochastischer Übergang. Zustand A wurde nur eingefügt, um deutlich zu machen, daß immer nur *ein* Thread im Zustand *Running* oder *Leaving* sein kann. Die resultierende Markovkette enthält bereits bei kleinen Werten [6] von R,N, L und C mehrere Millionen Zustände, so daß die Effizienz daraus numerisch kaum zu bestimmen ist. Unter gewissen, unkritischen Einschränkungen und Vereinfachungen kann jedoch eine exakte Lösung ermittelt werden [32]. Gleichung 3.2 gibt für das stochastische Modell den Sättigungspunkt an, also die Anzahl von Threads, die nötig ist, um die Effizienz des Prozessors zu maximieren. Dabei gibt α den Grad des Vertrauens darauf an, daß der Prozessor nicht in die Situtation kommt, keinen ausführbereiten Thread zur Verfügung zu haben. Bei $\alpha = 0$ ergibt sich wieder Gleichung 3.1; $\alpha = 1$ führt zu einer Effizienz, die für bei $N = N_{Sättigung}$ Threads bei 95% des theoretischen Maximums liegt.

$$N_{Sättigung} = \left(\frac{\alpha R + \sqrt{(\alpha R)^2 + 4L(R+C)}}{2(R+C)}\right)^2 + 1 \tag{3.2}$$

Die Berücksichtigung der Verwendung von Caches nähert das Modell weiter an die realen Verhältnisse an. Das wesentliche Ergebnis einer derartigen Erweiterung des analytischen Modells ist, daß die Effizienz bei steigender Anzahl von Threads nicht mehr gegen das theoretische Maximum konvergiert, sondern durch die zunehmend schlechtere Nutzung des Caches jenseits des Sättigungspunktes wieder absinkt. Wenn schließlich jeder Zugriff in den Cache einen Cache Miss erzeugt, bleibt die Effizienz auf einem konstanten, niedrigen Niveau. Es ensteht also im realen Einsatz von Multithreading zum Zwecke der Überlappung von Kommunikation und Berechnung das Problem, einen optimalen Grad der Nebenläufigkeit, sprich Anzahl der Threads, zu verwenden.

[6] Für $L = 128$, $C = 2$ und $N = 4$ hat die Markov-Kette bereits mehr als 9 Millionen Zustände.

Abschließend sind in den beiden Abbildungen 3.2 und 3.3 der Verlauf der Effizienz und die Zahl der zur Sättigung notwendigen Threads nach diesem Modell aufgetragen. Dazu wurden Werte der Kontextwechselzeit C und Seitenfehlerlatenz L verwendet, die in Kapitel 6 für Intel Paragon ermittelt wurden.

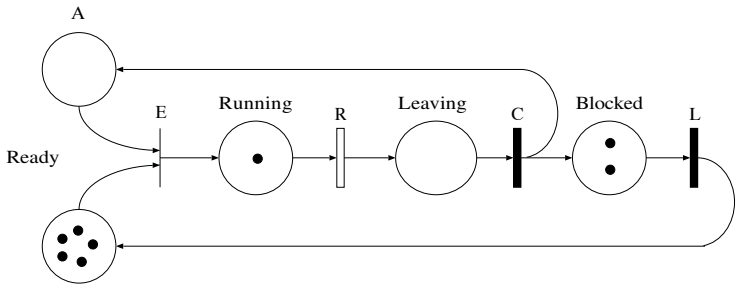


Abbildung 3.1: Multithreaded Prozessor als Petrinetz

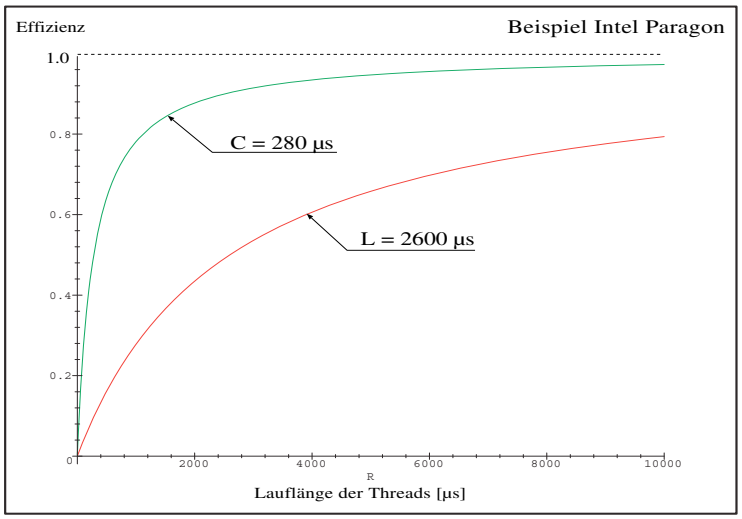


Abbildung 3.2: Effizienz als Funktion der Lauflänge R;
Latenz bei Speicherzugriff und Kontextwechsel als Parameter (Beispielwerte von Intel Paragon)

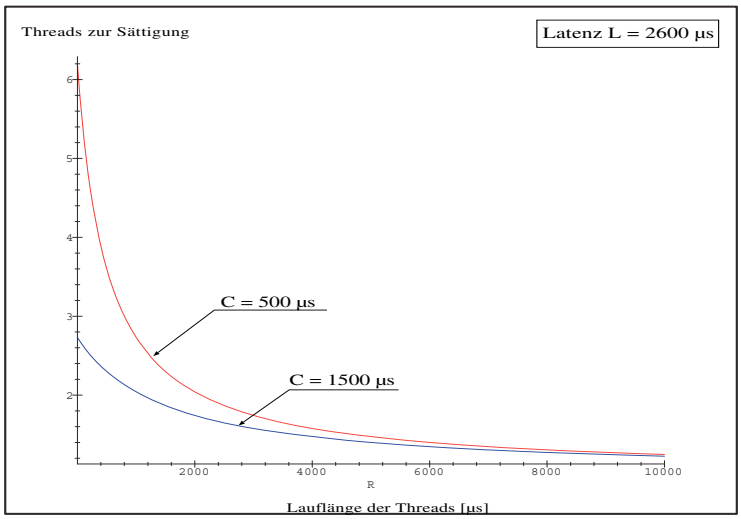


Abbildung 3.3: Zahl der zur Sättigung nötigen Threads als Funktion der Lauflänge R; Latenzen bei Kontextwechsel als Parameter (Beispielwerte von Intel Paragon)

3.5.2 Hardware-unterstütztes Multithreading

Wie im vorhergegangenen Abschnitt deutlich wurde, hängt die Effizienz des Multithreading wesentlich von dem Verhältnis der Lauflänge eines Threads zu der auftretenden Latenz beim Kontextwechsel ab. Eine wirkungsvolle Vergrößerung der Lauflänge ist auch mittels Hardwareunterstützung erreichbar. Dazu ist es wichtig, daß der Prozessor mehrere Kontexte zugleich speichern kann. Ansonsten tritt durch einen Kontextwechsel wiederum ein Zugriff auf den externen Speicher auf, um den nächsten Thread zu starten. In einem solchen Fall lohnt sich Multithreading nur, wenn es sich um eine NUMA-Architektur handelt und die Threadkontexte in einem schnelleren Speicher liegen als der Speicher, in den die Referenz erfolgt, die zum Kontextwechsel führte. Hier werden nun Strategien zum Hardware-unterstützten Kontextwechsel auf UMA-Systemen ohne und mit Caches vorgestellt.

Multikontext ohne Caches

Es gibt verschiedene Ansätze der Hardware-Unterstützung, um mittels Multithreading Latenzzeiten zu verstecken. Die einfachste Methode ist es, bei jeder Speicherreferenz den Kontext zu wechseln (*switch-on-load*). Hinsichtlich des kritischen Faktors bei der Effizienz des Multithreading, der Lauflänge der Threads, ist switch-on-load nicht optimal, da sofort nach jeder Referenz der Kontext gewechselt wird, auch wenn kurz darauf erneut eine Referenz erfolgt. *Switch-on-use* verbessert dieses Verhalten und erlaubt längere Laufzeiten der Threads, indem der Kontext erst dann gewechselt wird, wenn auf einen Wert zugegriffen wird, der zuvor angefordert wurde und noch nicht eingetroffen ist. Eine Variante von switch-on-use ist *explicit-switch*. Hierbei wird der Kontextwechsel explizit aufgerufen, im Gegensatz zum impliziten, aber Hardware-technisch aufwendigeren switch-on-use. Ein angepaßter Compiler kann damit Code erzeugen, in dem die Referenzen auf externe Speicherstellen (unter Beachtung der Datenabhängigkeiten) gruppiert werden, bevor der Kontext gewechselt wird. Ein Beispiel (ein Ausschnitt aus einem Löser für Laplace-Gleichungen) findet sich in Abbildung 3.4. Es ist deutlich, wie die Anzahl von Kontextwechseln durch den Einsatz der expliziten Switch-Anweisung gegenüber dem switch-on-load Verfahren reduziert werden kann. Detaillierte Untersuchungen finden sich in [31].

Multikontext mit kohärenten Caches

Der Einsatz von Caches bei Multithreading führt zu einem deutlichen Anstieg der Threadlauflängen, da ein großer Teil der externen Referenzen aus dem Cache bedient werden kann (*cache-hit*). Bei einem cache-hit ist ein Kontextwechsel nicht mehr effizient und unterbleibt. Diese erhöhte Lauflänge führt dazu, daß der Grad des Multithreading herabgesetzt werden kann und somit Hardware-Kontexte eingespart werden können. Dies führt zu einer Designentscheidung zwischen Größe und Art des Caches und Zahl der Hardware-Kontexte.

Die Kriterien zum Kontextwechsel sind zunächst ähnlich zu denen im vorherigen Abschnitt. Bei switch-on-miss wird zum nächsten Thread gewechselt, wenn eine Referenz

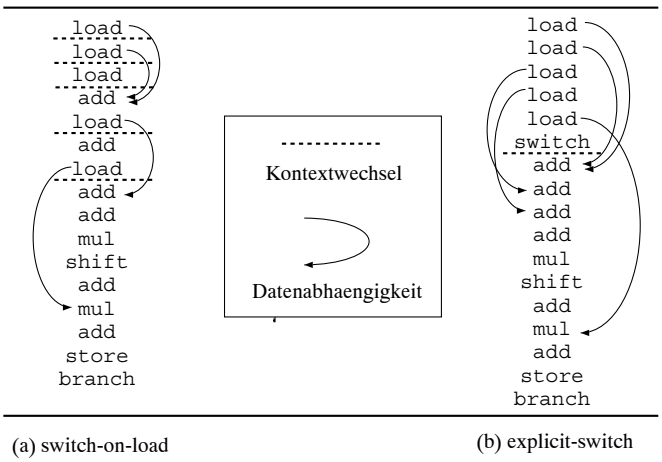


Abbildung 3.4: Reduzierung der Kontextwechsel durch explicit-switch

nicht aus dem Cache bedient werden konnte, bei switch-on-use-miss wird erst gewechselt, wenn der tatsächliche Zugriff auf diese Referenz erfolgt. Wenn man davon ausgeht, daß in jedem Fall genügend Hardware-Kontexte zur Verfügung stehen, um alle Threads im Prozessor zu halten, so liegt die Performance beim Einsatz von Caches im Bereich von wenigen Prozentpunkten unterhalb bis zu 15 Prozent oberhalb eines Systems ohne Cache [31]. Dies ist zurückzuführen auf die größere Lauflänge der Threads.

Ähnlich wie durch die Gruppierung von Referenzen auf externen Speicher und den anschließenden expliziten Kontextwechsel eine Leistungssteigerung bei fehlendem Cache erzielt werden konnte, kann dies bei vorhandenem Cache durch einen bedingten Kontextwechsel (*conditional-switch*) erfolgen. Hierbei wird nur dann ein Kontextwechsel durchgeführt, wenn eine der Referenzen nicht aus dem Cache bedient werden konnte. Die Leistungssteigerung durch conditional-switch ist jedoch weit weniger hoch wie bei explicit-switch, da nach den Ergebnissen von Boothe mitunter [31] 96% bis 99% der Referenzen aus dem Cache bedient werden[7], wobei jedesmal die conditional-switch Anweisung Taktzyklen verbraucht.

Aufgrund der längeren Laufzeiten der Threads beim Einsatz von Caches wird ein Modifikation des Scheduling wichtig, so daß nicht nur bei einem cache-miss ein Kontextwechsel durchgeführt wird. Sonst kann es vorkommen, daß ein Thread mit sehr langer Laufzeit andere Threads blockiert, die nur kurz laufen, da sie z.B. einen Lock belegen wollen.

Beispiele für Multikontext-Systeme

Tera MTA Ein seit geraumer Zeit in Entwicklung befindliches System, das das Multikontext-Prinzip in einer sehr ausgeprägten Weise zu verwirklichen verspricht, wurde

[7] Cachegröße 64 Kbyte, Programme `sieve` (Primzahlenberechnung), `sor` (Löser für Laplace-Gleichung), `Ugray` (Ray-Tracing), `water` (N-Körper Simulator mit limitiertem Interaktions-Radius), `Locus` (Routing für VLSI- Schaltungen) und `Barnes` (N-Körper Gravitations-Simulator). Andere Programme wie z.B. Matrixmultiplikation ergaben 84% Trefferquote.

von der Tera Computer Company [1] unter dem Namen *Tera MTA* vorgestellt. Es ist ein massiv-paralleles MIMD-System mit der Topologie eines dreidimensionalen Torus und verteiltem Speicher. MTA steht für Multithreaded Architecture, denn die Prozessoren des Tera-Systems führen bis zu 128 Instruktionsströme zur gleichen Zeit aus. Zwischen diesen Strömen, die hier gleichbedeutend mit hardware-verwalteten Threads sind, kann mit jedem Uhrentakt umgeschaltet werden. Wenn ein Thread eine Instruktion starten konnte, gelangt diese in die Prozessor-Pipeline. Anschließend wird der nächste ausführbereite Thread ausgewählt, d.h. ein Thread, der momentan keine Instruktion in der Pipeline hat, um seine nächste Instruktion auszuführen. Bei einer Pipeline-Latenz von etwa 70 Takten und verzögerungsfreien Wechseln zwischen den Threads werden somit rund 70 Threads benötigt, um den Prozessor voll auszulasten.

Die Wechsel zwischen den Threads erfolgen verzögerungsfrei, da der gesamte Hardwarekontext des Prozessors in 128facher Ausführung existiert, und somit jeder Thread seinen eigenen Kontext besitzt. Damit erfordert ein Kontextwechsel zwischen den Threads keine Aus- und Einlagerung von Registerinhalten, sondern nur ein Wechsel des aktiven Kontextes, der bereits im Prozessor vorliegt. Jeder dieser Kontexte besteht aus

- **1 64-Bit Statuswort**
  In der unteren Hälfte enthält es den 32-Bit-breiten Programmzähler und in den oberen 32 Bit verschiedene Zustandsindikatoren.
- **32 64-Bit Universalregister**
- **8 64-Bit Zielregister**
  Mit Hilfe der Zielregister kann ein Verzweigungsziel von der zugehörigen Instruktion getrennt werden. Dadurch können Instruktionen an der Zieladresse vorab geladen werden (*Prefetching*), und die eigentlichen Verzweigungsinstruktionen werden kleiner, was kleinere Schleifen bewirkt.

Insgesamt führt dies zu 5248 64-Bit Registern im Prozessor, die in Funktion und Menge mit Vektorregistern oder Cache-Worten verglichen werden können.

Weitere Eigenschaften des Tera-Systems, die das Multikontext-Prinzip unterstützen, sind

- *Explicit-Dependence Lookahead*
  Um eine gute Auslastung des Prozessors auch mit weniger Threads zu gewährleisten, müssen mehrere Instruktionen eines Threads gleichzeitig in der Prozessor-Pipeline verarbeitet werden. Dies erfordert die Beachtung von Abhängigkeiten zwischen diesen Instruktionen. Der herkömmliche Ansatz des *Scoreboarding* [12] kann hier nicht verwendet werden, da er bei der Zahl der Kontexte zu viel Bandbreite an Registern benötigen würde. Daher wurde die *ausdrückliche Abhängigkeit* eingeführt. Hierbei hat jede Instruktion ein 3 Bit breites Vorschau-Feld, in dem angegeben ist, wieviele Instruktionen in diesem Strom folgen, bevor eine Instruktion von der aktuellen Instruktion abhängig ist. Dieser Eintrag muß zur Compilezeit ermittelt werden. Da der maximal darstellbare Wert in diesem Feld 7 ist, können sich maximal 8 Instruktionen eines Stroms zur gleichen Zeit in der Pipeline befinden. In diesem Fall reichen bereits 9 Threads aus, um den Prozessor voll auszulasten.

- *Protection Domains*
  Auf jedem Prozessor können bis zu 16 Schutzbereiche eingerichtet werden, die den Programm- und Datenspeicher sowie die Zahl der Threads definieren. Dies stellt eine Art von Prozeß oder *virtuellem Prozessor* dar, der für die Anwendung transparent ist. Tatsächlich kann dieser virtuelle Prozessor von einem physikalischem Prozessor auf einen anderen transferiert werden. Innerhalb dieses Schutzbereichs können mit unpriviligierten Befehlen neue Threads erzeugt werden, wozu nur wenige Register initialisiert bzw. vom aktuellen Thread kopiert werden müssen.

- *Tagged Memory*
  Jedes 64-Bit Wort im Speicher besitzt zusätzlich 4 Statusbits, mit denen die Semantik des Speicherzugriffs erweitert wird. Zur sehr leichtgewichtigen Synchronisation der Threads dient ein *full/empty-Bit*, das angibt, ob das Datum gültig (full) oder ungültig (empty) ist. Mit Lade- und Speicherinstruktionen, die den Zustand dieses Bits berücksichtigen (z.B. das Datum nicht lesen, solange es als empty deklariert ist) und im Anschluß ggf. ändern, ist bereits eine Synchronisation erfolgt. Dabei kann zunächst eine einstellbare Zahl von Versuchen erfolgen, etwa eine Speicherstelle zu lesen, wenn sie als *empty* markiert ist. Dazwischen erfolgen weiterhin Kontextwechsel. Führen diese Versuche nicht zum Erfolg, wird eine schwergewichtigere Synchronisationsmethode verwendet.

SB-PRAM Das SB-PRAM-System [16] verfolgt eien ähnlichen Ansatz. Es handelt sich um ein skalierbares paralleles MIMD-System mit physikalisch verteiltem Speicher, der jedoch als gemeinsamer Speicher dargestellt wird. Bislang erfolgte der Ausbau auf 4 Prozessoren, weshalb nicht von massiver Parallelität gesprochen werden kann.

Der PRAM-Prozessor unterstützt 32 sogenannte *virtuelle Prozessoren (vP)*, die den Instruktionsströmen der Tera MTA entsprechen. Auch hier wird nach Abgabe einer Instruktion eines vPs an die Pipeline des physikalischen Prozessors ein Kontextwechsel zum nächsten vP durchgeführt. Die Besonderheiten des SB-PRAM sind einige Aspekte der Architektur (Speicherzugriff über Hash-Funktionen zur Vermeidung von Flaschenhälsen) und die vergleichsweise einfache Hardware-Technische Implementierung (off-chip Cache, echtes RISC mit gleicher Länge und Ausführungszeit aller Instrukionen), was ein kostengünstiges System ermöglicht.

3.5.3 Software-unterstüztes Multithreading

Auf vielen Systemen ist keine besondere Hardware-Unterstützung für Multithreading implementiert (Beispiel Intel Paragon), obwohl hier lange Latenzen beim Zugriff auf nichtlokalen Speicher auftreten[8] Hier kann nur Software-Mäßiges Multithreading eingesetzt werden. Natürlich ist hierbei die Latenz beim Kontextwechsel deutlich höher; wenn jedoch die zu versteckende Latenz beim Speicherzugriff hoch genug ist, kann man die

[8] Die Intel Paragon wurde vom Hersteller auch als NORMA-System entwickelt; der virtuell gemeinsame Speicher wurde nachträglich von Dritten [20] entwickelt.

Prozessorausnutzung und damit die Leistung denoch steigern. Die maximal erreichbare Leistung bei langen Speicherlatenzen (500 bis 1000 Taktzyklen) liegt dabei nach Untersuchungen in [31] allerdings selbst bei gleicher Kontextwechsellatenz unterhalb der Leistung bei kurzen (200 Taktzyklen) Speicherlatenzen. Dafür gibt es mehrere Gründe, die bei langen Speicherlatenzen besonders ins Gewicht fallen:

- **Kurze Laufzeiten:** wenn mehrere nacheinander ausgeführte Threads kurze Laufzeiten haben, kann die entstandene Speicherlatenz einfach nicht mehr überdeckt werden.
- **Unbalancierte Last:** bei unausgegelichener Lastverteilung auf die Threads erfährt derjenige Thread, der aufgrund der höheren Last als letzter läuft, die volle Speicherlatenz. In dieser Zeit müssen die anderen Threads zur Synchronisation auf ihn warten (siehe Abbildung 3.5).
- **Blockierung durch Lock:** wenn ein Speicherzugriff innerhalb eines durch einen Lock abgesicherten exklusiven Bereichs erfolgt, können die nachgeschalteten Threads nicht in diesen Bereich eintreten, sondern müssen die volle Speicherlatenz abwarten.

Trotz dieser Einschränkungen ist der Einsatz von Multithreading sinnvoll, da die Leistung gegenüber der Ausführung in nur einem Thread stark gesteigert werden kann (relativ gesehen sogar stärker als bei kürzeren Speicherlatenzen).

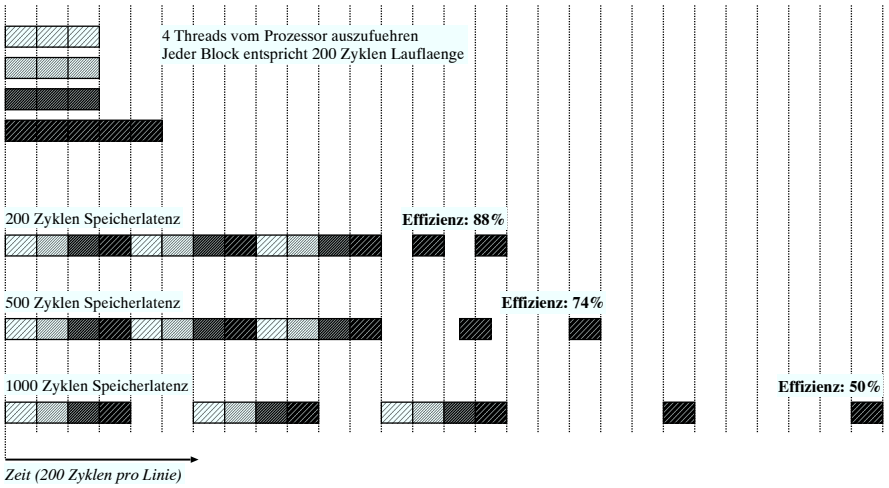


Abbildung 3.5: Verringerte Leistungszunahme durch Multithreading bei steigender Speicherlatenz und unausgeglichener Lastverteilung

3.5.4 Andere untersuchte Verfahren

Neben der Verwendung von Multithreading zur Verminderung der effektiven Speicherlatenz gibt es noch andere Verfahren, die dieses Ziel verfolgen . In den folgenden Abschnitten findet sich dazu ein vergleichender Überblick. Zwar basieren diese Verfahren

im wesentlichen auf Hardware und lassen sich daher in einem vorhandenen System nicht nachrüsten, jedoch ist es vorteilhaft, die in einem System bereits verwendeten Verfahren zur Reduzierung von Speicherlatenz zu kennen, um diese durch Software weiter zu reduzieren.

Kohärente Caches

In Systemen mit nur einem oder wenigen Prozessoren werden Hardware-mäßig kohärentgehaltene Caches inzwischen standardmäßig eingesetzt. Bei bus-basierten Multiprozessoren ist das Kohärenzproblem durch Verwendung von bus-snooping noch relativ leicht zu lösen. In massiv-parallelen Systemen mit einem Verbindungsnetzwerk ist das Kohärenzprotokoll jedoch weitaus aufwendiger, wodurch in vielen Systemen nur Software-mäßig kohärent gehaltene oder gar keine Caches zum Einsatz kommen.

Schon im Kapitel 3.5.2 wurde anhand der bei vielen Anwendungen zu erwartenden hohen Quote der Cachetreffer von bis zu 96% dargestellt, welch starken Einfluß ein Hardwaremäßig kohärent gehaltener Cache auf die effektive Speicherlatenz hat. Jedoch wird in den Untersuchungen in [10] selbst bei Trefferquoten von weniger als 70% eine Leistungssteigerung um über 100% erreicht. Dabei bleibt aber die effektive Prozessorauslastung immer noch unterhalb von 30%, was den Einsatz weiterer Techniken notwendig macht.

Schwache Speicherkonsistenz

Modelle zur Speicherkonsistenz in Multiprozessorsystemen sind, ebenso wie die kohärenten Caches, eine Designfrage der Hardware [14]. Die Frage ist dabei immer, wie sich die Reihenfolge der Befehlsausführung zu der Reihenfolge der enthaltenen Speicherzugriffe auf gemeinsamen Speicher verhält.

Das konservativste Speichermodell ist das der *sequentiellen* oder *strengen Konsistenz*. Hierbei werden alle Speicherzugriffe genau in der Reihenfolge des Auftretens in der Instruktionsfolge ausgeführt. D.h., ein Prozessor kann keinen Speicherzugriff ausführen, bevor nicht der letzte Schreibvorgang eines Prozessors im System abgeschlossen ist. Es kann immer nur ein einziger Prozessor auf den Speicher zugreifen, und dieser Zugriff erfolgt atomar. Obwohl dadurch große Speicherlatenzen auftreten, ist dieses Modell sehr verbreitet, da es vergleichsweise einfach zu implementieren ist.

Ein anderes Modell, das kürzere Speicherlatenz bietet, ist die *schwache Konsistenz* (Weak Consistency). Es gibt eine Anzahl von leicht unterschiedlichen Definitionen einer schwachen Konsistenz, die u.a. verschiedene Grade dieser nicht-sequentiellen Konsistenz definieren. Allgemein gilt aber, daß durch ein solches Modell ein Puffern oder Pipelining aufeinanderfolgender Zugriffe oder eine Ausführung in anderer Reihenfolge als vom Programmcode vorgesehen möglich ist. Meistens wird dabei zwischen Lese- und Schreibzugriffen unterschieden: beim TSO Modell [36], das in einigen Varianten der Sun SPARC Architektur verwirklicht wurde, werden Schreibzugriffe eines Prozessors in einem privaten FIFO-Puffer abgelegt, Lesezugriffe sind jedoch ungepuffert.

In [10] wurde eine andere Variante von schwacher Konsistenz , das *Release Consistency Modell*, quantitativ untersucht. In diesem Modell müssen Speicherzugriffe, die der Synchronisation dienen, als solche gekennzeichnet als *lock* (Lesezugriff, auch als Teil eines Lese-Ändere-Schreibe Zyklus) oder *unlock* (Schreibzugriff) klassifiziert werden. Zwischen diesen beiden Punkten können Speicherzugriffe sodann flexibel durch Pufferung und Pipelining beschleunigt werden. Dieses Modell beseitigt nach den Untersuchungen in [10] den Großteil der Schreiblatenz, was zu Leistungssteigerungen zwischen 9% und gut 50% führt. Es erfordert jedoch eine höhere Hardware-Komplexität und ein komlexeres Programmiermodell.

Prefetching

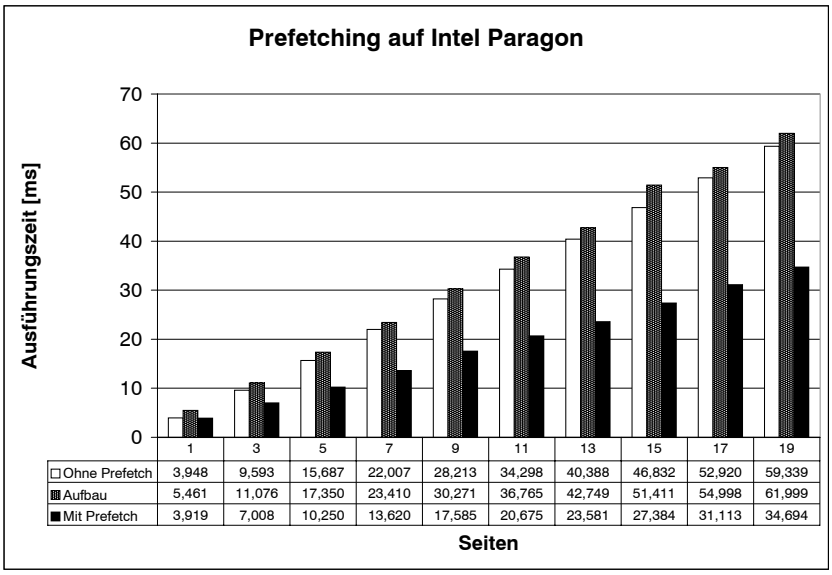
Beim Verfahren des *Prefetching*, also dem vorweggenommenen Laden von Daten und Befehlen in einen Speicherbereich muß unterschieden werden zwischen Software- und Hardware-Prefetching, wobei hier aber nur die Software-basierte Variante betrachtet wird.

Beim Software-Prefetch wird das Vorabladen durch eine explizite Anweisung ausgelöst, die entweder manuell oder durch einen Compiler eingesetzt werden kann. Die Untersuchungen von Gupta et al.[10] zeigen jedoch, daß bereits eine deutliche Leistungssteigerung erreicht werden kann, wenn an den entscheidenden Stellen, an denen vorhersagbare und eindeutig strukturierte Zugriffsmuster auftreten, einige Prefetchanweisungen plaziert werden. Diese Stellen können bereits gut vom Programmierer ermittelt werden, so daß ein Einsatz von Prefetching auch ohne angepaßten Compiler erfolgen kann. Bei Gupta wurden so Leistungssteigerungen von 14% bis 60% erreicht; die Kombination von Prefetching mit dem Release Consistency Modell erbrachte dabei durchweg die besten Ergebnisse.

Für den am hiesigen Institut entwickelten SVM-Fortran Compiler wurde Prefetching eingeführt. Ein synthetischer Benchmark (siehe Programm 3.2) zeigte dabei gute Ergebnisse, die aus Abbildung 3.6 ersichtlich sind. Dort sind die Ausführungszeiten eines Schleifendurchlaufs des angegebenen Benchmarks ohne und mit Prefetching dargestellt sowie unter *Aufbau* die für das Prefetching einmalig benötigte Zeit für den Aufbau der Prefetching-Liste inklusive des ersten Schleifendurchlaufs. Deutlich wird bei diesen Messungen aber auch, wie groß der Einfluß von Seitenfehlern auf die Systemleistung bei einem NORMA-System wie der Intel Paragon ist. Wenn es gelingt, diese Seitenfehler zu vermeiden oder verstecken, ist die Systemleistung für diesen Benchmark bereits doppelt so hoch. Prefetching rentiert sich allerdings erst bei mehrfachen Durchlaufen der zugehörigen Region sowie einer möglichst großen Anzahl von Seiten, wohingegen der Ansatz des Multithreadings bereits beim erstmaligen und möglicherweise einzigen Seitenfehler effektiv sein kann.

Algorithmus 3.2 Benchmark zur Ermittlung der Leistungssteigerung durch Prefetching in SVM-Fortran auf Intel Paragon

```
ohne Prefetching:
    do i=1, pages * 1024
       a(i)=
    enddo
    barrier

mit Prefetching:
    call prefetch_pfl(1)
    call create_pfl_start(1,a,pages * 1024 * 8)
    do i=1, pages * 1024
       a(i)=
    enddo
    call prefetch_pfl_stop(1)
    barrier
```



	1	3	5	7	9	11	13	15	17	19
Ohne Prefetch	3,948	9,593	15,687	22,007	28,213	34,298	40,388	46,832	52,920	59,339
Aufbau	5,461	11,076	17,350	23,410	30,271	36,765	42,749	51,411	54,998	61,999
Mit Prefetch	3,919	7,008	10,250	13,620	17,585	20,675	23,581	27,384	31,113	34,694

Abbildung 3.6: Leistungssteigerung durch Prefetching bei SVM-Fortran auf Intel Paragon

Kapitel 4

Synchronisation

Beim Betrieb von parallelen Systemen spielt die Synchronisation von Daten eine zentrale Rolle und ist daher ein sehr gründlich untersuchtes Feld. Unter einem parallelen System verstehen wir in diesem Zusammenhang einen Rechner mit mehreren Prozessoren (*Parallelrechner*), die über gemeinsamen Speicher oder Austausch von Nachrichten miteinander kommunizieren können (zur verwendeten Klassifizierung siehe Kapitel 2.3). Kommunikation zwischen den Prozessoren eines Parallelrechners tritt aus vielfältigen Gründen auf, sei es zur Signalisierung oder zum Zugriff auf gemeinsamen oder entfernten Speicher. Der im folgenden verwendete Begriff *Netzwerkoperationen* soll derartige Vorgänge zur Vereinfachung auf allen Architekturen bezeichnen, unabhängig davon, ob diese tatsächlich auf einem Netzwerk zur Kommunikation basieren oder nicht.

Synchronisation wird immer dann erforderlich, wenn eine Datenstruktur von mehreren Prozessoren oder auch von mehreren Prozessen auf einem Prozessoren bearbeitet wird. Dann muß gewährleistet werden, daß diese Datenstruktur zu jeder Zeit allen Prozessen identisch vorliegt. Dies ist dann der Fall, wenn immer nur ein Prozessor zur selben Zeit tatsächlich Daten in dieser Datenstruktur verändern kann. Es ist also notwendig, daß sich jeder Prozessor vor einem solchen Zugriff darüber versichert, sich also quasi für den Zugriff auf die Datenstruktur anmeldet. Diese Anmeldung erfolgt über *Locks*, die in Kapitel 4.2 vorgestellt werden.

Ein Grundprinzip beim parallelen Rechnen ist die Aufteilung eines Problems in Teilprobleme, die auf die Prozessoren verteilt werden. Leider sind diese Teilprobleme meistens voneinander abhängig, so daß im Laufe der Berechnungen ein Prozessor auch immer wieder Ergebnisse von anderen Prozessoren benötigt, die dem gleichen Stand der Berechnungen entsprechen. Diese Art von Synchronisation erfolgt über *Barrieren*. Ein Prozess kann erst dann mit Programmschritten hinter der Barriere weitermachen, wenn alle anderen beteiligten Prozesse ebenfalls an dieser Barriere angekommen sind. Diese impliziert natürlich auch Wartezeiten für die Prozesse, die früher an der Barriere angekommen sind. Kapitel 4.3 erläutert verschiedene Typen von Barrieren.

Grundlage sowohl von Locks als auch Barrieren sind *atomare Operationen*, während deren Ausführung ein Prozessor unter keinen Umständen unterbrochen werden kann[1].

[1] Hardware-technisch bedeutet das meistens, daß sie innerhalb eines Buszyklus abgearbeitet werden oder der Bus für diesen Zeitraum für andere Operationen blockiert wird.

4.1 Grundprimitive zur Synchronisation

Die Basis für alle Algorithmen zur Synchronisation sind *atomare Operationen*. Dies sind Operationen, die zwar tatsächlich aus mehreren Einzelanweisungen bestehen, wie sie üblicherweise im Befehlssatz von Prozessoren vorhanden sind, die jedoch garantiert ohne Unterbrechung hintereinander ausgeführt werden. Dies ist nötig, damit ein Prozeß in der Lage ist, einen Wert aus dem Speicher zu lesen, zu überprüfen und anschließend ggf. modifiziert zurückschreiben zu können, ohne daß innerhalb dieses Vorgangs ein anderer Prozeß (auf demselben oder einem anderen Prozessor) diesen Wert lesen kann. Damit wird Datenkohärenz gewährleistet.

Die Unterstützung für solche Operationen seitens der Hardware fällt unterschiedlich aus. Teilweise bieten die Prozessoren die atomaren Operationen auch auf Befehlssatzebene an (Sparc, Intel x86), andernfalls muß der Prozessor vorher in einen exklusiven Modus geschaltet werden (Intel i860), der den Prozessor oder bei Multiprozessoren den gesamten Bus sperrt. Solche Sperren müssen immer so kurz wie möglich gehalten werden, um eine übermäßige Blockade des Systems zu vermeiden.

Test_and_set

Dies ist die einfachste atomare Operation, die in vielen Prozessoren z.B. Motorola) als ein einzelner Befehl verfügbar ist. Auf moderneren Prozessoren (z.B. MIPS) kann diese Operation auch auf 2 Befehle, einer zum Schreiben und ein zweiter zum Abfragen des Ergebnisses, verteilt werden, was eine Nutzung der auftretenden Speicherlatenz durch eingeschobene Befehle ermöglicht. Test_and_set reicht prinzipiell bereits aus, um alle Aufgaben der Synchronisation zu ermöglichen, da mit ihr Locks (siehe Kapitel 4.2) und damit kritische Bereiche realisiert werden können. Bei `test_and_set` wird der Inhalt einer Speicherstelle mit einem Wert A verglichen, wobei es zwei Ergebnisse geben kann, gleich oder ungleich. Im gleichen Buszyklus wird der Wert A, unabhängig vom Ergebnis der Vergleichsoperation,in die Speicherstelle geschrieben. Das gewonnene Ergebnis kann direkt für einen Lock verwendet werden (siehe Kapitel 4.2.1).

Test_and_test_and_set

Bei Verwendung von `test_and_set` zur Realisierung eines Locks wird immer dann, wenn der gewünschte Lock nicht frei, der Wert in der Speicherzelle mit demselben Wert überschrieben. Durch diese Operation ändert sich im Speicher nichts, jedoch führt dieser Schreibvorgang dazu, daß in allen Caches, die diese Speicherstelle enthalten, die zugehörige Cachezeile als ungültig markiert wird. Dies führt zum einen zu Verwaltungsoverhead durch das Invalidieren der Cachezeilen, zum anderen zu einer hohen Belastung des Speichersystems, die eigentlich durch die Caches verringert werden soll. In vielen Fällen ist daher die Verwendung von `test_and_test_and_set` vorteilhaft. Diese Operation führt vor dem normalen `test_and_set` einen reinen Lesezugriff auf die fragliche Speicherstelle aus. Wenn dabei der Inhalt anzeigt, daß der Lock verfügbar ist, wird die eigentliche `test_and_set` Operation durchgeführt. Da `test_and_test_and_set`

nicht atomar ist, ist zwar nicht garantiert, daß man in jedem Fall nach einem erfolgreichen Test auch das folgende `test_and_set` erfolgreich ist, aber es wird auf jeden Fall die Zahl unnötiger Schreibvorgänge verringert.

Fetch_and_Φ

`Fetch_and_`Φ ist die allgemeine Bezeichnung von atomaren Operationen der Art, wie sie in den folgenden Abschnitten vorgestellt werden.

Compare_and_swap

Auch die `Compare_and_swap` Operation ist in einigen Prozessoren als atomarer Befehl vorhanden. Dabei liegen zwei Werte (Vergleichswert V und Aktualisierungswert A) in Prozessorregistern vor, ein dritter Wert S liegt im Speicher. Nun wird V mit S verglichen. Bei Gleichheit wird S durch Überschreiben mit A aktualisiert, bei Ungleichheit wird V mit dem Wert von A überschrieben. All dies läuft, wie für einen atomaren Befehl notwendig, innerhalb eines Buszyklus ab. Das Ergebnis des Vergleich kann auch für weitere Entscheidungen herangezogen werden. Eine Anwendung dieses Befehls wird in Kapitel 4.2.2 vorgestellt.

Fetch_and_add

Um einen zentralen Zähler zu verändern, also zu inkrementieren oder dekrementieren, muß der Zähler in ein Register des Prozessors geladen, dort bearbeitet und anschließend wieder zurückgeschrieben werden. Dies kann nicht, wie `test_and_set`, innerhalb eines Buszyklus geschehen und kann daher nicht ein atomarer Befehl sein. Wenn man jedoch diesen Zähler mit einem Lock sichert (der mittels `test_and_set` implementiert werden kann), erhält man eine neue atomare Operation, etwa im Falle des Inkrementierens `Fetch_and_add`.

Fetch_and_increment

Nachdem der Prozessor den Inhalt einer Speicherzelle gelesen hat, schreibt er den um Eins inkrementierten Wert zurück. Diese Operation stellt einen Spezialfall von `fetch_and_add` dar und ist für die atomare Behandlung von Zählern ideal.

Fetch_and_store

Diese Operation könnte auch `swap` heißen, aber der verwendete Name paßt besser in das bisher verwendete Schema zur Namensgebung. Seine Funktion ist offensichtlich: zwei Werte, ein Register des Prozessors und eine Speicherstelle, werden atomar gegeneinander ausgetauscht.

4.2 Locks

Ein zwangsläufiges Problem auf parallelen Systemen ist die Aufrechterhaltung der logischen Konsistenz von gemeinsamen Datenstrukturen. Eine Standardtechnik zur Gewährleistung der Konsistenz ist die Verwendung von *Locks*. Ein Prozeß, der auf die Datenstruktur schreibend zugreifen will, muß zuvor einen dazugehörigen Lock anfordern. Sobald er den Lock erhalten hat, wird ihm exklusiver Zugriff auf die Datenstruktur gewähleistet, bis er den Lock wieder abgibt. Dann kann ein anderer Prozeß, der möglicherweise bereits auf den Lock warten mußte, den Lock erhalten, und so auf die Datenstruktur zugreifen.

Ein Lock kann auch zur Errichtung von *kritischen Bereichen* genutzt werden. Dies sind Programmabschnitte, die nur von einem Prozeß zur Zeit durchlaufen werden dürfen. Dies liegt meistens auch darin begründet, daß auf gemeinsame Datenstrukturen exklusiv, weil schreibend, zugegriffen werden muß. Jedoch könnnen kritische Bereiche darüberhinaus beliebige Anweisungen enthalten. Sie sollten jedoch so kurz wie möglich gehalten werden, da sie ja der Parallelität in der Ausführung entgegenstehen.

Insgesamt kann man zur Bewertung eines Lock-Algorithmus folgende Kriterien heranziehen:

- Skalierbarkeit und damit verbundene Netzwerkbelastung
- Latenz für die konkurrenzfreie Belegung des Locks durch einen einzelnen Prozessor
- Speicherbedarf der verwendeten Datenstrukturen
- Fairness im Wettbewerb (FIFO) und bei Kontextwechseln durch den Scheduler (kann ein suspendierter Prozess alle anderen konkurrierenden Prozesse blockieren ?)
- Implementierbarkeit mit verfügbaren atomaren Operationen

Die Implementierung von Locks für Systeme mit gemeinsamen und verteilten Speicher unterscheidet sich grundlegend, weil die Kosten für Speicherzugriffe verschieden sind. Auf cc-NUMA- und vor allem NORMA-Systemen (verteilter Speicher) ist der Zugriff auf ein nicht-lokales Datum teuer. Auch bei UMA- Systemen ist der Zugriff auf ein Datum im gemeinsamen Speicher oft teurer als vermutet, wenn viele Prozessoren zur gleichen Zeit einen solchen Zugriff durchführen und es dadurch zu Konflikten kommt. Hier werden oft Caches eingesetzt, um bei einem Lesezugriff den Bus weniger zu belasten. Es ist also entscheidend, wie oft ein Prozessor auf ein Datum zugreifen muß, um einen Lock zu erhalten, wo dieses Datum abgelegt ist und welche Prozessoren insgesamt schreibend oder lesend darauf zugreifen.

Das Verhalten eines Prozesses beim Warten auf die Verfügbarkeit eines Locks kann in *Spinning* oder *Blocking* unterschieden werden. Beim Spinning wird der Zustand des Locks wiederholt abgefragt; beim Blocking gibt der Prozess nach einem erfolglosen Versuch, einen Lock zu erwerben, die Kontrolle für einen Zeitraum ab. Die Bestimmung

dieses Zeitraums stellt dabei das Problem dar: zu kurzes Warten verursacht zu häufiges Abfragen des Zustandes, zu langes Warten verschwendet Zeit. Bei der Entscheidung für ein Verfahren muß auch noch berücksichtigt werden, ob auf einem Prozessor nur ein Prozess läuft oder mehrere Threads ausgeführt werden (siehe Kapitel 4.2.2).

4.2.1 Locks für UMA-Systeme

Bei UMA Systemen, wo also der Zugriffspfad jedes Prozessors auf jede Speicherstelle gleich strukturiert ist, ist auch jeder Speicherzugriff mit der gleichen Latenz verbunden, solange keine Konflikte auftreten. In der Regel wird jedoch jedem Prozessor ein Cachespeicher zugeordnet (sei er exklusiv oder mit anderen Prozessoren geteilt), der den Speicherzugriff bei einem Cachetreffer stark beschleunigt und darüberhinaus bei einem Lesevorgang den Datenbus zum Speicher nicht belastet. Die Effizienz der Caches hängt jedoch stark von dem Verhalten der Prozessoren ab, da eine durch einen Schreibzugiff veränderte Speicherstelle in allen Caches als ungültig gekennzeichnet und bei erneutem Zugriff neu eingelesen werden muß, was eine starke Busbelastung bedeutet. Diese Busbelastung wiederum bedingt größere Latenzen für weitere parallel gestartete Speicherzugriffe. Bei der Suche nach dem optimalen Verfahren für einen Lock auf UMA Systemen muß also die Auslegung des Caches (Größe, Strukur und Strategie) Berücksichtigung finden, ebenso wie das verwendete Protokoll, das die Daten in den Caches kohärent hält. Andererseits gibt es UMA-Systeme ohne gemeinsamen Speicherbus und zugeordnete Caches, bei denen die Prozessoren über ein Schaltnetzwerk mit der gewünschten Speicherbank verbunden werden (CRAY, siehe Kapitel 2.3).

test_and_set Lock

Das einfachste Verfahren zum gegenseitigen Ausschluß (*Mutual Exclusion*, auch als *Mutex* abgekürzt) ist das wiederholte Abfragen (*spinning* oder gleichbedeutend *pollen*) einer zentralen Boolean-Variable in einer atomaren Operation. Dazu kann die `test_and_set` Operation verwendet werden. Der Lock, also eine allen Prozessoren zugängliche Speicherstelle, wird mit dem Wert für " Lock frei "initialisiert, etwa mit der Wert 1. Um den Lock zu erhalten, führt ein Prozessor die `test_and_set` Operation auf diese Speicherstelle mit dem Wert für " Lock belegt "(z.B. der Wert 0) aus. Liefert die Operation das Ergebnis Gleichheit, bedeutet dies, daß der Lock bereits durch einen anderen Prozessor belegt ist. Der Prozessor muß erneut versuchen, den Lock zu erhalten. Er erhält den Lock, wenn der Vergleich Ungleichheit liefert. Das ist dann der Fall, wenn zuvor ein Prozessor den erhaltenen Lock abgegeben hat, indem er den Wert 1 in die Speicherstelle schreibt. Anhand des Codebeispiels in Algorithmus 4.1 wird das Verfahren klar. Es wird auch deutlich, daß diese Art von Lock das Speichersystem stark belastet. Selbst wenn die Lesevorgänge durch die Verwendung von Caches wenig kosten, führen doch die in den meisten Fällen überflüssigen Schreibvorgänge zu teuren Cache-Invalidierungen.

Eine Verminderung dieser Belastung kann durch `test_and_test_and_set` Operationen erreicht werden, d.h. der Lock wird zuerst einmal nur gelesen. Nur wenn dabei ein Prozessor erkennt, daß der Lock frei ist, versucht er ihn durch eine `test_and_set`

Operation zu belegen. Dies führt zu einer starken Verminderung der Zahl der Schreibvorgänge. Um auch eine Verminderung der Zahl der Lesevorgänge zu erreichen, kann eine Verzögerung zwischen den `test_and_set` Operationen eingeführt werden. Die einfachste Lösung ist eine konstante Dauer dieser Verzögerung, bessere Ergebnisse werden aber durch eine exponentielle Verzögerung erzielt, wie sie in dem zweiten Beispiel in Algorithmus 4.1 verwendet wird.

Algorithmus 4.1 `test_and_set` Lock

```
#DEFINE LOCKED 1
#DEFINE FREE   0

shared char *Lock; /* zu FREE initialisiert */
private int delay;

basic_lock() {
   while (test_and_set (Lock, LOCKED)) {}

   /* jetzt haben wir den Lock */
   do_something();

   /* am Schluss den Lock wieder abgeben */
   *Lock = FREE;
}

exponential_lock() {
   delay = 1;
   while (test_and_set (Lock, LOCKED)) {
      pause (delay);
      delay *= 2;
   }

   /* jetzt haben wir den Lock */
   do_something();

   /* am Schluss reinitialisieren den Lock wieder abgeben */
   delay = 1;
   *Lock = FREE;

}
```

Ticket Lock

Der *Ticket Lock* [21] ähnelt dem Verfahren, das auch an Schaltern aller Art zur Einhaltung der Bedienreihenfolge verwendet wird: jeder neu eintreffende Kunde nimmt sich eine fortlaufende Nummer; auf einer Anzeige steht die Nummer des Kunden, der gerade

bedient wird. Dadurch kann der Kunde ungefähr abschätzen, wann er an der Reihe sein wird. Auf unser Problem übertragen sieht es so aus, das zwei Zähler existieren, die die Zahl der Anforderungen des Locks und der Zahl der Abgaben des Locks anzeigen. Ein Prozessor fordert einen Lock an, indem er mit einer `fetch_and_increment` Operation den Anforderungszähler erhöht und gleichzeitig sein Ticket erhält. Sobald nun der Abgabezähler mit seinem Ticket übereinstimmt, hat der Prozessor den Lock erhalten. Er gibt danach den Lock wieder ab, indem er den Abgabezähler inkrementiert. Dieses Verfahren gewährleistet neben dem gegenseitigen Ausschluß auch noch eine FIFO-Behandlung der Lockanforderungen.
Das Problem liegt nun darin, wie der Prozessor erfährt, daß der Abgabezähler den Wert seines Tickets angenommen hat. In einem netzwerkbasierten System könnte dies über das Versenden von Nachrichten erfolgen, wozu allerdings ein Prozessor als Verwalter des Locks eingesetzt werden müßte. In einem System mit gemeinsamen Speicher kann dieser zusätzliche Aufwand vermieden werden, wenn alle Prozessoren gleichberechtigt über `test_and_set` pollen. Dabei sollte aber wieder mit Verzögerungen gearbeitet werden, um die Belastung des Speichersystems in Grenzen zu halten. Im Gegensatz zu dem einfachen `test_and_set` Lock ist hier aber eine exponentielle Verzögerungsdauer ungünstig: durch das FIFO-Prinzip kann es dazu kommen, daß sich die Wartezeiten entlang der wartenden Prozesse übermäßig ausdehnen. Aber im Prinzip kann ein Prozessor ja abschätzen, wie lange er auf den Erhalt des Locks warten muß, wenn er die Differenz zwischen seinem Ticket und dem Wert des Abgabezählers betrachtet. Dazu fehlt natürlich noch die Zeit, die jeder der Prozesse vor ihm den Lock belegen wird. Diese Zeit ist leider i.d.R. nicht bekannt, sie kann höchstens geschätzt werden. Um zu vermeiden, daß ein Prozessor länger als nötig wartet, sollte als Erwartungswert die *minimale* Zeit, die ein Prozessor den Lock halten kann, verwendet werden. Eine Implementierung dieses Locks ist in Algorithmus 4.2 dargestellt[2]. Der im bezeichneten Algorithmus verwendete `wait_factor` ist dabei der kritische Parameter.

Feldbasierter Lock

Auf cc-UMA-Systemen ist ein Lockalgorithmus möglich, bei dem die Zahl der Speicheroperationen eine feste, obere Grenze hat [21] (siehe Algorithmus 4.3). Dies ist bei den Algorithmen der vorhergehenden Kapitel nicht der Fall, da die Ausführungszeit des kritischen, exklusiven Bereiches, den der Lock sichert, nicht bekannt ist.
Neben kohärenten Caches muß das System noch die atomare `fetch_and_store` Operationen unterstützen. In der zentralen `lock`-Struktur steht für jeden Prozessor ein Flag in einem Feld, wobei die Elemente dieses Feldes auf separate Cachezeilen verteilt werden sollten. Diese kann z.B. durch Einführen von Dummydaten geschehen. In `tail` ist jeweils der letzte Prozessor der Warteschlange über eine Referenz auf seinen Eintrag in dem `slots`-Feld eingetragen. Ein neu hinzukommender Eintrag liest in einer atomaren `fetch_and_store` Operationen diese Referenz und trägt sich gleichzeitig selber als neuer letzter Prozessor ein. Er pollt sodann das Flag seines Vorgängers, was aufgrund der kohärenten Caches keine Zugriffe auf das Speichersystem mit sich bringt.

[2] In dem Algorithmus wird die Problematik des Überlaufs nicht behandelt; sie betrifft nur die Subtraktion. Auf vielen Maschinen funktioniert der Algorithmus aber bereits so wie hier beschrieben.

Algorithmus 4.2 Ticket Lock mit proportionaler Verzögerung

```
struct t_lock {
   unsigned int next_ticket = 0;
   unsigned int now_serving = 0;
} *lock;

ticket_lock (t_lock *lock) {
   /* Lock anmelden */
   my_ticket = fetch_and_increment (lock->next_ticket);

   /* Warten, bis an der Reihe */
   do {
      pause (wait_factor * (my_ticket - lock->now_serving));
   } while (lock->now_serving - my_ticket);

   do_something();

   /* Lock abgeben */
   lock->now-serving += 1;
}
```

4.2.2 Locks für NUMA-Systeme

Auf NUMA- und NORMA-Systemen ist es besonders wichtig, das für einen Lock eine möglichst geringe Zahl von Netzwerkoperationen erforderlich ist, da diese gegenüber Zugriffen auf lokalen Speicher teuer sind. Ansonsten würde die Leistung durch die auftretende Latenz zu stark sinken, zumal Locks ein sehr häufig verwendetes Mittel der Synchronisation sind. Ein Problem der NORMA-Systeme ist zusätzlich, daß auf ihnen die in Kapitel 4.1 besprochenen Primitive nicht verwendet werden können.

Listenbasierter MCS-Lock

Ein Lock-Algorithmus, der die Vorteile der vorhergehenden Algorithmen in sich vereinigt, wird von Mellor-Crummey und Scott in [21] vorgestellt. Er arbeitet gleich effektiv auf Systemen mit und ohne kohärenten Caches, benötigt die `fetch_and_store` Operation und profitiert auch von einer vorhandenen `compare_and_swap` Operation, die dem Algorithmus garantiertes FIFO-Verhalten verleiht. Für eine Lock-Belegung benötigt dieser Algorithmus nur O(1) Netzwerkoperationen, da er nur lokale Flags pollt. Somit ist er gut für NUMA-Systeme geeignet, wobei er sich ebenso auf UMA-Systemen einsetzen läßt. Jeder Prozessor, der den Lock belegen will, legt eine `qnode`-Struktur in seinem lokalen Speicher an. Diese lokalen Strukuren sind einfach verkettet[3]. Dazu gibt es noch

[3] Diese Verkettung ist auf verteilten Systemen bei virtuell gemeinsamen Speicher auch nur ein einfacher Zeiger. Wenn der Speicher auch logisch verteilt ist, ist diese Verketttung aufwendiger als ein einfacher Zeiger in den Adreßraum.

Algorithmus 4.3 Feldbasierter Lock mittels fetch_and_store

```
#DEFINE P number_of_processors

struct lock {
  /* jedes Element von 'slots' sollte in einer anderen Cachezeile
     oder Speicherbank liegen */
  bool slots[P];  /* Initialisierung: slot[j] = TRUE */

  /* die Struktur 'tail' kann in ein Prozessorwort gepackt werden
     und so in einer atomaren Operation behandelt werden */
  struct tail {
    bool *who_was_last = NIL; /* zeigt auf ein Element in slots */
    bool this_means_locked = FALSE;
  }
}

/* eindeutige Prozessorkennung */
private int vpid;
private struct wait_for_lock {
  bool *who_is_ahead_of_me;
  bool what_is_locked;
} W;

void acquire_lock (lock *L) {
  /* hole den aktuellen Schwanz der Warteschlange
     und trage dich selber ein */
  W = fetch_and_store (&L->tail, (&slots[vpid], slots[vpid]));
  /* spin */
  while (*W.who_is_ahead_of_me == W.what_is_locked) {}
}

void release_lock (lock *L) {
  L->slots[vpid] = !L->slots[vpid]
}
```

eine globale `lock`-Struktur, die immer auf den letzten Prozessor in der Liste zeigt, aber dem ersten Prozessor in der Liste zugeordnet ist. Durch das Flag `locked` in der lokalen `qnode`-Strukur wird diese Zuordnung angezeigt. Ausschließlich beim ersten Prozessor in der Liste ist das Flag gesetzt. Entsprechend setzt dieser Prozessor das Flag in der `qnode`-Struktur seines Nachfolgers in der Liste, wenn er den Lock abgibt. Falls der Zeiger in der `lock`-Struktur auf NIL zeigt, ist der Lock keinem Prozessor zugeordnet und somit frei verfügbar.

Anstelle des lokalen Pollens eines Flags kann hier auch Signalisierung eingesetzt werden, da ja jeder Prozess weiß, welcher Prozeß nach ihm den Lock erhalten soll. Diesen kann der dann individuell aufwecken, was auf NUMA-Systemen Kosten in der gleichen Größenordnung wie das Schreiben in eine entfernte Speicherstelle mit sich bringt. Auf diese Möglichkeit wird im nächsten Absatz eingegangen.

Algorithmus 4.4 Listen-basierter MCS-Lockalgorithmus

```
struct qnode {
   qnode *next;
   bool  locked;
}
typedef *qnode lock;

/* Parameter Q zeigt auf eine qnode Struktur, die fuer den
   aufrufenden Prozessor lokal, aber global sichtbar ist */
void acquire_lock (lock *L, qnode *Q) {
   Q->next = NIL;
   predecessor = fetch_and_store (L, Q); /* Netzwerktransaktion */
   if (predecessor != NIL) {
      /* Schlange war nicht leer */
      Q->locked = TRUE;
      predecessor->next = Q;
   }
   while (Q->locked) {}   /* spin */
}

void release_lock (lock *L, qnode *Q) {
   if (Q->next = NIL) {
      /* es gibt keinen Nachfolger */
      if (compare_and_swap (L, Q, NIL))
         /* compare_and_swap liefert TRUE, wenn getauscht wurde */
         return;
      while (Q->next = NIL) {}
   }
   Q->next->locked = FALSE;
}
```

2-Phasen Algorithmus

Die bisher vorgestellten Locks sind alle vom Typ *Spin Locks*, zum Teil in Verbindung mit Verzögerung, d.h. sie ziehen sich für eine Zeit vom aktiven Abfragen des Zustandes zurück. Diese Zeit wurde aber unabhängig vom tatsächlichen Zustand des Locks, etwa der Zahl der wartenden Prozesse, dimensioniert. Auf single-threaded betriebenen Prozessoren mit Caches bieten diese Algorithmen in ihrer individuellen Anpassung hohe Effizienz.

Bei Multithreading hingegen kann Blocking die Effizienz deutlich steigern, auch in Verbindung mit Spinning. Eine gründliche Analyse der Effizienz solcher 2-Phasen-Wartealgortihmen findet sich in [18]. Der kritische Parameter ist hierbei die Zeit T_{spin}, nach der ein Thread vom Zustand Spinning in den Zustand Blocking übergehen soll. Beim Blocking tritt ein Kostenfaktor K auf: der erforderliche Kontextwechsel (Auslagern und späteres Wiedereinlagern des blockierten Threads) und damit verbunden die Signalisierung des Threads. Dann kann man schreiben:

$$T_{spin} = \alpha K$$

Zur Bestimmung des α berechnet man die zu erwartenden Kosten des Wartens über

$$E = \int_0^{\alpha K} t f(t) dt + \int_{\alpha K}^{\infty} (1+\alpha) K f(t) dt$$

Dabei ist $f(t)$ die Wahrscheinlichkeitsdichtefunktion der Wartezeit. Das erste Integral stellt den Beitrag zu den zu erwartenden Wartekosten für den Fall dar, daß die Wartezeit kürzer als αK ist; das zweite Integral ist der Beitrag der Wartezeiten dar, die größer als αK sind. Dieser setzt sich zusammen aus T_{spin} plus K. Daraus werden in der angegebenen Studie optimale statische Zeiten für T_{spin} zu $\alpha = 0.5$ bei exponentieller Verteilung der Wartezeiten der Threads, zu $\alpha = 0.62$ bei gleichförmiger Verteilung und zu $\alpha = 1$ bei unbekannter Wartezeitverteilung ermittelt.

Das Verfahren des Kontextwechsels und zur Signalisierung eines Threads ist abhängig von der Hardware-Architektur des Prozessors (mehrfache Hardware-Kontexte) und des Schedulers, so daß auf deren Effizienz kaum Einfluß genommen werden kann.

4.2.3 Locks für NORMA-Systeme

Auf NORMA-Systemen ist die zentrale Verwaltung eines Lock zwar auch möglich, aber nicht optimal. Denn Versuche, einen Lock zu belegen, müssen über den Austausch von Nachrichten erfolgen, die bei zentraler Verwaltung des Locks alle an einen einzigen Prozessor gerichtet werden. Das führt dazu, daß zum einen das Netzwerk ungleichmäßig belastet wird (was je nach Typ des Netzwerks verschiedene Auswirkungen auf die gesamte Kommunikation hat) und auch der betreffende Prozessor ist unter Umständen hauptsächlich mit der Verwaltung der Locks beschäftigt. Wenn man diese Effekte nicht tolerieren kann, ist die Verwendung eines *verteilten Locks* notwendig. Ein Überblick über die verschiedenen verteilten Locks wird in [37] gegeben.

Token-basierter Lock

Bei token-basierten Lock-Algorithmen ist ein einzelner, aber unbestimmter Prozessor im Besitzt des Tokens, also der Berechtigung zum Eintritt in den kritischen Abschnitt. Für einen Prozessor, der sich vor einem kritischen Abschnitt befindet, besteht das Problem nun darin, den Prozessor zu finden, der das Token besitzt. Dazu gibt es verschiedene Methoden (wobei im folgenden die Zahl der Prozessoren mit P, die Zeit zur Übertragung einer Nachricht mit T und die Zeit, die der Lock von einem Prozessor gehalten wird, mit E bezeichnet wird):

Broadcast: Ein Broadcast an alle Prozessoren führt zu P Nachrichten (abhängig von der Art der Implementierung des Broadcasts auf dem Netzwerk des Systems) für einen Tokenerwerb. Die Antwortzeit liegt bei $2T$ für niedrige Last und nähert sich bei hoher Last dem Wert $P * (T + E)$: alle anderen Prozessoren führen dann den kritischen Abschnitt aus, bevor der gleiche Prozessor wieder den Token erhält.

Logische Struktur Um den starken Nachrichtenverkehr zu vermeiden, wie er bei einem Broadcast auftritt, kann eine logische Kommunikationsstruktur eingerichtet werden, üblicherweise einen Ring, einen Baum oder ganz allgemein einen Graphen.

In einem Ring werden im Durchschnitt $\frac{P}{2}$ Nachrichten pro Tokenerwerb versandt, die Antwortzeit liegt aber bei $T * \frac{P}{2}$, weil die Nachrichten nicht parallel verschickt werden. Daher kann ein Ring sehr ineffizient sein.

Bei einem Baum werden die Prozessoren logisch zu einem gerichteten Baum verbunden, wobei der Token im Besitz des Prozessors an der Wurzel ist. Wenn der Token weitergegeben wird, werden auf dem Weg von der Wurzel zum Zielprozessor die Richtungen der passierten Kanten umgedreht, so daß der Prozessor mit dem Token die Wurzel des Baumes darstellt. Das Verfahren in seiner einfachen Form führt natürlich zu einem unausgeglichenen Baum, jedoch wurde in [30] ermittelt, das der Durchmesser (maximale Distanz zwischen zwei Knoten) eines derartigen, zufällig aufgebauten Baumes O($logN$) beträgt. Das führt zu einer Antwortzeit von $2T * logP$ bei geringer Last bis zu $P * (T + E)$ bei Volllast. Graphen werden prinzipiell genauso wie Bäume gehandhabt. Ihrem Vorteil der Redundanz (es können mehrere Wege vom Absender zum Adressaten führen) steht der Nachteil des höheren Nachrichtenaufkommens zur Vermeidung von Endlosschleifen von Nachrichten im Graphen gegenüber.

Der Nachteil all dieser Kommunikationsstrukturen ist die serielle Verarbeitung der Nachrichten, wodurch der Broadcast bei geringer Netzwerklast schneller ist. Andererseits wird durch das geringere Nachrichtenaufkommen der Fall hoher Auslastung erst später als beim Broadcasting erreicht.

Dynamische Ermittlung: Im Gegensatz zu den statischen Verfahren der vorhergehenden Abschnitte passen dynamische Verfahren die Kommunikationsstruktur so an, daß der Token möglichst schnell gefunden werden kann. In [29] wurde ein solcher dynamischer Algorithmus für Semaphoren implementiert und quantitativ analysiert. Da Locks

nichts anderes als binäre Semaphoren sind, können die Ergebnisse hier verwendet werden. Grundsätzlich wird folgendermaßen vorgegangen: eine Speicherseite enthält die Informationen über eine Anzahl m von Semaphoren (sieh Abb. 4.1), und jeder Prozessor hat einen solchen Datensatz zu jeder Semaphore. Jedoch sind die Inhalte dieser Datensätze nicht auf allen Prozessoren gleich. In diesen Datensätzen sind folgende Felder enthalten:

- **Semaphore:** Zeiger auf die eigentliche Semaphorenstruktur, wenn die Semaphore lokal vorliegt, ansonsten `NULL`.
- **probable_holder:** Nummer des Prozessors, der die Semaphore wahrscheinlich besitzt, d.h. lokal im Speicher hält. Wenn die Semaphore bereits auf dem Prozessor vorhanden ist, ist es natürlich die eigene Prozessornummer. Ansonsten ist es die Nummer des Prozessors, der nach dem Kenntnisstand dieses Prozessors die Semaphore besitzt.
- **block_req:** Schlange von Prozessor-Nummern, die auf die Freigabe der Semaphore warten.
- **timestamp:** gibt dem Feld `probable_holder` ein Alter, um die Aktualität mehrerer Felder auf verschiedenen Prozessoren beurteilen zu können.
- **requested:** ein Flag, das anzeigt, ob die Semaphore, wenn im lokalen Besitz, bereits von einem anderen Prozessor angefordert wurde.

Die Semaphoren erhalten eine initale Verteilung, danach können die Semaphoren zwischen den Prozessoren wandern. Durchzuführende P- (Belegung der Semaphore) oder V-Operationen (Freigeben der Semaphore) werden von einem Prozessor an den ihm bekannten Besitzer der Semaphore gerichtet und von dort ggf. weitergeleitet, bis die Anfrage den aktuellen Besitzer erreicht. Es ist sinnoll, wenn weiterleitende Prozessoren dabei die Nummer des Prozessors, von dem die Anforderung stammt, in ihr `probable_hold`-Feld eintragen, da diese Information in jedem Fall aktueller ist als die vorhandene. Wenn die Anforderung den Zielprozessor erreicht hat, kann eine V-Operation direkt durchgeführt werden, bei einer P-Operation wird dem Ausgangsprozessor die Semaphore bei Verfügbarkeit zugestellt, ansonsten wird er in die Schlange `blocked_req` der ausstehenden Anforderungen eingereiht.

Zusammen mit der Anfrage nach einer P- oder V-Operation kann ein Prozessor seine aktuellen Informationen (Prozessornummer und zughöriger Zeitstempel) über die Semaphoren einer Seite versenden, die im Verlauf des Weiterleitens der Anfrage zur Aktualisierung der jeweils lokalen Datenstrukturen genutzt werden können. Diese aktualisierten Datenstrukuren werden dann an den nächsten Prozessor weitergeleitet. Damit ähnelt der Algorithmus sehr stark dem *Dynamic Distributed Manager* in SVM-Systemen [17].

Die Untersuchungen zur Leistung dieses Algorithmus ergaben als qualitative Ergebnisse, daß der verteilte Algorithmus mit zunehmender Benutzungshäufigkeit der Semaphoren gegenüber dem zentralen Algorithmus kürzere Antwortzeiten liefert. Nur bei sehr schwacher Nutzung der Semaphoren ist der zentrale Algorithmus im Vorteil. Dies ist einsichtig, da beim Zentralalgorithmus weniger Nachrichten anfallen, der zentrale Prozessor jedoch

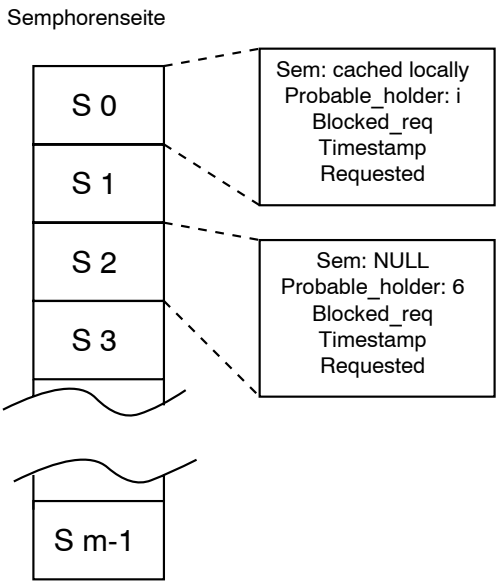


Abbildung 4.1: Datenstrukturen auf einer Semaphorenseite

bei steigender Semaphorennutzung zum Flaschenhals wird. Dieser Effekt wird bei zunehmender Prozessorzahl stärker, während der verteilte Algorithmus gut skaliert. Sein Durchsatz, gemessen in Semaphorenbelegungen pro Zeiteinheit, bleibt auch bei steigenden Prozessorzahlen (gemessen wurde von 10 bis 45 Prozessoren) konstant, während der zentrale Algorithmus in diesem Bereich bereits einen Rückgang des Durchsatzes aufwies.

Es liegt daher nahe, zur Erreichung maximaler Leistung eine Kombination der Algorithmen zu wählen: seltener benötigte Semaphoren werden zentral verwaltet, was wiederum den Speicherplatzbedarf für die Semaphorenverwaltung auf jedem Prozessor senkt.

Token-loser Lock

Im Gegensatz zu token-basierten Algorithmen ist bei token-losen Algorithmen nicht ein einzelner, unbestimmter Prozessor im Besitz des Tokens, sondern alle Prozessoren verfügen über Informationen über den Status der anderen Prozessoren. Grundsätzlich sind folgende Zustände möglich:

- der Prozessor hat den Lock angefordert
- der Prozessor ist im Besitz des Locks
- der Prozessor benötigt den Lock nicht

Insgesamt hat jeder Prozessor i drei Datensätze vorliegen, die jeweils aus einer Liste von Prozessorkennungen bestehen:

- *Anforderungsatz* R_i: die Menge der Prozessoren, bei denen der Lock angefordert werden muß
- *Informationssatz* I_i: die Menge der Prozessoren, die über eine Zustandsänderung im Sinne der oben beschriebenen Zustände informiert werden müssen
- *Statussatz* St_i: die Menge der Prozessoren, über die Prozessor i Informationen besitzt, d.h. es gilt für alle $i : S_i \in I_j \Rightarrow S_j \in St_i$. Hiermit kann also ermittelt werden, ob der Lock gerade belegt ist und welcher Prozessor ihn hält.

Aufbauend auf diesen Informationen verfährt der sogenannte *Maekawa*-Algorithmus folgendermaßen: der Prozessor i versendet an alle Prozessoren aus R_i eine Anforderungsnachricht mit einem prozessorglobalen Zeitstempel. Den Lock hat er erhalten, wenn er von all diesen Prozessoren eine Bestätigung bekommt. Wenn er den Lock wieder abgibt, versendet er eine entsprechende Statusnachricht an alle Prozessoren aus I_i. Auf den Prozessoren, die die Anforderungsnachricht erhalten, wird sie zu anderen Anforderungen in eine nach dem Zeitstempel geordnete Schlange eingereiht. Wenn anhand der Informationen in St_i der Lock verfügbar ist, wird an den vordersten Prozessor in der Warteschlange eine Bestätigungsnachricht verschickt und dieser aus der Schlange entfernt. Daneben müssen noch die Statusinformationen in St_i aktualisiert werden: wenn eine Bestätigungsnachricht an einen Prozessor versandt wurde, wird in St_i vermerkt, daß dieser Prozessor den Lock besitzt. Kommt später eine Statusnachricht an, wird wiederum St_i aktualisiert und ggf. eine neue Bestätigungsnachricht versandt.

Neben diesem Algorithmus gibt es als Variante den *Ricart-Agrawala*-Algorithmus. Er schickt nur dann auf eine Anforderungsnachricht nicht umgehend eine Bestätigung, wenn er eine eigene Anforderung höherer Priorität (also mit älterem Zeitstempel) abgesetzt hat. Bei diesem Verfahren kann also auch mehr als nur eine einzelne Bestätigung gegeben werden. Ein Prozessor hat aber erst dann den Lock erhalten, wenn er von *allen* Prozessoren die Bestätigung bekommen hat. Dieser Algorithmus kann variiert werden, indem ein Prozessor i, der von Prozessor j eine Bestätigung erhalten hat, diese solange als gültig betrachtet, bis i von j eine Anforderung erhält. Dadurch muß i an j in diesem Zeitraum keine Anforderungen mehr schicken, um den Lock zu benutzen.

Die Eigenschaften der vorgestellten verteilten Lock-Algoritmen sind abschließend in Tabelle 4.1 aufgeführt. Die verwendeten Beurteilungskriterien, die jeweils noch für *niedrige Last* (NL, keine Kollision von Anfragen) und *hohe Last* (HL, es erfolgen bereits neue Anfragen bevor der Lock wieder frei ist) aufgeteilt wurden, sind:

- **Nachrichten:** die Zahl der Nachrichten, die für einen Lockerwerb generiert wird.
- **Verzögerung:** die Zeit, die zwischen der Abgabe eines Locks und dem Erwerb dieses Locks durch den nächsten Prozessor vergeht. Diese Angabe ist nur für hohe Last relevant.
- **Antwortzeit:** die Zeit, die von dem Absetzen der Anfrage durch einen Prozessor bis zum Erhalt der Bestätigung durch ebendiesen vergeht.

	Token-basiert			Token-los	
	Broadcast	Logische Struktur	Dynamisch	Ricart-Agrawala	Maekawa
Nachrichten					
NL:	hoch	niedrig	O(1)[1]	hoch[2]	mittel
HL:	hoch	sehr niedrig	O(log P)[1]	hoch	hoch[3]
Verzögerung					
HL:	T	T	T	T	$2T$
Anwortzeit					
NL:	$2T$	$TlogP$	$O(T)^1$	$P-1$	$2T$
HL:	$P(T+E)$	$P(T+E)$	$O(P)^1$	$2(P-1)$	$P(T+E)$

Tabelle 4.1: Eigenschaften verteilter Lock-Algorithmen

Bemerkungen zur Tabelle:

1. Diese Werte für den dynamischen Algorithmus wurden aus der Simulation in [29] abgeleitet und nicht analytisch bestimmt.

2. Nachrichtenaufkommen kann durch die Verwendung der dynamischen Variante verringert werden

3. Augrund von auftretenden bzw. aufzulösenden Deadlocks bei hoher Last

4.2.4 Pagelocks

Auf NORMA-Systemen mit virtuell gemeinsamem Speicher ist der Zugriff auf entfernten Speicher um ein vielfaches teuerer als Zugriff auf lokalen Speicher. Dieses Problem wirkt sich besonders verheerend aus, wenn es zum *Thrashen* kommt, also dem wechselseitigen Anfordern der gleichen Seite von mehreren Prozessoren. In einem einfachen Szenario kann dies leicht dargestellt werden: zwei Prozessoren A und B greifen beispielsweise in einer Schleife auf verschiedene Felder zu, die aber in der gleichen Speicherseite liegen. Der lokale Zugriff kostet die Zeit t_{lokal}, der entfernte Zugriff (also das Übertragen der Seite t_{Seite} in den lokalen Speicher plus der lokale Zugriff selbst) kostet $t_{entfernt} = t_{Seite} + t_{lokal}$. Wenn nun im gleichen Zeitraum Prozessor A und B jeweils k Zugriffe auf die Felder durchführen, kostet dies im ungünstigsten Fall, wenn nach *jedem* Zugriff durch den einen Prozessor die Seite wieder auf den anderen Prozessor übertragen wird, jeden Prozessor die Zeit

$$\begin{aligned} T_{thrash} &= (2k-1) \cdot t_{Seite} + k \cdot t_{lokal}) \\ &\approx k \cdot (2t_{Seite} + t_{lokal}) \qquad \text{mit } k \gg 1 \end{aligned}$$

Hingegen beträgt die optimale Zeit T_{opt}, in der ein solcher Zugriff abgewickelt werden kann, unter der Vorraussetzung, daß die Seite auf keinem der beiden Prozessoren vorliegt:

$$T_{opt} = t_Seite + k \cdot t_{lokal}$$

Das Verhältnis $V = \frac{t_{Seite}}{t_{lokal}}$ liegt typischerweise in einer Größenordnung von 10^4, so daß die Näherung $2V + 1 \approx 2V$ zulässig ist. Damit ergibt für Verhältnis von T_{thrash} zu T_{opt}:

$$\frac{T_{thrash}}{T_{opt}} = 2 \cdot \frac{V \cdot k}{V + k}$$

Für den Fall $V = 10^4$ und $k = 10^3$ ergibt sich ein Laufzeitunterschied vom Faktor 1818. Auch wenn dies ein worst-case-Szenario ist, wird klar, daß ein solches Verhalten auf jeden Fall verhindert werden muß. Im ASVM [20], einer Implementation von virtuell gemeinsamen Speicher auf der Intel Paragon, wird zur Verminderung dieses Effekts ein *temporärer Lock* auf jede Seite gelegt, wenn sie gerade migriert ist, so daß sie für einen festen Zeitraum von 2 ms diesem Prozessor nicht entzogen werden kann. Allgemein kann solch eine Sicherung auch auf Ebene des Anwendungsprogramms durch den Einsatz der bereits bekannt Locks geschehen, jedoch ist dies in diesem Fall keine optimale Wahl. Denn wenn mit dem Lock ein kritischer Abschnitt gebildet wird (beispielsweise die gesamte Schleife abgesichert wird), ist eine parallele Verarbeitung der Schleife auch dann nicht möglich, wenn es gar nicht zu den beschriebenen Seitenkonflikten kommen würde. Wenn man dies verhindern will und daher jeden einzelnen Zugriff auf gemeinsame Felder durch einen Lock absichert, werden zum einen die Seitenkonflikte nicht sicher verhindert und zum anderen entsteht durch die damit verbundene exzessive Nutzung von Locks ein starker Overhead sowohl bei der Laufzeit als auch im Quelltext. Was man eigentlich braucht, ist eine Möglichkeit, alle Speicherseiten, in denen ein Objekt (z.B. ein Feld) liegt, einem Prozessor exklusiv zu dedizieren.

Ein solches Konstrukt ist ein *Pagelock*. Pagelocks machen, wie oben dargelegt, besonders auf NORMA-Systemen mit virtuell gemeinsamen Speicher (SVM) Sinn sind, wobei es sich dann um verteilte Locks handelt. Unter SVM wird der gesamte virtuelle Adreßraum auf physikalische, verteilte Speicherseiten abgebildet, die sich einfach durchnumerieren lassen. Die Verwaltung des Locks für eine jede Seite kann somit beispielsweise zyklisch auf die vorhandenen Prozessoren verteilt werden. Da der Speicherbereich (also die Adresse) eines Objekts bekannt ist, kann durch einfache Modulo-Division ebenso der zuständige Prozessor für den Pagelock ermittelt werden. Der Vorteil dieses Algorithmus mit einer festen Verteilung ist seine Einfachheit und der deterministische Kommunikationsaufwand und somit auch deterministisches Zeitverhalten bei unbelegtem Lock, denn es sind für jede Belegung eines Pagelocks nur zwei Nachrichten auszutauschen. Abhängig von der Nutzung des Locks (durch wieviele Prozessoren in welcher zeitlichen Abfolge) kann aber auch ein verteilter Lock vorteilhaft sein, da die Verwaltung eines Locks zum Inhaber migriert und dieser bei der folgenden Belegung des Locks gar keine Kommunikation betreiben muß, wenn der Lock zwischenzeitlich nicht angefordert wurde. Andererseits kann bei einem verteilten Algorithmus der Kommunikationsaufwand auch mehr als zwei Nachrichten betragen.

Neben NORMA-Systemen können Pagelocks aber auch auf allen anderen parallelen Systemen gewinnbringend eingesetzt werden, da sie die Möglichkeit bieten, eine kritische Region zu bilden, in der denoch parallel gearbeitet werden kann, solange nicht auf Speicherbereiche zugegriffen wird, die ein Prozessor durch einen Pagelock gesichert hat. Pagelocks sind in dieser Arbeit auch deshalb von Interesse, weil es sich unmittelbar anbietet, Multithreading einzusetzen, wenn ein Prozessor an einem Pagelock aufgehalten wird.

4.3 Barrieren

Barrieren sind ein Standardelement in der Implementierung von parallelen Algorithmen auf Multiprozessoren oder bei Verwendung von Multiprogramming/Multithreading auch auf Einzelprozessoren. Sie unterteilen ein Programm in einzelne logische Abschnitte oder trennen einzelne Schritte einer Iteration voneinander, indem sie gewährleisten, daß kein Prozessor bzw. Thread über einem bestimmten Punkt (die Barriere) hinaus mit der Programmabarbeitung fortfährt, ehe nicht alle beteiligten Prozessoren bzw. Threads an diesem Punkt angelangt sind. Dies impliziert Wartezeiten und Kommunikation. Die Wartezeiten liegen nicht in der Implementierung der Barriere begründet, sondern in nicht optimaler Lastverteilung durch den parallelen Algorithmus. Die verwendete Barriere kann jedoch eine sinnvolle Nutzung der auftretenden Wartezeit ermöglichen. An der Barriere auftretende Verzögerungen durch Kommunikation zwischen den Prozessoren lassen sich jedoch durch ein der Architektur des Multiprozessors angepaßtes Design der Barriere verringern. Dazu werden im folgenden verschiedene Barrierenalgorithmen vorgestellt. In Kapitel 4.3.6 werden anschließend die Eigenschaften der behandelten Barrieren zusammengefaßt dargestellt.

4.3.1 Grundlegende Algorithmen für Barrieren

Grundprinzipien der Barrieren Die zwei grundlegenden Konzepte für Barrieren sind *lineare Barrieren*, auch Zentralbarrieren genannt, und *baumstruktierte Barrieren*. Aus diesen beiden Varianten kann eine dritte Variante erzeugt werden, die *kombinierte Barriere*, die auf multiprogrammierten Parallelrechnern sinnvoll eingesetzt werden kann.

Zentralbarriere Die einfachste Implementierung einer Barriere ist ein akkumulierender Zähler. Dieser Zähler wird zu Null initialisiert, Prozessoren, die an der Barriere ankommen, erhöhen diesen Zähler in einer atomaren Operation. Anschließend müssen sie warten, bis alle Prozessoren an der Barriere angekommen sind. Dies wird durch ständigen Vergleich des Zählers mit der erwarteten Anzahl von Prozessoren erreicht. Wenn die Barriere komplett ist (und auch gewährleistet ist, daß alle Prozessoren dies erkannt haben), wird der Zähler wieder zu Null gesetzt.

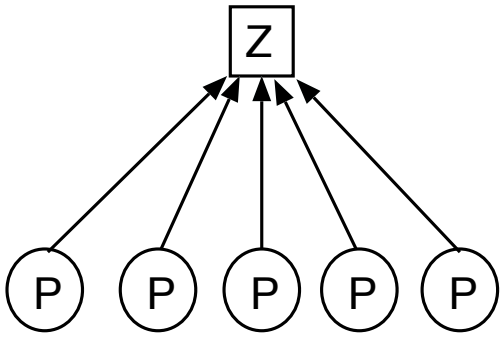


Abbildung 4.2: Struktur der Zentralbarriere bei 5 Prozessoren

Dieser Ansatz birgt offensichtlich zwei Nachteile in sich. Zum einen muß die Inkrementierung des Zählers atomar erfolgen, da es sonst zu *Race Conditions* kommen kann. Dazu muß entweder Unterstützung in der Hardware des Rechners vorhanden sein, oder der Zähler muß explizit gesperrt werden, etwa über einen Lock, was wiederum aufwendig ist. Der schwerwiegendere Nachteil ist die Belastung des Speicherbusses bzw. des Netzwerks durch den dauernden Zugriff einer zunehmenden Zahl von Prozessoren (am Ende sind alle Prozessoren beteiligt) auf eine Speicherstelle. Der dadurch erzeugte *Hot-Spot*, also eine Konzentration der Kommunikation auf eine einzige Stelle im System, kann zwar auf geeigneten Architekturen durch die Nutzung von lokalen Caches, die über ein Broadcast kohärent gehalten werden, entschärft werden. Dennoch benötigt diese *zentrale Barriere* zur Komplettierung bei P Prozessoren immer noch Zeit in der Größenordnung O(P), da insgesamt P serielle Zugriffe auf den zentralen Zähler erfolgen müssen.

Baumstrukturierte Barriere Als erster Schritt zur Verminderung der oben angesprochenen Problematik bietet es sich an, anstelle der flachen Zugriffshierarchie, bei der alle Prozessoren auf dieselbe Variable zugreifen, eine Baumstruktur einzuführen. Die zu synchronisierenden Prozessoren werden in Gruppen zu jeweils n Prozessoren eingeteilt (was

zu einem n-ären Baum führt, Beispiel für $n = 2$ in Abbildung 4.3) und bilden eine Subbarriere. Jeweils ein Prozessor dieser Gruppe, der entweder statisch oder dynamisch (der letzte Prozessor, der ankommt) bestimmt werden kann, führt ein weiteres Rendezvous mit den benachbarten Knoten im gemeinsamen Vaterknoten aus, bis die Wurzel des Baums erreicht ist. Damit sind alle Prozessoren an der Barriere angekommen, nun müssen noch die wartenden Prozessoren aufgeweckt werden. Das Verfahren für dieses Aufwecken muß sich an die Speicherarchitektur des Systems anpassen: auf einem System mit gemeinsamen Speicher und kohärenten Caches kann dazu einfach ein zentrales Busy-Flag verwendet werden. Dies ist jedoch keine leistungsfähige Lösung für ein NORMA-System, auf dem jeder nicht-lokale Speicherzugriff (wie es bei einer zentralen Speicherstelle der Fall ist) das Netzwerk belastet. Dort kann der Baum zurücktraversiert werden indem jeder Prozessor, der zu seiner Gruppe von wartenden Prozessoren zurückkehrt, diesen eine individuelle Nachricht schicken.

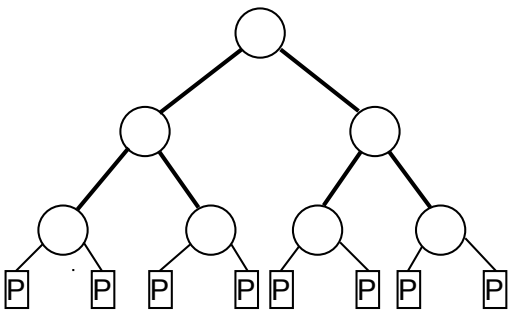


Abbildung 4.3: Struktur der Baumbarriere für $n = 2$

Eine Implementierung einer solchen Barriere wird in Algorithmus 4.5 dargestellt.

Kombinierte Barriere Wenn auf einem NUMA- oder NORMA-System eine Applikation sowohl mehrere Prozessoren belegt und auf diesen Prozessoren wiederum mehrere Threads anlegt, ist der Einsatz einer kombinierten Barriere sinnvoll. Dabei findet die Synchronisation auf dem Prozessor mit einer Zentralbarriere statt, die bei gemeinsamen Speicher sehr leistungsfähig ist. Sobald die Threads auf einem Prozessor synchronisiert sind, kann dieser Prozessor an der globalen Synchronisation aller Prozessoren über eine baumstrukurierte Barriere teilnehmen, die den Kommunikationsaufwand möglichst gering hält.

Die Laufzeit von linearen Barrieren ist von der Ordnung O(P), während baumstrukturierte Barrieren eine Laufzeit der Ordnung O(log P) erreichen können. Man könnte daher meinen, baumstrukturierte Barrieren seien grundsätzlich leistungsfähiger und daher vorzuziehen. Dabei muß aber berücksichtigt werden, daß die Baumstruktur unter Umständen mehr Overhead bei der Initialisierung und Verwaltung hat. So ist es aufwendiger, bei veränderter Zahl von Prozessoren, die an der Barriere teilnehmen sollen, die Baumstruktur anzupassen als bei der linearen Barriere einfach den zugehörigen Zähler zu ändern.

Algorithmus 4.5 Algorithmus der *Combining Tree* Barriere

```
typedef struct TREE_NODE {
   int fan_in;         /* Fan-in dieses Knotens */
   int count;          /* zu fan_in initialisiert */

   bool locksense;     /* zu FALSE initialisiert */
   struct TREE_NODE *parent;   /* NIL an der Wurzel */
   } tree_node;

shared  tree_node nodes[P]; /* die Knoten des Baumes, ggf. im
                               Speicher plazieren */
private bool sense = TRUE;
private treenode *my_node;

void combining_barrier(void) {
   combining_barrier_aux (my_node);
   sense = !sense;
};

void combining_barrier_aux (tree_node *this_node) {
   if (fetch_and_decrement (&this_node->count) == 1) {
     /* der letzte Prozess erreicht den Knoten */
     if (this_node->parent != NIL)
         combining_barrier_aux (this_node->parent)
      this_node->count = this_node->fan_in; /* Reinitialisieren */
      this_node->locksense = !this_node->locksense;
   }
   while (this_node->locksense != sense) {}   /* Polling */
}
```

Das gleiche gilt für die Frage, wie die Prozesse ihre Wartezeit an der Barriere verbringen sollen. Als Alternativen bieten sich hier *Polling* und *Blocking* an. Beim Polling, dem dauernden Abfragen einer Variable bis zum Eintreten des gewünschten Zustandes, verbraucht der Prozessor Rechenzeit und Speicherbandbreite und verrichtet neben dem Warten auf das Ereignis keine weitere Arbeit. Beim Blocking hingegen gibt der Thread bzw. Prozeß für die Wartezeit die Kontrolle über den Prozessor ab, der sich somit anderen Aufgaben widmen kann, oder der Prozess behält den Prozessor und widmet sich während der Wartezeit anderen Aufgaben, die nicht mit der laufenden Synchronisation in Zusammenhang stehen. Diese Lösung scheint prinzipiell vorteilhafter als das Polling zu sein, aber es ergeben sich auch Probleme, die diesen prinzipiellen Vorteil des passiven Wartens relativieren können. So weiß der Prozess nicht, wie lange er warten muß, wenn er die Kontrolle abgibt. Der doppelte Kontextwechsel kostet darüberhinaus eine Zahl von Taktzyklen, deren untere Grenze recht hoch sein kann. Wenn diese Zahl größer ist als die ursprüngliche Wartezeit, lohnt sich das Blocking nicht, sondern kostet mehr als Polling.

In den folgenden Abschnitten werden daher verschiedene Barrierenalgorithmen vorgestellt. Dabei wird auch zwischen Barrieren für netzwerkbasierte Rechner und für Rechner mit gemeinsamen Speicher unterschieden, da für diese Architekturen oftmals unterschiedliche Algorithmen die besseren Ergebnisse bringen. Die grundlegenden Bewertungskriterien für eine Barriere sind

- Länge des kritischen Pfades: der kritische Pfad ist die maximale Zahl von Kommunikationsschritten, die zum Erreichen der Barriere für einen Prozessor notwendig ist. Bei der Zentralbarriere ist diese Zahl 1, bei einem binären Baum beträgt sie $log_2(P)$.
- Gesamtanzahl der Netzwerkoperationen
- Belastung des Speichersystems / des Netzwerks
- Speicherbedarf der notwendigen Datenstrukturen
- Implementierbarkeit mit verfügbaren atomaren Operationen

Grundsätzlich tritt bei Barrieren der Widerspruch zwischen Minimierung der Konzentration auf eine einzelne Speicherstelle (bzw. einen einzelnen Prozessor in einem NORMA-System) und der gleichzeitigen Minimierung der Länge des kritischen Pfades auf. Hier gilt es, abhängig von der zugrundeliegenden Architektur, ein Optimum zu finden. Hierzu ist es hilfreich, wenn bereits zum Zeitpunkt der Auswahl des Barrierentyps bekannt ist, wie die Ankunftszeiten der beteiligten Prozesse verteilt sind. Dabei ist es oftmals einfacher, eine effiziente Synchronisation auf einer netzwerkbasierten Architektur zu realisieren, da hier die Kommunikation explizit vom Programmierer kontrolliert wird, was jedoch bei der Programmierung größeren Aufwand bedeutet. Hingegen haben in Systemen mit gemeinsamen Speicher die Caches, Prefetching und Invalidierung maßgeblichen Einfluß auf den zeitlichen Ablauf eines Speicherzugriffs, und der Ausgang von konkurierenden Zugriffen unterliegt häufig nicht der Kontrolle des Programmierers. Hier müssen für eine

effektive Implementierung Strukturen der Hardware, wie z.B. Aufbau und Kohärenzprotokoll der Caches und das Bussystem berücksichtigt werden, was zu sehr hardwarenahen Implementierungen führen kann.

In den folgenden Kapiteln werden Algorithmen für Barrieren vorgestellt, die alle mehr oder weniger auf den vorgestellten Basistypen aufbauen, aber in bestimmte Richtungen optimiert wurden.

4.3.2 Barrieren für UMA- und NUMA-Systeme

Auf UMA-Systemen sind folgende Randbedingungen für den Entwurf mit Barrieren zu beachten:

- Es gibt nur eine Kategorie von Speicher, so daß Zugriffe im Prinzip immer gleich teuer sind. Dennoch sollte der Speicheraufbau im Hinblick auf den parallelen Zugriff berücksichtigt und Hot-Spots vermieden werden

- Zentrale Flags, die von einem Prozessor beschrieben und von allen anderen gelesen werden, sind nur bei Einsatz von geeigneten Caches, die das Speichersystem von den Lesezugriffen auf das Flag entlasten, effizient, was die Wahl des Aufweckalgorithmus bestimmt.

cc-NUMA-Systeme erfüllen den zweiten Punkt oft ebenso wie UMA-Systeme mit Cache, wenn die Daten nur entsprechend verteilt werden können. Unter dieser Vorraussetzung kann man die Algorithmen für Barrieren für diese beiden Architekturen generell zusammenfassen, bei bestehenden Unterschieden wird darauf hingewiesen.

Ausstreuende Barrriere Die *ausstreuende Barriere* von Brooks [5] verwendet zur Kommunikation keine Baumstruktur, sondern eine Butterfly-Struktur. Brooks' Algorithmus wurde von Hensgen *et al.* [13] zur *Dissemination Barriere* erweitert, deren Kommunikationsstruktur in Abbildung 4.4 dargestellt ist. Dabei synchronisiert sich Prozessor i in Runde k mit Prozessor $(i + 2^k) mod P$, was zu $\lceil log_2 P \rceil$ Synchronisationsrunden führt. Eine Runde ist hierbei als eine Teilsynchronisation der Prozessoren nach einem vorgegebenen Kommunikationsmuster anzusehen; verschiedene Runden laufen unter verschiedenen Kommunikationsmustern ab. Damit werden insgesamt $Plog_2 P$ Kommunikationsschritte benötigt, was auf netzwerkbasierten Rechnern nicht optimal ist. Jedoch wird jedes Flag nur von einem Prozessor gepollt, und es ist statisch festgelegt, welche Flags jeder Prozessor pollt. Dadurch können diese Flags lokal angelegt werden. Dieser Barrierentyp ist folglich für UMA- und auch cc-NUMA-Systeme geeignet, wenn auch die Kommunikation noch relativ hoch ist.

Algorithmus 4.6 Algorithmus für die Dissemination Barriere

```
typedef struct FLAGS {
   bool my_flags [1][logP -1];
   bool partner_flags [1][logP -1];
} flags;

private int parity = 0;
private bool sense = TRUE;
private flags *local_flags;

shared flags all_nodes[P-1];
             /* all_nodes[i] ist im gemeinsamen Speicher,
                aber lokal fuer Prozessor i */

/* Auf Prozessor i zeigt jeweils local_flags auf all_nodes[i]
   all_nodes[i].my_flags[r][k] zu FALSE initialisiert
                               fuer alle i, r, k
   Wenn j == (i + 2^k) mod P ist fuer r = 0, 1:
      all_nodes[i].partner_flags[r][k] zeigt auf
        all_nodes[j].my_flags[r][k] */

void dissemination_barrier() {
   int instance;

   for (instance = 0; instance < log P; instance++) {
      *local_flags->partner_flags[parity][instance] = sense;
      /* spin */
      while (local_flags->my_flags[parity][instance] != sense)
        {}
   }
   if (parity == 1)
      sense = !sense;
   parity = 1 - parity;
}
```

Wettbewerbs-Barriere (Tournament Barrier) Ebenfalls Hensgen *et al.* [13] haben die Wettbewerbsbarriere entwickelt. Sie basiert auf einem binären Baum, an dessen Blättern die Prozessoren starten, was vergleichbar mit der Combining-Tree-Barriere bei einem Fan-In von 2 ist. In diesem Fall ist jedoch der Gewinner an jedem Knoten statisch festgelegt, was `fetch_and_`Φ Operationen überflüssig macht. In Runde k setzt Prozessor i ein Flag, das von Prozessor j abgefragt wird, wobei $i = 2^k$ und $j = i - 2^k$ ist (siehe Abbildung 4.4). Danach scheidet Prozessor i aus dem Wettbewerb aus und pollt ein globales Flag, das letztendlich von Prozessor 0 nach $\lceil log_2 P \rceil$ Runden gesetzt wird. In dieser Form ist die Barriere also nur für UMA-Systeme mit gemeinsamen Speicher ausreichend effizient. Mit Einführung eines binären Aufweckbaums kann jedoch erreicht werden, daß die Prozessoren auf lokalen Flags pollen und die Barriere für cc-NUMA-Systeme effizient macht. Letztendlich führt das zu einem Spezialfall der Combining-Barrier mit einem Fan-In von 2 und statisch bestimmten Gewinnern an jedem Knoten.

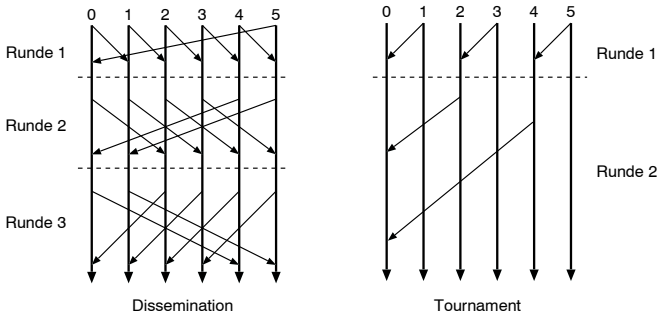


Abbildung 4.4: Kommunikationsstruktur der Dissemination und Tournament Barriere

Adaptive kombinierende Baumbarriere In [22] werden optimierte Varianten der adaptiven, kombinierenden Barriere vorgestellt. Die Optimierung hat folgende Aspekte:

- Minimierung der Zugriffe auf nicht-lokalen Speicher
- Möglichkeit, die Wartezeit aktiv zu nutzen
- Reduzierung der Kontextwechsel auf Prozessoren mit mehreren Threads

Scott und Mellor-Crummey haben diesen Algorithmus für ein System mit gemeinsamem Speicher implementiert; eine Umsetzung für verteilten Speicher ist jedoch möglich und wird in Kapitel 4.3.3 erläutert. Der zugrundeliegende Algorithmus von Gupta und Hill [11] baut auf einem doppelt verketteten binären Baum auf, in dem für jeden Prozessor genau ein beliebiges Blatt als Startposition vorgesehen ist. Die Struktur des Baumes ist jedoch nicht statisch, sondern wird von Prozessoren, die in die Barriere eintreten, so verändert, daß nachfolgende Prozessoren auf der Suche nach ihrer Warteposition weniger Ebenen durchlaufen müssen.

Unter Verwendung der Namensgebung aus Abbildung 4.5 läuft dieser Vorgang im Detail folgendermaßen ab: jeder Prozessor steigt von seinem Blatt n solange den Baum hinauf,

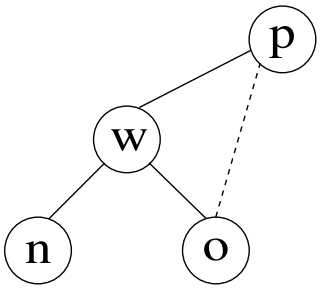


Abbildung 4.5: Namensgebung der Knoten bei der adaptiven Baumbarriere

bis er einen Knoten findet, der noch von keinem anderen Prozessor besetzt ist. Diesen Knoten w markiert er als besetzt und hängt sodann den Knoten o als neuen Sohn an den Knoten p. Tatsächlich muß bei diesem Vorgehen jeder Prozessor nur einen Schritt im Baum zurücklegen, um einen freien Knoten zu finden. Der letzte Prozessor, der an der Barriere ankommt, findet als Vater seines Blattes den NIL-Knoten, wodurch er weiß, daß die Barriere komplett ist. Daher setzt er in der Wurzel des Baumes das entsprechende Flag. Der zugehörige Prozessor pollt dieses Flag und setzt daraufhin bei seinen Söhnen ebenfalls diese Flags, wodurch sukzessiv alle Prozessoren erreicht werden. Dies setzt vorraus, daß die ursprüngliche, nicht adaptierte Struktur des Baumes gesichert wurde, da der adaptierte Baum kein regulärer binärer Baum mehr ist und das Rücktraversieren darin nicht mehr möglich ist.

Eine Optimierung betrifft die Möglichkeit, daß ein Prozessor während der Wartezeit an der Barriere andere Dinge rechnen kann[4] anstatt nutzlos zu warten. Eine derartige Barriere wird als *fuzzy Barrier* bezeichnet. Dazu müssen die Phasen des Eintretens in die Barriere und des Verlassens getrennt werden. Ein einfaches Trennen der Phasen ohne weitere Modifizierung des Benachrichtigungsalgorithmus hätte jedoch zur Folge, daß Prozessoren, die relativ früh in die Barriere eingetreten sind und daher einen Knoten in der Nähe des unteren Randes des Baumes haben, beim Wiedereintreten in die Barriere u.U. lange auf die Prozessoren aus dem oberen Bereich des Baumes nahe der Wurzel warten müssen. Es sollte also erreicht werden, daß die Prozessoren, die als erste in die Barriere eingetreten sind, auch als erste benachrichtigt werden, wenn die Barriere komplett ist. Dies ist besonders wichtig, da ja kein Prozess bei seinem Eintreffen an der Barriere abschätzen kann, wann der letzte Prozess die Barriere erreichen wird. Entsprechend kann es vorkommen, daß sich ein Prozeß zu lange mit seinen Nebenrechnungen aufhält und damit andere Prozesse in der Barriere blockieren würde. Als Lösung wird ein separater Benachrichtigungsbaum auf der Basis des ursprünglichen binären Baumes benutzt, in dem ein Prozess so lange in Richtung der Wurzel wandert wie es freie Knoten gibt. Damit ist in diesem Fall ein Knoten gemeint, auf dem nicht bereits ein anderer Prozess wartet.

Eine weitere Optimierung widmet sich der Minimierung der nichtlokalen Speicherzugriffe. Die nicht-fuzzy Variante der Barriere pollt bislang bei drei verschiedenen Gelegenheiten auf nicht-lokalen Speicherstellen:

[4]Diese Dinge dürfen natürlich nichts mit dem Ergebnis zu tun haben, das nach der Barriere allen Prozessoren vorliegt.

1. Während des Wartens auf die Vervollständigung der Barriere pollt jeder Prozess auf einem Flag eines dynamisch gesuchten Knotens
2. Um den Baum konsistent zu halten, wird jeder besuchte Knoten über `test_and_set` abgefragt
3. Bei der Suche nach dem Knoten, bei dem der Prozess warten kann, und dem anschließenden Ermitteln des Zwillingsknotens

Das Problem unter Punkt 1 kann gelöst werden, indem jeder Prozess ein lokales Flag anlegt und in dem Knoten des Baumes ein Zeiger darauf abgelegt. Die nichtlokalen Speicherzugriffe unter Punkt 2 und 3 können mittels Ersetzung der Locks durch `fetch_and_store` Operationen reduziert werden, wobei der Einsatz von `compare_and_swap` an dieser Stelle zusätzlich das Zurückschreiben eines Pointers, der fälschlicherweise überschrieben wurde, überflüssig machen.

Bei der fuzzy Variante der Barriere mit minimierter Zahl der nichtlokalen Speicherzugriffe kommt neben den oben angegebenen Maßnahmen noch hinzu, daß beim Eintritt in die Beendigungsphase der Barriere der Warteplatz nahe der Wurzel gefunden werden soll, ohne daß jeder Zugriff auf einen Knoten von einem Lock gesichert werden muß. In diesem Fall handelt es sich sogar um zwei Locks, nämlich den auf den Sohnknoten und den zugehörigen Vaterknoten, weil der Prozess nachsieht, ob der Vater bereits belegt ist. Ist das der Fall, belegt er wiederum den Sohn. Dieser zweifache Locken kann umgangen werden, indem auf dem Weg vom Blatt bis zum tatsächlichen Warteknoten einfach *jeder* Knoten, der dem Prozess unbelegt erscheint, mit dem Zeiger auf das eigene Flag versehen wird.

Eine letzte Optimierung betrifft die Methode zum Aufwecken. Wenn ein Prozess entdeckt, daß in einem Subbaum bereits ein Prozeß aktiv ist oder die Wurzel des Subbaums unbesetzt ist, bricht er den Aufweckprozeß für diesen Subbaum ab.

Statische f-fach Barriere In [9] wird als Variaton des Tournament-Algorithmus die statische f-fach Barriere vorgestellt. Sie benutzt nicht den festen Fan-In von 2, sondern bestimmt für jede Stufe individuell den optimalen Fan-In über den in Algorithmus 4.7 dargestellten Algorithmus. Dabei wird der Baum der f-fach Barriere im Gegensatz zur MCS-Barriere von den Blättern zur Wurzel hin aufgebaut (bottom-up). Während dadurch der Baum der MCS-Barriere nur dann ausgeglichen ist, wenn die Zahl der teilnehmenden Prozessoren P eine Potenz von 2 ist, ist dies bei der f-fach Barriere bereits der Fall, wenn P ein Vielfaches des Fan-In ist.

Dynamische f-fach Barriere Die im vorhergehenden Absatz beschriebene Barriere wird als statisch bezeichnet, weil die Rolle jedes Prozessors bei der Erstellung des Baumes festgelegt wird. Wenn ein designierter Vaterprozeß vor seinem Sohnprozeß an der Barriere ankommt, muß er die Ankunft seiner Söhne pollen. Dies führt zu Fehlzugriffen in den Cache beim Vater, wenn die Söhne ihre Ankunft in der zugeordneten Speicherstelle eintragen. Um diesen Schwachpunkt der statischen f-fach Barriere zu beheben, wird sie

Algorithmus 4.7 Bestimmung des statischen Fan-In der f-fach Barriere

```
/* wg. Effizienz: MAX_FANIN = Zahl der Bytes in Maschinenwort */
const MAX_FANIN = 8;
const MIN_FANIN = 2;
float measure, best_measure;

int fanin (int num_procs_left) {
   best_fanin = MAX_FANIN;
   current_fanin = best_fanin;
   best_measure = 0;

   while (current_fanin > MIN_FANIN) {
      remainder = num_procs_left % current_fanin; /* Modulo */
      if (remainder == 0)
         remainder = current_fanin;

      /* 0 < measure <= 1, wobei 1 ausgeglichenen Baum ergibt */
      measure = remainder / current_fanin;

      if (measure > best_measure) {
         best_measure = measure;
         best_fanin = current_fanin;
      }

      current_fanin -= 1;
   }

   return (best_fanin);
}
```

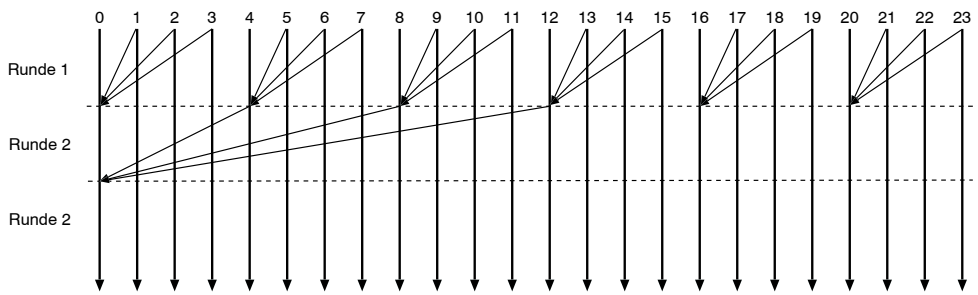


Abbildung 4.6: Kommunikationstruktur der f-fach Barriere

modifiziert, um die Vater-Sohn Rollenverteilung dynamisch optimal festzulegen [9]. Wie bei Combining-Tree-Barriere wird dem zuletzt eintreffenden Prozeß die Rolle des Vaters gegeben. Die Kommunikationsstruktur der statischen f-fach Barriere bleibt dabei erhalten. Jedoch muß statt des Flags (mit statischer Bedeutung) mit sense-reversing gearbeitet werden, um das Auftreten von Race-Conditions zu vermeiden.

4.3.3 Barrieren für NORMA-Systeme

MCS Baum Barriere Um mit der MCS-Barriere von Mellor-Crummey und Scott [21] P Prozessoren zu synchronisieren, werden zwei Bäume mit jeweils P Knoten verwendet. Ein Baum ist der Ankunftsbaum, der andere dient als Aufweckbaum, wobei die Bäume verschieden Fan-Ins bzw. Fan-Outs haben. Als Fan-In für den Ankunftsbaum wird 4 verwendet, was etwa auf 32-bit-Maschinen wegen der Zusammenfassung von 4 Statusbytes zu einem Maschinenwort günstig ist, da somit der Informationsgehalt einer Nachricht maximiert werden kann. Für den Aufweckbaum gilt, daß bei einem Fan-Out von k der letzte Prozessor nach $k\ log_k P$ Aufweckvorgängen erreicht wird. Dieser Ausdruck wird minimal für $k = 2$. Die Verknüpfung der Knoten erfolgt über Zeiger auf das jeweils zuständige Flag `child_not_ready` und `parent_sense`. Der Algorithmus der MCS-Barriere ist aus Abbildung 4.9 ersichtlich. Die in dem Algorithmus auftretenden Pointeroperationen auf ein gemeinsames Feld lassen sich NORMA-gerecht in Nachrichten umformen.

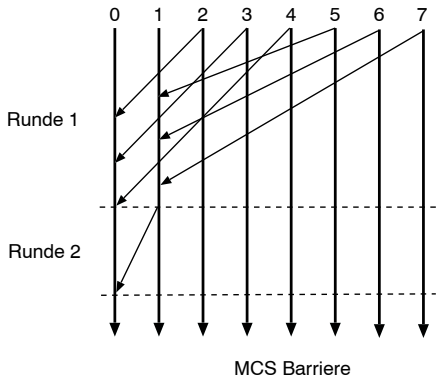


Abbildung 4.7: Kommunikationstruktur der MCS-Barriere

Adaptive Combining Tree-Barrier Die adaptive Baumbarriere wurde bereits in Kapitel 4.3.2 vorgestellt, unter anderem eine Variante, die die Zahl der Zugriffe auf nichtlokalen Speicher so weit als möglich reduziert. Dies ist Vorraussetzung dafür, daß sich eine Barriere auf einem netzwerkbasierten System effizient realisieren läßt, wenn diese Zugriffe jeweils einer Nachricht an einen entfernten Prozessor zugeordnet werden. Wenn es auf einem gegebenen System die Weglänge einer Nachricht einen Einfluß auf die Kommuniktionsleistung hat, sollte versucht werden, im Netzwerk benachbarte Knoten auch im Baum miteinander zu verbinden.

Algorithmus 4.8 Datentstruktur und Rendezvous der dynamischen f-fach Barriere

```
typedef union {
   long whole;
   boolean parts[MAX_FANIN];
} whole_and_parts;

typedef struct treenode_t{
   /* Zeiger auf das Flag in dem zu pollenden Flagwort */
   boolean *myPlace;
   struct treenode_t *parent;   /* Zeiger auf den Vater */

   /* Soehne, die bislang angekommen sind */
   whole_and_parts childSoFar;
   /* Soehne, die noch erwartet werden */
   whole_and_parts childExpected[2];

   unsigned char level;
   type_enum type;
} treenode_t;

void barrier()
{
   register treenode_t *node = myleaf;

   *myStartingPlace = local_sense;
   for(;;) {
      if (node->childSoFar.whole !=
          node->childExpected[local_sense].whole) {
         while (global_sense != local_sense);
         break;
      }
      if (node->type == ROOT) {  /* an der Wurzel angelangt */
         global_sense = local_sense;       /* signalisieren */
         break;
      }
      *node->myPlace = local_sense;
      node = node->parent;
   }
   local_sense ^= TRUE;  /* sense-reversal */
}
```

Algorithmus 4.9 Algorithmus für die MCS-Baumbarriere

```
typedef struct TREE_NODE {
   bool  parent_sense;
   bool *parent_ptr;
   bool *child_ptr[2];
   bool  have_child[4];
   bool  child_not_ready [4];
   bool  dummy      /* pseudo-data */
} tree_node;

shared tree_node nodes[P]; /* nodes[vpid] lokal fuer Prozessor i */
private int vpid;          /* virtuelle, aber eindeutige Proz.-ID */
private bool sense;

/* Initialwerte der Datenstrukturen:
  sense ist true auf Prozessor i und nodes[i] ist:
  have_child[j] = true wenn 4*i+j < P, sonst false
  parent_ptr = &nodes[floor((i-1)/4)].child_not_ready[(i-1) mod 4]
               oder &dummy falls i = 0
  child_ptr[0] = &nodes[2*i+1].parent_sense
                 oder =&dummy falls 2*i+1 >= P
  child_ptr[1] = &nodes[2*i+2].parent_sense
                 oder =&dummy falls 2*i+2 >= P
  child_not_ready = have_child
  parent_sense = false */

void tree_barrier (void) {
   /* warten auf die Soehne */
   while (nodes[vpid].child_not_ready[0]
          || nodes[vpid].child_not_ready[1]
          || nodes[vpid].child_not_ready[2]
          || nodes[vpid].child_not_ready[3]) {}
   /* Kopieren der Feldinhalte (Kurzfassung" fuer naechsten Lauf */
   nodes[vpid].child_not_ready = nodes[vpid].have_child;
   /* dem Vater Bescheid geben, dass der Sohn angekommen ist */
   *parent_ptr = FALSE;

   /* Alle ausser root warten auf Aufwecken durch den Vater */
   if (vpid) { while (nodes[vpid].parent_sense != sense) {} }

   /* Soehne aufwecken (via sense-reversal) */
   *nodes[vpid].child_prt[0] = sense;
   *nodes[vpid].child_prt[1] = sense;
   sense = !sense;
}
```

4.3.4 Barrieren für mehrere Threads auf einem Prozessor

Neben der Synchronisation einer Zahl von Prozessoren auf einem Parallelrechner ist auch bei der Ausführung mehrerer Threads eines Prozesses auf einem einzelnen Prozessor (*Multithreading*) eine Synchronisation erforderlich, wenn die einzelnen Threads Teile eines Gesamtproblems bearbeiten. Im Gegensatz zu den Situationen in den vorherigen Abschnitten besteht hier aber nicht die Wahl zwischen blockierenden und pollenden Algorithmen, da ein pollender Thread die Rechenzeit aller anderen Threads unnötig beschränkt. Hingegen taucht das Problem des *Hotspots* nicht auf, da alle Threads zu gleichen Kosten auf den lokalen Speicher zugreifen könnnen und ohnehin sequentiell bearbeitet werden. Die Verwendung von Signalen oder Bedingungsvariablen ist daher vorteilhaft, da diese Art der Kommunikation auf einem Prozessor nicht blockierend ist und nur wenige Kosten verursacht. Darüberhinaus bieten die verfügbaren Programmierschnittstellen für Threads Funktionen zur Synchronisation an.

In den folgenden Abschnitten werden einige Verfahren vorgestellt und bewertet. In den zugehörigen Beispielen wird das pThreads-API verwendet, definiert im POSIX Standard 1003.1c (Kapitel 3.4).

Fork & Join Die einfachste Methode, Threads zu synchronisieren ist die Verwendung des Fork & Join-Verfahrens. Der Prozeß erzeugt die erforderliche Anzahl von Threads, die sogleich in die ihnen zugewiesene Funktion springen und diese abarbeiten (siehe Algorithmus 4.10). Danach werden die Threads vernichtet, was die Methode wegen des anfallenden Overheads zumeist uninteressant macht. In [19] wird Fork & Join jedoch mit speziellen Threads verwendet, bei denen der Aufwand zur Erzeugung und Vernichtung vermieden wird. Ein Beispiel, wie derartige Threads in den Algorithmus der adaptiven Quadratur zur Berechnung der Fläche unter einer Kurve der Funktion f eingesetzt werden können, ist in Algorithmus 4.11 dargestellt.

FIFO-Lock Barrier Eine erste, sehr einfache Barriere, bei der es nicht nötig ist, für jeden Barrierendurchlauf die zugeordneten Threads neu zu starten, läßt sich über einen FIFO-Lock wie Ticket Lock (Kapitel 4.2.1) realisieren. Dazu wird ein Thread zum Schlüsselthread bestimmt, der den Mutex gleich zu Beginn belegt. Die Barriere besteht nun einfach aus dem Versuch eines jeden Thread, diesen Mutex zu belegen. Ein Zähler verhindert, daß der Schlüsselthread den Lock zu früh abgibt oder wieder erhält, wodurch ein Deadlock oder zumindest inkorrektes Verhalten der Barriere entsteht.

Eine solche Barriere ist in einer Multithreading-Umgebung aufgrund ihres pollenden Charakters nur bei sehr kurzen Wartezeiten an der Barriere effizient, aber es ist interessant zu ermitteln, wie sie sich ihre Leistung gegenüber fortgeschritteneren Barrierentypen ausnimmt.

Zentralbarriere Die Barriere von Marejka (siehe Algorithmus 4.12) arbeitet mit Bedingungsvariablen, Mutex Locks und einem zentralen Zähler. Sie hat gegenüber der einfachen Mutex Lock Barriere den Vorteil, daß kein ausgezeichneter Thread benötigt wird und auch die Mutex Locks nicht gepollt werden.

Algorithmus 4.10 Synchronisation mittels Fork & Join unter POSIX

```
void * j_thread( void *arg ) {
   /* Erledigung der Aufgabe */
   do_something();

   return( 0 );
}

main()
{
   /* Threads erzeugen und mit Funktion j_thread() starten */
   for ( j = 0; j < number_threads; ++j )
      if ( n = pthread_create( &thread_id[j], NULL, j_thread,
           (void *)NULL )) {
         errno   = n;
         perror( "pthread_create" );
         exit( 1 );
      }

   /* Auf die Rueckkehr eines jeden Threads warten
      (hier koennte man  anhand der Thread ID's
      auch nur auf bestimmte Threads warten) */
   for ( j = 0; j < number_threads; ++j )
      pthread_join( thread_id[j], NULL );
}
```

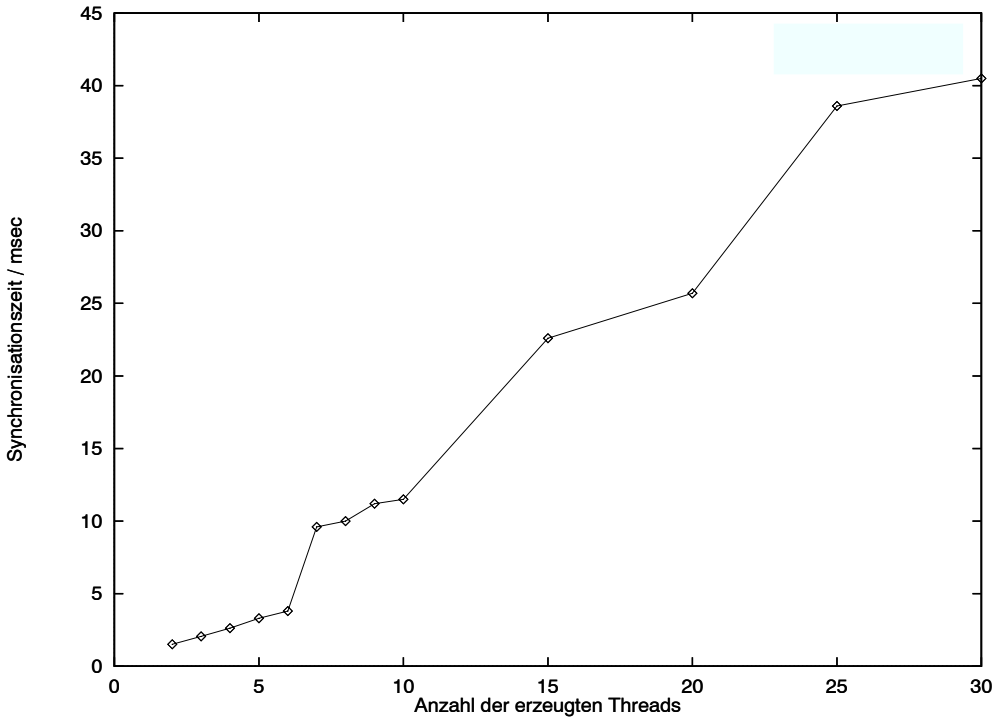


Abbildung 4.8: Threadsynchronisation über POSIX Fork & Join
Messungen auf Sun Sparcstation 20

Algorithmus 4.11 Adaptive Quadratur mit Fork & Join-Synchronisation (Pseudocode, nach [19])

```
main()
{
   real links, rechts, *antwort, flinks, frechts, start_flaeche

   Eingabe von left und right
   berechne flinks, frechts und start_flaeche
   antwort = thread_fork (quad, links, rechts, flinks, frechts,
                          start_flaeche)
}

quad (real a, b, fa, fb, area)
{
   real *links, *rechts, fm, m, alinks, arechts

   berechne Mittelpunkt m und flaechen alinks und arechts
   if (genau genug)
      return (alinks + arechts)
   else {
      /* starte 2 neue Threads mit Unterprogramm (rekursiv) */
      links = thread_fork (quad, a, m, fa, fm, alinks)
      rechts = thread_fork (quad, m, b, fm, fb, arechts)
      /* warte auf Rueckkehr */
      thread_join()
      return (*links, *rechts)
}
```

Ein Problem kann es sein, wenn ein Thread vom Scheduler suspendiert wird, während er den Lock hält: andere Threads werden in dieser Zeit auf den Lock warten ohne eine Chance, ihn zu erhalten. Dadurch wird Zeit vergeudet, ein Deadlock kann dadurch allerdings nicht entstehen. Die Verwendung des Mutex Locks kann vermieden werden, wenn mit geeigneten atomaren Operationen auf den Zähler zugegriffen wird. Da solche Operationen aber meist Maschinenspezifisch sind, geht mit einer derartigen Maßnahme die Portierbarkeit auf POSIX-Ebene verloren.

Diese Barriere stellt eine Variante der typischen Barriere auf multiprogrammierten Prozessoren dar. Je nach gegebenen Mitteln zur Synchronisation und deren Leistungsfähigkeit sowie der Art der Nutzung der Barriere kann der Algorithmus bei gleichem Grundprinzip optimiert werden.

4.3.5 Hardware-Unterstützung für Barrieren

Neben der in den vorrangegangenen Kapiteln besprochenen softwaretechnischen Realisierung von Barrieren ist auch die Nutzung von spezieller Hardware möglich. Rein logisch ist eine Barriere eine einfache UND-Verknüpfung, die sich in Hardware theoretisch einfach realisieren läßt. In der Praxis ergibt sich aber durch die zusätzlich nötigen Signale und den Wunsch nach Skalierbarkeit ein höherer Aufwand, der gegen den tatsächlichen Nutzen aufzuwiegen ist.

Cray T3D vs. Cray T3E In dem Parallelrechner T3D hat Cray zur Synchronisation ein eigenes baumstrukturiertes Netzwerk parallel zum eigentlichen Kommunikationsnetzwerk realisiert [24]. Über dieses Netzwerk ist sowohl Barriersynchronisation (entspricht dem logischen UND) als auch Eurekasynchronisation (entspricht dem logischen ODER) möglich. Das Netzwerk wurde als Baum von UND-Gattern mit einem Fan-In von 4 realisiert, da sich eine UND-Funktion für sämtliche Prozessoren des Systems (maximal 1024) keinesfalls in einer einzigen Stufe realisieren läßt und ein solches Gatter auch nicht skalierbar und partitionierbar wäre. Das derart erzeugte Barriersignal wird ebenfalls über einen Baum mit einem Fan-Out von 4 an die Prozessoren zurückgeleitet. Da eine einzige Hardwarebarriere nicht ausreicht, wenn das System in mehrere unabhängige Partitionen aufgeteilt wird, wurden insgesamt 16 parallele Barrieren implementiert[5].

Bei Verwendung von negativer Logik kann die gleiche Hardware auch zur Eurekasynchronisation genutzt werden. Dazu ist allerdings eine Barriere vor und nach dieser Synchronisation erforderlich, da die Eurekasynchronisation nicht synchron ist und somit Raceconditions auftreten könnten.

Bei dem Nachfolgemodell der T3D, der T3E, hat Cray sämtliche Funktionen zur Kommunikation in ein einziges Netzwerk eingebettet; das separate Synchronisationsnetzwerk der T3D wurde weggelassen [25]. Dadurch wurde die grobe Granularität der T3D bezüglich der Partitionierbarkeit als auch der Aufrüstbarkeit überwunden sowie der Leitungsbedarf verringert. Auf jeder Prozessoreinheit (PE) wurde eine Schaltung für Barrier- und

[5] Zur Einsparung von Leitungen wurden sie aber nicht hardwaremäßig voll parallel implementiert, sondern über 4 Leitungen zeitlich gemultiplext

Algorithmus 4.12 Zentralbarriere nach Marejka

```
typedef struct {
   int maxcnt;  /* maximum number of runners    */
   struct _sb {
      pthread_cond_t  wait_cv;  /* cv for waiters at barrier    */
      pthread_mutex_t wait_lk;  /* mutex for waiters at barrier */
      int runners;              /* number of running threads    */
   } sb[2];
   struct _sb *sbp;              /* current sub-barrier */
} pbarrier_t;

int pbarrier_init( pbarrier_t *bp, int count, int type, void *arg )
{
   bp->maxcnt = count;
   bp->sbp    = &bp->sb[0];

   for ( i = 0; i < 2; ++i ) {
      struct _sb *sbp = &(bp->sb[i]);
      sbp->runners = count;

      if (n = pthread_mutex_init( &sbp->wait_lk, NULL ))
         return (n);
      if (n = pthread_cond_init( &sbp->wait_cv, NULL ))
         return (n);
   }
   return (0);
}

int pbarrier_wait (register pbarrier_t *bp) {
   register struct _sb *sbp = bp->sbp;
   register int count = bp->maxcnt;

   pthread_mutex_lock( &sbp->wait_lk );
   if ( sbp->runners == 1 ) {   /* der letzte Thread ist da ! */
      if ( bp->maxcnt != 1 ) {
         sbp->runners = bp->maxcnt; /* Reinitialisierung */
         bp->sbp = (bp->sbp == &bp->sb[0])
                   ? &bp->sb[1] : &bp->sb[0];

         pthread_cond_broadcast( &sbp->wait_cv );  /* Aufwecken */
      }
   } else {
      sbp->runners--; /* einer weniger, auf den zu warten ist*/
      while (sbp->runners != bp->maxcnt)
         pthread_cond_wait( &sbp->wait_cv, &sbp->wait_lk );
   }
   pthread_mutex_unlock(&sbp->wait_lk);
}
```

Eurekasynchronisation (BES) implementiert. Diese Schaltung arbeitet unabhängig vom Prozessor, wickelt ihre Kommunikation, die für die Synchronisation anfällt, aber über das normale Netzwerk ab. Diese Synchronisationsnachrichten haben jedoch die höchste Priorität [35]. Da hier ein größerer Teil der Synchronisation in Software abgewickelt wird, muß eine PE bzw. die darauf befindliche BES als Wurzel der Barriere bestimmt werden. Der Prozessor auf der PE kann das Erreichen der Barriere wahlweise pollend oder über interruptgesteuert in Erfahrung bringen.

Während in der T3D das Kommunikationsnetzwerk zur Synchronisation überhaupt nicht benutzt werden muß, treten dazu bei der T3E eine Anzahl von Nachrichten auf dem Netzwerk auf. Die genaue Zahl hängt von der Struktur der Partion ab, die in die Synchronisation involviert ist. Die PE der Partition bilden einen Baum, in dem jedoch ein Fan-In im Bereich von 0 bis 6 auftreten kann. Man kann also nur von O(log P) sprechen. In der T3D dauert eine Barriersynchronisation aufgrund der Schaltzeiten der Gatter im Schnitt 24 Taktzyklen. Die Schwankungen dieser Zeit werden durch die gemeinsame Nutzung von nur 4 Leitungen für 16 Barrieren verursacht.

Wie sich diese unterschiedlichen Ansätze auf die Leistung auswirken, wurde durch Messungen ermittelt (siehe Abbildung 4.9). Es wird deutlich, daß die Zeit für Barrier-Synchronisation auf der T3D auch mit steigender Knotenzahl konstant bleibt, da offentsichtlich die (mit steigender Prozessorzahl logarithmisch zunehmende) Tiefe des Hardwarebaumes kaum ins Gewicht fällt gegenüber dem restlichen (konstanten) Overhead der Messung[6] Die Barriere auf der T3E skaliert hingegen mit O(log Knotenzahl), allerdings bewegen sich die Synchronisationszeiten in der gleichen Größenordnung wie bei dem Vorgängermodell. Die Tatsache, daß bei dem synthetischen Benchmark die Busbelastung nur aus den Synchronisationsnachrichten besteht, ist dabei keine wesentliche Einschränkung, da auch bei realen Anwendungen die eigentliche Kommunikation erst nach der Barriere durchgeführt wird und die Synchronisationsnachrichten auch im realen Betrieb die höchste Priorität haben. Insofern erscheint die Entscheidung von Cray, den Aufwand des separaten Synchronisationsnetzwerkes einzusparen, vertretbar.

SGI Power Challenge Silicon Graphics hat in seiner Power Challenge UMA Architektur auf jeder Prozessoreinheit einen Synchronisationszähler implementiert [26]. Durch ein Broadcast-Signal auf dem Bus, das von jedem Prozessor beim Erreichen der Barriere ausgegeben wird, wird dieser Zähler inkrementiert. Der an der Barriere angekommene Prozessor pollt sodann diesen (lokalen) Zähler, bis dieser die Zahl der involvierten Prozessoren anzeigt, womit die Barriere komplett ist. Dadurch ruft auf diesem System eine Synchronisation an einer Barriere nur N Buszugriffe hervor gegenüber N^2 Zugriffen bei Verwendung eines zentralen Zählers und kohärenten Caches. Allerdings kann tatsächlich nur eine einzige Barriere auf dem gesamten System auf diese Art genutzt werden. Das heißt, das das System zur Verwendung dieser Barriere von einer Applikation exklusiv genutzt werden muß.

[6] Das zur Messung verfügbare System hatte nur 32 Knoten.

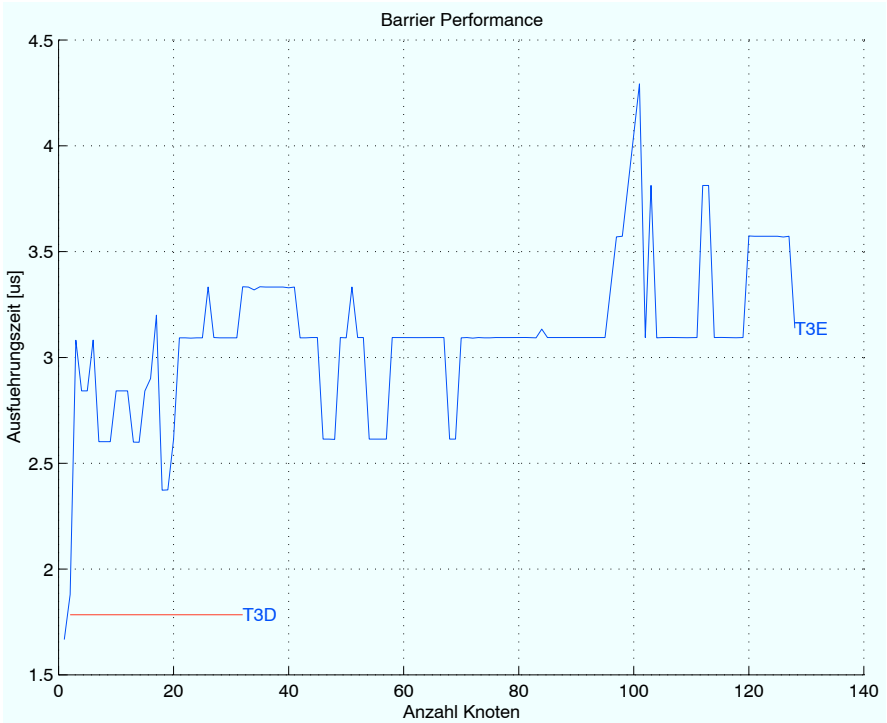


Abbildung 4.9: Barrier-Synchronisation auf Cray T3D und T3E

4.3.6 Vergleich der behandelten Barrieren

In diesem Kapitel werden in den Tabellen 4.2 und Tabellen 4.3die Eigenschaften der zuvor behandelten Barrieren tabellarisch gegenübergestellt. Die betrachteten Kriterien sind:

1. **Spezialhardware:** benötigt der Algorithmus spezielle Hardware-Instruktionen, die über die atomare Operation `test_and_set` hinausgehen?
2. **Kritischer Pfad:** die Länge des kritischen Pfades
3. **Transaktionen:** die Zahl der Kommunikationsvorgänge, also Nachrichten über das Netzwerk oder Zugriffe auf nicht-lokalen Speicher
4. **Transaktionen mit Cache:** Abschätzung der effektiven Zahl der im vorherigen Punkt genannten Speicherzugriffe bei Verwendung kohärenter Caches
5. **Zahl der Schreiber:** wieviele Prozesse schreiben zur Synchronisation auf eine Speicherstelle ?
6. **Hierarch. Speicher:** ist der Algorithmus geeignet für Systeme mit hierachischem Speicher (NUMA)?
7. **Kommunikationsstrukur:** wird die Kommunikationsstruktur statisch bei der Initialisierung der Barriere oder dynamisch während ihrer Laufzeit festgelegt?

8. **Speicherbedarf:** Ordnung der Größe des Speicherbedarfs im gemeinsamen Speicher

Wenn ein Barrierenalgorithmus unter Punkt 1 eine komplexere atomare Operation benötigt, läßt er sich in der Regel denoch mit den einfachen Primitiven realisieren, da damit Locks erzeugt werden können, die jede Art von synchronisiertem Zugriff erlauben. Eine solche Lösung hat aber eine schlechtere Leistung als es mit den speziellen Instruktionen der Fall ist und machen damit u.U. den Algorithmus obsolet.

Die unter Punkt 7 aufgeführte Eigenschaft einer statischen oder dynamischen Kommunikationsstruktur ist kein eindeutiges Zeichen für mehr oder weniger Leistung: eine dem einzelnen System optimal angepaßte statische Kommunikationsstruktur[7] kann sogar weniger Overhead mit sich führen. Eine dynamische Struktur, die sich also während der Barrierensynchronisation anhand der Ankunftszeiten der Prozesse oder einem anderen Kriterium ändert, ist vor allem bei Barrieren mit unterschiedlicher Laufzeit der Prozesse vorteilhaft. Die Eigenschaft *unbestimmt* bedeutet hier, daß der Algorithmus keine Kommunikationsstruktur vorgibt.

	Fork&Join	**Mutex Lock**	**Zentral**
Spezialhardware	Nein	Nein	Vorteilhaft
Kritischer Pfad	1	1	1
Transaktionen		P	P^2
Transakt. mit Cache		P	
Zahl der Schreiber		P	
Hier. Speicher	Möglich	Nein	Nein
Komm.struktur	umbestimmt	unbestimmt	unbestimmt
Speicherbedarf	0	1	1

Tabelle 4.2: Vergleich von Algorithmen für Barrieren bei Multiprogramming

[7] Diese statische Struktur wird allerdings auch erst zur Laufzeit erstellt, ist also quasi-dynamisch

	Combining	**Dissem.**	**Tourn.**	**MCS**
Spezialhardware	Ja	Nein	Nein	Nein
Kritischer Pfad	$\lceil log_f P \rceil$	$\lceil log_2 P \rceil$	$\lceil log_2 P \rceil$	$\lceil log_f P \rceil$
Transaktionen	f$\lceil \frac{P-1}{f-1} \rceil$	P$\lceil log_2 P \rceil$	P-1	P-1
Transakt. mit Cache	f$\lceil \frac{P-1}{f-1} \rceil$	2P$\lceil log_2 P \rceil$	2(P-1)	$\lceil \frac{P-1}{f} \rceil \frac{3f-2}{2}$
Zahl der Schreiber	f	1	1	f
Hierarch. Speicher	Ja	Nein	Nein	Ja
Komm.struktur	Dynamisch	Statisch	Statisch	Statisch
Speicherbedarf	P	P$\lceil log_2 P \rceil$	$\frac{P}{2} \lceil log_2 P \rceil$	P

	Adaptiv	**Statisch f-fach**	**Dynamisch f-fach**
Spezialhardware	Nein	Nein	Nein
Kritischer Pfad	$\lceil log_2 P \rceil$	$\lceil log_f P \rceil$	$\lceil log_f P \rceil$
Transaktionen	?	P - 1	f$\lceil \frac{P-1}{f-1} \rceil$
Transakt. mit Cache	?	$\lceil \frac{P-1}{f-1} \rceil \frac{3f-2}{2}$	f$\lceil \frac{P-1}{f-1} \rceil$
Zahl der Schreiber	1	f - 1	f
Hierach. Speicher	Ja	Nein	Nein
Komm.struktur	Dynamisch	Statisch	Dynamisch
Speicherbedarf	2P	$\frac{P}{f} \lceil log_f P \rceil$	$\frac{P}{f} \lceil log_f P \rceil$

Tabelle 4.3: Vergleich von Algorithmen für Barrieren auf Multiprozessorsystemen

Kapitel 5

Einbindung in SVM-Fortran

SVM-Fortran [4] ist eine Erweiterung von Fortran 77 (inklusive CRAY-Erweiterungen), die die Programmierung paralleler Systeme erlaubt. Die Zielmaschinen sind Parallelrechner mit gemeinsamem Speicher, inbesondere virtuell gemeinsamem Speicher. Das Datenmodell, auf dem die Erweiterung basiert, ist das eines einzigen, globalen Adreßraums, auch wenn die Daten physikalisch in lokalen Speichern der Prozessoren gehalten werden. Als Programmiermodell wird *Single-Program-Multiple-Data* (SPMD) verwendet, wodurch auf allen Prozessoren dasselbe Programm ausgeführt wird.

5.1 Programmiermodell und -umgebung von SVM-Fortran

Prozessorfelder In SVM-Fortran kann der Benutzer bestimmen, auf welchen Prozessoren ein Ausführungsblock, beispielsweise die Iterationen einer parallelen Schleife, ausgeführt werden soll. Ein Hilfmittel hierzu sind *Prozessorfelder* erreicht, mit denen abstrakte Prozessoren angesprochen werden können. Diese werden schließlich auf die realen Prozessoren des parallelen Systems abgebildet. Durch die abstrakten Prozessoren ist der Parallelismus im SVM-Fortran-Programm unabhängig vom Parallelismus, der auf dem System physikalisch Verfügbar ist. Die Prozessorfelder wiederum dienen der Vereinfachung der Handhabung und der Abbildung von Datenstrukturen auf die Prozessoren, wie es bei der Verwendung von *Templates* geschieht[1].

Jedem Block im Programm ist genau ein Prozessorfeld zugeordnet, welches diesen Block ausführt. Dieses Feld wird als *Active Processor Set* (APS) dieses Blocks bezeichnet. Es können mehrere Blöcke parallel durch mehrere APS ausgeführt werden, wobei die Schnittmenge dieser APS leer sein muß.

Synchronisation Zur Synchronisation innerhalb des APS stehen alle gängen Mittel wie Barrieren, Locks und kritische Sektionen zur Verfügung (mehr zur Synchronisation im Kapitel 4).

[1] Templates dienen dem (vorbestimmten oder dynamischen) irregulären Scheduling von parallelen Schleifen zur effektiven Verarbeitung irregulärer Datenstrukturen.

Ausführungsmodus Die Ausführung eines Blocks des Programms erfolgt, bezogen auf das APS, in einem von zwei Modi:

- **exklusiver Modus:** Der Block wird nur von einem einzigen Prozessor ausgeführt (welcher nicht bestimmt ist). Die anderen Prozessoren des APS überspringen diesen Block und synchronisieren sich hinter dem exklusiven Block mit dem ausführenden Prozessor.
- **replizierter Modus:** Alle Prozessoren des APS bearbeiten den Code dieses Blockes. Jedoch kann der Kontrollfluß der beteiligten Prozessoren verschieden sein, sei es durch entsprechendes Scheduling einer Schleife oder explizite Auswertung der Prozessornummer.

Datenspeicherung Jede Variable in SVM-Fortran ist entweder als *private* (privat) oder *shared* (gemeinsam) definiert. Im ersten Fall besitzt jeder (abstrakte) Prozessor eine private Instanz dieser Variablen, auf die er exklusiven Zugriff hat. Gemeinsame Variablen können hingegen von allen Prozessoren des APS gelesen und geschrieben werden, wobei das Modell der *strengen Kohärenz* angewendet wird.

Die **SVM-Fortran Programmierumgebung** besteht aus fünf Teilen:

- **SVMF-Compiler:** Ein *source-to-source* Compiler, der das mit den SVMF-Direktiven versehene Programm in ein Fortran 77-Programm übersetzt.
- **SVMF-Laufzeitbibliothek:** Diese Bibliothek enthält alle Funktionen, die zur Allokation der Daten durch die Prozessoren, die Synchronisierung und das Scheduling der Prozessoren, Steuerung des Kontrollflusses und allen weiteren Aufgaben im Zusammenhang mit der Nutzung des virtuell gemeinsamen Speichers dienen.
- **SVM-Implementierung** der Zielarchitektur: der gemeinsame Adreßraum aller Prozessoren, der virtuell oder real sein kann, wird durch die standardisierten *System V Shared-Memory*-Schnittstelle angesprochen. Die zugehörige Funktionalität wird auf der Intel Paragon durch ASVM bereitgestellt; auf anderen Systemen wie Sun und SGI Origin existiert Hardware-mäßig gemeinsamer Speicher.

Neben diesen Komponenten, die notwendig sind, um ein SVM-Fortran-Programm ablaufen lassen zu können, existieren noch Werkzeuge zur Analyse und Visualisierung:

- **OPAL** wird zur Leistungsanalyse eingesetzt und kann quelltextbezogen Leistungsdaten darstellen. Dazu zählen neben Ausführungszeiten von bestimmbaren Regionen u.a. auch Angaben zu Seitenfehlern und deren Bediendauer.
- **PARvis** kann die Leistungsdaten des SVM-Systems grafisch anzeigen und ermöglicht so, komplexe Vorgänge im SVM-System einfacher zu überblicken.

5.2 Einsatzmöglichkeiten von Threads in SVM-Fortran

In SVM-Fortran existieren verschiedene Möglichkeiten, ein Programm zu parallelisieren:

- **parallele Schleifen** (PDO-Direktive)
  Dies ist die am häufigsten verwendete Methode zur Parallelisierung. Sie ist es aus dem Grund, weil in den Schleifen in der Regel der weitaus größte Teil der Rechenzeit eines Programms verbraucht wird und daher eine parallele Ausführung von Schleifen den größten Gewinn bringt.

- **parallele Sektionen** (PSECTION-Direktive)
  Parallele Sektionen ermöglichen es, unterschiedliche Programmabschnitte parallel abzuarbeiten. Dieser Funktionsparallelismus wird aber weniger häufig als der Datenparallelismus der PDO-Direktive verwendet.

- **replizierte Regionen** (REPLICATED_REGION-Direktive)
  Bei der Verwendung von replizierten Regionen wird identischer Code auf allen Prozessoren parallel abgearbeitet. Hier liegt meistens kein echter Parallelismus vor; dieses Konstrukt dient vielmehr dazu, private Daten der Prozessoren gleichartig zu behandeln.

Von diesen Möglichkeiten bietet sich jedoch einzig die parallele Schleife an, prozessorlokal von mehreren Threads abgearbeitet zu werden, denn es macht keinen Sinn, auf einem Prozessor den gleichen Code auf den gleichen Daten mehrmals auszuführen (bei replizierten Regionen). Es wäre zwar denkbar, bei Verwendung paralleler Sektionen die Sektionen weiter auf Threads zu verteilen. In der Praxis ist die Granularität der parallelen Sektionen jedoch so grob, daß diese vollständig auf die verfügbaren Prozessoren verteilt werden können. Falls innerhalb der Sektionen parallele Schleifen auftreten, könnnen diese natürlich auf Threads verteilt werden.

Parallele Schleifen jedoch eignen sich besonders dazu, auf Threads verteilt zu werden: es genügt, die Menge von Iterationen der Schleife, die ein Prozessor erhält, wiederum auf die Zahl der verwendeten Threads zu verteilen. Da in parallelen Schleifen auf gemeinsame Daten zugegriffen wird, kann hier das Verstecken von Kommunikationslatenzen durch Multithreading besonders wirksam werden (was in parallelen Sektionen ohne Schleifen in der Praxis nicht der Fall ist).

5.3 Spracherweiterung

Ein Ziel bei der Einbindung von Multithreading in den SVM-Fortran-Compiler war es, den dadurch entstehenden zusätzlichen Aufwand für den Programmierer so gering wie möglich zu halten. Aus dieser Perspektive wäre eine transparente Einbindung des Multithreading, also implizit bei jeder parallelisierten Schleife, ideal[2]. Allerdings gewinnt nicht

[2]Eine solche Einbindung ist durch die Compileroption `-threads` auch möglich. Sie ist jedoch primär zu Testzwecken implementiert.

jede parallele Schleife durch Multithreading an Leistung, wie die Untersuchungen in Kapitel 6 zeigen. Somit ist die Leistung eines Programms, in dem parallele Schleifen auch durch Threads lokal parallelisiert werden, durch den entstehenden Overhead häufig geringer als die Leistung des Programms, das ohne Threads arbeitet, wenn in der Schleife nur wenige oder keine Seitenfehler auftreten.

Daher wurde die Möglichkeit geschaffen, jede Schleife individuell von einer bestimmten Zahl von Threads ausführen zu lassen, wobei diese Zahl von 0 (d.h. Ausführung allein durch den initialen Thread des Prozess) bis zu einem definierbaren Maximum reicht. In Kapitel 6.2 wird gezeigt, daß es auf den untersuchten Systemen keine Beschränkung des möglichen Leistungsgewinns darstellt, wenn dieses Maximum bei 8 Threads liegt.

5.4 Erweiterung der PDO-Direktive

Die aufgeführten Überlegungen führen dazu, die Threads über eine neue Option der PDO-Direktive in parallele Schleifen einzubinden.

5.4.1 THREADS-Option

Diese Option gibt an, daß die Schleife durch Threads ausgeführt werden soll. Diese Option wurde wie folgt definiert (bezugnehmend auf die Notation in [4], Seite 18):

parloop-option **is** ...
or THREADS (*int-expr [, int-expr]*)

Dabei korrespondieren die Parameter der THREADS-Option (ganzzahlige Ausdrücke) von links nach rechts mit den Schachtelungen der Schleife von außen nach innnen für den Fall, daß eine pefekt geschachtelte, mehrdimensionale Schleife durch eine einzige PDO-Direktive parallelisiert wird. Es ist möglich, jede Dimension einer geschachtelten Schleife durch Threads bearbeiten zu lassen oder innere Dimensionen der Schleife nicht mit Threads zu parallelisieren. Letztere Möglichkeit sollte verwendet werden, wenn die Zahl der Seitenfehler den entstehenden Overhead des mehrdimensonalen Threadschedulings (wobei jeder Thread wiederum weitere Threads für die nächste Schleifenebene einsetzt) nicht rechtfertigt. Mehrdimensionale Schleifen, die in jeder Dimension mit Threads arbeiten, bringen aufgrund des vermehrt anfallenden Overheads des Thread-Schedulings häufig weniger Leistung als die gleiche Schleife, bei der nur die äußere Dimension auf Threads verteilt wurde.

5.4.2 THREAD_PRIVATE-Option

Da die Threads in einem gemeinsamen Adressraum arbeiten, besitzen sie gemeinsamen Zugriff auf alle Variablen aus dem Kontext, in dem sie eingesetzt werden. Zu diesen Variablen zählt auch die Laufvariable der Schleife. Wenn jedoch die Threads alle die gleiche

Laufvariable benutzen (thread-globale Variable), führt dies zu einem unkorrekten Programm. Daher muß die Möglichkeit gegeben sein, Variablen thread–lokal zu deklarieren, so daß jeder Thread auf seiner privaten Instanz der Variablen arbeitet. Zu diesem Zweck wurde eine weitere Option zur PDO-Direktive eingeführt, mit der sich Variablen, die in der Schleife referenziert werden, als thread-lokal deklarieren lassen:

parloop-option **is** ...
or THREAD_PRIVATE (*variable-name-list*)

Die Entscheidung darüber, welche Variablen thread-lokal zu deklarieren sind, liegt beim Programmierer. Einzig die Laufvariablen der durch Threads zu bearbeitenden Schleifenebenen werden automatisch vom Compiler thread-lokal deklariert, da es für sie immer erforderlich ist.

5.5 Compilererweiterung

Der SVM-Fortran-Compiler wurde entworfen, um Fortran77-Programme auf mehreren Prozessoren eines Parallelrechners ausführen zu können. Zum damaligen Zeitpunkt war die Verwendung von Multithreading noch nicht beabsichtigt; daher mußte diese Fähigkeit nachträglich integriert werden. So waren der Compiler und die integrierten Tracing-Vorrichtungen nicht darauf eingestellt, daß Unterprogramme entstehen können, die nicht vom Programmierer entworfen wurden, oder das Teile des Fortran-Codes von ihrer ursprünglichen Position entfernt wurden. Diese Probleme sollten möglichst integriert, also ohne Spezialbehandlungen verschiedener Fälle, behandelt werden.

5.5.1 Konzept

Damit eine Schleife überhaupt von einem Thread angesprungen und ausgeführt werden kann, ist eine definierte Einsprungadresse notwendig. Diese kann mit den Sprachmitteln von Fortran und C nicht erlangt werden, wenn es sich bei dem von den Threads auszuführenden Programmsegment nicht um ein Unterprogramm handelt. Daher ist es notwendig, daß der Compiler die von den Threads auszuführende Schleife in ein Unterprogramm (in Fortran eine `SUBROUTINE`) transformiert, dessen Adresse als funktionaler Parameter an die Laufzeitbibliothek und von dort an die Threads übergeben werden kann.

5.5.2 Datenstrukturen

Der SVM-Fortran-Compiler verarbeitet den Quelltext in Form eines abstrakten Syntaxbaumes, der von dem portablen Fortran-Parser PAFF [3] erzeugt wird. PAFF stellt auch die notwendigen Funktionen zur Bearbeitung und Erweiterung dieses Baumes, also der enthaltenen Datenstrukturen, zur Verfügung. Darüberhinaus verfügt der Compiler jedoch auch über weitere, interne Datenstrukuren. Das nachträgliche Einfügen der

Multithreading-Eigenschaften verlangte Veränderungen an vielen Stellen im Quelltext, an denen aber nicht immer die notwendigen Informationen zur Einfügung der Threads zur Verfügung standen. Daher wurde eine zusätzliche Datenstruktur eingeführt, die all diese Informationen bereitstellt. Somit war es ausreichend, diese Struktur an alle Unterprogramme zu übergeben, die mit der Einführung von Threads zu tun haben. Diese Datenstruktur ist in Quelltext 5.1 dargestellt. Von dieser Datenstruktur wird für jede der maximal 7 zulässigen Schachtelungsebenen eine Instanz angelegt, wobei sich die Datenstruktur durch die interne Verzeigerung auch problemlos für eine andere, flexible Zahl von Schachtelungsebenen einsetzen ließe. In diese Instanzen werden während des Verarbeitung der PDO-Direktive die gewonnenen Informationen eingetragen.

Erklärungsbedürftig sind an dieser Stelle die Verweise auf die Schleife der nächsten Ebene sowie die Querverweise bezüglich der Reduktionsvariablen. Erstere sind Zeiger auf die Zeiger, welche im Verlauf des prozessorglobalen Schedulings auf den DO- und ENDDO-Knoten der betreffenden Schleife weisen. Diese Verweise müssen aber geändert werden, wenn die betreffende Schleife von ihrer ursprünglichen Stelle im Code in ein Unterprogramm verlagert wird und dabei neue DO- und ENDDO-Knoten erhält. Die Querverweise sind notwendig, um den Zusammenhang zwischen den Originalvariablen, die reduziert werden sollen, und den zu diesem Zweck notwendigerweise eingeführten Hilfsvariablen zu erhalten (siehe auch Kapitel 5.5.5).

Quelltext 5.1 Datenstruktur `thread_pdo_t`

```
typedef struct Thread_PDO_t {
  /* Ausdruck zur Zahl der Threads */
  Node *nbr_threads;
  /* Scheduling-Strategie **/
  scheduling_strategy_t strategy;

  /** Informationen zum mehrdimensionalen Scheduling **/
  int scheduling_dim;    /* Zahl der Scheduling-Ebenen */
  int current_dim;       /* Akutelle Scheduling-Ebene  */
  Node **do_n,**enddo_n; /* Verweis auf die Schleife der
                            naechsten Ebene */

  /** Informationen zur Reduktion **/
  Symbol_Table thrdloc_symtab;  /* Thread-lokale Variablen */
  Sym_crossref
    *origSym_Xref,         /* Querverweis zwischen den Original- */
    *redSym_Xref;             /* und Reduktionsvariablen        */
  Symbol_Table *redvar_symtab;  /* Original Reduktions-Variablen */
  int nbr_redvars;              /* Zahl der Reduktions-Variablen */

  /* Verknuepfung zu den anderen Scheduling-Ebenen */
  struct Thread_PDO_t *prev, *next;
} thread_pdo_t;
```

5.5.3 Erzeugung des Unterprogramms

Die notwendigen Schritte zur Erzeugung des Unterprogramms sind in Abbildung 5.1 dargestellt. Grundsätzlich stellt der Vorgang eine transparente Erweiterung des PDO-Scheduling dar, der im Anschluß an die Verteilung der Iterationen der Schleife auf die verwendeten Prozessoren erfolgt.

5.5.4 Verarbeitung der Parameter

Alle Variablen, auf die in der nunmehr in ein Unterprogramm transformierten Schleife zugegriffen wird und die nicht thread-lokal sind, müssen als Parameter an das Unterprogramm übergeben werden. Dazu kommmen noch die Iterationsgrenzen und ggf. die Schrittweite der Schleife, was zu weiteren Variablen innerhalb des Unterprogramms führt. Die Punkte, die daher bei der Parameterübergabe beachtet werden mußten, sind im folgenden erläutert; eine Übersicht der Vorgehensweise wird in Abbildung 5.2 anhand eines Beispiels gegeben.

- **indirekter Aufruf**
  Der Aufruf des Unterprogramms erfolgt nicht direkt, also von Fortran nach Fortran, sondern indirekt über einen Aufruf der Funktion `SVM_SCHEDULE_THREADS()` der SVM-Laufzeitbibliothek (siehe Kapitel 5.6). Daher müssen sämtliche Parameter zunächst an diese Funktion übergeben werden.

- **call-by-reference Parameter**
  Grundsätzlich werden in Fortran, im Gegensatz zu C, alle Parameter an Unterprogramme als Zeiger übergeben; call-by-value Übergabe ist nicht möglich. In diesem Fall stellt es aber kein Problem dar, da sich die Semantik des Programms durch das Multithreading ja auch nicht ändern soll. D.h., Variablen, deren Werte vorher in der Schleife verändert wurden, werden im Unterprogramm genauso geändert (da sie mittels Zeigern übergeben wurden). Eine Ausnahme bilden die thread–lokalen Variablen, die nicht mit übergeben werden dürfen.

- **variable Zahl von Parametern**
  Da der indirekte Aufruf nur eine feste Anzahl von Übergabeparametern erlaubt, tatsächlich jedoch eine variable Zahl von Parametern erforderlich ist, wird ein Zeiger auf ein Feld von Zeigern auf die eigentlichen Fortran-Parameter übergeben. Ein weiterer Parameter an die Laufzeitfunktion gibt deren Anzahl an. Die Threads rufen über einen mitübergebenen funktionalen Parameter das Fortran-Unterprogramm mit der aktuellen Anzahl von Parametern auf (Abbildung 5.2). Dazu wurde eine Anzahl von C-Prototypen definiert. Somit entsteht aber auch eine obere Grenze der Zahl der thread-globalen Variablen, die auf 20 festgelegt wurde[3].

- **Übergabe nur teilweise transparent**
  Der Großteil der Fortran-Parameter, die über die Laufzeitbibliothek übergeben werden, ist für diese transparent, d.h. es werden nur die Zeiger in der erforderlichen

[3] Bei Überschreitung dieser Grenze gibt der Compiler einen Fehler aus.

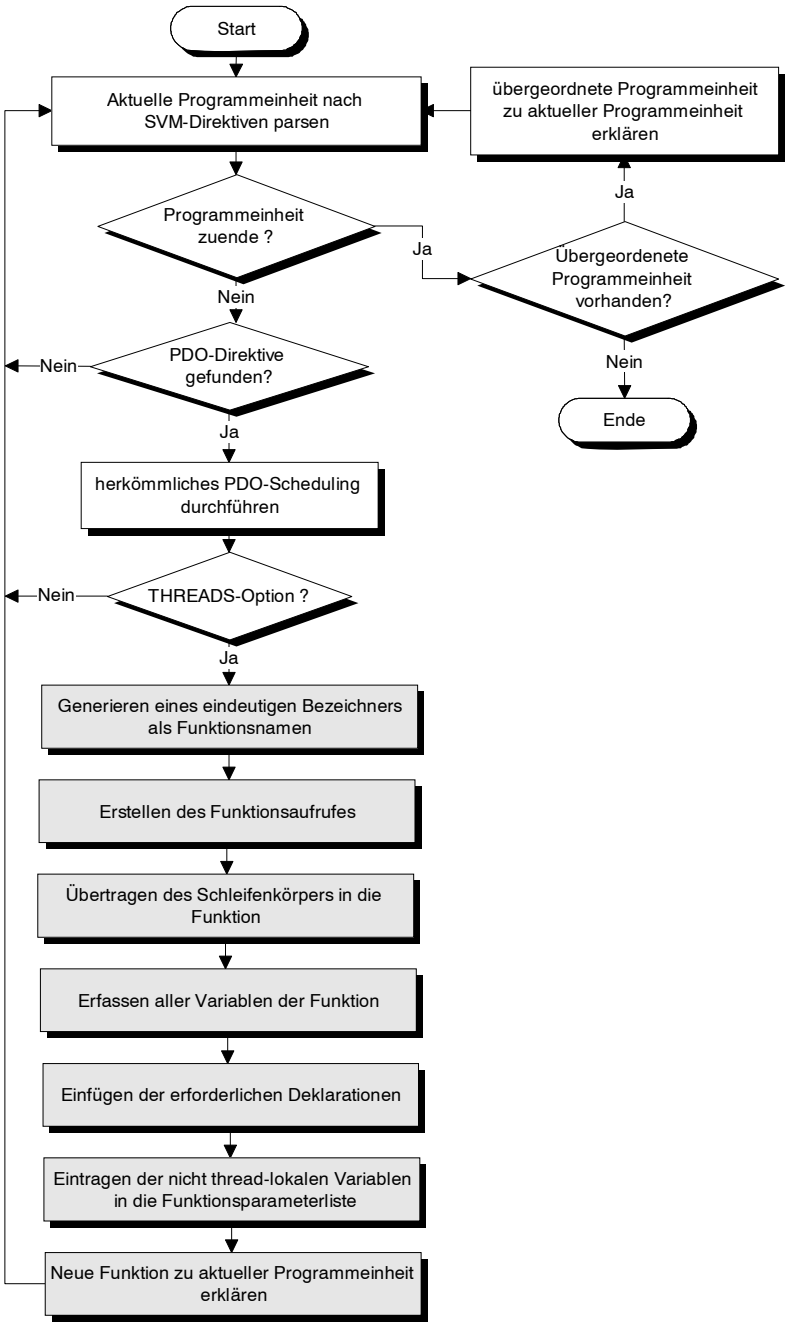


Abbildung 5.1: Transformierung der Schleife in ein Unterprogramm

Reihenfolge an das Fortran-Unterprogramm übergeben. Die Itererationsgrenzen der (parallelen) Schleife jedoch, die mit den anderen Parametern zusammen in dem Zeigerfeld übergeben werden, müssen von der Scheduling-Funktion der Laufzeitbibliothek erkannt werden, um sie wiederum auf die Threads zu verteilen. Dazu mußten weitere Parameter an die Laufzeitbibliothek übergeben werden, um die Iterationsgrenzen zu identifizieren.

- **keine Typprüfung**
  Eine Typprüfung der Parameter in der Laufzeitbibliothek ist nicht möglich, jedoch auch nicht erforderlich, da nur die Zeiger und die als Integerzahlen vorliegenden Iterationsgrenzen behandelt werden.

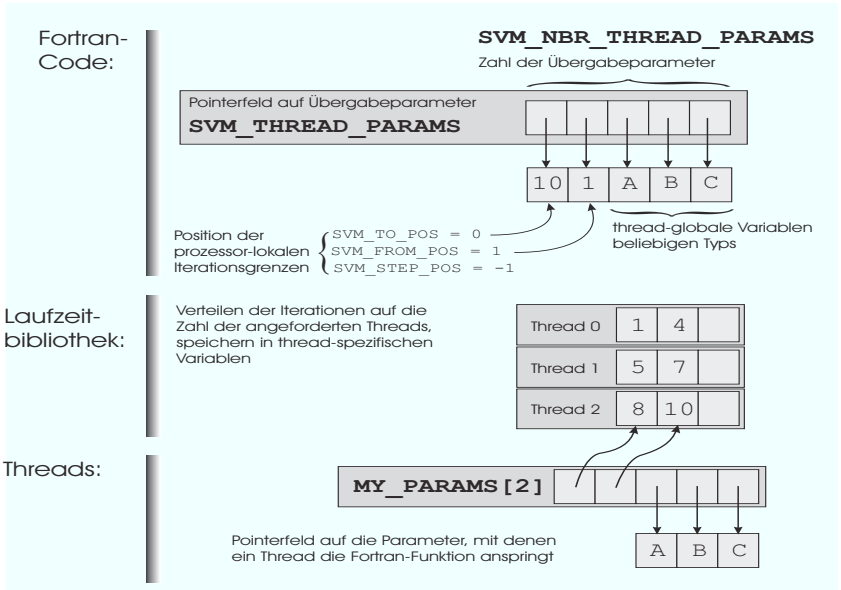


Abbildung 5.2: Behandlung der Übergabeparameter an das Thread-Unterprogramm (siehe auch Quelltext 5.6)

5.5.5 Durchführung von Reduktionen

Wenn in der parallelen Schleife mittels der PDO-Option `REDUCTION()` eine Reduktionsoperation durchgeführt wird, erfordert dies eine besondere Behandlung, wenn die Schleife mit Threads arbeitet. Dann muß nicht nur, wie bisher, eine prozessorglobale Reduktion durchgeführt werden, sondern zusätzlich müßen zuvor die Ergebnisse der Threads auf dem einzelnen Prozessor zusammengefaßt werden. Die Realisierung dieses Vorgangs wird in Abbildung 5.3 dargestellt.

5.6 Erweiterung der Laufzeitbibliothek

Zur Verwendung des Multithreadings wurde die Fortran-Schnittstelle der Laufzeitbibliothek um einen Funktionsaufruf sowie einige Datenstrukturen, die dieser Funktion als Pa-

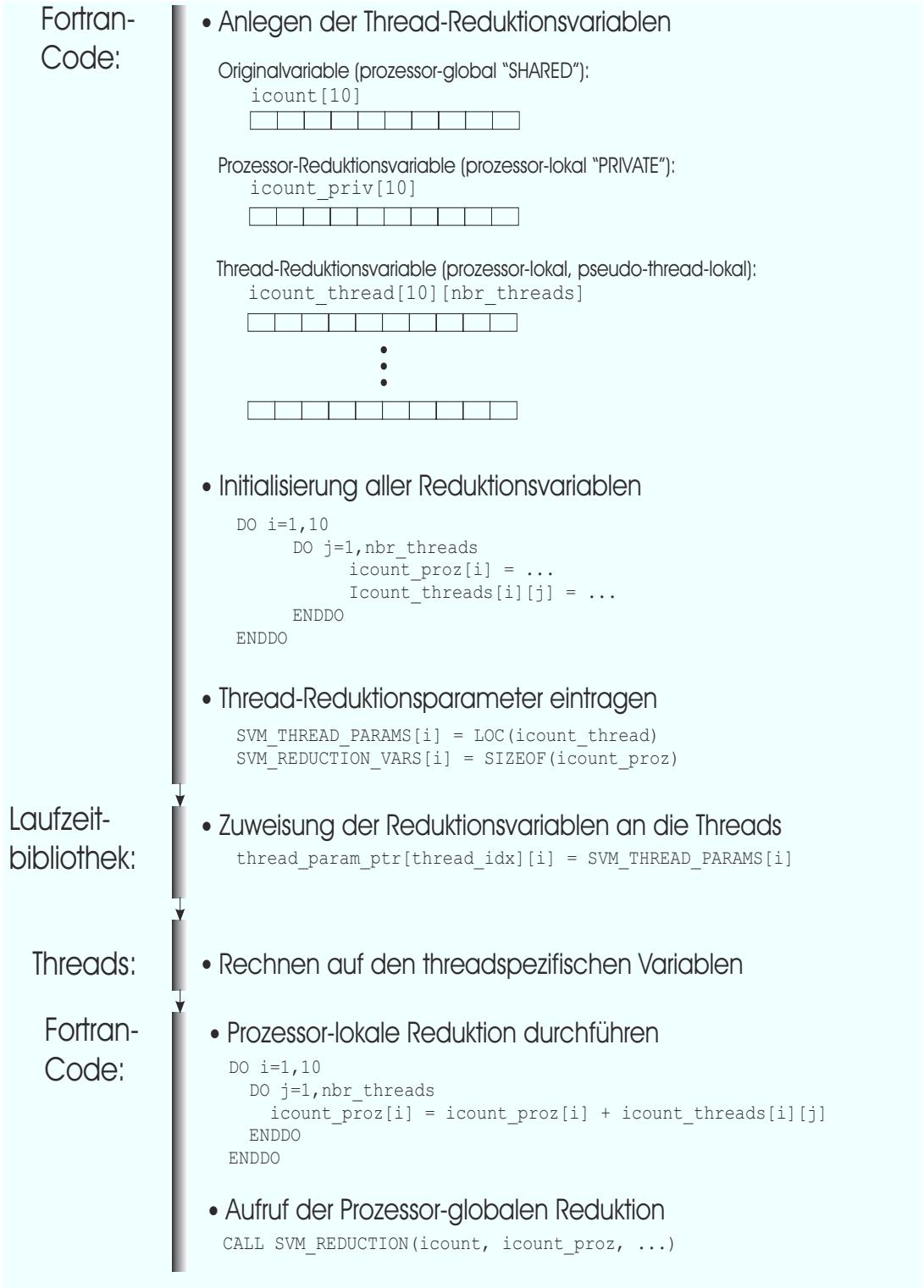


Abbildung 5.3: Behandlung von Reduktionsvariablen

rameter übergeben werden, erweitert.

5.6.1 Konzept und Ablauf des Scheduling

Um die erzeugte Fortran-Funktion von Threads abarbeiten zu lassen, könnte man jeweils die gewünschte Anzahl von Threads erzeugen und ihnen die Fortran-Funktion als Startpunkt übergeben. Diese würde abgearbeitet, anschließend beendet sich der Thread. Dieser Ansatz ist vom Scheduling- und Synchronisationsaufwand her am einfachsten, scheitert jedoch bereits an der eingeschränkten Möglichkeit der Parameterübergabe der entsprechenden Funktion des POSIX-API (siehe Kapitel 3.4). Zudem würde das wiederholte Erzeugen und Beenden übermässig viel Zeit kosten (siehe Kapitel 6.1.1).

Daher wird bei der Initialisierung der Bibliothek eine Anzahl von Threads erzeugt[4]. In ihrer Startfunktion erwerben sie sodann eine eindeutige Kennung, tragen sich in die Schlange der verfügbaren Threads ein und warten sodann an ihrer individuellen Bedingungsvariablen, die sie zusammen mit der Kennung erhalten haben. Wenn die Schedulingfunktion `SVM_SCHED_THREADS()` der Laufzeitbibliothek aufgerufen wird (siehe Abbildung 5.4), wählt sie aus dem Pool verfügbarer Threads die erforderliche Zahl von Threads aus. Dabei kann es zu folgenden Situationen kommen:

- Die Zahl der verfügbaren Threads ist größer oder gleich dem Wert des mit der THREADS-Option angegebenen Ausdrucks. In diesem Fall wird eben die gewünschte Zahl von Threads verwendet.

- Die Zahl der verfügbaren Threads ist kleiner als die gewünschte Zahl zu verwendender Threads, jedoch erfolgt in dieser Schleife keine Reduktion[5] und die Zahl der verfügbaren Threads ist größer oder gleich einer konfigurierbaren unteren Schranke der lokalen Parallelität. In diesem Fall wird die Schleife mit den aktuell verfügbaren Threads ausgeführt, um so den Overhead der Erzeugung neuer Threads zu vermeiden.

- Wenn zuwenig ausführbereite Threads zur Verfügung stehen und gleichzeitig eine Reduktion verlangt wird oder die untere Schranke der lokalen Parallelität unterschritten wird, werden zur Laufzeit die nötigen Threads erzeugt. Da Threads mit dem POSIX-API nur einzeln erzeugt werden können, sind potentiell effizientere Methoden wie das vorsorgliche Erzeugen von mehreren Threads oder der zugehörigen Strukturen (siehe Kapitel 3.3.4) mittels eines einzigen Aufrufes (und somit eines einzigen Wechels vom User- in den Kerneladreßraum) nicht möglich.

Anschließend werden die Iterationen, die der Prozessor erhalten hat, auf die Threads verteilt. Dabei kommt eine einfache Blockverteilung zum Einsatz, da dies an dieser Stelle

[4] Diese Zahl läßt sich, da sie unter Paragon OSF/1 die Leistung der Synchronisation der Threads beeinflußt, durch die Konfigurationsdatei `svmf.config` der Laufzeitbibliothek festlegen.

[5] Bei einer Reduktion *muß* die Schleife mit der verlangten Anzahl von Threads ausgeführt werden, da die zur Reduktion angelegten Datenstrukturen im Fortran-Code auf dieser Zahl basieren

auch die sinnvollste Verteilung ist. Denn zu diesem Zeitpunkt soll ja nicht mehr eine stärkere Lokalität der Daten bezüglich der Prozessoren erreicht werden, sondern eine gute Verteilung der Seiten, auf denen diese Daten liegen, zwischen den Threads. Gut bedeutet in diesem Fall, daß die Threads möglichst mit Daten arbeiten sollen, die auf verschiedenen Seiten liegen. Dadurch soll gewährleistet werden, daß nicht mehrere Threads auf der selben Seite arbeiten, was im Falle eines Seitenfehlers den erwünschten Effekt des Versteckens der Kommunikationslatenzen verhindern würde. Außerdem sollen die Threads natürlich auch möglichst lokale Datenzugriffe auf wenige Seiten haben, damit es zu maximalen Lauflängen der Threads kommt, sofern eine Seite lokal vorliegt. Ein zyklische oder noch komplexere Verteilung erfüllt diesen Zweck über einen generellen Ansatz nicht. Der individuelle Ansatz, bei dem weitergehende Kenntnisse über die Datenverteilung in das Scheduling einfließen, wurde jedoch bereits beim vorgeordneten Prozessorscheduling berücksichtigt.

Zur Synchronisation der Laufzeitbibliothek und der verwendeten Threads wird zudem eine Barriere benötigt, die ebenfalls in einem Pool von bei der Initialisierung angelegten Barrieren vorgehalten wird. Mehr als eine Barriere zur gleichen Zeit wird allerdings nur bei geschachtelten Schleifen, bei denen mehr als eine Ebene mit Threads arbeitet, benötigt.

Schließlich erfolgt die Aktivierung der Threads über die Signalisierung der Bedingungsvariablen, auf die jeder der ausgewählten Threads wartet. Nach der Signalierung wartet die Laufzeitbibliothek an der zuvor erworbenen Barriere auf die Rückkehr der Threads aus der Fortran-Routine. Die Barriere wird sodann zurück in den Pool gegeben, während die Threads sich bereits selber wieder in den Pool der verfügbaren Threads eingefügt haben.

5.6.2 Datenstrukturen

Die gesamte Abarbeitung der in das Unterprogramm transformierten Schleife durch die Threads erstreckt sich also, wie auch aus Abbildung 5.4 ersichtlich wird, über 3 Ebenen, nämlich

1. dem (compilierten) Fortran-Code, der das eigentliche Benutzerprogramm darstellt
2. der in C geschriebenen Laufzeitbibliothek, die die Verwaltung der Threads übernimmt
3. die Threads, die zwar von der Laufzeitbibliothek erzeugt wurden, aber unabhängig von dieser laufen und nur über Synchronisationsmechanismen (Bedingungsvariablen, Mutexe und darauf aufbauenden Barrieren) mit der Bibliothek in Verbindung stehen.

Für eine effiziente Kommunikation zwischen diesen Ebenen, die ja alle auf dem selben Prozessor und somit im selben physikalischen Speicher liegen, kommt die Nutzung eines

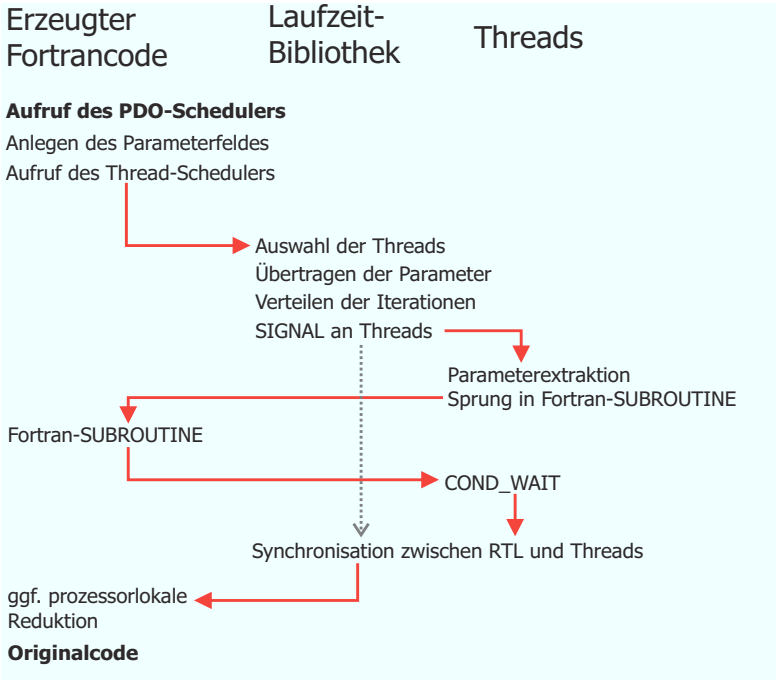


Abbildung 5.4: Ablauf des Threadschedulings zur Laufzeit

gemeinsamen Speichers in Frage. Dies führt natürlich zu globalen Variablen. Es wurde versucht, die Zahl der globalen Variablen so klein wie möglich zu halten, um die Übersichtlichkeit des Quelltextes zu gewährleisten und um die Fehlerwahrscheinlichkeit durch die Möglichkeit der Veränderung der Variablen in verschiedenen Programmeinheiten zu minimieren. Neben den globalen Variablen findet noch die Kommunikation durch die Verwendung von Zeigern als Übergabeparameter Verwendung. Hierbei muß besonderer Wert auf die korrekte Interpretation der Zeiger gelegt werden, da sie typenlos und z.T. in Feldern übergeben werden und somit der Compiler keine Möglichkeit hat, eventuelle Fehler zu erkennen. Es werden u.a. folgende Datenstrukturen verwendet:

- **Barriere zur Thread-Synchronisation** (Quelltext 5.2)
  Zur Synchronisation von mehreren Threads auf einem Prozessor reicht eine Mutex zum Schutz der kritischen Daten, eine Bedingungsvariable sowie eine Zählervariable und eine Booleanvariable aus. Hier wurden die Synchronisationsprimitive jedoch doppelt ausgeführt, um eine schnelle Wiederverwendbarkeit der Barriere nach erfolgter Synchronisation zu gewährleisten. Dazu kommen noch Datenelemente zur Verwaltung der Barriere. Diese Datenstruktur wird an Funktionen wie `pbarrier_set()` (um die Zahl von zu synchronisierenden Threads festzulegen) oder `pbarrier_wait()` übergeben, die die notwendigen Operationen durchführen.

- **Warteschlangen für Threads und Barrieren** (Quelltext 5.3)
  Die sogenannten *Pools*, in denen unbeschäftigte Threads und Barrieren gelagert werden, sind durch einfach verkettete Listen als Warteschlangen realisiert. Die

Quelltext 5.2 Datenstruktur `pbarrier_t` zur Threadsynchronisation

```
typedef struct PBarrier {
  volatile int maxcnt;    /* maximale Zahl der Threads */
  volatile int aborted;   /* Synchronisation abgebrochen ? */
  volatile int complete;  /* Alle Threads angekommen ? */
  int id;                 /* Kennung der Barriere */

  /* doppelte Sub-Barriere zur schnellen Wiederverwenbarkeit */
  struct _sb {
    pthread_cond_t  wait_cv;  /* Bedingungsvariable, an der
                                 gewartet wird */
    pthread_mutex_t wait_lk;  /* Mutex fuer kritischen Bereich */
    volatile int    runners;  /* Zahl noch laufender Threads */
  } sb[2];
  struct _sb *sbp;            /* momentane Sub-Barriere */
} pbarrier_t;
```

Schlange für die Threads enthält noch die Kennung des betreffenden Threads zur Referenzierung der verbundenen Bedingungsvariablen und anderer Verwaltungszwecke sowie den funktionalen Parameter (der von der Laufzeitbibliothek dort eingetragen wird), über den der Thread in den Fortran-Code springt.

Quelltext 5.3 Datenstrukturen für Thread- und Barrierenpools

```
/* Warteschlange fuer unbeschaeftigte Threads */
typedef struct ThreadQueue {
  struct ThreadQueue *next;

  int thread_id;
  void (*thread_work)();
} pthread_queue_t;

/* Warteschlange f"ur ungenutzte Barrieren */
typedef struct BarrierQueue {
  struct BarrierQueue *next;

  pbarrier_t *barrier;
} pbarrier_queue_t;
```

- **Übergabestrukturen für Funktionsparameter** (Abbildung 5.2)
  In Kapitel 5.5.4 wurde bereits die Methode erläutert, wie mit Hilfe von Zeigerfeldern die notwendige, variable Anzahl von Funktionsparametern an das Thread-Unterprogramm durch die Laufzeitbibliothek hindurch an die Threads weitergegeben wird. Dazu werden folgende Datenstrukturen benutzt:

- `SVM_THREAD_ARGPTR` ist ein im Fortrancode statisch angelegtes Zeigerfeld für die Übergabe der Funktionsparameter. Durch das statisch angelegte Feld ergibt sich zwar eine obere Grenze fuer die Zahl der Parameter, die bei einer dynamischen Erzeugung nicht aufträte. Jedoch kostet der erforderlich Aufruf der Laufzeitbibliothek, um dynamisch Speicher zu allokieren, zusätzlich Zeit. Unter diesem Aspekt stellt die geringe Verschwendung von Speicher (8 Byte pro nicht genutztem Feldelement) bei einem statischen Feld mit 32 Einträgen einen akzeptablen Weg dar. Der Compiler gibt eine Fehlermeldung aus, wenn festgestellt wird, daß eine Threadfunktion mehr als die maximal mögliche Zahl von Übergabeparametern benutzt.

- jede `Thread-Reduktionsvariable` ist ein Feld, das zuerst als Dummy statisch angelegt wird und dabei der Original-Reduktionsvariablen mit einer zusätzlichen Dimension für die Threads entspricht. Zur Laufzeit wird dem Feld dynamisch allokierter Speicher zugewiesen. Dieses Feld muß, trotz der Kosten für den Aufruf der Laufzeitbibliothek, dynamisch angelegt werden, denn seine Dimension hängt von dem Parameter der `THREADS()`-Option ab (der Ausdruck für die Zahl der Threads). Dessen Wert steht erst zur Laufzeit fest. Mit der oben vorgetragenen Argumentation könnte man aber auch an dieser Stelle ein statisches Limit setzen, da die Zahl der sinnvoll gleichzeitig einsetzbaren Threads unterhalb von 16 liegt. Aber der Speicherbedarf dieses Feldes, das je nach Typ der Reduktionsvariablen auch mehr als eine Dimension haben kann[6], ist unter Umständen mit mehreren Megabyte recht hoch, so daß eine akzeptable statische Grenze zu übermäßiger Speicherverschwendung führen könnte.

- `SVM_THREAD_REDUCTION` ist ebenfalls ein im Fortran-Code statisch angelegtes Integerfeld. Es gibt an, ob eine der mittels des Zeigerfelds `SVM_THREAD_ARGPTR` übergebenen Variablen eine Reduktionsvariable darstellt. Wenn ja, ist der korrespondierende Integerwert ungleich Null und gibt gleichzeitig die Größe der Reduktionsvariablen in Bytes an. Dieser Wert wird in der Laufzeitbibliothek benötigt, um jedem Thread seine individuelle Reduktionsvariable, enthalten in dem dynamisch angelegten Feld, zuzuweisen.

- `thread_iter_params` ist ein statisches, globales Integerfeld in der Laufzeitbibliothek, in das die Thread-spezifischen Iterationsparameter eingetragen werden.

- `thread_params` ist ebenfalls ein statisches, globales Feld in der Laufzeitbibliothek, enthält aber Zeiger. Diese Zeiger zeigen sowohl auf die Thread-globalen Variablen, die aus dem Kontext des Fortran-Codes stammen, als auch auf die Thread-spezifischen Iterationsparameter in `thread_iter_params` und stellen alle Parameter dar, die an das Fortran-Unterprogramm übergeben werden müssen.

[6] Die Thread-Reduktionsvariable hat immer eine Dimension mehr als die ursprüngliche Reduktionsvariable.

In Abbildung 5.5 ist als Beispiel eine etwas komplexere, zweidimensionale DO-Schleife dargestellt, die auf beiden Ebenen mit Threads arbeiten soll. Der vom SVMF-Compiler erzeugte Code, der aus Gründen der Übersichtlichkeit stark vereinfacht wurde, ist in Abbildung 5.6 (Vorbereitung und Durchführung des Aufrufs der Thread-Scheduling-Funktion) und in Abbildung 5.7 (in Unterprogramme transformierte Schleifenebenen) zu sehen.

```
CSVM$ REDISTRIBUTE (BLOCK,BLOCK) ONTO proc2:: templ2
CSVM$ PDO(LOOPS(i,j),REDUCTION(icount),
CSVM$+    STRATEGY(ON_HOME(templ2(i,j))))
CSVM$+    THREADS(nthread,nthread))
            DO i=1,N1
               DO j=1,N2
                  icount = icount + 1
               ENDDO
            ENDDO
```

Abbildung 5.5: Parallele zweidimensionale DO-Schleife in SVM-Fortran

```
      CALL SVM_PRIV_ALLOC(%REF(PTR_ICOUNT_02),
     +                    4*NTHREAD*NTHREAD, 1)
      DO I_01=1,NTHREAD*NTHREAD
         ICOUNT_01=0
         ICOUNT_02(I_01)=0
      ENDDO
100   CONTINUE
      CALL SVM_GET_BOUNDS(SVM_FROM, SVM_TO, LAST_BLOCK)
      SVM_THREAD_ARGPTR(1) = LOC(SVM_TO)
      SVM_THREAD_REDUCTION(1) = 0
      SVM_THREAD_ARGPTR(2) = LOC(NTHREAD)
      SVM_THREAD_REDUCTION(2) = 0
      SVM_THREAD_ARGPTR(3) = LOC(SVM_FROM)
      SVM_THREAD_REDUCTION(3) = 0
      SVM_THREAD_ARGPTR(4) = LOC(ICOUNT_02)
      SVM_THREAD_REDUCTION(4) = 4*NTHREAD
      THREAD_SCHED_STRATEGY = 1
      CALL SVM_SCHED_THREADS(SVM_THREAD_WORK_0,
     +                       THREAD_SCHED_STRATEGY,
     +                       NTHREAD,2,0,0,SVM_THREAD_ARGPTR,4,
     +                       SVM_THREAD_REDUCTION)
      DO I=1,1
      ENDDO
      DO I_01=1,NTHREAD*NTHREAD
         ICOUNT_01=ICOUNT_01+ICOUNT_02(I_01)
      ENDDO
      IF (LAST_BLOCK .EQ. 0) GOTO 100
      CALL SVM_PRIV_FREE(PTR_ICOUNT_02,4*NTHREAD*NTHREAD,1)
      CALL SVM_REDUCTION(ICOUNT,ICOUNT_01)
```

Abbildung 5.6: Vorbereitung und Aufruf der Threadschedulingfunktion und anschließende Reduktion

```
      SUBROUTINE SVM_THREAD_WORK_0(SVM_TO,NTHREAD,SVM_FROM, ICOUNT_02)
      EXTERNAL SVM_THREAD_WORK_1
      INTEGER SVM_THREAD_REDUCTION(20), SVM_THREAD_ARGPTR(20)
      INTEGER THREAD_SCHED_STRATEGY, LAST_BLOCK
      INTEGER J, I, ICOUNT_02
      INTEGER SVM_FROM, SVM_TO, SVM_FROM_01, SVM_TO_01, NTHREAD
      DO I=SVM_FROM,SVM_TO,1
200     CONTINUE
        CALL SVM_GET_BOUNDS(SVM_FROM_01,SVM_TO_01,LAST_BLOCK)
        SVM_THREAD_ARGPTR(1) = LOC(SVM_FROM_01)
        SVM_THREAD_REDUCTION(1) = 0
        SVM_THREAD_ARGPTR(2) = LOC(ICOUNT_02)
        SVM_THREAD_REDUCTION(2) = 4
        SVM_THREAD_ARGPTR(3) = LOC(SVM_TO_01)
        SVM_THREAD_REDUCTION(3) = 0
        THREAD_SCHED_STRATEGY = 1
        CALL SVM_SCHED_THREADS(SVM_THREAD_WORK_1,THREAD_SCHED_STRATEGY,
     +                         NTHREAD,0,2,0,SVM_THREAD_ARGPTR,3,
     +                         SVM_THREAD_REDUCTION)
        DO J=1,1
        ENDDO
        IF (LAST_BLOCK .EQ. 0) GOTO 200
      ENDDO
      RETURN
      END

      SUBROUTINE SVM_THREAD_WORK_1(SVM_FROM_01, ICOUNT_02,SVM_TO_01)
      INTEGER SVM_TO_01, SVM_FROM_0010
      INTEGER J, ICOUNT_02
      DO J=SVM_FROM_01,SVM_TO_01,1
        ICOUNT_02 = ICOUNT_02+1
      ENDDO
      RETURN
      END
```

Abbildung 5.7: Aus der mehrdimensionalen Schleife hervorgegangene Subroutinen

Kapitel 6

Effekte auf die Leistung

In diesem Kapitel wird anhand synthetischer und praktischer Benchmarks untersucht, wie sich die Verwendung von Threads in SVM-Fortran auf die erzielte Leistung der Programme auswirkt.

6.1 Kennwerte der verwendeten Systeme

Um die Leistungswerte und deren Veränderungen durch Multithreading auf den verwendeten parallelen Systemen richtig interpretieren zu können, ist es erforderlich, die relevanten Kennwerte zu berücksichtigen. Wie schon in dem theoretischen Modell in Kapitel 3.5.1 dargelegt wurde, sind dies die *Kontextwechselzeit* zwischen den Threads und die *Latenzzeit* beim Zugriff auf den nicht-exklusiven oder nicht-lokalen Speicher. Das ist bei UMA-Systemen der gemeinsame Hauptspeicher der zugeordneten Prozessoren, bei NUMA, COMA und NORMA-Systemen mit virtuell gemeinsamem Speicher wird der Speicher mit der größten Latenzzeit betrachtet. In unserem Fall nicht ausschlaggebend ist die *Erzeugungszeit* für einen neuen Thread, weil die Threads i.d.R. nur einmalig während der Initialisierung der Laufzeitbibliothek erzeugt werden, was auf jedem System wenige Millisekunden[1] dauert und gegenüber der Gesamtlaufzeit eines typischen Programms nicht ins Gewicht fällt. Sehr wichtig ist jedoch die Leistung bei der *Synchronisation* mehrerer Threads, da diese bei jedem Aufruf der Schedulingfunktion stattfindet. Zur Synchronisation werden auf allen Systemen die POSIX-Funktionen pthread_cond_wait(), pthread_cond_signal() und pthread_cond_broadcast() verwendet.

6.1.1 Paragon

Die Paragon [7] stellt ein typisches NORMA-System dar, das in der verwendeten Konfiguration mit ASVM über virtuell gemeinsamen Speicher [20] programmiert werden kann. Dabei wird der logisches Adreßraum in Seiten zu 8 KB aufgeteilt. Diese Seiten werden

[1] Die Erzeugung eines POSIX-Threads dauert auf Intel Paragon etwa 1,5 ms und auf der Sun Sparcstation 20 rund 350 μs.

mittels eines Kohärenzprotokolls unter den Prozessoren verteilt. Das Betriebssystem der Paragon ist eine angepaßte Version von OSF/1 mit einem Mach-Kernel. Dieser bietet neben CThreads auch DCE-Kernel-Threads an (also ein POSIX-ähnliches API), wobei das ins Betriebssystem integrierte ASVM bei einem Seitenfehler automatisch das Scheduling der Threads auslöst[2].

Threadsynchronisation Überraschend ist das Synchronisationsverhalten der Threads auf Intel Paragon: abhängig von der Zahl der insgesamt im Programm erzeugten Threads steigt der Zeitbedarf, um eine feste Zahl von Threads zu synchronisieren, sublinear an. In Abbildung 6.1 ist dies für die Zahl von 2 Threads zu sehen. Ebenso sind die Zeit zur Threadsynchronisation und die Zahl der zu synchronisierenden Threads nahezu linear miteinander verknüpft, wohingegen dies bei anderen Systemen (Sun Solaris) nicht der Fall ist (Abbildung 6.2). Die Ursache für dieses Verhalten konnte nicht innerhalb der Thread-Bibliothek gefunden werden und muß daher im Kernel des OSF/1-Betriebssystems auf der Intel Paragon vermutet werden, dort im Scheduler oder beim Wechsel zwischen Benutzer- und Systemadreßraum.

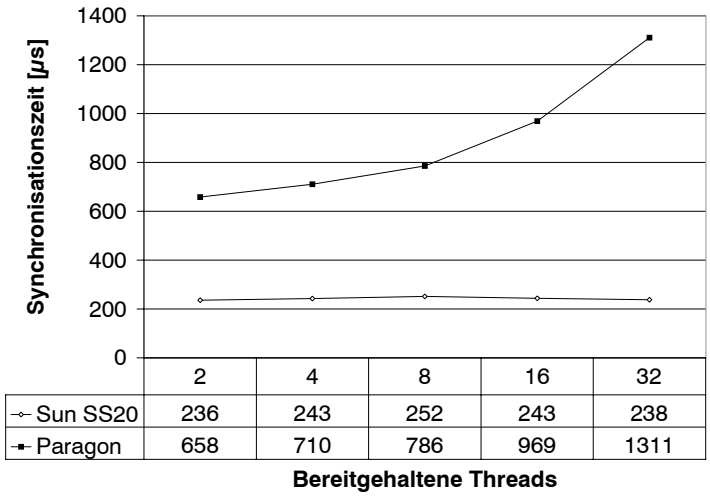


Abbildung 6.1: Synchronisationszeit für 2 Threads in Abhängigkeit der Zahl der im System vorgehaltenen Threads

Die langen Synchronisationszeiten legen nahe, auf der Paragon pro Schleife so wenig Threads wie möglich zu verwenden (2 bis 4), wenn keine extremen Seitenfehlerzahlen ($\gg 10$) pro Schleife auftreten. Diese Vermutung wird auch in Kapitel 6.2 experimentell bestätigt. Mittels Tracing-Werkzeugen wie OPAL [8] kann festgestellt werden, in welchen Schleifen eines Programms Seitenfehler in welcher Zahl auftreten, um dann eine entsprechende Zahl von Threads einzusetzen.

[2]Es wurde jedoch festgestellt, daß sich dies nicht immer wie erwartet und nicht optimal verhält (siehe unten).

Seitenfehler- und Kontextwechsellatenz Ebenfalls unerfreulich hoch ist die Zeit zum Kontextwechsel zwischen den Threads. Unter diesem Aspekt sind die Seitenfehlerlatenzen der Paragon gerade noch groß genug, um tatsächlich den gewünschten Effekt des Versteckens von Kommunikationslatenzen zu erzielen.

Dazu muß die Kontextwechselzeit kleiner sein als die Latenzzeit. Um zu überprüfen, ob diese Grundbedingung vorliegt, wurde der time_access-Benchmark (siehe Kapitel 6.2.2) mit der ZERO-Verteilung auf 2 Prozessoren ausgeführt. Neben den Daten, die am Ende des Programmlaufes ausgegeben werden, bietet der Benchmark die Möglichkeit, den Zeitpunkt eines Zugriffs auf eine Seite sowie den Zeitpunkt, wann dieser Zugriff abgeschlossen wurde, sehr effizient zu protokollieren[3]. Diese Daten können sodann den verwendeten Threads zugeordnet werden. Mittels dieser Daten kann genau ermittelt werden, bei welchem Zugriff ein Seitenfehler auftrat, wie lange die Behandlung dieses Seitenfehlers dauerte, und welcher Thread in ausgelöst hat. Ebenso wichtig ist aber, daß damit das Schedulingverhalten der Threads (auf der Paragon durch den Kernel verwaltet) verfolgt werden kann.

Die Ergebnisse dieser beiden Versuche können den Abbildungen 6.3 (Bedienzeiten für ASVM-Seitenfehler bei geringer Netzwerkbelastung), 6.4 (Bedienzeiten bei hoher Netzwerkbelastung) und 6.5 (Latenz für seitenfehlerbedingte Kontextwechsel) entnommen werden. Der Erwartungswert der Seitenfehlerlatenz liegt unter geringer Netzwerklast (SERVER-Verteilung) bei 2,7 ms, während er unter hoher Belastung des Netzwerks (ZERO-Verteilung) bei 3,8 ms liegt. Die Latenz eines Kontextwechsels liegt unabhängig von der Netzwerkbelastung bei durchschnittlich 1,6 ms, wobei diese Zeit auch die Behandlung des Seitenfehlers durch das ASVM beinhaltet. Ein Kontextwechsel, der nicht durch einen Seitenfehler verursacht wurde[4], läuft im Mittel innerhalb von 280 μs ab, wobei die Streuung sehr viel geringer als bei durch Seitenfehler bedingten Kontextwechseln ist.

Damit ist die angesprochene Grundvorraussetzung erfüllt, Multithreading *kann* sich lohnen. Die gemessenen Latenzzeiten (Abb. 6.3 und 6.4) für die Seitenfehler entsprechen recht gut der darübergelegten zugehörigen Normalverteilung. Nur der Peak bei 2,3 ms fällt heraus. Die Zeiten für die Kontextwechsel sind weniger gut verteilt; der Großteil der gemessenen Zeiten verteilt sich zwischen 0,6 ms und 1,8 ms. Viele Zeiten liegen aber noch oberhalb von 1,8 ms, was darauf schließen läßt, daß nicht zu jedem Zeitpunkt ein ausführbarer Thread zur Verfügung stand. In einigen Fällen kann man jedoch den Seitenzugriffsdaten entnehmen, daß durchaus ein ausführbereiter Thread zur Verfügung stand, dieser jedoch nicht aktiviert wurde und somit übermäßig lange Kontextwechselzeiten auftreten.

Schedulingstörungen auf Paragon Das Scheduling der Threads kann durch die Protokollierung der Speicherzugriffe durch den `time_access` Benchmark gut beobachtet werden. Zum einen werden die Threads auf der Paragon nach abgelaufenem Zeitquantum

[3] Auf der Paragon erfolgt die Zeitermittlung dazu über den globalen 10 MHz-Hardwaretimer, der einen lokalen 64-Bit-Zähler taktet.

[4] Die POSIX-Threads werden auf Intel Paragon durch ein Prioritäten-gesteuertes Zeitscheiben-Scheduling verwaltet. Die Priorität nimmt mit der Laufzeit der Threads ab [15].

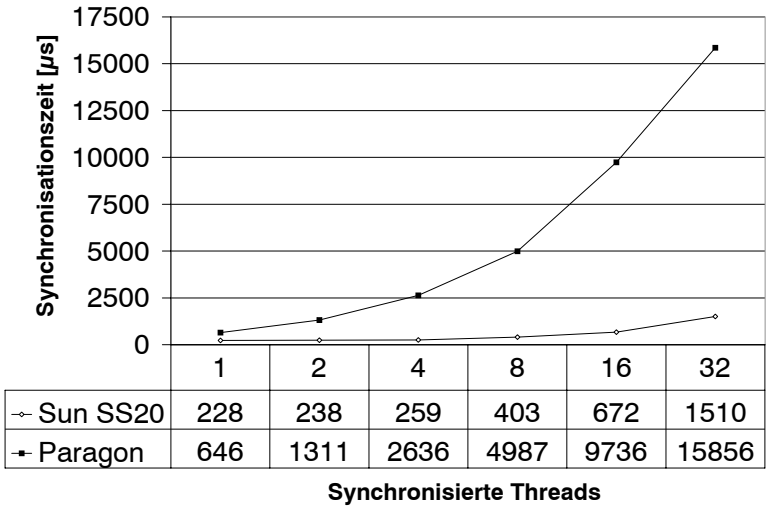


Abbildung 6.2: Synchronisationszeit von Threads in Abhängigkeit der Zahl der zu synchronisierenden Threads

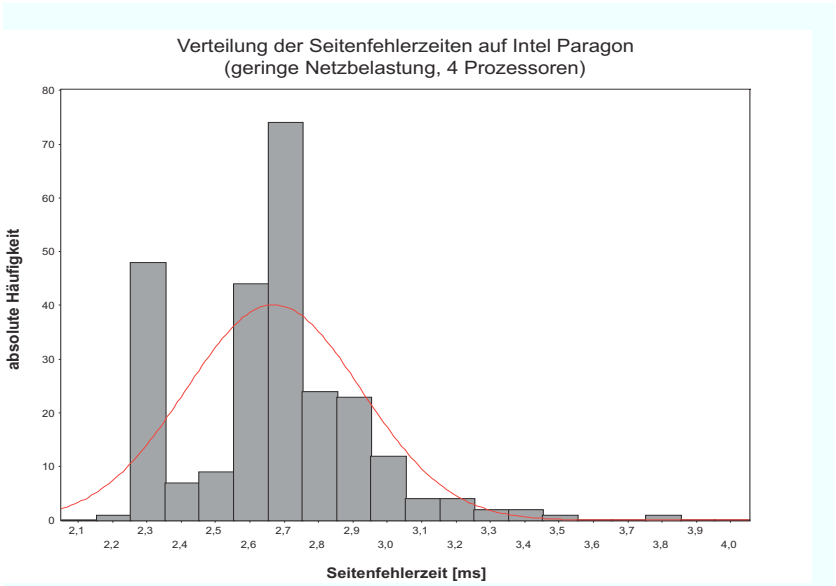


Abbildung 6.3: Verteilung der Bedienzeiten für SVM-Seitenfehler auf Intel Paragon bei geringer Netzwerkbelastung

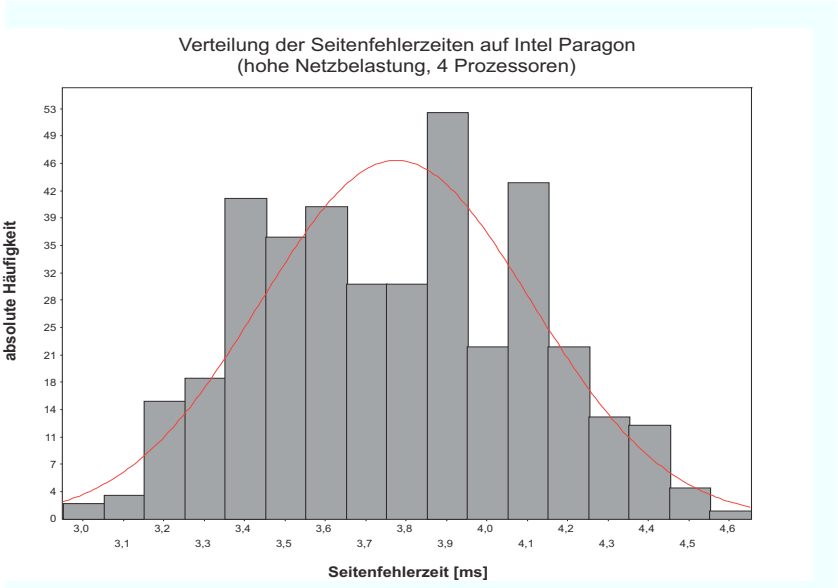


Abbildung 6.4: Verteilung der Bedienzeiten für SVM-Seitenfehler auf Intel Paragon bei hoher Netzwerkbelastung

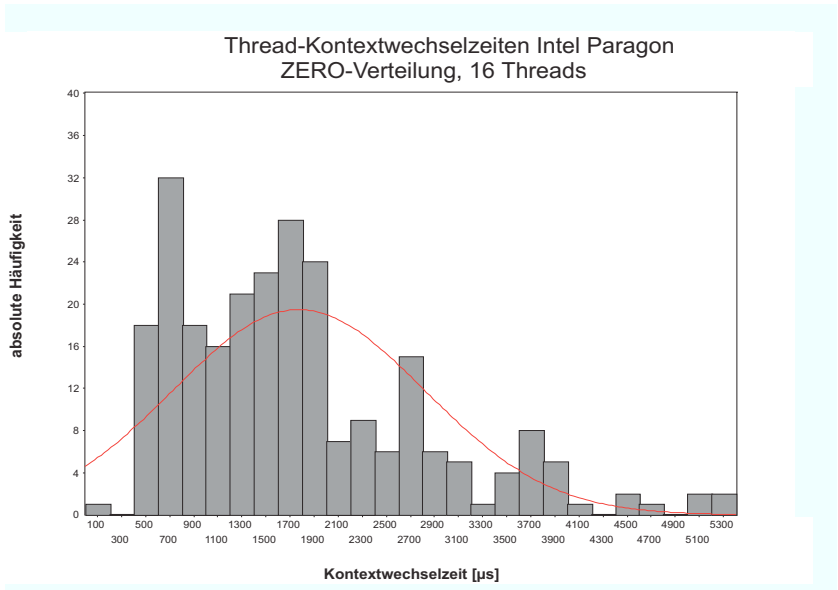


Abbildung 6.5: Zeiten für seitenfehlerbedingte Kontextwechselzeiten auf Intel Paragon

gescheduled, zum anderen führt auch ein Seitenfehler, den ein Thread auslöst, zum Aufruf des Thread-Schedulers im Betriebssystem, der sodann den nächsten ausführbereiten Thread aktiviert. Im allgemeinen verhält sich dabei das System auch wie erwartet, jedoch konnte mitunter beobachtet werden, daß der Scheduler ungünstig vorgegangen ist. Dabei wurde folgendes Problem offenbar: obwohl ausführbereite Threads zur Verfügung stehen, wird auf die Behandlung eines Seitenfehlers gewartet, anstelle einen Kontextwechsel durchzuführen. Dieses Problem wirkt sich natürlich negativ auf die Leistung aus, da wartende Threads nicht zur Ausführung gelangen und genau die Wartezeiten entstehen, die durch Multithreading vermieden werden sollen. Das Programm wird jedoch korrekt abgearbeitet. Eine Ursache für dieses sporadisch auftretende Verhalten konnte nicht ermittelt werden.

Overhead durch Threadscheduling Mittels des `time_access` Benchmarks mit NOMISS-Verteilung (siehe Kapitel 6.2.2) konnte durch den Vergleich der Laufzeiten mit und ohne Threads der anfallende Scheduling-Overhead gemessen werden, der sich aus vier Komponenten zusammensetzt:

- Aufwecken und anschließende Synchronisation der Threads
- Kontextwechsel zwischen denen im Falle der NOMISS-Verteilung (keine Seitenfehler) sequentiell ausgeführten Threads, d.h.

$$Anzahl_{Kontextwechsel} = Anzahl_{gescheduleter\ Threads} - 1$$

- Aufruf und die Abarbeitung der Threadschedulingfunktion der Laufzeitbibliothek
- Abarbeitung des hinzugekommenen Codes im Originalprogramm
- Allokierung von privatem Speicher im Falle von Reduktionen.

Dabei ist der Anteil der ersten beiden Punkte am Overhead abhängig von der Zahl der gescheduleten Threads und auf der Intel Paragon auch von der Zahl der vorgehaltenen Threads; die drei restlichen Anteile sind von der Zahl der verwendeten Threads nicht abhängig. Das Maß dieses Overheads ist in Abbildung 6.6 dargestellt. Aus dem auf den einzelnen Thread bezogenen Overhead wird deutlich, daß die Synchronisation- und Kontextwechselzeiten überwiegen: mit zunehmender Zahl der Threads fällt der restliche Teil des Overheads (alle aufgeführten Punkte außer der Synchronisation) pro Thread weniger ins Gewicht, bevor die superlinear ansteigende Synchronisationszeit bei 16 Threads den Overhead wieder leicht ansteigen läßt. Insgesamt liegt der Overhead bei 1 ms pro Thread, so daß zusammen mit der Kontextwechselzeit es nicht möglich sein wird, bei 1 oder 2 Seitenfehlern in einer Schleife selbst mit nur 2 Threads eine Leistungsverbesserung zu erreichen.

6.1.2 Sun Sparcstation 20

Die Sun Sparcstation 20 ist eine Dualprozessor-Workstation mit UMA-Architektur. Daher treten bei Speicherzugriffen nur vergleichsweise geringe Latenzzeiten auf, so daß sich durch Multithreading keine positiven Veränderungen der Leistung ergeben sollten. Dennoch wurde in Kapitel 6.3.1 ein interessantes Verhalten beobachtet, das den Einsatz von Threads rechtfertigen könnte.

Threadsynchronisation Die Zeiten zur Synchronisation von Threads sind bereits in den Abbildungen 6.1 und 6.2 im Vergleich zur Intel Paragon angegeben. Hier steigt die Synchronisationszeit nur sublinear mit der Zahl der Threads; die Synchronisationszeit für eine feste Zahl von Threads ist zudem erwartungsgemäß unabhängig von der Zahl der insgesamt generierten Threads.

Die Ursache für das sehr viel bessere Verhalten der Threads unter Solaris liegt in der Implementation dieser Threads als Userthreads, die an einen Kernelthread gebunden sind. Dadurch werden mehrere Userthreads auf einen Kernelthread abgebildet, was bei Synchronisation und Scheduling zeitaufwendige Wechsel zwischen Benutzer- und Kernadreßraum vermeidet. Da keine Seitenfehler auftreten, ist die direkte Verwendung von Kernelthreads bzw. die 1:1 Abbbildung von POSIX-Threads auf Kernelthreads auch nicht erforderlich, da keine Kommunikationslatenzen überdeckt werden können.

Kontextwechsellatenz Da Seitenfehler oder größere Speicherlatenzen auf einem UMA-System mit 2 Prozessoren nicht auftreten, wird hier von den Kenngrößen nur die Kontextwechsellatenz für die eingesetzten Threads betrachtet. Auf dem verwendeten Testsystem Sparcstation 20 liegt die typische Zeit für einen Kontextwechsel bei 31,5μs, wobei die Streung sehr gering ist. Auf einem leistungsfähigerem System (Sparcstation Ultra) wurden Kontextwechselzeiten von 13,9 μs gemessen.

Overhead durch Threadscheduling Der Overhead für den Einsatz von Threads in einer Schleife liegt auf der Sun Sparcstation deutlich unterhalb des Overheads, der auf der Intel Paragon ermittelt wurde (Abbildung 6.7). Wiederum stellen die gemessenen Zeiten den Unterschied in der Ausführungszeit des `time_acccess` Benchmarks ohne und mit Threads dar. Allerdings fällt auf, daß der Overhead im Fall von 2 Threads nicht linear mit der Zahl der Threads skaliert. Dies kann nicht auf die Synchronisationszeit zurückgeführt werden, da diese sich beim Übergang von 2 auf mehr Threads nur geringfügig ändert. Hier müssen interne Techniken des Scheduler im Betriebssystem als Ursache vermutet werden, wie sie auch schon bei der Messung der Synchronisationszeit über Fork&Join (siehe Abbildung 4.8) als Sprung bei 5 Threads beobachtet wurden. Es wird durch den starken Rückgang des Overheads pro Threads offensichtlich, daß der von der Zahl der gescheduleten Threads abhängige Anteil am Overhead aufgrund der leistungsfähigeren Synchronisation geringer als bei Intel Paragon ist.

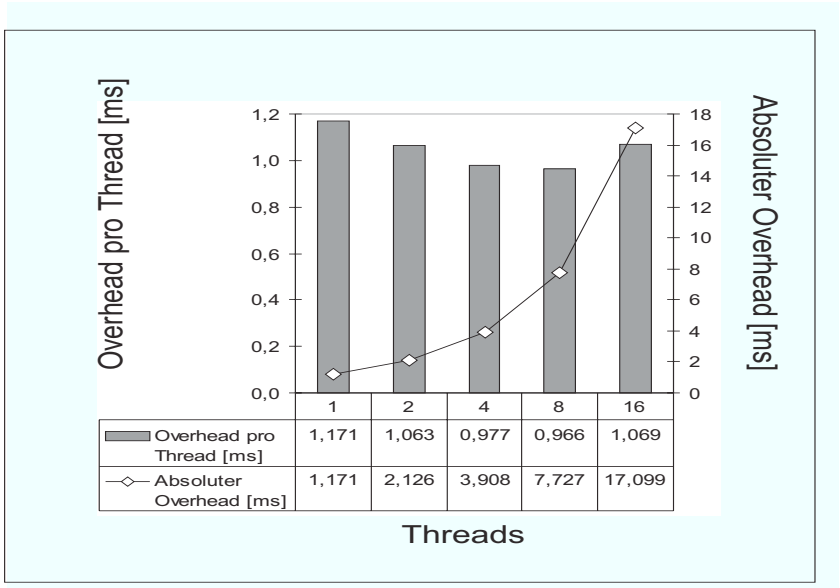


Abbildung 6.6: Overhead für Threadscheduling auf Intel Paragon

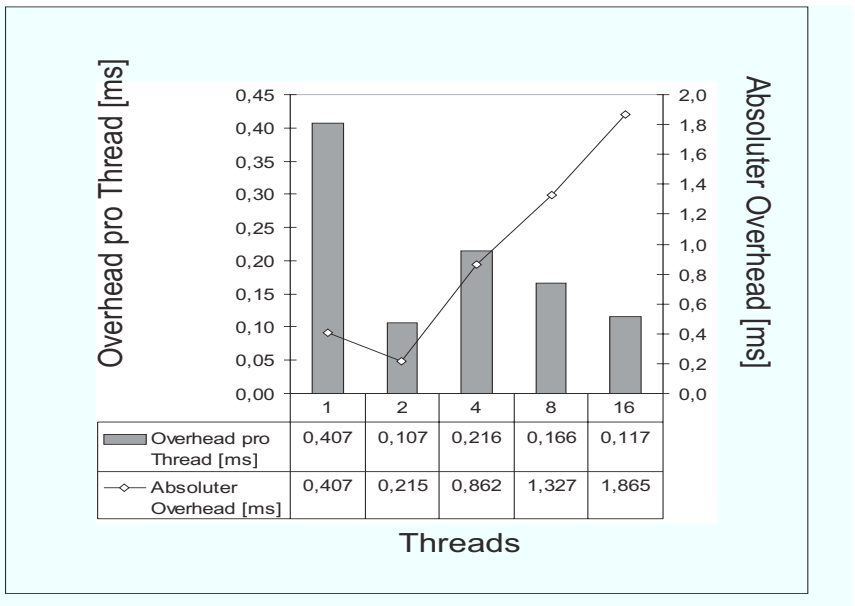


Abbildung 6.7: Overhead für Threadscheduling auf Sun Sparcstation 20

6.2 Synthetische Benchmarks

Synthetische Benchmarks sind Testprogramme, die keiner realen Applikation entstammen, sondern speziell geschrieben wurden, um bestimmte Eigenschaften eines Systems zu untersuchen und Probleme zu reproduzieren und zu lokalisieren. Ihre Ergebnisse, in der Regel Leistungswerte wie MFLOPS oder MB pro Sekunde, können daher auch nicht pauschal auf das System übertragen werden bzw. es darf nicht pauschal erwartet werden, daß mit einer realen Anwendung gleiche Leistungswerte erreicht werden.

6.2.1 Korrektheitstests

Die Grundvorraussetzung für die Verwendung von Threads ist natürlich, daß die Ergebnisse der Programme weiterhin korrekt sind. Da dies für alle SVMF-Erweiterungen gilt, wurde eine Serie von Programmen geschrieben, die die Funktion dieser Erweiterungen in verschiedenen Variationen auf Korrektheit prüfen. In realen Programmen ist eine solche Prüfung in der Regel nicht einfach zu integrieren und ist auch nicht sinnvoll. Zwar geben diese relativ einfachen Tests keine Garantie, daß eine Direktive (also SMVF-Erweiterung) unter allen Umständen ein korrektes Ergebnis liefern wird, aber es hat sich gezeigt, daß sehr viele Probleme schon durch diese systematischen und vollständigen Tests aufgezeigt wurden. Vollständig heißt hier, daß alle Konstrukte in sinnvollen Kombinationen verwendet werden. So werden beispielsweise, was die Threads betrifft, alle Schedulingverfahren sowohl bei ein- als auch bei zweidimensionalen Schleifen getestet, nicht jedoch bei Schleifen noch höherer Ordnung, da bereits der Schritt von ein- auf zweidimensional die meisten Probleme, die bei mehrdimensionalen Schleifen mit Threads auftreten können, impliziert. Diese Probleme sind das rekursive Scheduling (die Datenstrukturen in der Laufzeitbibliothek müssen *Thread-sicher* sein), eine sehr viel größere Zahl von gleichzeitig aktiven Threads und Synchronisationseinrichtungen sowie für den SVMF-Compiler ein komplizierteres Scheduling und umfangreichere Parameterübergabe, um die Sichtbarkeitsbereiche der Variablen korrekt zu behandeln.

Für die Korrektheitstests wurden die Programme *pdo_thread.f* (Scheduling paralleler Schleifen), *cflow_thread.f* (Kontrollflußsteuerung auf parallelen Systemen) und *reduction_thread.f* verwendet. Sie sind mit den Programmversionen, die keine Threads verwenden, identisch bis auf die THREADS()-Option in der PDO-Direktive. Die Testprogramme lieferten sowohl ohne als auch mit Verwendung von Threads korrekte (identische) Ergebnisse bei allen implementierten Direktiven und Konstrukten von Direktiven.

6.2.2 time_access.f

Neben der Korrektheit der Programme war natürlich die Messung des erhofften Effekts der Überlappung von Kommunikation und Berechnung der wesentliche Punkt. Diese Messungen sollten mit bestimmten Betriebszuständen des Systems in Zusammenhang gebracht werden, um die Ursachen für den jeweils beobachteten Effekt zu ermitteln. Die wesentlichen Parameter für den Betriebszustand eines parallelen Systems sind

- **Prozessorauslastung**
  Hier wird unterschieden, ob ein Prozessor auf lokal vorhandenen Daten[5] rechnet oder ob er auf Daten zugreifen will, die nicht lokal vorliegen, sondern bei denen ein Zugriff Kommunikation im System auslöst. Die Zeitdauer, die ein Prozessor ohne Kommunikation rechnet, wird in dem Modell des Multithreading (Kapitel 3.5.1) als *Lauflänge* bezeichnet.

- **Netzwerkbelastung**
  Die Netzwerkbelastung hängt einerseits von der Prozessorauslastung ab (Greift überhaupt ein Prozessor auf Daten zu?) und zum anderen von der Verteilung dieser Daten im Speicher (Welche Netzwerkaktivität löst ein Zugriff auf Daten aus im Hinblick auf die räumliche und zeitliche Verteilung der Kommunikationsvorgänge?).

Der wichtigste Wert bei der Beurteilung der Effizienz von Multithreading zum Verstecken von Kommunikationslatenzen ist der sogenannte *Seitengewinn*, definiert als

$$\frac{Ausführungszeit_{ohne\ Threads} - Ausführungszeit_{mit\ Threads}}{maximale\ Seitenfehlerzahl\ auf\ einem\ Prozessor}$$

Der Seitengewinn ist das entscheidende Maß für das Leistungsverhalten des Codes, der mit Threads arbeitet. Durch ihn kann unabhängig von der Laufzeit des Programms und der Zahl der auftretenden Seitenfehler angegeben werden, wie effizient die Verwendung von Threads ist. Theoretisch kann hier als Bestwert die durchschnittliche Bedienzeit eines Seitenfehlers erwartet werden. Davon muß jedoch einerseits der Overhead abgezogen werden, der durch die Verwendung der Threads anfällt, andererseits fällt ein Teil der Bedienzeit auch bei der Verwendung von Threads an (der Seitenfehler muß ja weiterhin zunächst einmal behandelt und die Anforderung der Seite abgesetzt werden). Es sind natürlich auch negative Werte für die Seitengewinne möglich, wenn der Overhead den erzielten Gewinn durch Überlappung übersteigt.

Zur Ermittlung der Effizienz des Multithreadings auf der Intel Paragon mit ASVM wurde der synthetische Benchmark `time_access`[6] entwickelt, bei dem sowohl die Prozessorauslastung (die Lauflänge der Threads) als auch die Netzwerkbelastung (über die Verteilung der Daten) parametrisiert sind. Die Prozessorauslastung wird über eine Schleife wählbarer Zeitdauer realisiert, deren Iterationszahl zu Beginn des Benchmarks über eine Kalibrierung bestimmt wird, um die vorgegebene Zeit mit Berechnungen zu verbringen. Der Einsatz von System-Timern hat sich hier als nicht sinnvoll erwiesen, da diese auch weiterlaufen, wenn ein Thread den Prozessor gar nicht besitzt. Die Datenverteilung erfolgt ebenfalls in der Initialisierungsphase des Programms, indem jeder Prozessor schreibend auf die ihm entsprechende der gewählten Verteilung zugeteilten initialen Daten zugreift, so daß die Daten vom Betriebssystem in seinen lokalen Speicher transferiert werden[7].

[5] Daten, auf die der Prozessor ohne Konkurrenz zugreifen kann, etwa in Registern, Caches oder privatem Speicher

[6] Der Name des Benchmarks hat sich historisch entwickelt; er basiert auf einem Programm zur Messung von Zugriffszeiten.

[7] Bei dem wichtigsten Testsystem, der Intel Paragon, wird der virtuell gemeinsame Speicher in Seiten zu je 8 KB verwaltet, so daß *ein* schreibender Zugriff auf eine Seite genügt, um die Seite in den Speicher des entsprechenden Prozessorknotens zu bringen und bei allen anderen Prozessoren zu invalidieren.

Die Daten sind als zweidimensionales Feld organisiert, auf das in einer DO-Schleife zeilenweise zugegriffen wird. Dabei entsprich jeweils eine Zeile einer Seite (8 KB auf Intel Paragon).

Anschließend wird die DO-Schleife parallel ausgeführt, und die Prozessoren greifen ebenfalls schreibend auf die ihnen über die Verteilung der Iterationen zugewiesenen Daten zu. Dabei werden nicht lokal vorhandene Seiten vom entsprechenden entfernten Prozessor angefordert (Seitenfehler).

Insgesamt wurden sieben verschiedene Datenverteilungen implementiert[8], die in Abbildung 6.8 in einer grafischen Übersicht dargestellt sind. Die Diagramme zeigen jeweils in der oberen Zeile vier Prozessoren, denen in der Zeile darunter ihr lokaler Speicher zugeordnet ist, in der Breite entsprechend der Zahl der enthaltenen Seiten skaliert. Die Seiten, auf die ein Prozessor zugreifen möchte, sind mit seinem Muster aus der oberen Zeile schattiert und befinden sich z.T. in den lokalen Speichern der anderen Prozessoren. Wenn ein lokaler Speicher Seiten enthält, auf die kein Prozessor zugreifen wird, ist dieser Speicher schräg schraffiert.

Mit jeder dieser Verteilungen wurden Testläufe auf Intel Paragon durchgeführt. Als Parameter wurden die Lauflänge (10, 100, 1000 und 10000 μs, die den Bereich von üblichen Schleifenlaufzeiten[9] mit darin auftretenden Seitenfehlern abdecken) und die Zahl der Threads (keine Threads sowie 1, 2, 4, 8 und 16 Threads) variiert. Es wurden jeweils 512 Seiten zu je 8 KB auf 4 Prozessoren verteilt, so daß keine Auslagerung auf Plattenspeicher auftreten konnte und gleichzeitig die laufenden Prozessoren noch genügend Iterationen zugeteilt bekamen. Jeder dargestellte Meßwert basiert auf mindestens 4 Messungen, wenn das Ergebnis nur wenig streute. Bei unerwarteten oder stark streuenden Meßwerten wurden entsprechend mehr Meßläufe durchgeführt (bis zu 20). In den folgenden Kapiteln werden die Eigenschaften der Verteilungen dargestellt und mit den zugehörigen Meßwerten für die Seitengewinne in Beziehung gesetzt. Anschließend werden die wichtigsten Erkenntnisse zusammengefaßt.

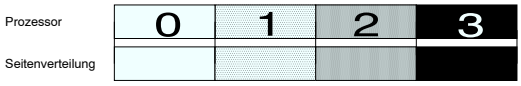
NOMISS (Abb. 6.8.1) Diese Verteilung erzeugt keinen Seitenfehler und stellt daher die Referenzverteilung dar, die angibt, wie groß die optimale Ausführungszeit ohne Seitenfehler, d.h. Netzwerkkommunikation zur Übermittlung von Daten des virtuell gemeinsamen Speichers, ist.

Anhand dieser Verteilung kann auch der Overhead ermittelt werden, der durch den Einsatz der Threads anfällt. Dazu wird die Ausführungszeit ohne Threads mit der Auführungszeit bei der Benutzung einer bestimmten Zahl von Threads in Beziehung gesetzt. Die Ergebnisse wurden bereits in Abbildung 6.6 (Intel Paragon) und 6.7 (Sun Sparcstation 20) aufgeführt.

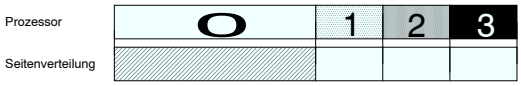
SERVER (Abb. 6.8.2) Bei dieser Verteilung sind alle Prozessoren außer Prozessor 0 reine Datenserver, die selber nicht rechnen und auch keine Seiten anfordern. Solche Zugriffe treten bei Reduktionen auf.

[8] Die Verwendung der Begriffe Daten und Seiten erfolgt äquivalent, da die Daten auf NORMA-Systemen mit virtuell gemeinsamem Speicher seitenweise organisiert sind.

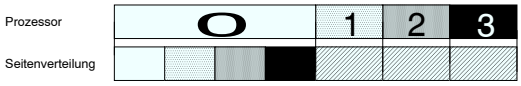
[9] Ermittelt durch Tracing mit OPAL



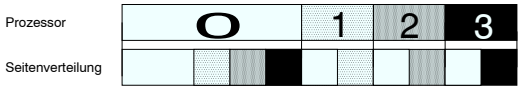
6.8.1: NOMISS-Verteilung



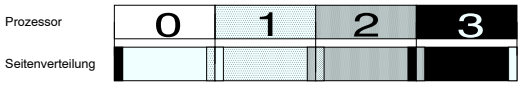
6.8.2: SERVER-Verteilung



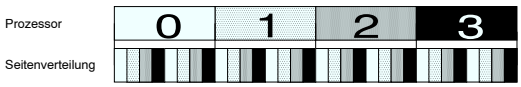
6.8.3: ZERO-Verteilung



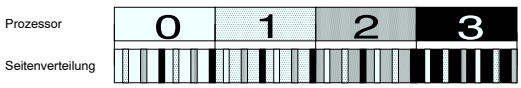
6.8.4: HALF-Verteilung



6.8.5: BORDER-Verteilung



6.8.6: SCATTER-Verteilung



6.8.7: PARTLY-Verteilung

Abbildung 6.8: Verteilungsschemata des synthetischen Benchmarks

Die Messungen bei dieser Verteilung zeigen ein ausgesprochen einheitliches Bild: die maximalen Seitengewinne liegen zwischen 1,12 und 1,17 ms, wobei der Gewinn nur wenig über die Zahl der verwendeten Threads und die Lauflänge variiert. Trotz der vielen Seitenfehler (jeder Zugriff auf eine Seite erzeugt einen Seitenfehler) stellt sich bereits bei 2 oder 4 Threads das Maximum der erzielten Seitengewinne, also die Sättigung ein. Die Ursache hierfür liegt in den kurzen Bedienzeiten der Seitenfehler durch die anderen Prozessoren. Weitere Threads erzeugen jedoch auch noch eine gute Überlappung, mögliche höhere Seitengewinne werden durch den anfallenden Overhead kompensiert.

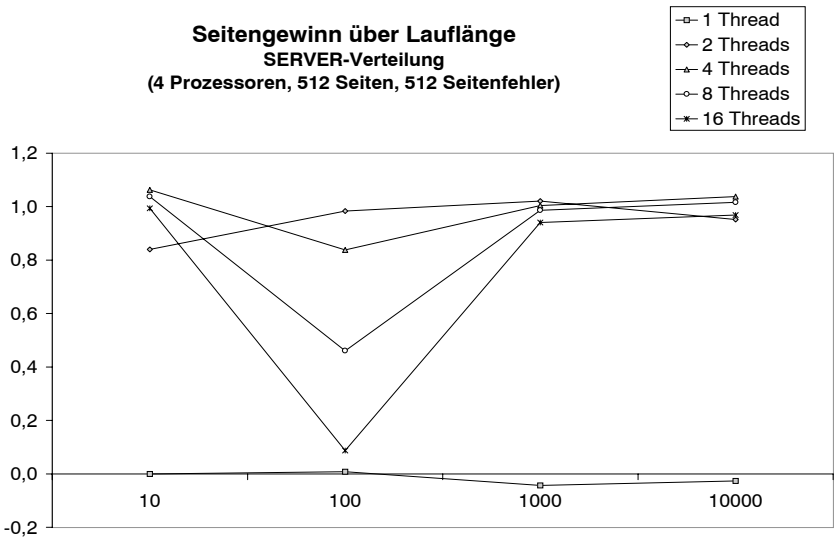


Abbildung 6.9: Seitengewinne SERVER-Verteilung

ZERO (Abb. 6.8.3) Diese Verteilung ist sozusagen das Gegenstück zu der SERVER-Verteilung, denn hier greifen alle Prozessoren auf Daten zu, die sämtlich im lokalen Speicher von Prozessor 0 vorliegen. Die beiden Verteilungen sind insofern verwandt, als daß auch hier jeder Zugriff auf eine Seite einen Seitenfehler auslöst. Hiermit soll untersucht werden, wie stark der Einfluß der Konzentration aller Seitenanforderungen auf einen einzigen Prozessor ist.

Tatsächlich ist bei den Lauflängen von 10, 100 und 1000 μs die Charakteristik der Änderung der Seitengewinne über die Zahl der Threads ähnlich der SERVER-Verteilung, nur die Höhe des Seitengewinns ist aufgrund der längeren Bedienzeit der Seitenfehler geringer und liegt zwischen 0,71 und 0,86 ms. Das Verhalten bei 10000 μs Lauflänge weicht jedoch stark ab: es werden mit maximal 0,102 ms kaum noch Seitengewinne erreicht. Es wurde daher die Entwicklung der Seitengewinne zwischen 1000 und 10000 μs genauer untersucht.

Die Ursache für diese unterschiedliche Entwicklung der Seitengewinne mit größer werdender Lauflänge liegt in der unterschiedlichen Netzwerk- und Prozessorbelastung durch

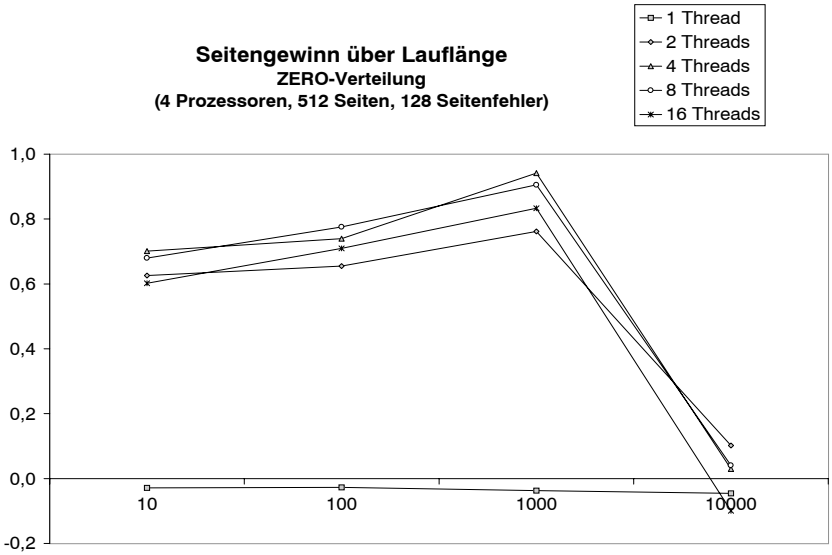


Abbildung 6.10: Seitengewinne ZERO-Verteilung

die Seitenanforderungen. Die Bedienzeit für die Seitenfehler beträgt bei der SERVER-Verteilung über alle Lauflängen hinweg rund 2,7 ms, da die Seitenanforderungen nur von Prozessor 0 aus an die restlichen Prozessoren gehen. Da diese und das Netzwerk dadurch nicht überlastet werden, bleibt die Zeit konstant. Bei der ZERO-Verteilung hingegen sinkt die Bedienzeit für die Seitenfehler von 3,6 ms bei der Lauflänge von 1000 μs auf 2,9 ms bei der Lauflänge von 10000 μs (siehe Abbildung 6.11). Denn bei der kürzeren Lauflänge (wenn gilt: Lauflänge < Bedienzeit für Seitenfehler) treffen die Seitenanforderungen der Prozessoren in einer so hohen Frequenz bei dem alleinig bedienenden Prozessor 0 ein, daß dieser und das Netzwerk überlastet sind. Sobald die Lauflänge größer als die Bedienzeit wird, was bei der Lauflänge ab 3000 μs der Fall ist, sinkt die Bedienzeit ab. Damit sinken jedoch auch die Seitengewinne, da die Effizienz des Benchmarks *ohne* Threads steigt.

HALF (Abb. 6.8.4) Wie bei der ZERO-Verteilung fordern auch bei der HALF-Verteilung alle Prozessoren Seiten ausschließlich von Prozessor 0 an. Der Unterschied ist jedoch, daß die zweite Hälfte der Seiten, auf die zugegriffen wird, bereits bei den Prozessoren vorliegt und keine Seitenfehler verursacht, so daß gerade bei den größeren Lauflängen eine gute Überlappung der Kommunikation aufgrund der fehlenden Seiten und der Arbeit auf den vorhandenen Seiten zu beobachten sein sollte.

Dies ist bei den Lauflängen von 10, 100 und 1000 μs und Verwendung von mehr als 2 Threads der Fall, wenn auch die Seitengewinne mit knapp 0,6 ms deutlich unterhalb des Optimums von etwa 2 ms liegen. Der Grund, daß 2 Threads noch keine Seitengewinne erzielen, liegt darin begründet, daß in diesem Fall der eine Thread alle Seitenfehler erzeugt, während der andere Thread in dieser Zeit seine Iterationen über die Matrix durchführt. Anschließend wartet der erste Thread nur noch auf die Bedienung seiner Seitenfehler. Dieser Effekt ist auch für die relativ geringen Seitengewinnne bei 4 und mehr Threads verantwortlich.

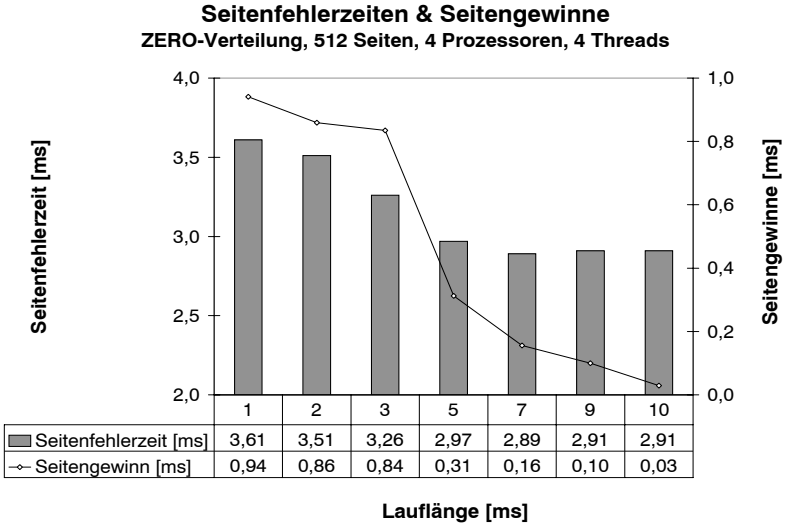


Abbildung 6.11: Entwicklung der Seitengewinne über die Lauflänge (ZERO-Verteilung mit 512 Seiten auf 4 Prozessoren)

Bei der Lauflänge von 10000 μs bringt die Verwendung von Threads keine Seitengewinne mehr, wie es auch bei der ZERO-Verteilung zu beobachten war. Bei beiden Verteilungen steigen die Seitengewinne auch bei noch längeren Lauflängen nicht wieder an, und tatsächlich ist die HALF-Verteilung in der unteren Hälfte der Seiten mit der ZERO-Verteilung identisch und zeigt daher in diesem Bereich die gleichen Eigenschaften.

BORDER (Abb. 6.8.5) Hierbei greift jeder Prozessor auf einen Bereich von 10% der angrenzenden Daten seines Nachbarprozessors zu. Der benachbarte Prozessor wird über die logische Prozessorkennung ermittelt. Dieses Verfahren liefert zwar nicht unbedingt einen tatsächlichen, physischen Nachbarprozessor (wenn ein solcher überhaupt als ausgezeichneter Prozessor existiert), aber da in realen Applikationen ebenfalls auf diese Weise vorgegangen wird, sind die Ergebnisse durchaus relevant. Diese Art von Datenverteilung und Zugriff entspricht dem Abgleich von Randwerten beim Bearbeiten eines zerlegten Gebietes.

Die gemessenen Werte entsprechen den Erwartungen: bei kürzeren Laufzeiten wird das Optimum bereits mit 2 Threads erreicht, da in diesem Fall derjenige Thread, der schließlich die Seitenfehler am oberen Ende der Iterationen erzeugt, keine ausführbereiten Threads mehr neben sich hat. Erst bei größeren Lauflängen ist dies der Fall, da dann die Threads vor Beendigung ihrer Aufgabe gescheduled werden, und so wird bei 1000 und 10000 μs das Maximum des Seitengewinns von 0,733 ms mit 4 Threads erreicht. Die Seitengewinne fallen nicht so hoch wie etwa bei der SERVER-Verteilung auf, da sich der Overhead auf weniger Seitenfehler verteilt und somit stärker ins Gewicht fällt. Der Einsatz von 8 und 16 Threads ist nicht effektiv, da die zusätzlichen Threads nur im Iterationsbereich arbeiten, in dem keine Seitenfehler auftreten.

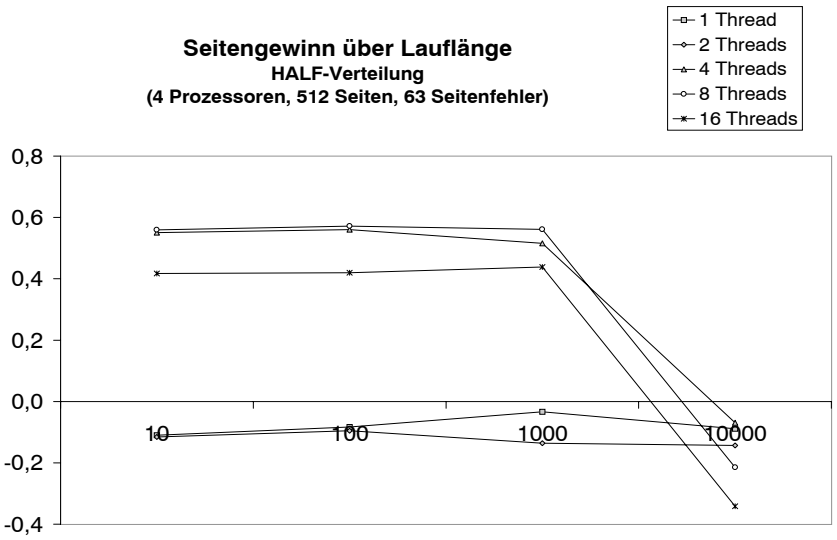


Abbildung 6.12: Seitengewinne HALF-Verteilung

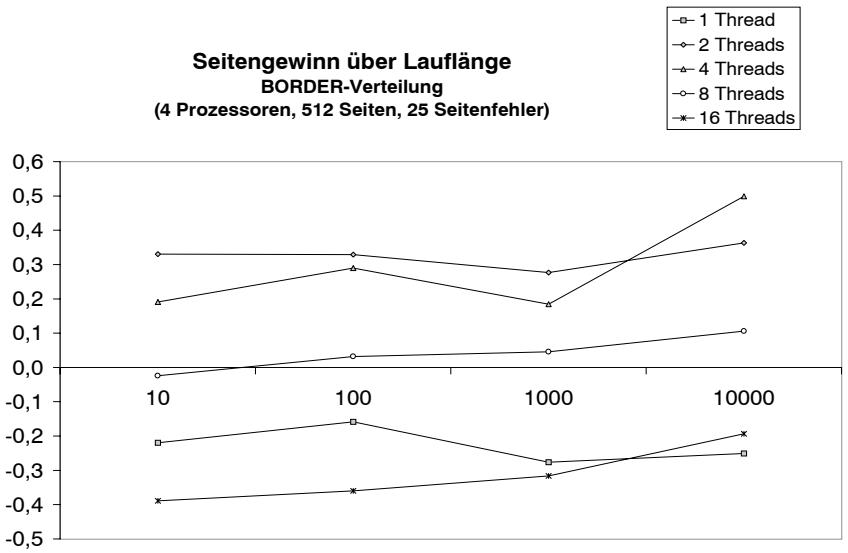


Abbildung 6.13: Seitengewinne BORDER-Verteilung

SCATTER (Abb. 6.8.6) Mittels der SCATTER-Verteilung soll ein irreguläres Zugriffsmuster erzeugt werden, in dem die Prozessoren über ihre Iterationen von wechselnden Prozessoren Seiten anfordern. Die Netzbelastung ist daher zeitlich und räumlich gleichmäßiger verteilt, insgesamt sollte sich also die Laufzeit mit Threads deutlich verkürzen.

Die Messungen bestätigen diese Vermutung, indem mit 4 Threads die höchsten Seitengewinne überhaupt von 1,82 bis zu 2,15 ms erreicht werden, was schon nahe dem theoretischen Maximum liegt. Wenn man sich die graphische Darstellung der Seitenverteilung ansieht, wird klar, daß das Optimum des Seitengewinns mit 2 Threads nicht erreicht werden kann, da es 8 gleichgroße Bereiche von Seiten gibt, die bei 4 verschiedenen Prozessoren liegen. Und tatsächlich liefern 4 und 8 Threads die besten Resultate, und selbst der starke Overhead bei 16 Threads verhindert nicht, daß die mit dieser Konfiguration erzielten Seitengewinne im gleichen Bereich liegen. Entscheidend ist hier die gleichmäßige Verteilung der Seitenfehler auf die Threads.

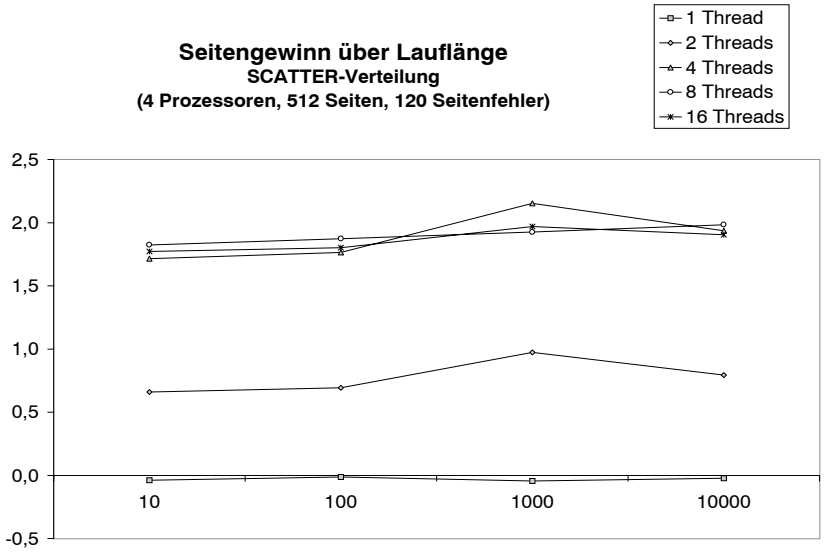


Abbildung 6.14: Seitengewinne SCATTER-Verteilung

PARTLY (Abb. 6.8.7) Die PARTLY-Verteilung hat Ähnlichkeit mit der SCATTER-Verteilung, jedoch treten hier weniger Seitenfehler auf, und der Prozessor arbeitet entsprechend länger auf lokalen Daten. Es ist also zu erwarten, daß die wenigen Seitenfehler, die auftreten, besonders gut versteckt werden. Es werden Seitengewinne von maximal 0,75 ms erzielt, was auf den höheren Overhead pro Seitenfehler (gegenüber der SCATTER-Verteilung) zurückzuführen ist. Dies bewirkt auch, daß die Seitengewinne bei Verwendung von mehr als 4 Threads zurückgehen, da dann der Overhead überwiegt.

Auffällig sind die nur sehr geringen Seitengewinne bei der Lauflänge von 1000 μs, wobei sich aber das qualitative Verhalten mit dem der anderen Lauflängen deckt.

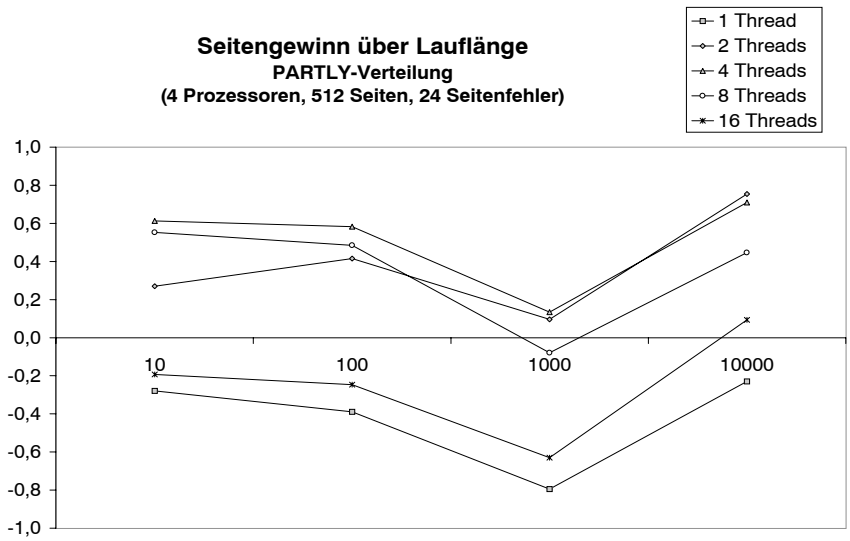


Abbildung 6.15: Seitengewinne PARTLY-Verteilung

Zusammenfassung der Ergebnisse Die Ergebnisse der verschiedenen Seitenverteilungen des synthetischen Benchmarks auf der Intel Paragon mit ASVM lassen sich wie folgt zusammenfassen:

- Die Verwendung von mehr als 4 Threads hat in keinem Fall Vorteile gebracht.
- Auch bei der Verwendung von 4 und weniger Threads können die erzielbaren Seitengewinne stark variieren (SCATTER-, BORDER- und HALF-Verteilung). Nur eine gleichmäßige Verteilung der Seitenfehler auf die eingesetzten Threads bringt optimale Ergebnisse.
- Die Lauflänge der Threads hat einen starken, aber nicht einfach voherbestimmbaren Einfluß auf die erzielten Seitengewinne (ZERO-, HALF- und PARTLY-Verteilung). Nach dem (einfachen) theoretischen Modell sind größere Lauflängen der Threads leistungsfördernd, dieses Modell wurde sowohl bestätigt (PARTLY-Verteilung) als auch widerlegt (ZERO- und HALF-Verteilung). Es kommt aber auch zu Leistungseinbußen bei mittleren Lauflängen (SERVER-Verteilung).
- Der maximal erzielbare Seitengewinn liegt bei etwas 2 ms, der typische Seitengewinn zwischen 0,5 und 0,8 ms. Aber auch mit Leistungseinbußen durch den Einsatz von Threads muß gerechnet werden (HALF-Verteilung).

6.3 Praktische Benchmarks

Nachdem die Versuche mit den synthetischen Benchmarks gezeigt haben, daß zumindest auf der Intel Paragon das Verhältnis von Seitenfehler- zu Kontextwechsellatenz einen Leistungsgewinn beim Einsatz von Multithreading ermöglicht, soll anhand von zwei praktischen Programmen untersucht werden, wie sich das Multithreading beim Einsatz in regulären Programmen verhält. Da diese nicht nur aus einer parallelen Schleife bestehen, in der eine große Zahl von Seitenfehlern erzeugt wird, sind schwächere Leistungszuwächse zu erwarten, bei ungünstigen Konfigurationen auch Leistungsbeinbußen durch den Overhead des Threadscheduling.

FIRE Der Fire-Benchmark besteht aus dem Kern eines Programms zur Simulation von strömungsdynamischen Vorgängen in Brennkammern von Verbrennungsmotoren. In diesem Kern werden dünnbesetzte, lineare Gleichungssysteme gelöst. Dieses Programm wurde in zwei unterschiedlich parallelisierten Versionen getestet. Dazu wurden verschiedene Datensätze verwendet; die hier gezeigten Ergebnisse wurden mit dem Datensatz PENT erzielt, da sich dieser aufgrund seiner Größe auf Prozessorzahlen von 4 bis 16 effektiv einsetzen läßt.

SHALLOW-WATER Als weiteres Testprogramm wurde der bekannte Shallow-Water-Benchmark von Paul N. Swartzrauber[10] herangezogen, der mittels Templates parallelisiert wurde. In dem Programm werden insgesamt 14 Matrizen (7 Matrizen jeweils in alter und neuer Version) miteinander verknüpft, um ein einfaches Klimamodell zur Wettervorhersage zu berechnen.

6.3.1 AVL-Fire

Die erste Version von Fire, *AVL-Fire*, wurde sehr einfach (und somit schnell) parallelisiert, indem alle Schleifen mit einer PDO-Direktive und Block-Scheduling versehen wurden. Da hierbei keine Rücksicht auf die Datenverteilung genommen wurde, kommt es zu einer hohen Zahl von Seitenfehlern und damit verbunden einem schlechten Speedup.

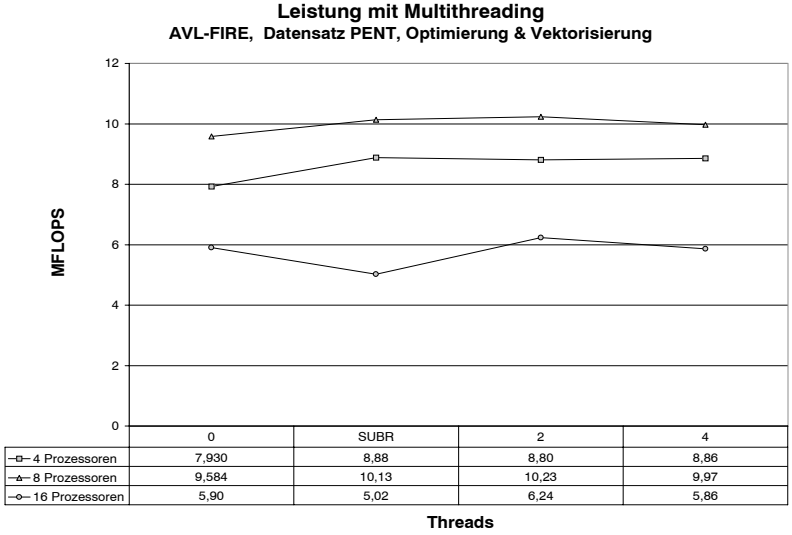
Intel Paragon

Testläufe mit der nicht durch Compileroptionen optimierten Version von AVL-Fire ergaben Seitenfehlerzahlen, die stark abhängig waren von der Verwendung von Threads: wurden Threads verwendet, stiegen die Seitenfehlerzahlen um den Faktor 2 bis 4 gegenüber der Version ohne Threads an. Für dieses Verhalten konnte ohne detaillierte Analyse des erzeugten Assemblercodes keine Erklärung gefunden werden. Allerdings hat die Verwendung von nicht optimierten Programmen auch keine praktische Bedeutung, insbesondere auf Systemen mit Vektoreinheiten wie der Intel Paragon.

[10] Das Modell basiert auf der Veröffentlichung [34].

Die optimierte und vektorisierte Version[11] von AVL-Fire bietet bezüglich der Zahl der Seitenfehler ein homogenes Bild, indem die Zahl der Seitenfehler mit und ohne Threads gleich hoch ist (Abbildung 6.17). Aber die Leistungszunahme der Version mit Threads ist nicht allein auf die Überlappung von Kommunikationslatenzen zurückzuführen, sondern liegt auch in der unterschiedlichen Vektorisierung durch den Fortran77-Compiler begründet (weitere Erläuterungen dazu im folgenden Kapitel bei der Diskussion von SVM-Fire). Daher wurde als Vergleichsbasis zur Beurteilung des Multithreading-Effekts eine AVL-Fire-Version verwendet, die durch Verlagerung der Schleife in ein Unterprogramm identisch vektorisiert werden konnte und in den Diagrammen mit SUBR gekennzeichnet ist. Die Version, die herkömmlich compiliert wurde, ist in Abbildung 6.16 mit 0 gekennzeichnet und wird in den anschließenden Leistungsvergleichen wegen fehlender Vergleichbarkeit nicht mehr aufgeführt.

Die Version von AVL-Fire, die mit 2 Threads arbeitet, erbringt auf 8 und 16 Prozessoren die höchsten Leistungen (Abbildung 6.16). Dabei ist zu beachten, daß der Datensatz ideal auf 8 Prozessoren arbeitet und hier eine maximale Leistung von 10,23 MFLOPS bei Verwendung von 2 Threads erbringt, wozu ein Seitengewinn von knapp 0,8 ms beiträgt (Abbildung 6.19). Auf 16 Prozessoren wird aufgrund der zu geringen Größe des Datensatzes und des damit verbundenen schlechten Verhältnisses von Kommunikation zu Berechnung ein negativer Speedup erzielt. Die vielen Seitenfehler, die hierbei erzeugt werden, können jedoch durch die Threads gut versteckt werden, was zu einem Seitengewinn von gut 1,2 ms führt.



	0	SUBR	2	4
4 Prozessoren	7,930	8,88	8,80	8,86
8 Prozessoren	9,584	10,13	10,23	9,97
16 Prozessoren	5,90	5,02	6,24	5,86

Abbildung 6.16: Leistung von AVL-Fire in Abhängigkeit der Verwendung von Threads

[11] Compileroption `-O3 -Mvect`

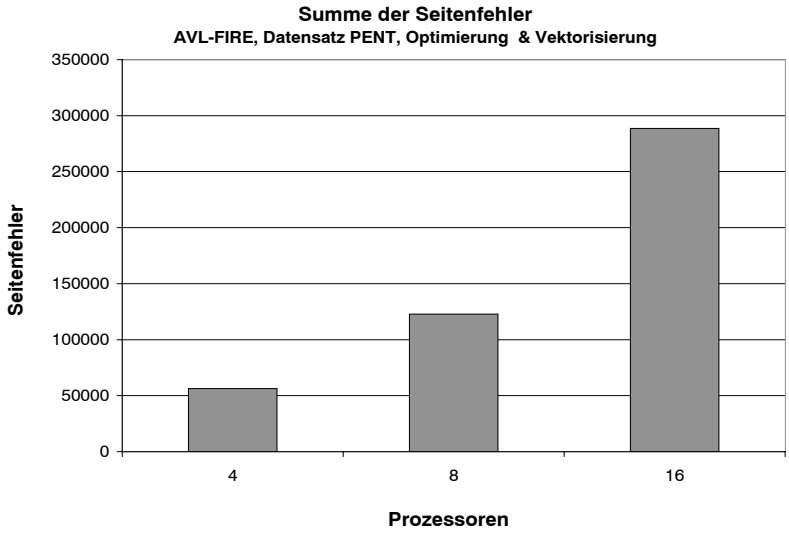


Abbildung 6.17: Zahl der Seitenfehler von AVL-Fire

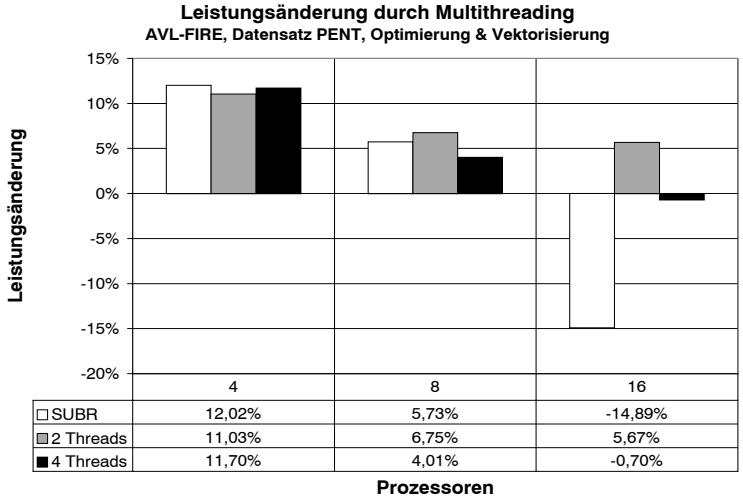


Abbildung 6.18: Änderung der Leistung von AVL-Fire durch Multithreading

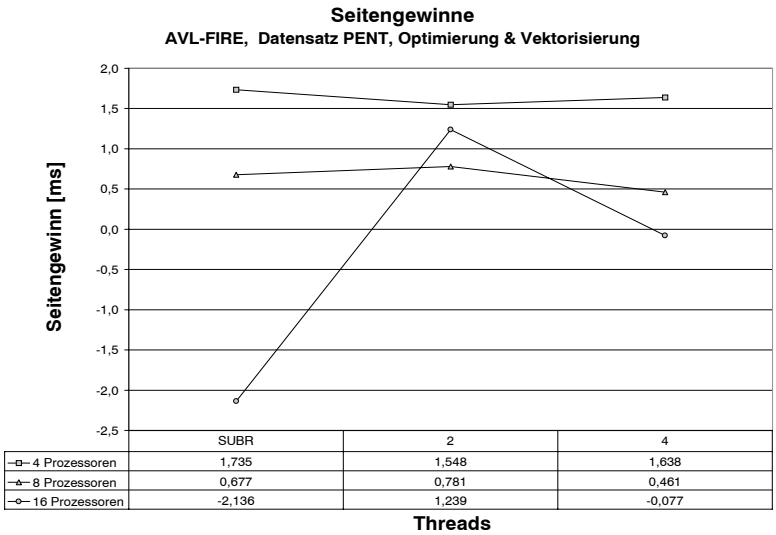
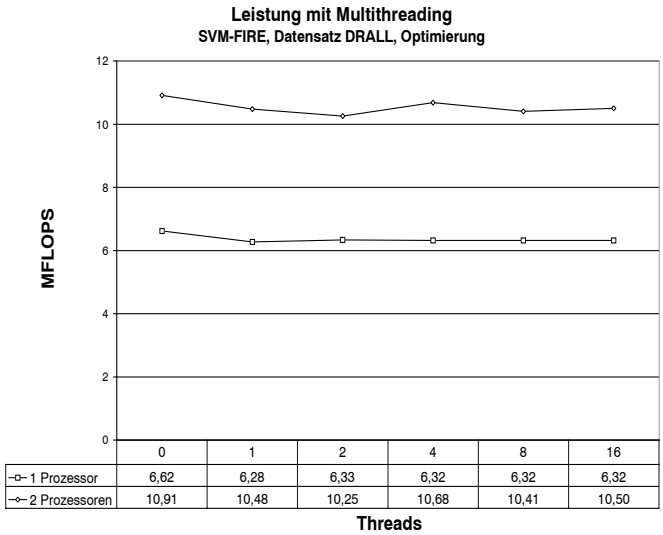


Abbildung 6.19: Erzielte Seitengewinne bei AVL-Fire

Sun Sparcstation 20

Auf de UMA-System Sun Sparcstation 20 zeigen die gemessenen Werte in Abbildung 6.20, daß die Verwendung von Threads erwartungsgemäß keinen nennenswerten Einfluß auf die Leistung hat. Das der Speedup von einem auf zwei Prozessoren nur gut 1,6 beträgt, ist vor allem auf die anfallende Synchronisation und Reduktion und kaum auf etwaige Speicherkonflikte zurückzuführen[12].



	0	1	2	4	8	16
1 Prozessor	6,62	6,28	6,33	6,32	6,32	6,32
2 Prozessoren	10,91	10,48	10,25	10,68	10,41	10,50

Abbildung 6.20: Leistung von AVL-Fire auf Sun Sparcstation 20

[12]Durch Tracing konnte der Laufzeitanteil der Barrier-Synchronisation zu 12% und der Reduktion zu 3,5% bestimmt werden

6.3.2 SVM-Fire

Die zweite Version (*SVM-Fire*) wurde durch Blockausrichtung aufwendig optimiert, indem mittels Templates die Datenverteilung an den Seitengrenzen ausgerichtet werden. Somit wird *false sharing*, wechselseitige Schreibvorgänge von mehreren Prozessoren auf dieselbe Seite, verhindert und es treten sehr viel weniger Seitenfehler auf.

Intel Paragon

Die optimierte Version zeigt auf Intel Paragon das erwartete Verhalten, weil sich die Zahl der Seitenfehler mit und ohne Threads gegenüber AVL-Fire deutlich reduziert. Es ist zu erwarten, daß die relativ wenigen Seitenfehler nur noch geringe Spielräume für Leistungszuwächse bieten.

Der Vergleich der erreichten MFLOPS-Leistungen ergibt, wie bei AVL-Fire, für die mit Threads arbeitende Versionen jedoch einen Leistungszuwachs, der nicht mehr allein auf die Verwendung von Threads zur Überlappung der auftretenden Seitenfehler erklärt werden kann: ein Seitengewinn von durchschnittlich 10 ms ist weit über dem, was theoretisch erreicht werden kann. Näheren Aufschluß lieferte die Analyse der durch den Fortran77-Compiler vorgenommenen Optimierung, insbesondere der Vektorisierung [23]. Dabei stellte sich heraus, daß die kritische Schleife, in der die Seitenfehler erzeugt werden und die durch Threads verarbeitet werden soll, nur bei der Verwendung von Threads vektorisiert wurde - und diese Version bereits dadurch deutlich mehr Leistung brachte. Die Ursache für dieses Verhalten des Compilers liegt in der Speicherverwaltung des SVM-Systems begründet, das gemeinsame Variablen (`SHARED`-Direktive) im virtuell gemeinsamen Speicher über Pointer verwaltet. Diese Verwaltung wird aus Quelltext 6.1 deutlich, die den aus der `SHARE`-Direktive erzeugten Fortran77-Code zeigt. Ein Feld, auf das indirekt über ein Zeiger-referenziertes Feld zugegriffen wird (`DIREC1(LCC(X,Y))`, siehe Quelltext 6.3) kann jedoch mit den verfügbaren Techniken des Compilers nicht vektorisiert werden[13]. Die Schleife stellt sich dem Compiler jedoch anders dar, wenn sie in ein Unterprogramm transformiert wurde, wie es bei Verwendung von Threads geschieht (Quelltext 6.2). Innerhalb der erzeugten `SUBROUTINE SVM_THREAD_WORK_()` wird auf die Felder in Fortran-Standardart zugegriffen, da die Felder auch in der üblichen Weise deklariert sind.

Um diesen Effekt zu überprüfen, wurde zusätzlich eine Version des Programms generiert, die zwar auch die Schleife in ein Unterprogramm verlegt hat, aber ohne Threads arbeitet und das betreffende Unterprogramm direkt aufruft. Die Schleife wurde daher vektorisiert; die Ergebnisse dieser Version sind in den Diagrammen mit SUBR (anstelle der Zahl der Threads) gekennzeichnet und wurde als Vergleichsbasis zur Beurteilung der Effizienz des Multithreadings verwendet.

[13] Ein explizites Abschalten etwaiger Prüfungen des Compilers auf Datenabhängigkeiten, welche eine Vektorisierung verhindern können, mittels einer Compilerdirektive führte ebenfalls nicht zu einer Vektorisierung.

Quelltext 6.1 Deklaration der Variablen der Hauptschleife im SVM-Fire Code

```
      DIMENSION DIREC1(60394)
      POINTER(PTR_DIREC1,DIREC1)
      COMMON /CC003/PTR_DIREC1
      DIMENSION DIREC2(47312)
      POINTER(PTR_DIREC2,DIREC2)
      COMMON /CC004/PTR_DIREC2
      DIMENSION LCC(6,1:47312)
      POINTER(PTR_LCC,LCC)
      COMMON /C0010/PTR_LCC

      PTR_DIREC1 = SHARED_VAR_PTR(5)
      PTR_DIREC2 = SHARED_VAR_PTR(6)
      PTR_LCC = SHARED_VAR_PTR(19)
```

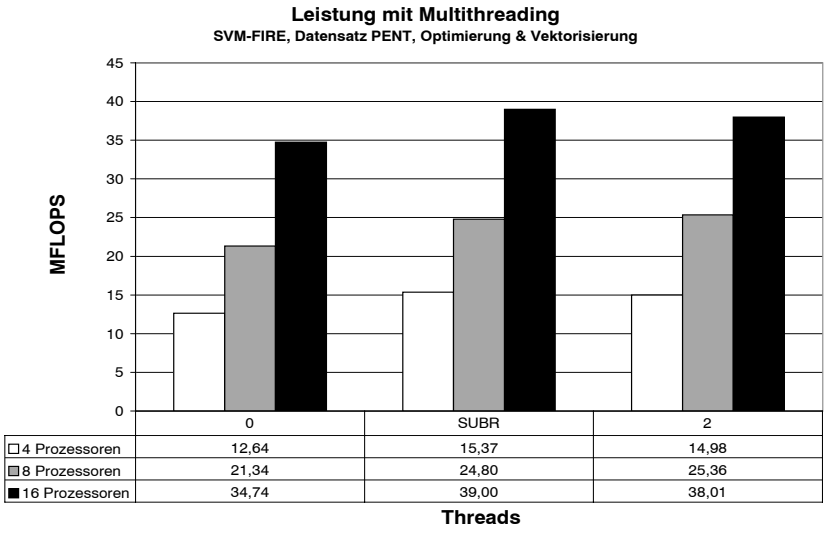
Quelltext 6.2 Verarbeitung der Übergabeparameter und Zugriff auf gemeinsame Variablen in der Subroutine

```
      SVM_THREAD_ARGPTR(1) = LOC(DIREC1)
      SVM_THREAD_ARGPTR(2) = LOC(DIREC2)
      SVM_THREAD_ARGPTR(3) = LOC(LCC)


      SUBROUTINE SVM_THREAD_WORK_(DIREC1,DIREC2,LCC,BE,BH,
     +SVM_FROM_0023,BL,BN,BP,BS,BW,SVM_TO_0024)
      DOUBLEPRECISION DIREC2(47312)
      DOUBLEPRECISION DIREC1(60394)
      INTEGER LCC(6,47312)
```

Quelltext 6.3 Indirekter Zugriff auf gemeinsame SVM-Variablen in Fortran77

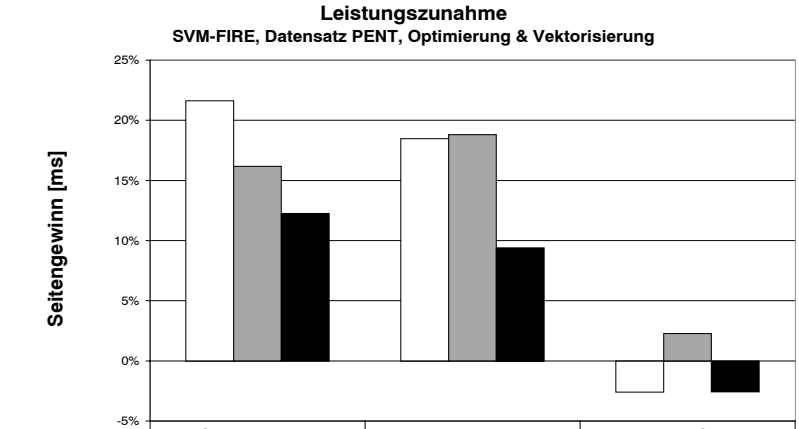
```
CSVM$ SHARED:: DIREC1, DIREC2, LCC

CSVM$ PDO(STRATEGY(ON_HOME(templ_nncell(nc))))
      DO NC=NINTCI,NINTCF
         DIREC2(NC)=BP(NC)*DIREC1(NC)
     X         -BS(NC)*DIREC1(LCC(1,NC))
     X         -BW(NC)*DIREC1(LCC(4,NC))
     X         -BL(NC)*DIREC1(LCC(5,NC))
     X         -BN(NC)*DIREC1(LCC(3,NC))
     X         -BE(NC)*DIREC1(LCC(2,NC))
     X         -BH(NC)*DIREC1(LCC(6,NC))
      ENDDO
```

Die Messungen mit SVM-Fire lieferten die Ergebnisse, die in den Abbildungen 6.21, 6.22 und 6.23 dargestellt sind. Hier wurde von vorneherein mit den Compileroptionen `-O3 -Mvect` auf höchster Stufe optimiert sowie vektorisiert. Auf Messungen mit mehr als 2 Threads wurde hier verzichtet, da Probeläufe lediglich die erwarteten Leistungseinbußen durch den zusätzlichen Overhead ergaben. Wiederum erbrachte die Version mit 2 Threads auf 8 Prozessoren die besten Leistung und lieferte durch einen Seitengewinn von 0,75 ms gegenüber der SUBR-Version um 2,3% mehr Leistung. Bei der Ausführung auf 4 und 16 Prozessoren lohnte sich der Einsatz von Threads jedoch nicht und verringerte die Leistung gegenüber der SUBR-Version um jeweils rund 2,5%. Da die maximale Seitenfehlerzahl eines Prozessors bei allen Prozessorzahlen identisch ist, muß die Ursache für dies Einbußen in der ungleichmäßigeren Verteilung der Seitenfehler auf die Threads liegen.



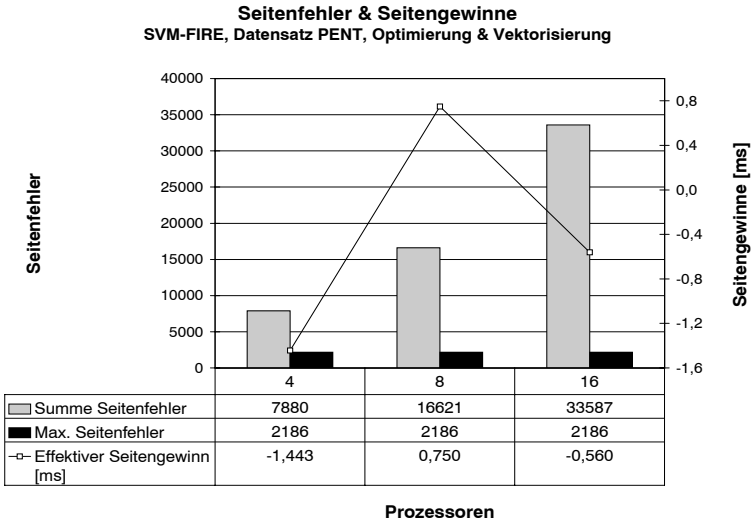
	0	SUBR	2
4 Prozessoren	12,64	15,37	14,98
8 Prozessoren	21,34	24,80	25,36
16 Prozessoren	34,74	39,00	38,01

Abbildung 6.21: Leistung von SVM-Fire in Abhängigkeit der Verwendung von Threads



	SUBR gegen 0	2 gegen 0	2 gegen SUBR
4 Prozessoren	21,6%	18,5%	-2,6%
8 Prozessoren	16,2%	18,8%	2,3%
16 Prozessoren	12,3%	9,4%	-2,5%

Abbildung 6.22: Leistungsteigerung durch Multithreading bei SVM-Fire



	4	8	16
Summe Seitenfehler	7880	16621	33587
Max. Seitenfehler	2186	2186	2186
Effektiver Seitengewinn [ms]	-1,443	0,750	-0,560

Abbildung 6.23: Seitengewinne durch Multithreading bei SVM-Fire

Sun Sparcstation 20

Auf der Sparcstation 20 zeigte die SVM-Version des Fire-Codes ein unerwartetes Verhalten. Obwohl es auf diesem System wegen des physikalisch gemeinsamen Speichers nicht zu Seitenfehlern kommt und der Sparc-Prozessor auch keine Vektorverarbeitung kennt, die den Compiler zu Problemen wie auf der Intel Paragon leiten könnte, verbessert sich die Leistung des Programm mit Verwendung von Threads spürbar, wenn nicht die maximale Optimierung durch den Fortran77-Compiler verwendet wird. Wird die Compilation mit der höchsten verfügbaren Optimierung durchgeführt, bleibt die Leistung nahezu unabhängig von der Verwendung von Threads (Abbildung 6.24). Die Ursache für diese Leistungssteigerung um bis zu 8% trotz des anfallenden Overheads muß also in den Optimierungen des Fortran-Compilers liegen, die im Bereich der indirekten Zugriffe fallen: bei der schwächeren Optimierung wird, wie beim Compiler auf Intel Paragon, die Schleife mit den indirekten Feldzugriffen über Pointervariablen nicht optimiert. Bei der maximalen Optimierung werden auch diese Zugriffe optimiert, allerdings gibt der Compiler eine Warnung aus, daß dies unvorhergesehenes Verhalten hervorrufen kann. Dies wurde in diesem Fall jedoch nicht festgestellt.

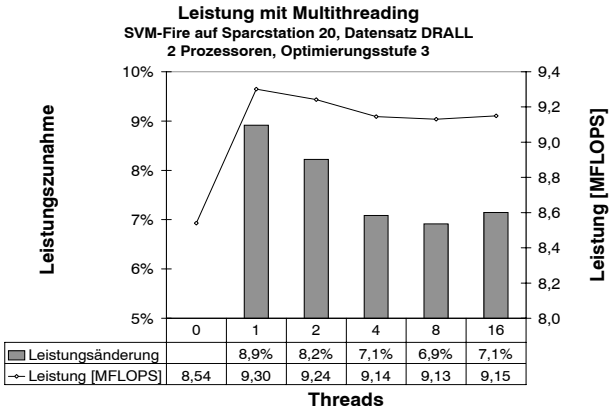
6.3.3 SHALLOW-WATER

Für die Testläufe auf der Intel Paragon wurde eine Matrixgröße von 2 MB gewählt (mittels Dimensionierung von 512 x 512), was eine Gesamtdatengröße von gut 28 MB ergab. Die Instrumentierung mit OPAL ergab, daß nur eine Schleife für die Ausführung durch Threads in Betracht kam (Quelltext 6.4). In dieser Schleife wurden pro Durchlauf 3 - 4 Seitenfehler durch Schreibzugriffe ausgelöst.

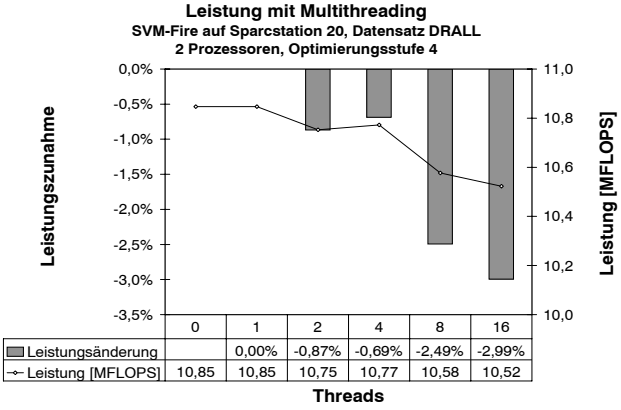
Bei der Ausführung des mit Stufe 3 optimierten, aber nicht vektorisierten Programms ergab keine Leistungsgewinne durch die Ausführung mit Threads, sondern nur leichte Einbußen von 1 bis 2%. Offenbar konnten die Seitenfehler nicht verdeckt werden, da sie nicht gleichmäßig auf die ausführenden Threads verteilt waren, sondern alle von einem Thread behandelt wurden. Denn anhand des SVM-Fire-Benchmarks kann man sehen, daß es sich grundsätzlich lohnen kann, Schleifen mit nur 3 Seitenfehlern durch Threads ausführen zu lassen.

Der Shallow-Water-Benchmark in SVM-Fortran verhält sich jedoch sehr ungewöhnlich, wenn man ihn durch den Compiler vektorisieren läßt. In dem Fall erzeugt der Code ohne Threads sehr viel mehr Seitenfehler als der Code mit Threads, welcher weiterhin die gleiche Zahl von Seitenfehlern wie der Code ohne Vektorisierung erzeugt (Abbildung 6.28). Dies liegt allerdings nicht, wie bei SVM-Fire, an indirekten Feldzugriffen, da derartige Zugriffe in Shallow-Water nicht auftreten (siehe Quelltext 6.4).

Die Leistungszuwächse durch die erfolgreiche Vektorisierung, die durch die Verwendung des Multithreading möglich wurde, sind jedoch extrem. Ursache dafür sind die zusätzlichen Seitenfehler, denn bei der Version ohne Threads erzeugt durch die Vektorisierung beim Ablauf mehr als die doppelte Zahl von Seitenfehlern als das gänzlich unvektorisierte Programm. Die Version mit Threads erzeugt auch nach der Vektorisierung keine zusätzlichen Seitenfehler und kann daher voll von der Vektorisierung profitieren und ist auf 8



6.24.1: Optimierungsstufe 3



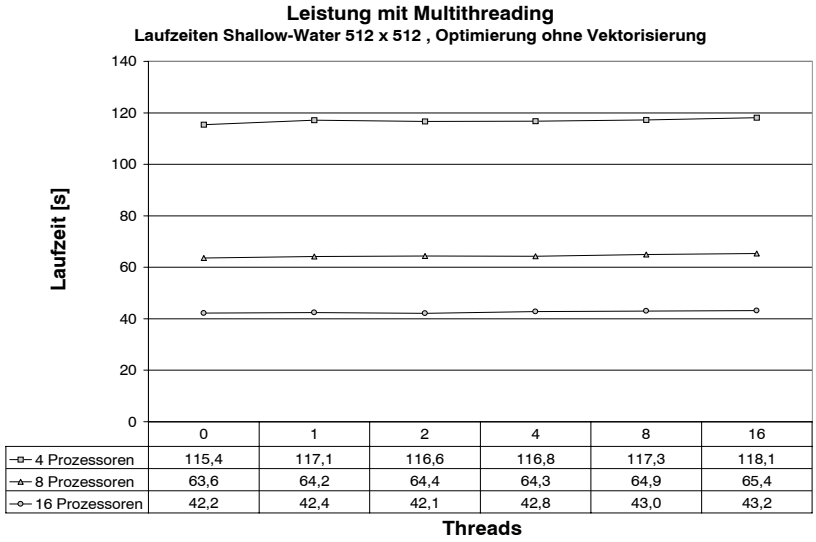
6.24.2: Optimierungsstufe 4

Abbildung 6.24: Leistung von SVM-Fire auf Sun Sparcstaion 20

Prozessoren mehr als doppelt so schnell wie die Version ohne Threads. Dem Verstecken von Kommunikationslatenzen kommt dabei allerdings nur noch eine Nebenrolle zu, da die Ausführungszeit mit zwei Threads kaum schneller ist als die mit nur einem Thread.

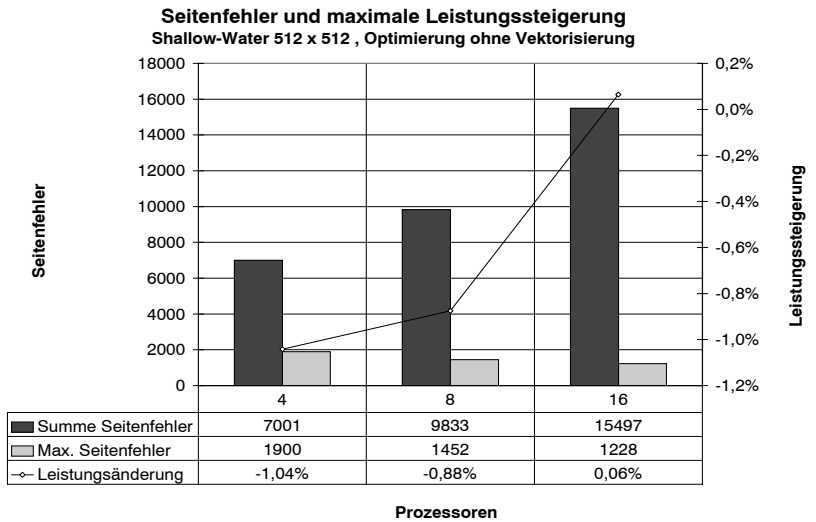
Quelltext 6.4 Mit Threads arbeitende Schleife im SVM-Fortran Shallow-Water-Benchmark

```
CSVM$ SHARED,ALIGN:: U,V,P,UNEW,VNEW,PNEW,UOLD,VOLD,POLD

CSVM$ PDO(LOOPS(J),STRATEGY(ON_HOME(templ(j))),NOBARRIER
CSVM$+   ,THREADS(nbr_t), THREAD_PRIVATE(j,i))
      DO J=1,N
        DO I=1,M
          UOLD(I,J) = U(I,J)+ALPHA*(UNEW(I,J)-2.*U(I,J)+UOLD(I,J))
          VOLD(I,J) = V(I,J)+ALPHA*(VNEW(I,J)-2.*V(I,J)+VOLD(I,J))
          POLD(I,J) = P(I,J)+ALPHA*(PNEW(I,J)-2.*P(I,J)+POLD(I,J))
          U(I,J) = UNEW(I,J)
          V(I,J) = VNEW(I,J)
          P(I,J) = PNEW(I,J)
        ENDDO
      ENDDO
```



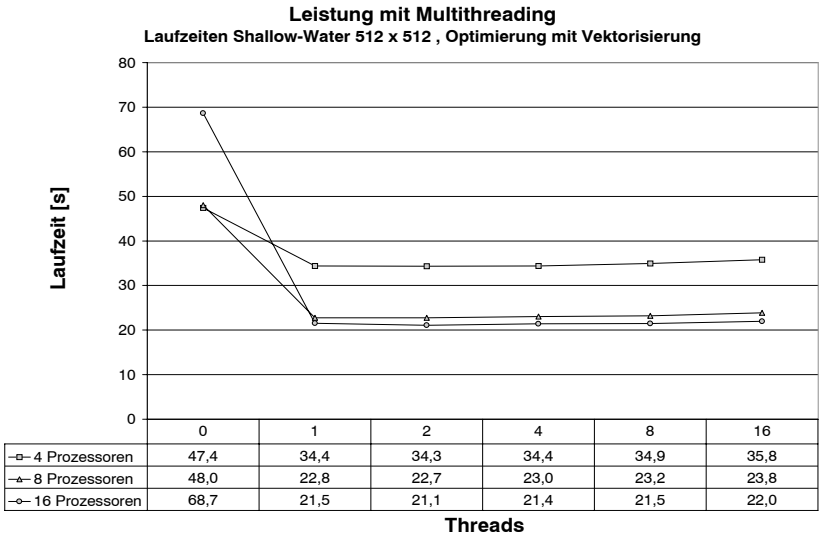
	0	1	2	4	8	16
4 Prozessoren	115,4	117,1	116,6	116,8	117,3	118,1
8 Prozessoren	63,6	64,2	64,4	64,3	64,9	65,4
16 Prozessoren	42,2	42,4	42,1	42,8	43,0	43,2

Abbildung 6.25: Laufzeiten Shallow-Water-Benchmark ohne und mit Threads (ohne Vektorisierung)



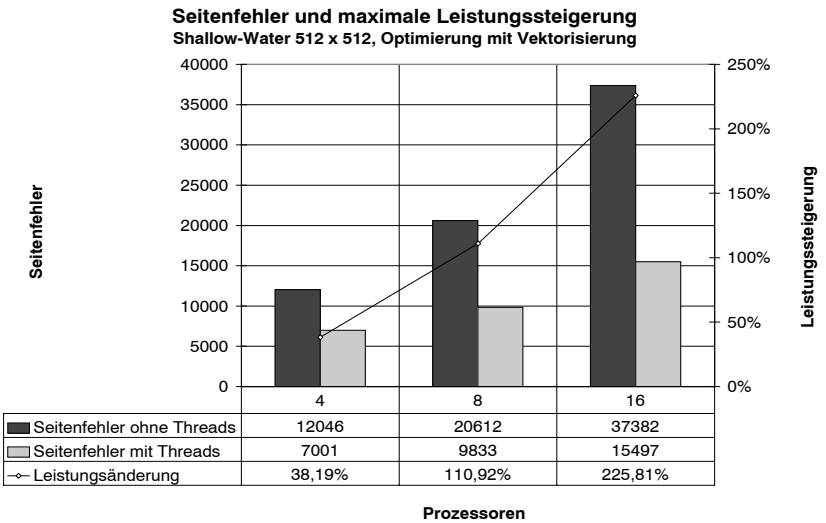
	4	8	16
Summe Seitenfehler	7001	9833	15497
Max. Seitenfehler	1900	1452	1228
Leistungsänderung	-1,04%	-0,88%	0,06%

Abbildung 6.26: Seitenfehler und Leistungsänderung bei Shallow-Water-Benchmark (ohne Vektorisierung)



	0	1	2	4	8	16
4 Prozessoren	47,4	34,4	34,3	34,4	34,9	35,8
8 Prozessoren	48,0	22,8	22,7	23,0	23,2	23,8
16 Prozessoren	68,7	21,5	21,1	21,4	21,5	22,0

Abbildung 6.27: Laufzeiten Shallow-Water-Benchmark ohne und mit Threads (mit Vektorisierung)



	4	8	16
Seitenfehler ohne Threads	12046	20612	37382
Seitenfehler mit Threads	7001	9833	15497
Leistungsänderung	38,19%	110,92%	225,81%

Abbildung 6.28: Seitenfehler und Leistungsänderung bei Shallow-Water-Benchmark (mit Vektorisierung)

Kapitel 7

Zusammenfassung und Ausblick

Die Untersuchungen haben ergeben, daß durch Einsatz von Software-basiertem Multithreading prinzipiell das gewünschte Ziel erreicht werden kann, auf Parallelrechnern mit (virtuell) gemeinsamen Speicher kommunikationsbedingte Latenzzeiten mit sinnvoller Berechnung zu überbrücken. Dabei ist jedoch erforderlich, daß die Kontextwechsellatenzen der verwendeten Threads sowie der Overhead durch die Synchronisation insgesamt kleiner sind als die Kommunikationslatenz. Diese Bedingung wird auf der Intel Paragon mit ASVM als virtuell gemeinsamen Speicher erfüllt. Es können pro auftretender Kommunikationslatenz, hier mit Seitenfehler als Ursache, bis zu 2 ms eingespart werden, was den Erwartungen an erzielbare Gewinne bei einer durchschnittlichen Bedienzeit für die Seitenfehler um 3 ms entspricht.

Die Verwendung der Threads in SVM-Fortran ist für den Programmierer sehr einfach, da das Hinzufügen einer einzigen Option zu bereits erstelltem Code genügt. Dabei kann jedes PDO-Konstrukt mit beliebigen Schedulingstrategien auf diese Weise lokal parallelisiert werden. Ein effizienter Einsatz der Threads erfordert jedoch eine Instrumentierung und Analyse der Schleifen, in denen Kommunikation erzeugt wird, die mit OPAL als Werkzeug möglich ist. Der Aufwand beim Einsatz von Threads liegt folglich nicht in der programmiertechnischen Einbindung in das Programm, sondern in der Entscheidung und Analyse, an welcher Stelle der Einsatz effizient ist. Aufgrund der beobachteten Probleme mit der Vektorisierung durch den Fortran77-Compiler sollte diese Option in der Analyse berücksichtigt werden.

In realen Anwendungen, die hinsichtlich der Datenverteilung bereits auf minimale Seitenfehler optimiert wurden, bewegt man sich auf dem primären Zielsystem, der Intel Paragon, schnell an der Grenze, an der der Overhead des Einsatzes der Threads den Gewinn aufzehrt. Dies liegt in der langsamen Synchronisation der Threads im Verhältnis zu den bei gut optimieren Programmen wenigen Seitenfehlern pro Schleifendurchlauf begründet. Auch wenn hier positive Seitengewinne erzielt werden, ist die Auswirkung auf die Ausführungszeit des gesamten Programms oftmals recht gering.

Erhebliche Leistungsverbesserung könnte auf Intel Paragon durch die Verwendung eines angepaßten Thread-Paketes erreicht werden. Denn für unsere Zwecke benötigt ein Thread so gut wie keinen Stack (i.d.R. nur für die Thread-privaten Laufvariablen), was

den Speicherbedarf reduzieren würde. Ein Scheduling bräuchte nur aufgrund von Seitenfehlern zu erfolgen, was die beobachteten Probleme mit dem Scheduler lösen würde, da keine Prioritäten und Laufzeiten berücksichtigt werden müßten. Die Probleme mit der Synchronisation (geringe Leistung und starke Abhängigkeit von der Zahl der Threads) kosten ebenfalls viel Leistung.

Eine Portierung des Multithreadings auf SGI Origin 2000 ist vorgesehen und kann interessante Aufschlüsse über die Effizienz von Software-basierten Threads gegenüber Hardware-basiertem gemeinsamem Speicher geben.

Im Bereich der Synchronisation bietet sich zuvorderst eine Implementierung des verteilten Locks (Kapitel 4.2.3) zur effizienteren Verwendung von Pagelocks unter hoher Last an. Hierbei könnte ebenfalls die Verwendung von Threads untersucht werden, wenn ein Thread durch den nächsten Thread abgelöst wird, sobald er den Lock anfordert.

Literaturverzeichnis

[1] R. Alverson, D. Callahan, et al.: The Tera Computer System, in Proceedings of the International Conference on Supercomputing, S. 1–6, http://www.tera.com/tera/ftp.html, Juni 1990, ACM.

[2] A. Arnold, M. Röth: User's Guide to PARvis, ZAM, Forschungszentrum Jülich GmbH, Zweite Auflage, 1996.

[3] R. Berrendorf: Der FORTRAN-Parser PAFF als wiederverwendbares Modul für Programmier-Tools, Technischer Bericht Jül-Spez-537, ZAM, KFA Jülich, 1989.

[4] R. Berrendorf, M. Gerndt: SVM-Fortran Reference Manual, Technischer Bericht KFA-ZAM-IB-9510, ZAM, Forschungszentrum Jülich, April 1995.

[5] E. Brooks: The Butterfly Barrier, International Journal of Parallel Programming, 15(4):295 – 307, 1986.

[6] D. Engler, G. Andrews, D. Lowenthal: Filaments: Efficient Support for Fine-Grain Parallelism, Technischer Bericht TR 93-13a, Departement of Computer Science, University of Arizona, Februar 1994.

[7] M. Gerndt, R. Berrendorf, et al.: Intel Paragon XP/S - Architecture, Software Environment and Performance, Technischer Bericht IB-9409, ZAM, Forschungszentrum Jülich, 1994.

[8] M. Gerndt, A. Krumme, S. Özmen: Performance Analysis for SVM-Fortran with OPAL, in International Conference on Parallel and Distributed Processing Techniques and Applications, Athens, Georgia, USA, November 1995.

[9] D. Grunwald, S. Vajracharya: Efficient Barriers for Distributed Shared Memory Computers, Diplomarbeit, University of Colorado, September 1993.

[10] A. Gupta, et al..: Comparative Evaluation of Latency Reducing and Tolerating Techniques, in Proceedings of International Symposium on Computer Architecture, Mai 1991.

[11] R. Gupta, C. Hill: A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree, International Journal of Parallel Programming, 18(3):161 – 180, Juni 1989.

[12] J. Hennessy, D. Patterson: Rechnerarchitektur - Analyse, Entwurf, Implementierung, Bewertung, Kapitel 6.7, vieweg Verlag, 1990.

[13] D. Hensgen, R. Finkel, U. Manber: Two Algorithms for Barrier Synchronization, International Journal of Parallel Programming, 17(1):1 – 17, 1988.

[14] K. Hwang: Advanced Computer Architecture, Kapitel 5.4, S. 248 – 256, International Editions, McGraw-Hill, Zweite Auflage, 1993.

[15] Intel Corporation, Paragon System User's Guide, Mai 1995, Nr. 312489-004.

[16] J. Keller, W. Paul, D. Scheerer: Realization of PRAMs: Processor Design, in Proceedings of WDAG'94, 8th International Workshop on distributed Algorithms, Nummer 857 in LNCS, S. 17–27, http://www-wjp.cs.uni-sb.de/sbpram/papers.html, September 1994, Springer Verlag.

[17] K. Li: Shared Virtual Memory on Loosely Coupled Multiprocessors, Dissertation, Yale University, September 1986.

[18] B.-H. Lim, A. Agarwal: Waiting Algorithms for Synchronization in Large-Scale Multiprocessors, ACM Transactions on Computer Systems, 11(3):253 – 294, August 1993.

[19] D. Lowenthal, V. Freeh, G. Andrews: Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing, Technischer Bericht TR 95-14, Department of Computer Science, University of Arizona, Dezember 1995.

[20] M. Mairandes: Virtuell gemeinsamer Speicher mit integrierter Laufzeitbeobachtung, Dissertation Jül-3199, ZAM, Forschungszentrum Jülich, Februar 1996.

[21] J. Mellor-Crummey, M. Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, in Proceedings of TOCS 91, Band 9, S. 21 – 65, Februar 1991.

[22] J. Mellor-Crummey, M. Scott: Fast, Contention-Free Tree Combining Barriers, Technischer Bericht TR 429, University of Rochester, Rice University, Juni 1992.

[23] A. Mylinski-Wiese: Leistungsuntersuchung des iPSC/860 RISC-Knoten-Prozessors: Architekturanalyse und Programmoptimierung, Technischer Bericht Jül-2766, ZAM, Forschungszentrum Jülich, Mai 1993.

[24] N.N.: Interne Dokumentation Cray T3D, Technischer Bericht, Cray Research, Inc., 1992.

[25] N.N.: Interne Dokumentation Cray T3E, Technischer Bericht, Cray Research, Inc., 1995.

[26] N.N.: Power Challenge, Technischer Bericht, Silicon Graphics Inc., 1995.

[27] N.N.: Origin and Onyx2 - Theory of Operations Manual, Silicon Graphics Inc., 1997, Document Number 007-3439-001.

[28] J. Philbin, J. Edler: Very Lightweight Threads, in HIPS'96 Proceedings, S. 95 – 103. IEEE Computer Society, April 1996.

[29] M. Ramachandran, M. Singhal: Decentralized Semaphore Support in a Virtual Shared-Memory System, Journal of Supercomputing, 9(1):51–70, 1995.

[30] K. Raymond: A tree-based algorithm for mutual exclusion, ACM Transaction on Computer Systems, 7(1):61 – 77, Februar 1989.

[31] R.F.Boothe: Evaluation of Multithreading and Caching in Large Shared Memory Parallel Computers, Dissertation, University of California at Berkeley, Juli 1993.

[32] R. Saavedra-Barrera, D. Culler: An Analytical Solution for a Markov Chain Modeling Multithreaded Execution, Technischer Bericht 91-623, University of California, Berkeley, 1991.

[33] R. Saavedra-Barrera, D. Culler, T. van Eicken: Analysis of Multithreaded Architectures for Parallel Computing, Technischer Bericht 90-569, University of California at Berkeley, 1990.

[34] R. Sadourny: The Dynamics of Finite-Difference Models of the Shallow-Water Equations, Journal of Athmospheric Science, 32(4), april 1975.

[35] S. Scott: Synchronization and Communication in the T3E Multiprocessor, in Proceedings ASPLOS-VII, Cambridge, MA, Oktober 1996.

[36] P. Sindhu, J. Frailong, M. Cekleov: Scalable Shared-Memory Multiprocessors, Kapitel Formal Specification of Memory Modules, Kluwer Academic Publishers, Boston (MA), 1992.

[37] M. Singhal: A Taxonomy of Distributed Mutual Exclusion, Journal of Parallel and Distributed Computing, 18:94–101, 1993.

[38] E. Smirni, C. Childers, E. Rosti, L. Dowdy: Thread Placement on the Intel Paragon: Modeling and Experimentation, Technischer Bericht 94-05, Vanderbilt University, Nashville, 1994.

[39] A. Tannenbaum: Moderne Betriebssysteme, Studienbücher der Informatik, Hanser Verlag, Zweite Auflage, 1995.

[40] U. Vahalia: UNIX Internals - The New Frontiers, Kapitel 3, Prentice Hall, 1992.

Danksagung

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

Daher gilt mein erster Dank dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld sowie Herrn Dr. W. Nagel, die mit Ihren Lehrveranstaltungen mein Interesse an der Parallelverarbeitung geweckt und mir anschließend die Möglichkeit zur Anfertigung dieser Arbeit gegeben haben. Mein besonderer Dank für die Betreuung dieser Arbeit gilt Herrn R. Berrendorf, der mir wesentliche Einsichten in das Themengebiet gab, sowohl theoretischer als auch praktischer Natur.

Für die wertvollen Anregungen und Unterstützung bei Problemen möchte ich Herrn Dr. M. Gerndt sowie Herrn H. Bast von der Firma Intel danken. Ebenso gilt mein Dank allen weiteren Mitarbeitern des Lehrstuhls, die am Gelingen der Arbeit beteiligt waren.

Außer Konkurenz ist der Dank bei meinen Eltern, die mir dieses Studium durch ihre finanzielle und ideelle Unterrstützung ermöglicht haben.