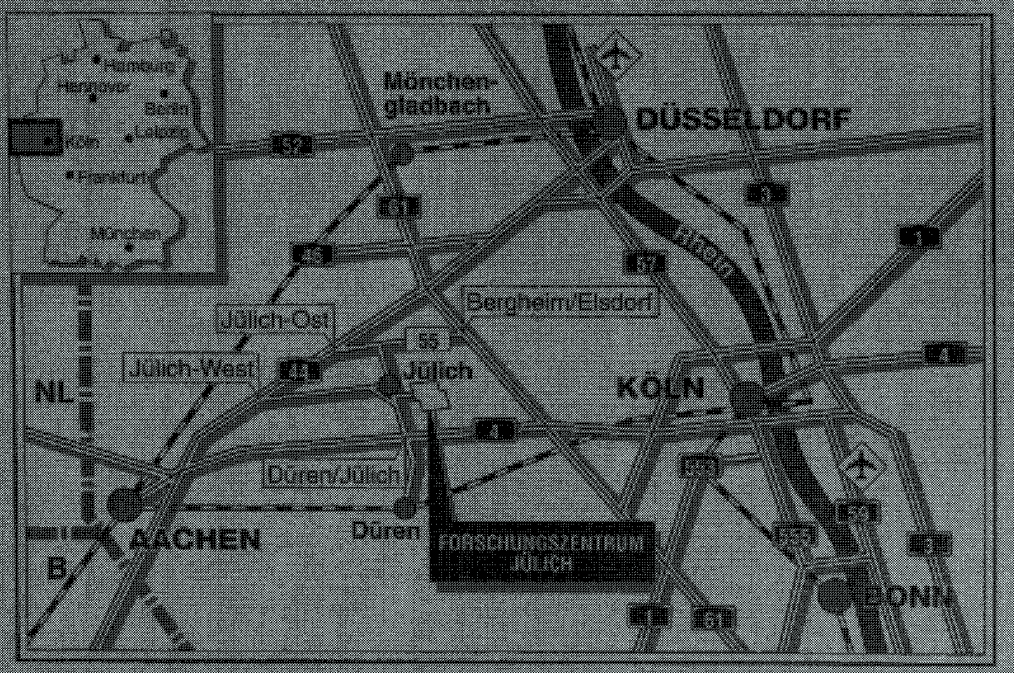
Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Untersuchung von Scheduling-Algorithmen für massiv-parallele Systeme mit virtuell gemeinsamem Speicher

Alfred Arnold

JUL-3143



Berichte des Forschungszentrums Jülich ; 3438
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3438
D 82 (Diss. RWTH Aachen)

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

Kurzfassung

Massiv-parallele Rechnersysteme mit physikalisch verteiltem Hauptspeicher stellen zwar einen kostengünstigen Weg dar, hohe Rechenleistungen zu erzielen, erfordern aber auf der anderen Seite neue Betriebssystemkonzepte, Programmiermodelle und Programmiersprachen, um in Produktionsumgebungen gewinnbringend einsetzbar zu sein. Der virtuell gemeinsame Speicher (SVM) ist ein Programmiermodell, das diesen Anforderungen nach vereinfachter Programmierung von Rechnersystemen mit physikalisch verteiltem Speicher gerecht zu werden versucht. Für einen erfolgreichen Einsatz von SVM sind allerdings Anpassungen der Anwendungssoftware und der Betriebssysteme erforderlich. Während die Optimierung paralleler Anwendungen auf SVM bereits in einer Reihe von Arbeiten behandelt wurde, sind bisher noch vergleichsweise wenig Anstrengungen unternommen worden, parallele Betriebssysteme und insbesondere deren Scheduling-Algorithmen auf eine effiziente Kooperation mit den Hardware-Gegebenheiten in einem SVM-System abzustimmen. In der vorliegenden Arbeit wurde daher untersucht, wie sich bestehende, parallele Betriebssysteme um Komponenten erweitern lassen, die eine effizientere Abarbeitung SVM-basierter Anwendungen ermöglichen.

Die mit Hilfe eines ereignisgesteuerten, im Rahmen dieser Arbeit entwickelten Simulationssystemes durchgeführten, Untersuchungen konzentrierten sich auf zwei Problemkreise: Zum einen wurde untersucht, inwieweit durch lokales Scheduling sequentieller Anwendungen Prozessorleerlaufzeiten einer parallelen Anwendung genutzt werden können, so daß sich die Auslastung eines parallelen Rechnersystems verbessert; zum anderen wurden Algorithmen verglichen, die es einem parallelen Betriebssystem für SVM-basierte Rechnersysteme erlauben, Situationen mit hoher Seitenfehlerrate zu erkennen und korrigierend einzugreifen.

Es hat sich in den Untersuchungen zum einen gezeigt, daß eine asynchrone Ausführung sequentieller Hintergrundlasten unter bestimmten, auch in der Praxis antreffbaren Bedingungen zu einer Verbesserung der Systemauslastung führen kann. Zum anderen ergab ein Vergleich der untersuchten Verfahren zur Reduktion von hohen Seitenfehlerraten, daß alle in dieser Arbeit neu entwickelten Scheduling-Strategien das primär angestrebte Ziel der Seitenfehlerreduktion erreichen. Darüber hinaus sind jedoch bei einer geeigneten, sich dem Programmverhalten dynamisch anpassenden Parameterwahl auch verringerte Laufzeiten für die einzelnen Anwendungen möglich.

Da die entwickelten Algorithmen sich leicht als zusätzliche Komponenten in ein bestehendes, paralleles Betriebssystem einbetten lassen, werden sie eine wertvolle Ergänzung für die Betriebssysteme zukünftiger, auf SVM basierender Parallelrechner darstellen.

Abstract

Analysis of Scheduling Algorithms for Massively-Parallel Systems with Virtually Shared Memory

Massively-parallel systems with a physically distributed main memory present a cost-effective way to achieve high computing powers, but however also require new operating system concepts, programming models, and languages to be effectively usable in production environments. Virtually shared memory (SVM) is a programming model that tries to satisfy the demand for distributed-memory systems with simplified programming. A successful application of SVM however requires adaptions of application software and operating systems. While the topic of application optimization for SVM has already been treated in an number of papers, comparably few attempts have been made to tune parallel operating systems and especially their scheduling algorithms for an efficient cooperation with the given hardware of an SVM-based system. This thesis therefore investigated how existing parallel operating systems can be extended by components that allow a more efficient execution of SVM-based application programs.

The investigations were done with an event-driven simulation system specially developed for this work and focused on two issues: On one hand, it was examined to what extend local scheduling of sequential background load can be used to exploit processor idle times, leading to a higher CPU utilization; on the other hand, algorithms were compared which enable a parallel operating system to detect situations with excessive page fault rates and to initiate corrective arrangements.

The investigations showed on the one hand that under certain conditions, an asynchronous execution of sequential background loads can lead to a higher system utilization. These conditions can also be found on real-world systems. On the other hand, comparisons of the algorithms to reduce page fault rates showed that all algorithms achieved their primary target of page fault reduction. Moreover, reduced execution times for individual applications are possible if the algorithm parameters are chosen in a suitable, preferably adaptive way.

Since the developed algorithms can be easily embedded into an existing parallel operating system, they will prove to be a valuable extension for operating systems of future SVM-based parallel computers.

Inhaltsverzeichnis

1 Motivation **9**

2 Funktionsweise von SVM **13**
2.1 Basisfunktion 13
2.2 Gründe für Performance-Verluste bei SVM 15
2.3 Variationen des Grundalgorithmus 18

3 Übersicht über Scheduling-Verfahren auf parallelen Rechnersystemen **21**
3.1 Generelle Scheduling-Hierarchie 22
3.1.1 Job-Scheduling 22
3.1.1.1 Batch versus interaktiv 22
3.1.1.2 Verfahren zum Batch-Scheduling 23
3.1.1.3 Verfahren zum interaktiven Scheduling 27
3.1.1.4 Allokation und Migration 27
3.1.1.5 Verfahren zur Allokation 27
3.1.1.6 Verfahren zur Lastbalancierung 31
3.1.2 Task-Scheduling 38
3.1.3 Thread-Scheduling 42
3.2 Varianten der Grundstruktur 45
3.3 Existierende parallele Betriebssysteme 48
3.3.1 Mach 48
3.3.2 Chorus 50
3.3.3 Cray UNICOS 52
3.4 Wertung der vorgestellten Verfahren 56

4 Untersuchte Scheduling-Algorithmen **57**
4.1 Ausnutzung der Wartezeiten bei Seitenfehlern 58
4.2 Behandlung von Programmen mit extrem vielen Seitenfehlern 58
4.2.1 Erkennungsverfahren 59
4.2.2 Verfahren zur Unterdrückung 59
4.2.2.1 Migration von Threads 59
4.2.2.2 Verriegeln von Seiten 60
4.2.2.3 Anhalten von Threads 61
4.2.3 Untersuchte Strategien 61
4.2.3.1 Thread-Unterbrechung mit zentraler Steuerung . . . 61
4.2.3.2 Thread-Unterbrechung mit dezentraler Steuerung . . 64

5 Beschreibung des entwickelten Simulationssystems 67
5.1 Basisarchitektur . . . 68
5.2 Struktur der Prozessoren . . . 70
5.2.1 Programmiersprache . . . 71
5.2.2 Abarbeitung auf Maschinenebene . . . 78
5.2.3 Lokale Prozeßverwaltung . . . 79
5.2.4 Speicherverwaltung . . . 81
5.2.5 Steuerung des Nachrichtenverkehrs . . . 83
5.3 SVM-Management . . . 85
5.4 Verbindungsnetzwerk . . . 87
5.5 Globales Scheduling . . . 89
5.6 Migration . . . 91
5.7 Lastspezifikation . . . 92
5.8 Simulator-Konfiguration . . . 93
5.9 Ergebnisauswertung . . . 94

6 Ergebnisse der Untersuchungen 97
6.1 Benutzte Maschinenkonfiguration . . . 97
6.2 Verwendete Testprogramme . . . 98
6.2.1 Matrixmultiplikation . . . 98
6.2.2 LU-Zerlegung . . . 100
6.3 Ausnutzung von Leerlaufzeiten durch sequentielle Hintergrundlasten . 102
6.3.1 Matrixmultiplikation . . . 105
6.3.2 LU-Zerlegung . . . 112
6.4 Behandlung von Programmen mit extrem vielen Seitenfehlern . . . 116
6.4.1 Matrixmultiplikation . . . 116
6.4.1.1 Variante mit zentraler Steuerung . . . 119
6.4.1.2 Variante ohne zentrale Steuerung . . . 125
6.4.2 LU-Zerlegung . . . 129
6.4.3 Adaptive Festlegung von Anhaltezeiten . . . 133
6.5 Strukturelle Einordnung der Ergebnisse . . . 135

7 Zusammenfassung 141

Kapitel 1

Motivation

Die Entwicklung massiv-paralleler Rechner ist durch zwei Trends gekennzeichnet: Zum einen wird durch verfeinerte Entwicklungswerkzeuge die Programmierung dieser Rechner immer weiter erleichtert, zum anderen haben die Betriebssysteme für massiv-parallele Rechnersysteme einen Grad an Stabilität und Funktionalität erreicht, wie er vormals nur bei sequentiellen oder moderat-parallelen Systemen anzutreffen war. Das Resultat dieser Entwicklungen ist, daß massiv-parallele Rechnersysteme den reinen Forschungsbereich zu verlassen beginnen und sich auch kommerziellen Anwendern als preiswerte Antwort auf die überall steigenden Ansprüche an Rechenleistung und Hauptspeicherkapazität anbieten.

Mit diesem erweiterten Kundenkreis tritt aber auch eine neue Anwenderschicht mit einem anderen Anwendungsprofil an Hardware und Software massiv-paralleler Systeme heran: Im Forschungsbereich ist es die Regel, daß eine Anwendung für ein neues paralleles System entweder vollständig neu erstellt wird oder bei der Portierung drastische Änderungen erfährt, um die bestmögliche Leistung auf diesem System zu erreichen. Die Minimierung der für ein solches Vorgehen erforderlichen Arbeitsleistung hat einen vergleichsweise geringen Einfluß auf die Wahl der Mittel und ist deshalb von untergeordneter Bedeutung. Im kommerziellen Bereich ist die Gewichtung der Argumente beim Rechnereinsatz jedoch völlig anders: Hier kommt es auf eine in der Summe möglichst preiswerte Gesamtlösung an. Da Personalkosten bei der Software-Entwicklung einen sehr hohen Anteil haben, ist es durchaus möglich, daß eine weniger effiziente Implementierung gewählt wird, die aber wesentlich schneller zu erstellen und in vielen Fällen auch wesentlich einfacher zu warten ist. Die geringere Leistung dieser Gesamtlösung kann dann meist durch eine entsprechend leistungsfähigere Hardware ausgeglichen werden, deren Mehrpreis durch die eingesparten Personalkosten mehr als kompensiert wird.

In der Praxis bedeutet dies meist, daß eine Portierung bestehender Anwendungen auf einen neuen Rechner mit möglichst geringem Aufwand vorzunehmen ist.

Muß vor einer ersten, funktionierenden portierten Version erst die vollständige Anwendung umgeschrieben werden, so wird von einer Portierung häufig Abstand genommen. Gewünscht ist eine initiale Portierung, die mit möglichst wenig Aufwand vorzunehmen ist und danach eine inkrementelle Verbesserung bis zur gewünschten Performance erlaubt.

In der Praxis bedeuten diese Vorbedingungen für massiv-parallele Systeme einen deutlichen Hinderungsgrund. Dieser Grund ist in der gegenüber den auch im kommerziellen Bereich verbreiteten moderat-parallelen Systemen anderen Architektur zu suchen. Während Vektorrechner mit mehreren, parallel arbeitenden Prozessoren oder symmetrische Multiprozessorsysteme im allgemeinen mit einem physikalisch gemeinsamen Speicher arbeiten, auf den alle Prozessoren gleichberechtigt und ohne Verzögerung zugreifen können, ist der Speicher bei massiv-parallelen Systemen privat und verteilt, d.h. jeder Prozessor besitzt ein eigenes Hauptspeichermodul, auf das er exklusiv Zugriff hat. Die Kommunikation zwischen den einzelnen Prozessoren kann allein durch Austausch von Nachrichten auf einem die Prozessoren verbindenden Netzwerk erfolgen (eine seltene Ausnahme von diesem Paradigma stellen Architekturen wie die Cray T3D/T3E dar [Cr93], die Zugriffe auf entfernte Speichermodule in Hardware auf Nachrichten abbilden und damit die Illusion eines gemeinsamen Adreßraumes herstellen können). Es ist einsichtig, daß ein Programm, welches für eine Architektur mit gemeinsamem Speicher geschrieben wurde, nur mit hohen Aufwand auf ein nachrichtenbasiertes Modell übertragbar ist.

Das Programmiermodell des verteilten Hauptspeichers mit explizitem Message-Passing kommt der Hardware-Struktur massiv-paralleler Systeme am nächsten und läßt sich daher mit vergleichsweise wenig Aufwand effizient implementieren. Aus der Sicht des Anwendungsprogrammierers hat es jedoch besonders für numerische Anwendungen gegenüber einem globalen Adreßraum deutliche Nachteile. Parallelismus existiert in numerischen Anwendungen insbesondere in der Form des Datenparallelismus, bei dem auf großen Datenfeldern vergleichsweise einfache Operationen durchgeführt werden. Diese Form des Parallelismus steht im Gegensatz zum Funktionsparallelismus, bei dem ein Programm in sich abgeschlossene Prozesse aufgeteilt wird, die parallel zueinander ablaufen und sich über den Austausch von Nachrichten synchronisieren. Während Funktionsparallelismus sehr einfach auf einen Parallelrechner mit verteiltem Hauptspeicher abgebildet werden kann, lassen sich datenparallele Anwendungen einfacher auf Parallelrechner mit einem globalen Adreßraum abbilden. Die dabei verwendete Strategie besteht im allgemeinen darin, den Iterationsraum von Schleifen auf die einzelnen Prozessoren aufzuteilen. Die im Schleifenrumpf vorhandenen Speicherzugriffe können in erster Näherung unbetrachtet bleiben, da durch den gemeinsamen Hauptspeicher jedem Prozessor zu jeder Zeit der volle Datensatz zur Verfügung steht; der gemeinsame Hauptspeicher sorgt für die notwendige Kohärenz. Eine Veränderung des Schleifenrumpfes ist nur nötig, wenn eine Änderung der Spei-

cherzugriffsmuster vorgenommen werden soll, um Zugriffskonflikte auf der Ebene der Hardware zu reduzieren [Oed85].

Während also die Parallelisierung einer sequentiellen Anwendung mit einem gemeinsamen Hauptspeicher inkrementell ohne großen initialen Aufwand vorgenommen werden kann, ist die Situation bei einem verteilten Hauptspeicher erheblich schwieriger. Da der Hauptspeicher eines einzelnen Prozessors meist nicht groß genug ist, um den ganzen Datensatz einer Anwendung aufzunehmen, müssen die Daten vom Programmierer auf die einzelnen Prozessoren verteilt werden. Diese Verteilung muß der gewählten Arbeitsverteilung angepaßt werden; in Fällen, in denen sich die Diskretisierung zur Laufzeit automatisch der Struktur des Problemes anpaßt, kann es sogar erforderlich sein, die Datenverteilung während des Programmlaufes dynamisch zu ändern, um Lastungleichgewichte zu vermeiden. Will ein Prozessor nun auf Datenelemente zugreifen, die nicht in seinem lokalen Speicher verfügbar sind, so muß er bei explizitem Message-Passing eine Nachricht verschicken, die diese Daten anfordert. Aber selbst wenn der Datensatz einer Anwendung so klein ist, daß er in den lokalen Speicher eines Prozessors paßt und damit auf jeden Prozessor repliziert werden kann, muß der Programmierer zusätzlichen Code einfügen, der die Kohärenz der lokalen Kopien garantiert und so verhindert, daß ein Prozessor mit veralteten Daten arbeitet.

Anhand dieser Ausführungen wird klar, daß gerade für den kommerziellen Bereich das Programmiermodell des verteilten Speichers bisher in vielen Fällen einen erheblichen Hinderungsgrund für den Einsatz massiv-paralleler Rechner darstellt [HoNa95]. Diesem Argument steht aber entgegen, daß ein Parallelrechner mit verteiltem Hauptspeicher deutlich billiger zu realisieren ist als ein Parallelrechner vergleichbarer Leistung mit gemeinsamem Speicher. Ein Speicher, der von mehreren Prozessoren quasi-gleichzeitig angesprochen werden kann, ist technisch sehr aufwendig aufzubauen und findet in der Praxis momentan seine Grenze bei etwa dreißig Prozessoren; im Gegensatz dazu sind bereits Parallelrechner mit verteiltem Speicher verfügbar, die aus einigen tausend Prozessoren bestehen. Ab einer gewissen Grenze für die benötigte Rechenleistung sind nur noch Parallelrechner mit verteiltem Speicher als Basis für ein Hochleistungsrechnen denkbar. Durch die fortschreitende Entwicklung der Technik verschiebt sich diese Grenze zwar weiter nach oben, gleichzeitig wachsen aber auch die benötigten Rechenleistungen für die Lösung von großen Problemen.

Es gibt eine Reihe von Ansätzen, um dieses Dilemma zu überwinden und dem Programmierer auch auf einem Parallelrechner mit physikalisch verteiltem Speicher einen globalen Adreßraum anzubieten. Die Realisationsvarianten können vereinfacht darin unterschieden werden, an welcher Stelle die Auflösung globaler Adressen erfolgt. Bei einer *compilerbasierten* Realisierung werden die globalen Referenzen anhand einer automatisch erzeugten oder vom Programmierer vorgegebenen Datenverteilung vom Compiler in explizites Message-Passing umgesetzt. Der Vorteil dieser

Variante ist, daß das erzeugte Programm ohne den Overhead eines Laufzeitsystems zur Verwaltung und Speicherung der Datenverteilung auskommt; der Nachteil ist jedoch, daß durch die Festlegung der Datenverteilung zur Zeit der Übersetzung dynamische oder irreguläre Verteilungen nur schwierig oder überhaupt nicht abzubilden sind. Ein compilerbasierter globaler Adreßraum eignet sich daher besonders für Anwendungen mit regelmäßigen Datenzugriffsmustern. Die bekannteste compilerbasierte Realisierung ist das *High-Performace-Fortran* (HPF) [HPF94], ein herstellerübergreifender Standard für eine Erweiterung der Fortran-90-Sprache um parallele Konstrukte.

Leider existieren aber in der Praxis auch eine Reihe von Anwendungen, die in ihren Zugriffsmustern so irregulär sind, daß sie sich nicht von einem Compiler vorab analysieren lassen. Soll für derartige Anwendungen ein globaler Adreßraum bereitgestellt werden, so kann dies nur zur Laufzeit erfolgen. Dem von Kai Li vorgeschlagenen virtuell gemeinsamem Speicher (*Shared Virtual Memory*, SVM)[Li86] liegt ein solcher Ansatz zugrunde, der in Kapitel 2 näher beschrieben werden soll. Da der virtuell gemeinsame Speicher zur Laufzeit realisiert wird, ist es für eine gute Performance wichtig, daß die Implementation effizient in die Betriebssystemumgebung eingebettet ist und mit ihr kooperiert. Insbesondere betroffen ist davon das Scheduling-System: Da unter SVM die einzelnen Prozessoren dynamisch Speicherseiten austauschen, können falsche Scheduling-Entscheidungen zu einer empfindlichen Störung dieses Datenverkehrs und damit zu einer deutlichen Performance-Verschlechterung führen.

Aufgrund der Neuheit des Konzeptes von SVM und der Tatsache, daß SVM bisher noch keine weite Verbreitung in kommerziellen Parallelrechnersystemen gefunden hat, sind bisher nur einige wenige Arbeiten über das Laufzeitverhalten von SVM-Programmen in einer Multiprogramming-Umgebung durchgeführt worden. Demzufolge liegen bisher auch keine Erkenntnisse über Auslegungskriterien für Scheduler auf SVM-basierten Rechnersystemen vor. Thema der hier vorgestellten Arbeit ist daher die Untersuchung der Auswirkungen verschiedener Scheduling-Strategien auf den Ablauf von parallelen Programmen, die SVM als Programmiermodell nutzen.

Nach einer genaueren Einführung in die Arbeitsweise von SVM wird zu diesem Zweck im dritten Kapitel ein Überblick über gängige Scheduling-Verfahren auf parallelen Rechnersystemen gegeben. In Kapitel 4 werden die im Rahmen dieser Arbeit untersuchten Verfahren vorgestellt, und Kapitel 5 beschreibt die zur Analyse verwendete Simulationsumgebung, die im Rahmen dieser Arbeit erstellt wurde. Kapitel 6 faßt die mit dem Simulationssystem erzielten Ergebnisse zusammen. Das abschließende Kapitel gibt eine Zusammenfassung der in dieser Arbeit erzielten Ergebnisse.

Kapitel 2

Funktionsweise von SVM

Das Konzept des virtuell gemeinsamen Hauptspeichers (**S**hared **V**irtual **M**emory, SVM) basiert darauf, einem Anwendungsprogramm auf einem Parallelrechner zur Laufzeit einen globalen Adreßraum zur Verfügung zu stellen, auch wenn es sich um ein paralleles System mit verteiltem Hauptspeicher handelt. SVM wurde erstmals in der Arbeit von Kai Li [Li86] vorgestellt. Das Konzept des virtuell gemeinsamen Hauptspeichers ist seitdem bereits mehrfach in verteilte Betriebssysteme implementiert worden, so z. B. in Mach [OSF92] und Chorus [AAO92]. SVM ist für die Anwendung vollständig transparent, d.h. die Anwendung muß zunächst keine Rücksicht auf die physikalisch andere, verteilte Speicherstruktur nehmen. Unter ungünstigen Bedingungen kann es allerdings zu extremen Performance-Einbußen kommen. Deshalb ist ein detailliertes Verständnis der Eigenschaften von SVM und der durch dieses Konzept verursachten Effekte für eine effiziente Programmierung derartiger Systeme unabdingbar. Das Feld der dafür notwendigen Untersuchungen und Anpassungen beschränkt sich nicht nur auf die Programmierung von Anwendungen; eine Beurteilung von in Betriebssystemen verwendeten Scheduling-Algorithmen in Bezug auf ihre Eignung für SVM ist ebenfalls notwendig.

2.1 Basisfunktion

Die Grundidee von SVM beruht darauf, den aus der Sicht des Programmierers globalen Adreßraum in Seiten gleicher Größe aufzuteilen, die dann zwischen den Prozessoren nach vorgegebenen Regeln wandern dürfen. Die Tragfähigkeit dieses Konzepts beruht auf der Annahme, daß die auf den einzelnen Prozessoren ablaufenden Teile eines parallelen Programmes eine Datenlokalität aufweisen, so daß sie über längere Zeit auf der gleichen Menge von Seiten arbeiten und sich ihre Seitenmengen nur geringfügig überlappen. Wenn eine von einem Prozessor benötigte Seite nicht vorhanden sein sollte, wird ein *Seitenfehler* ausgelöst, und das SVM-Laufzeitsystem

transferiert bzw. kopiert diese Seite vom bisherigen Nutzer zum anfordernden Prozessor. Nach diesem Vorgang kann der auf diesem Prozessor unterbrochene Programmablauf wieder fortgesetzt werden.

Dieses Konzept hat eine große Ähnlichkeit mit dem auf sequentiellen Rechnern bekannten und dort erfolgreich betriebenen *Demand Paging*, bei dem einem einzelnen Prozeß ein virtueller Hauptspeicher vorgespiegelt wird, von dem nur die momentan benötigten Seiten im Hauptspeicher tatsächlich vorliegen und der Rest auf einem Sekundärspeicher gelagert wird. Heute sind SVM-Implementierungen meist in Software realisiert, die die auf modernen Mikroprozessoren vorhandene, für *Demand Paging* konstruierte Speicherverwaltungseinheit (MMU) nutzen (z.B. ASVM auf Intel Paragon [Zei93]). Im Unterschied zum virtuellen Speicher wandern die Seiten bei SVM jedoch nicht zwischen realem Haupt- und Sekundärspeicher, sondern *zwischen* den verteilten Hauptspeichern der einzelnen Prozessoren. Um zu verhindern, daß Prozessoren mit inkonsistenten Daten arbeiten oder Seiten verloren gehen, existiert ein Kohärenzprotokoll, dessen Einhaltung von einer speziellen Instanz, dem *SVM-Manager*, überwacht wird. Abbildung 2.1 zeigt die Behandlung eines Seitenfehlers durch den Manager anhand eines Beispieles:

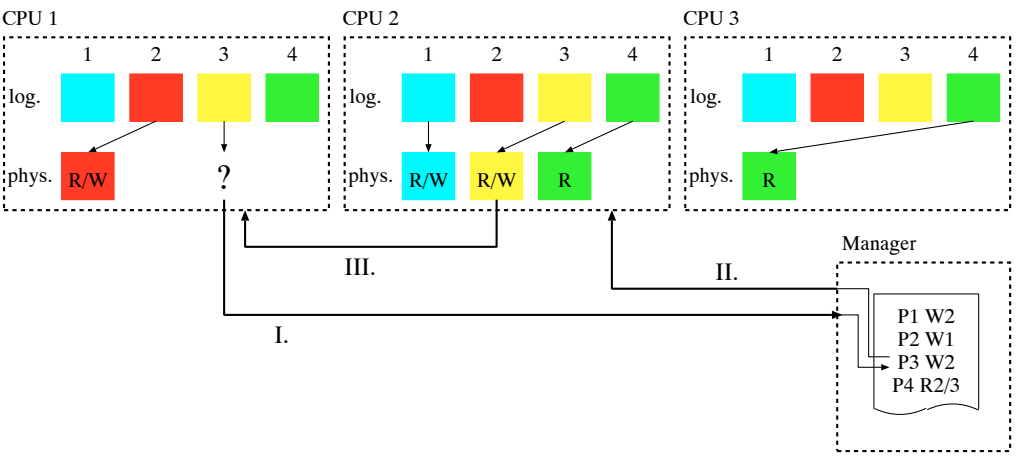


Abbildung 2.1
Arbeitsweise eines SVM-Systems

Im Beispiel greifen drei Prozessoren auf einen globalen Adreßraum zu, der aus 4 Seiten besteht. CPU 1 versucht, auf ein Datum zuzugreifen, das auf der momentan lokal nicht verfügbaren Seite 3 liegt. Dieser Seitenfehler wird dem SVM-Manager gemeldet (Schritt I), der anhand seiner Tabelle ermittelt, daß Seite 3 sich momentan bei CPU 2 befindet. Deswegen schickt der Manager an CPU 2 eine Nachricht, Seite 3 an CPU 1 zu schicken (Schritt II). Mit dem Transfer der Seite (Schritt III) ist der Seitenfehler vollständig behandelt worden, der Manager muß jetzt noch seine Tabelle entsprechend korrigieren.

Wie an der Beispielverteilung deutlich wird, unterscheidet SVM, ob ein Prozessor eine Seite momentan mit Lese- oder Schreib-/Leseberechtigung besitzt. Der Grund für diese Unterscheidung ist, daß es in vielen Programmen Datenfelder gibt, auf die nur lesend zugegriffen wird. In einer solchen Situation ist es erwünscht und auch sicher, auf unterschiedlichen Prozessoren jeweils eine Kopie dieser Seite zu haben, da sich beim Auslesen der Daten keine Inkonsistenzen einstellen können. Bei der Behandlung von Seitenfehlern ist demzufolge zwischen Schreib- und Lesefehlern zu unterscheiden. Je nach vorheriger Konstellation kann es dann für den Manager erforderlich sein, vor der eigentlichen Aufforderung zum Seitentransfer zusätzliche Steuernachrichten zu versenden: Besitzen mehrere Prozessoren eine Lesekopie einer Seite und fordert ein Prozessor diese Seite zum Schreiben an, so muß der Manager zuerst alle anderen Prozessoren auffordern, ihre Lesekopien zu invalidieren, bevor der Anforderer das Schreibrecht erhält. Umgekehrt muß der Manager ein vorhandenes Schreib-/Leserecht für eine Seite auf ein Leserecht reduzieren, bevor ein weiterer Prozessor eine Lesekopie der Seite erhalten kann.

Es ist nicht zwingend notwendig, die Verwaltung aller Seiten von einem Manager vornehmen zu lassen, da die Verwaltungsoperationen für verschiedene Seiten logisch voneinander völlig unabhängig sind. Für eine Skalierbarkeit dieses Konzeptes ist es im Gegenteil unabdingbar, daß die Management-Funktionen auf mehrere Prozessoren verteilt werden. Falls die Zugriffsfrequenz auf einzelne Speicherbereiche während des Programmablaufs stark variiert, kann es sogar sinnvoll sein, die Management-Funktionen während des Programmablaufes dynamisch wandern zu lassen, um so einen Lastausgleich zu erzielen.

2.2 Gründe für Performance-Verluste bei SVM

SVM in der bisher vorgestellten Variante hat den Vorteil, für die Anwendung vollständig transparent zu sein, d.h. der vom SVM-Laufzeitsystem zur Verfügung gestellte globale Adreßraum verhält sich logisch völlig äquivalent zu dem physikalisch gemeinsamen Speicher eines moderat parallelen Systems. Angenehm ist weiterhin, daß vom Anwender keine explizite Datenverteilung vorgegeben werden muß; diese stellt sich durch den Programmablauf von selbst ein. Es ist klar, daß auf einem parallelen System mit physikalisch verteiltem Speicher mit SVM in der Praxis nie die Effizienz eines reinen Message-Passing-Programmes erreicht werden kann, da zur Auflösung eines Seitenfehlers immer drei Nachrichten erforderlich sind und immer eine ganze Seite übertragen wird, auch wenn vielleicht nur ein einziges Datum von dieser Seite benötigt wird. Die Praxis zeigt aber auch, daß bei vielen Anwendungen nach Optimierungen des Quell-Codes in vergleichsweise kurzer Zeit eine Performance erreicht werden kann, die in der gleichen Größenordnung liegt wie für reine Message-Passing-Programme [GeBe95]; mit SVM tauscht man also einen mäßigen Verlust an

Laufzeiteffizienz gegen eine erheblich verkürzte Implementationszeit ein. Optimierungen sind allerdings bei der Portierung bestehender Programme immer notwendig, denn die funktionale Äquivalenz zum physikalisch gemeinsamen Speicher bezieht sich nicht auf das Zeitverhalten; insbesondere bei ungünstigen Zugriffsmustern kann es zu Konfliktsituationen kommen, die die Laufzeit eines Programmes um Größenordnungen verschlechtern. Ursache hierfür ist meist ein als *false sharing* bekannter Effekt, der dadurch verursacht wird, daß der SVM-Manager Kohärenz nicht für einzelne Datenworte, sondern nur für ganze Seiten garantieren kann; dies führt dazu, daß eine Seite immer nur für einen Prozessor mit Schreibrecht zur Verfügung steht, auch wenn die Datenbereiche, die verschiedene Prozessoren auf dieser Seite bearbeiten wollen, sich gar nicht überlappen (siehe Abbildung 2.2): Die Seite wandert ständig zwischen den beiden Prozessoren hin und her, was den Programmablauf drastisch verlangsamt.

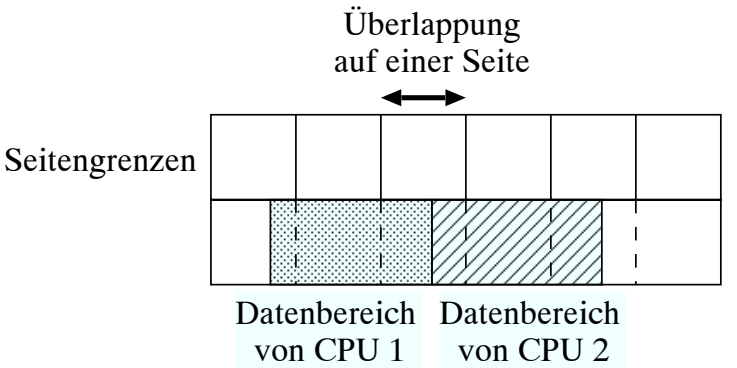


Abbildung 2.2
Beispiel für *False Sharing*

Die Wahrscheinlichkeit, daß *false sharing* auftritt, wächst naturgemäß mit der Seitengröße, weshalb Software-Implementationen von SVM, die üblicherweise mit einer durch die Hardware vorgegebenen Seitengröße von einigen Kilobyte arbeiten müssen, von diesem Effekt stärker betroffen sind als Hardware-Implementationen von SVM wie beispielsweise auf der KSR-1; da solche Rechner von vornherein auf SVM ausgelegt wurden, konnte dort eine für SVM günstigere Seitengröße von einigen -zig Byte vorgesehen werden. Die Seitengröße darf bei SVM-Systemen allerdings auch nicht beliebig klein werden, da sonst der Verwaltungsaufwand im Manager zu hoch wird. Da sich *false sharing* also nie ganz ausschließen läßt, muß man manuell Modifikationen der Anwendung vornehmen, um diesen Effekt zu vermeiden. Eine Möglichkeit ist die Ausrichtung von Datenfeldern auf Seitengrenzen, so daß keine Überlappung eintritt. Wird z.B. bei einem zweidimensionalen Feld jede Zeile von einem anderen Prozessor bearbeitet, so kommt es bei dichtestmöglicher Ablage der Feldelemente zu einer Überlappung, falls eine Zeile des Feldes nicht zufällig ein mehr-

faches der Seitengröße lang ist. Läßt man aber nach jeder Zeile im Hauptspeicher so viel Platz, daß die folgende Zeile wieder auf einer Seitengrenze beginnt, so existiert keine Überlappung mehr. Dieses Verfahren der Ausrichtung (*padding*) ist für ein zweidimensionales Feld in Abbildung 2.3 dargestellt.

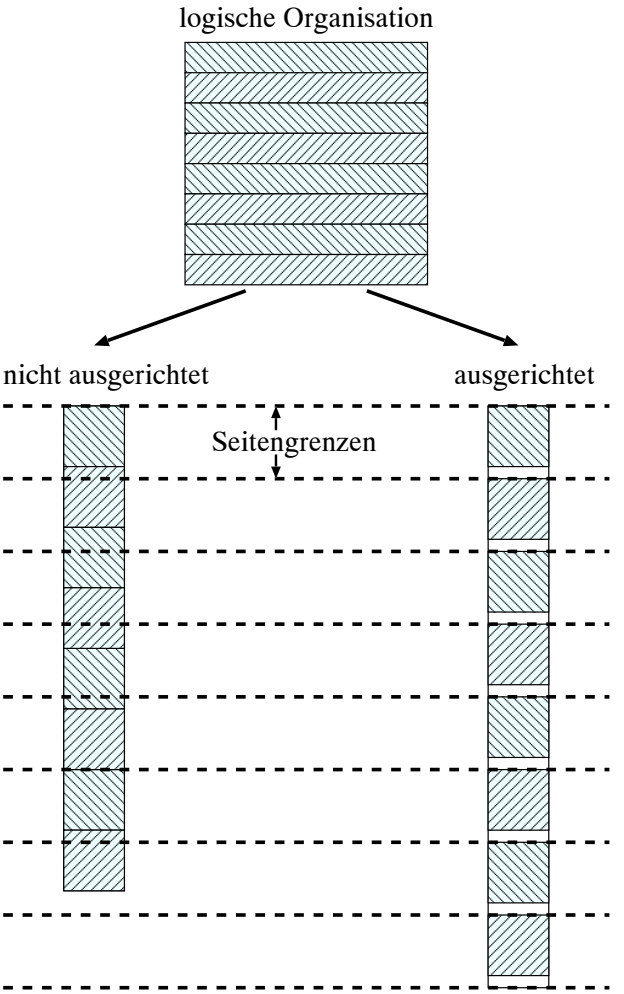


Abbildung 2.3
Ausrichtung eines zweidimensionalen Feldes auf Seitengrenzen

Eine Ausrichtung auf Seitengrenzen muß im allgemeinen durch den Compiler zur Verfügung gestellt werden, da sonst nicht die Ausrichtung des Feldanfangs auf eine Seitengrenze garantiert werden kann. Insbesondere bei kleinen Feldern kann aber der durch Ausrichtung entstehende Speicherverschnitt inakzeptabel groß werden, so daß andere Verfahren zur Vermeidung von *false sharing* eingesetzt werden müssen. Eine Weiterentwicklung dieser Ausrichtung stellen z.B. die in [Li94] vorgestellten Verfahren dar, die darauf abzielen, die Verteilung von Feldelementen auf

Speicherseiten den Zugriffsmustern des Programmes in einer Weise anzupassen, daß auf einer Seite liegende Daten möglichst nur noch von einem Prozessor bearbeitet werden. Diese Verfahren basieren auf angepaßten Funktionen, die Feld-Indices in lineare Speicheradressen umrechnen und anhand des Datenzugriffsmusters eines Programmes bestimmt werden müssen.

2.3 Variationen des Grundalgorithmus

Eine weitere Möglichkeit zur Vermeidung von *false sharing* besteht darin, die strengen Konsistenzbedingungen des Standard-Protokolls zu lockern, das auch unter dem Namen *strong coherence* bekannt ist und in jeder Situation garantiert, daß ein Lesezugriff auf ein Datum im globalen Adreßraum immer den zuletzt geschriebenen Wert liefert. Wenn jedoch bekannt ist, daß Prozessoren in einem bestimmten Programmabschnitt nur auf disjunkte Teile einer Seite zugreifen, so kann zugelassen werden, daß zeitweise mehrere Kopien einer Seite mit Schreibrecht existieren. Diese Variante wird im Gegensatz zum Standardprotokoll als *weak coherence* bezeichnet und ist in Abbildung 2.4 beispielhaft dargestellt.

Ausgehend von einer Originalseite erhalten mehrere Prozessoren eine Kopie mit Schreibrecht, die sie nun unabhängig voneinander verändern. Am Ende der Sequenz erhält ein zusätzlicher Prozeß die Originalseite zusammen mit allen modifizierten Seiten und kann durch Vergleich mit der Originalseite feststellen, welcher Prozessor welche Teile verändert hat. Daraus kann dann die neue Seite konstruiert werden. Sowohl Anfang als auch Ende des Programmbereiches, für den schwache Kohärenz verwendet werden soll, müssen durch Direktiven im Programm markiert werden. Außerdem muß die Anwendung garantieren, daß sich die modifizierten Bereiche nicht überlappen. Dies bedeutet, daß SVM unter Verwendung schwacher Kohärenz für die Anwendung nicht mehr transparent ist. Da außerdem die Zusammenführung der einzelnen Seiten einen erheblichen Aufwand darstellt, wird schwache Kohärenz vergleichsweise selten verwendet.

In der Praxis häufiger anzutreffen sind hingegen Varianten des SVM-Protokolls, die den - durch den separaten SVM-Manager verursachten - zusätzlich induzierten Aufwand zu eliminieren suchen. Wenn es gelingt, Funktionen des Managers und des Besitzers (d.h. des Prozessors, der für das Verschicken einer Seite zuständig ist) in einem Prozessor zusammenzulegen, kann von den drei Nachrichten, die üblicherweise zu der Auflösung eines Seitenfehlers erforderlich sind, eine eingespart werden. In [Zei93] werden Protokolle beschrieben, die bei einem Seitenfehler die Management-Aufgaben auf den anfragenden Prozessor übergehen lassen. Eine Aktualisierung der Information über den Besitzer/Manager findet nicht bei jeder Seitenwanderung statt; vielmehr besitzt jeder Prozessor eine Liste über vermutliche

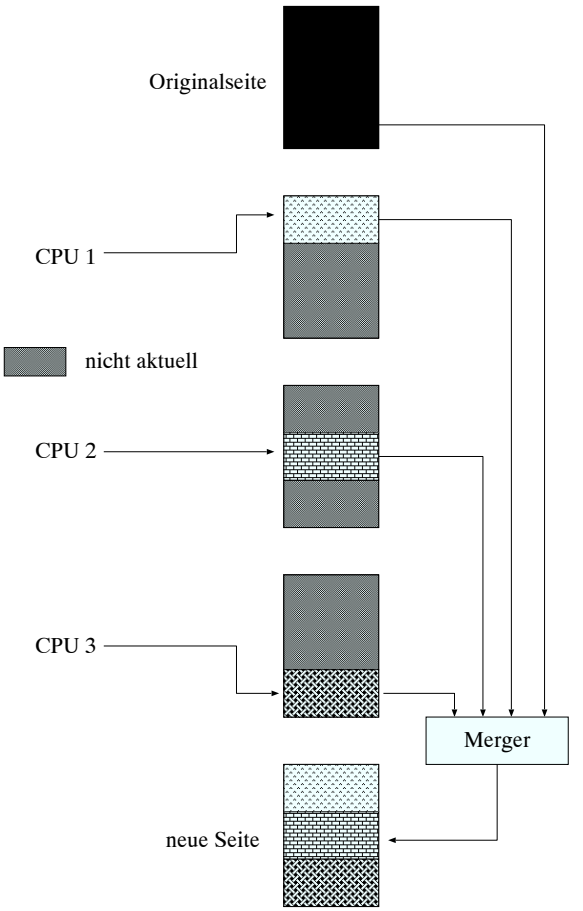


Abbildung 2.4
Arbeitsweise von SVM bei schwacher Kohärenz

Besitzer von Seiten. Da ein Prozessor beim Verschicken einer Seite gespeichert hat, an welchen Prozessor er die Managementaufgaben weitergereicht hat, ist er in der Lage, ankommende Anfragen korrekt weiterzureichen. Gleichzeitig kann er seine eigenen Tabellen korrigieren, da er mit dieser Anfrage die Information bekam, daß der anfragende Prozessor der neue Manager wird. Zu Beginn ist die Liste eines Prozessors über vermutliche Besitzer leer, weshalb beim ersten Fehler auf dieser Seite eine Broadcast-Anfrage an alle anderen Prozessoren verschickt wird. Vorteilhaft an diesem Verfahren ist, wie in [Zei93] hervorgehoben wird, daß ein Knoten lokal keine Daten für Seiten speichern muß, auf die er niemals zugreift. Im Gegensatz dazu muß beim Standardverfahren jeder Knoten eine Tabelle über die für die Seiten zuständigen Manager führen. Nachteilig gegenüber dem Standardverfahren ist das höhere Nachrichtenaufkommen durch die Broadcast-Operationen sowie möglicherweise längere Antwortzeiten, wenn eine Anfrage erst eine Kette von vorherigen Besitzern durchlaufen muß.

Diese Verbesserungen des SVM-Konzeptes können zwar die Effizienz in bestimmten Fällen verbessern, ändern jedoch nichts an der Tatsache, daß ein Betriebssystem bei seinen Scheduling-Entscheidungen die Schwachpunkte von SVM berücksichtigen muß. Dazu gehört naturgemäß zum einen, Entscheidungen so zu treffen, daß die sich in einem gut auf SVM optimierten Programm einstellende Seitenverteilung nicht zerstört wird. Zum anderen ist es durch die Kooperation der Knoten untereinander auch wichtig, das Scheduling so zu koordinieren, daß der Seitenaustausch möglichst reibungslos vonstatten geht. Wird einem einzelnen Prozeß einer parallelen Anwendung z.B. der Prozessor entzogen, so kann dieser u.U. keine Seitenanfragen mehr bedienen, was damit auch den Ablauf auf anderen Prozessoren behindert. Umgekehrt können Statistiken über den Datenverkehr zwischen den Prozessoren, die vom SVM-Laufzeitsystem zur Verfügung gestellt werden, wertvolle Informationen für das Scheduling einer Anwendung liefern. Um festzustellen, welche Informationen dies sein können, ist es notwending, die auf parallelen Systemen zur Anwendung kommenden Scheduling-Prinzipien zu kennen.

Kapitel 3

Übersicht über Scheduling-Verfahren auf parallelen Rechnersystemen

Die Ablaufplanung von parallelen Programmen auf Parallelrechnern hat sich als sehr komplexe Aufgabenstellung erwiesen, und mit einer Vielzahl von unterschiedlichen Verfahren, die die jeweiligen Anforderungen und Randbedingungen im Einzelfall berücksichtigen, gibt es bereits eine Reihe von spezialisierten Resultaten. So hängen diese Strategien beispielsweise vom erwarteten durchschnittlichen Anwendungsprofil ab und nehmen auf Zugangsart zum Rechnersystem (Batch oder Interaktiv) oder in Echtzeitsystemen auf die vom Scheduler einzuhaltenden Antwortzeiten Rücksicht. Da in dieser Arbeit primär Scheduling-Strategien für massiv-parallele Systeme mit Anwendungsschwerpunkt im numerischen Bereich behandelt werden sollen, können Echtzeit-Systeme hier ausgespart werden. Ebenso werden Betriebssysteme, die im wesentlichen sequentielle Programme abarbeiten und die mehrere Prozessoren nur zur Durchsatzsteigerung nutzen, nicht gesondert berücksichtigt. Dies bedeutet natürlich nicht, daß ein Scheduler für ein massiv-paralleles System nicht in der Lage sein sollte, sequentielle Programme effizient auszuführen; in einem massiv-parallelen Rechner mit Multiprogramming ist ein störungsfreies Nebeneinander von parallelen und sequentiellen Anwendungen ein wichtiger Gesichtspunkt; von einem Scheduler wird im folgenden aber verlangt, daß er die besonderen Bedürfnisse paralleler Programme berücksichtigt, indem er ein paralleles Programm nicht nur als eine Ansammlung von sequentiellen Teilen auffaßt, die voneinander unabhängig behandelt werden können, sondern auf die durch die parallele Ausführung entstehenden Randbedingungen geeignet eingeht und sie in die Ablaufplanung mit einbezieht.

Da das Konzept des virtuell gemeinsamen Speichers noch vergleichsweise neu ist und bisher nur in einem kommerziellen Rechner (der KSR-1/2) angewendet wurde, finden sich in der Literatur nur sehr wenige Arbeiten über Scheduling unter SVM;

in diesem Kapitel werden daher bevorzugt Algorithmen beschrieben, die entweder für Parallelrechner mit physikalisch gemeinsamem oder ausschließlich privatem Arbeitsspeicher konzipiert wurden. Da SVM Eigenschaften von beiden Programmiermodellen beinhaltet, lassen sich einige dieser Algorithmen, natürlich in modifizierter Form und auf die geänderten Randbedingungen angepaßt, auch auf SVM-Systeme anwenden.

3.1 Generelle Scheduling-Hierarchie

Den meisten heute verfügbaren parallelen Rechnern und ihren Betriebssystemen ist gemeinsam, daß die Gesamtaufgabe des Schedulings in mehrere, hierarchisch geordnete Ebenen aufgeteilt ist. Dies ist im allgemeinen notwendig, da sich mit einer Entscheidungsebene die Forderungen nach Skalierbarkeit und globaler Beherrschbarkeit des Systems nicht vereinbaren lassen. Herausgebildet hat sich dabei eine Strukturierung in drei Ebenen, nämlich das *Job*-, *Task*- und *Thread*-Scheduling.

3.1.1 Job-Scheduling

Das Job-Scheduling stellt die oberste Ebene der Scheduling-Hierarchie dar. Sein Aufgabenbereich umfaßt Operationen, für die ein Gesamtüberblick über das vollständige Parallelrechnersystem erforderlich ist, also Annahme und Beendigung von Jobs, die Auswahl von geeigneten Anfangsverteilungen der parallelen Programmteile auf physikalische Knoten sowie die Überwachung einer globalen Lastbalance. Ihm obliegt also die Entscheidung, *welche* Programme *wann* auf *welchen Prozessoren* ausgeführt werden. Je nach Auslegung des Parallelrechnersystems kann die Strategie eines Job-Schedulers also auf die Erreichung gänzlich anderer Optimierungskriterien ausgerichtet sein.

3.1.1.1 Batch versus interaktiv

Da Parallelrechner zumeist möglichst gut ausgenutzt werden sollen, werden viele Parallelrechner auch heute noch ganz oder teilweise im *Batch*-Betrieb betrieben. Bei dieser Betriebsart erreichen die Arbeitsaufträge das Parallelrechnersystem in Form von in sich abgeschlossenen Arbeitsaufträgen, die ohne weitere Eingriffe des Anwenders abgearbeitet werden. Batch-Systeme werden üblicherweise durch ein System von Warteschlangen gesteuert, die es zum einen ermöglichen, momentan nicht bearbeitbare Jobs zur späteren Ausführung aufzubewahren, zum anderen kann in diesem System von Warteschlangen aber auch eine größere Menge von Jobs akkumuliert

werden, bevor für diese dann ein gemeinsamer Ablaufplan erstellt wird. Ein Beispiel für diese Variante des Batch-Scheduling wäre ein Parallelrechner, der tagsüber interaktiven Zugang bietet, aber auch gleichzeitig die Submission von Batch-Jobs erlaubt, die dann in der folgenden Nacht abgearbeitet werden sollen. Ist in der Nachtphase kein interaktiver Zugriff mehr möglich, können auch keine weiteren Jobs zur Ausführung mehr submittiert werden. Dies bedeutet, daß zu Beginn der Nachtperiode alle auszuführenden Jobs mit ihren Ressourcenanforderungen (maximale Rechenzeit sowie Prozessoranzahl) bekannt sind und der Scheduler zu Beginn der Batch-Periode alle Jobs einmalig verplanen kann. Aufgrund der Singularität dieses Vorganges ist eine etwas längere Laufzeit für einen besseren Algorithmus vertretbar. Preemption, d.h. die Unterbrechung eines Programmes, um die Prozessoren einem anderen Programm zur Verfügung zu stellen, ist im allgemeinen auch nicht erforderlich, da diese nur für eine gleichmäßige und gerechte Rechenzeitverteilung im interaktiven Betrieb erforderlich ist; wird ein Job jedoch nachts im Batch abgearbeitet, so ist es für den Anwender völlig unerheblich, zu welchen genauen Zeitpunkt in der Nacht der Job das System verläßt, solange er am nächsten Tag fertiggestellt ist.

3.1.1.2 Verfahren zum Batch-Scheduling

Sofern alle auszuführenden Jobs mit ihren Laufzeiten vorab bekannt sind, ist es theoretisch immer möglich, die Kombination zu ermitteln, die eine mimimale Gesamtlaufzeit ergibt. Die Bestimmung dieser optimalen Kombination ist allerdings *NP-hart*[GaJo79], d.h. es ist kein Algorithmus bekannt, der einen optimalen Ablaufplan berechnen kann, ohne dabei alle möglichen Kombinationen zu untersuchen. Die Möglichkeit, alle Kombinationen miteinander zu vergleichen, wird für größere Prozessor- und Job-Zahlen aber schnell unrealistisch, da die Zahl der zu testenden Möglichkeiten exponentiell steigt. Es wurden daher Algorithmen gesucht, die mit abschätzbarem Zeitaufwand eine Lösung finden, deren Güte bezüglich eines bestimmten Kriteriums ebenfalls abschätzbar ist.

Einen vergleichsweise allgemeinen Ansatz verfolgen Schwiegelshohn, Ludwig et al. in ihrer Serie von Arbeiten [TLWF94, TSWY94, SLWT] zur Minimierung des sogenanten *Makespans*, d.h. der Zeit, nach der der letzte Job das parallele Rechnersystem verläßt, bzw. zur Minimierung der mittleren Antwortzeit für einen einzelnen Job. Die Antwortzeit für einen einzelnen Job ist dabei definiert als Zeit vom Beginn der Abarbeitung des *ersten* Jobs bis zur Fertigstellung des betreffenden Jobs, da davon ausgegangen wird, daß alle Jobs beim Scheduling-Vorgang vorliegen. Voraussetzung für den Algorithmus sind Programme, denen vom Scheduler eine variable Zahl von Prozessoren zugewiesen werden kann. Die Grundidee besteht darin, den einzelnen Jobs maximal die Hälfte aller verfügbaren Prozessoren zuzuweisen. Dies

verschlechtert zwar evtl. die Ausführungszeit einzelner Jobs maximal um einen Faktor 2, verhindert aber andererseits, daß große Jobs den kompletten Rechner blockieren und damit die Antwortzeiten für kleinere Jobs überproportional verschlechtern. Im schlimmsten Falle verlängert sich der Makespan gegenüber einem Scheduler, der die einzelnen Jobs sequentiell mit maximaler Prozessorzahl laufen läßt, um einen Faktor 2. Im allgemeinen ist dieser Faktor jedoch kleiner, da die meisten parallelen Anwendungen einen sublinearen Speedup besitzen, d.h. ihre Laufzeit sinkt nicht umgekehrt proportional zur steigenden Prozessorzahl, sondern schwächer. Umgekehrt bedeutet dies, daß sich die Laufzeit eines Programmes bei Halbierung der Prozessorzahl weniger als verdoppelt (der genaue Faktor ist natürlich programmabhängig).

Um eine möglichst niedrige mittlere Antwortzeit zu erreichen, reicht die Begrenzung auf maximal $p/2$ Prozessoren aber noch nicht aus. Es muß zusätzlich eine Reihenfolge der Jobs festgelegt werden, da die Antwortzeit für einen einzelnen Job sowohl von der Startzeit als auch von der zugewiesenen Zahl von Prozessoren abhängig ist. Um eine Kombination von Job-Reihenfolge und Prozessorzuteilung zu ermitteln, die eine möglichst geringe mittlere Antwortzeit erzielt, wird in einem ersten Schritt für jeden Job i die Prozessorzahl ermittelt, für die sich eine minimale Antwortzeit ergibt, falls Job i die Position s in der Job-Liste hat. Dies ist nicht unbedingt die maximal mögliche Prozessorzahl, da ein Job bei der Anforderung von weniger Prozessoren unter Umständen früher zur Ausführung gelangen kann. Die im ersten Schritt ermittelten minimalen Antwortzeiten werden dann als Kanten in einen Graphen eingetragen, wobei eine Kante jeweils einen Knoten mit der zugehörigen Job-Nummer und einen Knoten mit der zugehörigen Position in der Job-Liste verbindet. Die Einordnung in einen Graphen erlaubt, auf effiziente Weise über eine Minimierung des Gewichtes die Reihenfolge zu ermitteln, die insgesamt eine möglichst niedrige Antwortzeit ergibt. Reduziert man im so gefundenen Ablaufplan die Prozessorzuordnung auf maximal $p/2$, so erhält man im Ergebnis einen Ablaufplan mit minimaler Antwortzeit, dessen Gesamtlaufzeit im schlechtesten Falle um einen Faktor 2 länger ist als bei einer bestmöglichen Planung.

Um dieses Ergebnis zu erzielen, mußten die Autoren aber einige Einschränkungen vornehmen: Es wurde vorausgesetzt, daß den Programmen eine beliebige Zahl von Prozessoren zugeordnet werden kann, und daß die Programme sich auf die Prozessorzahl einstellen können. Dies ist in der Praxis bei Rechnern mit verteiltem Speicher jedoch oft nicht gegeben, da durch die explizite Verteilung von Daten auf die lokalen Speicher die Prozessorzahl häufig schon zum Zeitpunkt der Übersetzung festgelegt wird und danach nicht mehr geändert werden kann. Weiterhin wurden alle Prozessoren als identisch und in jeder Situation gleichwertig vorausgesetzt. Auf einem Rechner mit verteilten Speicher und Netzwerkstruktur ist es aber direkt einsichtig, daß Situationen existieren können, in denen zwar genügend Prozessoren vorhanden sind, ein Start aber dennoch nicht sinnvoll ist, da die Prozessoren aufgrund vorheriger Belegungen und Freigaben unzusammenhängend über das Netz verstreut sind.

Für die Kommunikation der Prozessoren untereinander (die in dieser Arbeit ebenfalls nicht berücksichtigt wird) ist eine solche Verteilung naturgemäß ungünstiger als ein Block von direkt benachbarten Prozessoren.

Im zweiten Artikel wird deshalb ein anderer Algorithmus namens *SMART* vorgestellt, der zum Scheduling von Jobs mit festgelegter Prozessorzahl und bekannter Ausführungszeit geeignet ist. *SMART* basiert auf dem Einsortieren von Jobs in *Fächer*, d.h. Gruppen von Jobs, die zur gleichen Zeit gestartet werden und deren summarische Prozessoranforderungen demzufolge kleiner oder gleich der Gesamtprozessorzahl sind (Abbildung 3.1). Da die Jobs in einem Fach erst dann gestartet werden, wenn der letzte Job des vorherigen Faches beendet wurde, ist es für eine gute durchschnittliche Auslastung wichtig, daß Jobs in einem Fach zusammengefaßt werden, deren Laufzeiten sich möglichst wenig voneinander unterscheiden. Im Falle von dem in diesem Artikel vorgestellen Algorithmus *SMART* werden dazu die Jobs in Gruppen eingeteilt, deren Laufzeiten kleiner als eine Zweierpotenz von Zeiteinheiten sind, d.h. in einer ersten Gruppe von Fächern werden alle Jobs einsortiert, deren Laufzeit kleiner oder gleich einer Zeiteinheit T ist, in eine zweite Gruppe alle Jobs mit $t <= 2T$ usw. Für den Fall, daß nicht alle Jobs einer Gruppe in einem Fach abgelegt werden können, müssen entsprechend mehrere Fächer erzeugt werden, wobei Jobs mit einer kleineren Prozessorzahl bevorzugt werden. Nach dieser initialen Zuordnung werden die Fächer nach folgender Vorschrift iterativ zusammengeschoben:

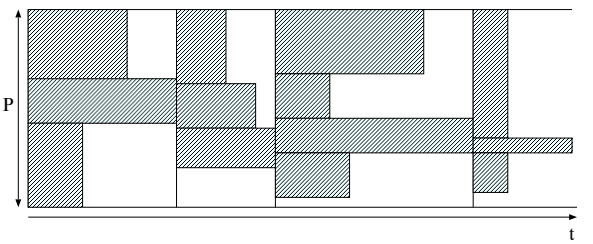


Abbildung 3.1
Shelf-basiertes Scheduling

- Sofern die Prozessoranforderungen von zwei Fächern in der Summe kleiner sind als die Anzahl vorhandener Prozessoren, können diese beiden Fächer zu einem einzigen zusammengefaßt werden.
- Sofern ein einzelner Job sowohl von seiner „Breite" (Prozessorzahl) als auch „Höhe" in ein niedrigeres Fach paßt, kann er dorthin verschoben werden.

Mit diesem Algorithmus ergibt sich eine Zuordnung, von der die Autoren nachweisen, daß sie im schlimmsten Falle um einen Faktor 32 schlechtere, mittlere Antwortzeiten

erbringt als eine optimale Lösung; sobald garantiert wird, daß keiner der Jobs mehr als die Hälfte aller verfügbaren Prozessoren belegt, reduziert sich der Faktor auf 16. Reihenuntersuchungen, die SMART auf zufällig ausgewählten Job-Kombinationen anwendeten und das Ergebnis mit der optimalen Kombination verglichen, ergaben jedoch in keinem Fall einen höheren Faktor als 4,27; die in dieser Arbeit theoretisch ermittelte Grenze scheint also noch eine relativ grobe Abschätzung darzustellen. In der dritten Arbeit gelang es, diese theoretische Abschätzung von 32 auf 11 zu verbessern, gleichzeitig wurde aber auch eine Variante des Zuordnungsalgorithmus vorgestellt, dessen Abschätzung bei 8 liegt. Diese Variante beruht darauf, die Jobs einer Gruppe nach ihrer „Fläche", d.h. dem Produkt aus Rechenzeit und Prozessorzahl anstatt nur nach der Prozessorzahl aufsteigend zu sortieren; die anschließende Kompaktierung ist identisch zum Originalalgorithmus. Dieser Algorithmus wird auch für den *gewichteten* Fall betrachtet, d.h. die Antwortzeit einzelner Jobs wird mit unterschiedlichen Faktoren bewertet. In diesem Falle ergibt sich eine obere Abschätzung von 10,44.

Allen vorgestellten Algorithmen ist gemein, daß diese ein nicht unterbrechend arbeitendes Betriebssystem voraussetzen, d.h. wenn eine Gruppe von Prozessoren mit der Ausführung eines Jobs begonnen hat, behalten diese den Job bis zum Programmende auf dem letzten Prozessor. Diese Annahme erlaubt es, Jobs als einfache, rechteckige Bereiche zu betrachten und vereinfacht die Abschätzungen, wird den realen Verhältnissen in einem Parallelrechner aber nicht vollständig gerecht. Zum einen wird nicht berücksichtigt, daß die ein paralleles Programm abarbeitenden Prozessoren untereinander kommunizieren und die Lage der Prozessoren zueinander im Netzwerk die Geschwindigkeit der Kommunikation beeinflußt; es kann daher sinnvoll sein, auch bei genügend freien Prozessoren einen Job nicht zu starten, wenn die freien Prozessoren weit verstreut im Netzwerk liegen. Zum anderen muß in der Realität aber noch berücksichtigt werden, daß ein Prozeß I/O-Phasen besitzt, in denen der Prozessor bereits mit der Ausführung anderer Jobs beginnen könnte, sofern der Hauptspeicher auf den Prozessoren für beide Prozesse ausreicht. Es ist auch denkbar, daß einzelne Prozessoren nicht während des gesamten Programmablaufes benötigt werden und phasenweise an Synchronisationsstellen warten.

Diese Gedanken gehen aber bereits über die Vorstellung eines strengen Batch-Betriebes mit sequentieller Abarbeitung hinaus und leiten zur zweiten Form des Job-Schedulings über, dem Online-Scheduling. Diese Betriebsart ist kennzeichnend für den interaktiven Betrieb von Rechnern, bei dem fortlaufend Jobs zur Ausführung abgegeben werden, die auch möglichst sofort ausgeführt werden sollen. Ein Aufsammeln von Jobs in Warteschlangen ist hier unüblich; sofern nicht genügend Prozessoren vorhanden sind und auch kein Multitasking auf den Rechenprozessoren möglich ist, wird ein Job meist zurückgewiesen. Im interaktiven Betrieb ist eine gerechte Verteilung der Ressourcen ebenfalls deutlich wichtiger: Während es für den Batch-Betrieb ausreicht, daß in der zur Verfügung stehenden Zeit möglichst viele Jobs

bearbeitet wurden (d.h. die Effizienz im Vordergrund steht), soll im interaktiven Betrieb jedem einzelnen Benutzer die Maschine gleichermaßen zur Verfügung gestellt werden.

3.1.1.3 Verfahren zum interaktiven Scheduling

Das Problem des „interaktiven Scheduling" ist mathematisch wesentlich komplizierter zu beschreiben, da die zur Verfügung stehenden Informationen spärlicher sind: Beim Scheduling eines Jobs steht neben den Ressourcenanforderungen dieses Jobs als Information nur die aktuelle Auslastung des Parallelrechners, eventuell mit den Restlaufzeiten der momentan ausgeführten Jobs zur Verfügung. Es ist weder bekannt, welche Jobs in Zukunft submittiert werden sollen, noch der Zeitpunkt der Submittierung. An die Stelle mathematisch exakter Aussagen und einer Vorausplanung der Prozessorbelegung müssen daher die Überwachung der laufenden Jobs sowie ein Mechanismus treten, der die Rechen- und Kommunikationslast möglichst gleichmäßig auf das Gesamtsystem verteilt und damit lokale Engpässe vermeidet. Zur Erreichung dieses Ziels hat ein Scheduler zwei Eingriffsmöglichkeiten: Die anfängliche Zuordnung (*Allokation, Mapping*) und die Umordnung während der Ausführung (*Migration*).

3.1.1.4 Allokation und Migration

Während unter Allokation in diesem Zusammenhang die initiale Festlegung der durch ein Programm belegten Prozessoren zu verstehen ist, werden bei der Migration Teile eines laufenden Programmes auf andere Prozessoren verlagert. Migrationen können erforderlich werden, wenn sich die Lastsituation im Rechnersystem so verändert hat, daß eine gleichmäßigere Verteilung den Zeitaufwand für die Migration selbst aufwiegt. Insbesondere in einem Parallelrechner mit verteiltem Speicher ist eine Migration eine sehr zeitaufwendige Operation, die nicht nur den zu migrierenden Programmteil zwischenzeitlich anhält, sondern auch alle anderen Programmteile behindert, die mit ihm kommunizieren.

3.1.1.5 Verfahren zur Allokation

Eine günstige Verteilung von Prozessen ist Grundvoraussetzung für die effiziente Ausführung paralleler Programme, unabhängig vom verwendeten Programmiermodell. Da die Kommunikation der einzelnen Prozessoren untereinander bei parallelen Programmen üblicherweise sehr intensiv ist, ist es sinnvoll, die ein Programm parallel abarbeitenden Prozesse auf benachbarten Prozessoren anzuordnen. Dies kann

einerseits die Latenzzeiten verringern, sofern das verwendete Routing-Verfahren eine zusätzliche Verzögerung pro Routing-Schritt einführt, andererseits reduziert eine möglichst zusammenhängende Anordnung die Wahrscheinlichkeit der Kollision von Nachrichten verschiedener paralleler Applikationen.

Ein weit verbreiteter Algorithmus zur Allokation von Prozessoren wurde im Zusammenhang mit Hypercube-Architekturen entwickelt und wird als *Buddy*-Algorithmus bezeichnet. Er beruht in seiner Originalform darauf, daß ein Hypercube aus einer Struktur von 2^n Prozessoren besteht, die sich in 2 gleichartige Strukturen von 2^{n-1} zusammenhängenden Prozessoren zerlegen läßt. Diese Unterstrukturen lassen sich nach dem gleichen Verfahren weiter zerlegen, bis man zu Partitionen kommt, die nur noch aus einem Prozessor bestehen. Für die ankommenden Jobs wird nun vorausgesetzt, daß die von ihnen angeforderte Prozessorzahl immer eine Zweierpotenz ist; nötigenfalls muß die angeforderte Zahl auf die nächstliegende Zweierpotenz aufgerundet werden. Der Algorithmus führt $n+1$ Listen, in denen noch freie Partitionen der Größen 2^0 bis 2^n verzeichnet sind. So enthält z.B. bei einem unbenutzten Rechner nur die letzte Liste einen Eintrag, der den kompletten Rechner beschreibt. Erreicht nun ein Job mit der Anforderung von 2^k Prozessoren den Rechner, so wird zuerst in der k-Liste nach einer freien Partition gesucht. Ist dort eine freie Partition vermerkt, so wird diese belegt, ansonsten werden die Listen $k+1$ bis n nach einer freien Partition durchsucht. Wird dabei eine Partition in Liste l gefunden, so wird diese aus Liste l ausgetragen und in zwei Partitionen der Größe 2^{l-1} geteilt. Von diesen Unterpartitionen wird eine in Liste $l-1$ eingetragen und die andere wieder geteilt. Dieser Vorgang setzt sich fort, bis eine Partition der gewünschten Größe 2^k übrigbleibt. Beim Freigeben müssen umgekehrt kleinere, zusammengehörige Partitionen wieder zusammengefaßt werden.

Bei der Übertragung dieses Algorithmus auf andere Topologien muß nach entsprechenden, unterteilbaren Strukturen gesucht werden. Eine in [LiCh91] vorgestellte Variante erlaubt die Anwendung des Buddy-Prinzips auf zweidimensionale Netze, wobei es sich aber sowohl beim Gesamtnetz als auch bei den Partitionen um quadratische Netze der Kantenlänge 2^k handeln muß. Diese Einschränkung bedeutet, daß der Unterschied zwischen den einzelnen Partitionsgrößen bereits einen Faktor von 4 beträgt. Beim Aufrunden auf die von einem Job angeforderte Prozessorzahl entstehen also noch größere Ineffizienzen als in einem Hypercube. Praktisch hat dieses Verfahren deshalb kaum Anwendung gefunden, zumal die Einschränkung, daß es sich bei der Gesamtprozessorzahl um eine Viererpotenz handeln muß, das Grundproblem von Hypercubes noch verschärft: Durch die Sprünge in Zweier- bzw. Viererpotenzen ist ein Rechnerausbau nur in festen, immer teureren Schritten möglich, „Zwischenlösungen", wie sie Gitterstrukturen ohne spezielle Geometrievorgaben erlauben würden, sind nicht möglich. Es besteht daher der Wunsch nach Algorithmen, die in einem Gitter beliebiger Größe Partitionen beliebiger Höhe und Breite belegen können, so daß

gleichzeitig auch kein Aufrunden der Prozessorzahl mehr erforderlich ist. Ein einfaches Verfahren, welches diese Kriterien erfüllt, wurde in [CuTz91] unter dem Namen *Frame-Sliding* vorgestellt. Bei diesem Verfahren wird das Rechteck der zu belegenden Partition in Schritten seiner Breite bzw. Höhe über die Prozessormatrix geschoben, bis ein entsprechendes, freies Feld gefunden wird. Die vergleichsweise groben Schritte sorgen zwar dafür, daß der Rechenaufwand nur moderat steigt, verhindert aber, daß alle möglichen freien Subnetze gefunden werden. Der Allokationsalgorithmus, der z.B. in der Intel Paragon zur Belegung von Partitionen vorgegebener Geometrie benutzt wird, verwendet deswegen bei der Suche nur Schritte der Länge 1 in beiden Richtungen. Auf diese Weise werden zwar alle Möglichkeiten durchsucht, der Rechen- und Vergleichsaufwand steigt jedoch linear mit Höhe *und* Breite des Netzes. Der Grund für dieses schlechte Zeitverhalten im Vergleich zu Buddy-Algorithmen ist darin zu suchen, daß neben der eigentlichen Belegungsmatrix keine zusätzlichen Datenstrukturen angelegt und verwaltet werden, die freie Partitionen speichern und so den Suchvorgang beschleunigen würden. Ein erster Ansatz in dieser Richtung waren die *first fit*- bzw. *best fit*-Strategien aus [Zhu92], die für mögliche Partitionsgrößen Bitfelder über Knoten mit einer geeigneten Anzahl von freien „Nachbarn" verwalten. Erweiterte Formen solcher Algorithmen finden in der Realität Anwendung, z.B. in einem erweiterten Allokationsalgorithmus der Intel Paragon. Dieser Algorithmus ist in der Lage, neben der Anforderung rechteckiger Partitionen auch Anforderungen zu bedienen, in denen nur die Anzahl der Prozessoren, nicht aber die exakte Geometrie der Partition spezifiziert wird. Aus allen Wahlmöglichkeiten wird dann diejenige gewählt, die einem Quadrat am nächsten kommt. Es werden ferner Vorkehrungen getroffen, eine Fragmentierung soweit als möglich zu vermeiden. Dies wird z.B. dadurch erreicht, daß eine Partition so gewählt wird, daß sie an möglichst vielen Seiten an bereits belegte Bereiche grenzt.

Trotz derartig ausgefeilter Allokationsalgorithmen kann in der Praxis durch wiederholtes Belegen und Freigeben von Partitionen unterschiedlicher Größe eine Fragmentierung des Rechnersystems entstehen. Trifft nun ein Job ein, dessen Prozessorbedarf nur durch Belegung einer nicht zusammenhängenden Partition zu befriedigen wäre, so bestehen prinzipiell drei Reaktionsmöglichkeiten:

- Zurückweisung bzw. Speicherung des Jobs;
- Zuweisung einer möglichst „unzerstreuten" Gruppe von Prozessoren;
- Umlagerung von laufenden Jobs, so daß eine zusammenhängende Partition der benötigten Größe entsteht.

Von allen Varianten ist die dritte in einem Rechnersystem mit verteiltem Speicher die aufwendigste und wird deshalb nur in Einzelfällen eingesetzt, zum Beispiel wenn der neu eintreffende Job hochpriorisiert ist (d.h. weder zurückgewiesen noch aufgeschoben werden kann) und die Zuweisung einer nicht zusammenhängenden Partition eine deutliche Verlangsamung bedeuten würde. Je nach Kommunikationsintensität kann die durch Zerstreuung bewirkte Verlangsamung unterschiedlich stark ausfallen, und das Kommunikationsaufkommen eines Jobs ist, im Vergleich zur Laufzeit, für das Betriebssystem im allgemeinen noch viel weniger a priori bekannt. Es ergibt sich daher die Frage, wie hoch die Zeitverluste durch nicht-zusammenhängende Partitionen durchschnittlich sind.

In [LiWi94] wurde diese Frage auf einer Intel Paragon für verschiedene Algorithmen untersucht. Den Untersuchungen gingen Experimente voraus, in denen gezielt versucht wurde, auf den Kommunikationsleitungen Überschneidungen zu erzielen, wie sie bei der Belegung nicht-zusammenhängender Partitionen entstehen würden. Das Ergebnis war, daß auch unter dem effizienteren SUNMOS- Betriebssystem, das deutlich höhere Datenraten erreicht als das originale OSF/1, nennenswerte Zeitverluste nur bei der Übertragung großer Nachrichten auftraten; bei kleinen Nachrichten ist die Wahrscheinlichkeit hoch, daß diese sich nicht überschneiden, da die einzelnen Programme die Nachrichten unsynchronisiert absetzen. Statistiken über das durchschnittliche Nachrichtenaufkommen einer Intel Paragon im Produktionsbetrieb ergaben weiterhin, daß der überwiegende Teil aller Nachrichten kürzer als 1 KByte ist und daher kaum von Überschneidungen betroffen wäre.

Basierend auf dieser Erkenntnis wird von den Autoren eine Erweiterung des zweidimensionalen Buddy-Algorithmus namens MBS (Multiple Buddy Strategy) vorgeschlagen. Dieser Algorithmus spaltet (für den Fall von zweidimensionalen Netzen) die angeforderte Prozessorzahl in Viererpotenzen auf und führt den normalen Buddy-Algorithmus für jede sich ergebende Teilkomponente aus. Zusätzlich wird aber erlaubt, daß die Belegung einer Komponente auch durch mehrere, kleinere und nicht zusammenhängende Bereiche erfolgen darf, sofern kein zusammenhängender Bereich der gewünschten Größe mehr frei ist. Auf diese Weise versucht MBS, den Zusammenhang der Gesamtpartition soweit als möglich zu erhalten und Partitionen nur dann zusammenzustückeln, wenn es unvermeidlich ist. Die Autoren gehen allerdings nicht darauf ein, wie es erreicht wird, daß für die im ersten Schritt entstandenen Komponenten möglichst aneinandergrenzende Bereiche belegt werden.

Dieser Algorithmus wird von den Autoren in einer Reihe von Experimenten mit den vorher erwähnten Strategien, die nur zusammenhängende Bereiche belegen können, verglichen. Das Ergebnis ist, daß selbst bei der Simulation kommunikationsintensiver Algorithmen MBS bessere Durchsätze liefert, d.h. der vorgegebene Job-Mix ist schneller abgearbeitet. Dieses Ergebnis ist darauf zurückzuführen, daß

die Belegung nicht-zusammenhängender Partitionen es erlaubt, Jobs bereits zu einem Zeitpunkt zur Ausführung zu bringen, zu dem eine einfache Strategie den Job noch zurückstellen müßte. Es zeigt sich aber auch, daß der Grad der Beschleunigung vom Kommunikationsaufkommen der Anwendungen abhängt: Je höher es ist, desto mehr führt die Belegung nicht-zusammenhängender Partitionen dazu, daß sich die einzelnen Programme gegenseitig auf dem Verbindungsnetzwerk behindern. Für den Fall, daß Anwendungen aufgrund ihres Kommunikationsverhaltens sehr stark von einer regelmäßigen, zusammenhängenden Partition profitieren, sinkt der Zeitgewinn auf etwa zehn Prozent.

Dieses Ergebnis zeigt also, daß die Wahl des Partitions-Layouts stark von den Laufzeit-Eigenschaften der Anwendung beeinflußt wird, die aber vorab für ein Betriebssystem nur schwer zu ermitteln sind. Es bietet sich daher an, im Fall erkennbar ungünstiger Layouts eine Umverteilung vorzunehmen. Diese Situation ist Teil des Gesamtkomplexes der *Lastbalancierung*.

3.1.1.6 Verfahren zur Lastbalancierung

Unter Lastbalancierung ist allgemein der Vorgang zu verstehen, die zur Verfügung stehenden Prozesse in einem parallelen Rechnersystem möglichst so dynamisch umzuverteilen, daß die Unterschiede in der Auslastung einzelner Bereiche minimiert werden. Übliche Auslastungsmaße sind dabei z.B. die Anzahl rechenbereiter Prozesse auf einem Prozessor bzw. die von ihnen genutzte Rechenzeit. Es sind aber auch andere Meßgrößen denkbar, wie z.B. das Kommunikationsaufkommen, die Menge des freien Speichers auf den Rechenknoten oder eine Kombination aus mehreren Maßen.

Inhalt eines Lastbalancierungsverfahrens ist es, im Rechnersystem entstehende Ungleichgewichte zu erkennen, zu bewerten und entsprechende Gegenmaßnahmen einzuleiten. Da eine Auflösung von Ungleichgewichten nur durch die Umverteilung von Job-Teilen möglich ist, wird im allgemeinen vorausgesetzt, daß einem einzelnen Prozessor mehrere Prozesse zugeordnet werden dürfen, zwischen denen die Rechenzeit mit Hilfe eines lokalen Schedulers aufgeteilt wird. Hätte man nur einen Prozeß pro Prozessor, so hätte der Auslastungszustand eines Prozessors nur zwei Zustände, nämlich keine oder volle Auslastung. Eine Verschiebung eines Prozesses wäre damit nur auf einen leeren Prozessor möglich, und der bisherige Prozessor wäre fortan leer. Das Ungleichgewicht wäre damit nur verschoben, aber nicht aufgehoben.

Ein Lastbalancierungsalgorithmus muß in der Praxis eine Reihe von Problemen lösen. Das erste ist das Problem einer angepaßten Datenerhebung. Wird der Lastausgleich für alle Prozessoren an einer zentralen Stelle vorgenommen, so kann sich diese Stelle bei einer Erhöhung der Prozessorzahl als Flaschenhals erweisen; erfolgt sie

dagegen dezentral und Extremfall nur unter direkten Nachbarn, so ist die Auflösung großer, globaler Lastimbalancen entweder überhaupt nicht oder nur nach einer Vielzahl von Zeitschritten möglich. Unabhängig von diesen Gesichtspunkten existiert in verteilten Systemen das Problem der Bewertung der ermittelten Daten; da die Lastinformationen über ein Kommunikationsnetz mit Verzögerungen übermittelt werden, sind sie beim Eintreffen prinzipiell veraltet, und es ist unmöglich, auf einem bestimmten Prozessor den exakten, aktuellen Lastzustand zu ermitteln. Diese Inkonsistenzen erschweren die nötige Voraussage des Systemverhaltens in der nahen Zukunft.

Eine systemabhängige Größe, die in die Entscheidung über Umverteilungen eingeht, ist der Zeitbedarf für den Transfer eines Prozesses von einem Prozessor auf einen anderen. Während in einer shared-memory-Architektur der Transfer nur aus dem Umlagern des Prozeßkontextes (CPU-Register, MMU-Tabellen) besteht, muß auf einem Rechner mit verteiltem Speicher ein vollständiger Speicherabzug übertragen werden. Ein Lastbalancierungsalgorithmus muß also den durch Migration voraussichtlich erzielbaren Zeitgewinn in Relation zu ihrem Overhead setzen.

Ein weiteres, wichtiges Kriterium für Lastausgleichsalgorithmen ist ihre *Stabilität*. Da ein Lastausgleichsalgorithmus sich als Regelkreis verstehen läßt, der anhand einer meßbaren, zu minimierenden Größe (nämlich das Lastungleichgewicht zwischen den Prozessoren) eine Stellgröße (die Verteilung von Programmen auf Prozessoren) variiert, besteht prinzipiell die Gefahr des *Reglerschwingens*, d.h. durch eine Überreaktion wird ein Lastungleichgewicht an einer Stelle in ein verstärktes Ungleichgewicht an einer anderen Stelle verwandelt, was wiederum eine verstärkte Rückreaktion auslöst; ein ausgeglichener Zustand kann sich nicht einstellen. Unter der Stabilität eines Lastausgleichsalgorithmus ist daher zu verstehen, daß ein bestehendes, beliebiges Ungleichgewicht vom Algorithmus in endlicher Zeit in einen ausgeglichenen Zustand überführt wird. Als eine Verschärfung dieses Kriteriums kann verlangt werden, daß dieser Prozeß monoton ablaufen soll, d.h. es sollen auch keine „gedämpften Schwingungen" zugelassen werden. Der theoretische Nachweis der Stabilität eines Lastausgleichsalgorithmus ist im allgemeinen schwierig, da es sich sowohl beim Lastmaß als auch bei der Lastverteilung nicht um Skalare, sondern um Vektoren handelt, deren Komponenten von Algorithmus sowie Systemgröße und -topologie abhängt. Ein theoretischer Nachweis der Stabilität kann daher immer nur für eine festgelegte Kombination aus Algorithmus, Systemtopologie und Anregung gelingen.

Mangelnde Stabilität ist zum Beispiel ein Problem des in [Smi80] vorgestellten *Bidding*-Algorithmus. Bei diesem dezentralen Lastausgleichsverfahren stellen die einzelnen Prozessoren unabhängig voneinander fest, ob ihre Last zu niedrig ist, und fordern in einem solchen Fall ihre Nachbarn auf, zusätzliche Prozesse zu ihnen zu

übertragen. Dabei kommt es sehr häufig vor, daß der anfragende Prozessor als Reaktion von seinen Nachbarn so viele Prozesse erhält, daß er in einen überlasteten Zustand geht und eine entsprechende „Rückreaktion" erfolgt. Im Ergebnis wandern Prozesse ständig zwischen Prozessoren hin und her. Dieses *Prozeß-Trashing* ist unbedingt zu vermeiden, da es auf die Performance ähnlich vernichtende Auswirkungen hat wie das bei der Beschreibung von SVM bereits angesprochene Page-Trashing.

Um die negativen Eigenschaften des *Bidding*-Algorithmus zu beseitigen, wurde er in [NXG85] zum sogenannten *Drafting*-Algorithmus erweitert. Hauptmerkmal des *Drafting*-Algorithmus ist die Kommunikation, die sich zwischen einem leicht belasteten Prozessor und seinen Partnern, die ihm potentiell zusätzliche Lasten schicken könnten, abspielt. Ein solcher Zyklus beginnt, wenn der Lastzustand eines Prozessors in den Zustand „niedrig" wechselt (*Drafting* unterscheidet lediglich zwischen drei Lastsituationen: hoch, normal und niedrig). Jeder Prozessor verwaltet lokal eine Tabelle, in der die letztgültigen Lastzustände aller Prozessoren verzeichnet sind; Inkonsistenzen sind dabei durchaus zugelassen. Aus dieser Tabelle werden nach einem Übergang die Prozessoren herausgesucht, deren Lastzustand als „hoch" vermerkt ist. Alle angesprochenen Prozessoren antworten daraufhin mit ihren *draft age*, einem Zahlenwert, der anzeigt, wie vorteilhaft es wäre, einen Prozeß von diesem Prozessor zu migrieren. Die Berechnung eines *ages* ist systemabhängig und nicht Teil des *Drafting*-Algorithmus. Der unterlastete Prozessor kann in einem zweiten Schritt den bestmöglichen Migrationspartner auswählen und einen Prozeß anfordern. Dabei ist es möglich, daß ein Prozessor inzwischen nicht mehr überlastet ist und den Transfer verweigert, worauf aus der Liste der nächstbeste Partner ausgewählt wird.

Stabilität wird durch die Nebenbedingung erreicht, daß ein Prozeß nur einmal migriert werden kann. Obwohl dieser Algorithmus an einer Reihe von Stellen Variationsmöglichkeiten zur Anpassung an die reale Situation bietet, besitzt er einige vorgegebene Eigenschaften, die zu nicht-optimalen Ergebnissen führen können:

- Die Auslastungsbereiche für „hoch", „mittel" und „niedrig" sind starr vorgegeben. Jeder Prozessor entscheidet unabhängig, wie sein Zustand ist und berücksichtigt in der Migrationsentscheidung nicht die Auslastung der anderen Prozessoren. Je nach Gesamtauslastung des Systems können die Auslastungsgrenzen unterschiedlich zu ziehen sein.

- Die Vorgabe, daß ein Prozeß während seiner Laufzeit nur einmal migriert werden kann, ist bei länger laufenden Anwendungen nicht akzeptabel.

- Bei einem hohen Aufwand für eine Migration kann es sinnvoll sein, ein möglicherweise nur temporäres Lastungleichgewicht nicht sofort zu beseitigen, sondern erst nach einer gewissen Zeit Gegenmaßnahmen einzuleiten. Eine solche Verzögerung existiert beim *drafting* nicht.

Die Starrheit des *drafting* hat zur Entwicklung von Lastbalancierungsalgorithmen geführt, die sich an die momentanen Lastverhältnisse dynamisch anpassen können. A. Stankovic hat zu diesem Zweck in [Sta85] das Konzept eines *stochastischen Lernautomaten* vorgeschlagen, mit dem sich ein dezentrales Lastbalancierungssystem aufbauen läßt. Kennzeichen dieses Automaten ist seine Lernfähigkeit, d.h. er registriert die Reaktion des restlichen Systems auf eine seiner Aktionen und speichert diese für den Fall, daß eine ähnliche Konstellation wieder auftritt. Anhand der bei der ersten Reaktion durchgeführten Bewertung kann jetzt eine mehr oder weniger modifizierte Reaktion erfolgen. Um Geschwindigkeit und Reaktionsvermögen dieses Verfahrens zu beeinflussen, existieren zwei Zeitperioden: Die erste bestimmt die Frequenz, mit der ein Prozessor seinen Lastzustand an die anderen Prozessoren meldet, die andere legt den Rhythmus fest, in dem ein Automat den augenblicklichen Systemzustand untersucht und eventuell entsprechende Reaktionen einleitet. Problematisch ist an diesem Verfahren die Tabelle, in der die gelernten Reaktionen auf verschiedene Systemzustände abgespeichert werden. Je feiner die Abstufung der unterschiedenen Lastzustände ist, desto größer wird die auf jedem Prozessor vorzuhaltende Tabelle. Mit der Größe der Tabelle wächst naturgemäß auch die Lernzeit des Systems, da für eine korrekte Reaktion auf einen bestimmten Zustand dieser mindestens einmal vorher aufgetreten sein muß, ein Automat aber pro Zeitschritt nur einen Zustand lernt. Wird aber die Zahl der möglichen Zustände zu klein gewählt, besteht die Gefahr von Fehlentscheidungen aufgrund fehlender Differenzierung. Als kritisch erweist sich in der Praxis die Wahl der Zeitperiode, in der die lokalen Automaten den Systemzustand überprüfen: Wird sie zu klein gewählt, kann es zu Prozeß-Trashing kommen, wird sie zu groß gewählt, reagiert das System zu träge.

Zusammenfassend kann gesagt werden, daß die Grundidee eines lernfähigen Lastbalancierungsautomaten zwar gut ist, in der hier vorgestellten Reinheit aber nicht brauchbar ist. Es ist notwendig, einen Lastbalancierer mit Wissen auszustatten, das er nicht lernen muß, und das Zeitraster seiner Reaktionen muß sich den Gegebenheiten anpassen.

Eine verfeinerte Methode der Übertragung von Lastinformationen und der Entscheidungsbildung findet sich in [ShHo94]. Die Auslastung eines Prozessors kann dort in eine beliebige Zahl K von Regionen unterteilt werden. Ein Knoten verschickt nur Statusmeldungen, wenn sein Zustand sich gegenüber der letzten Meldung um mindestens 2 Stufen verändert hat. Dieses Verfahren vermindert das Kommunikationsaufkommen zum einen dadurch, daß Informationen nur neu gesendet werden, wenn sie sich auch wirklich verändert haben; zum anderen wird verhindert, daß ein Prozessor, dessen Zustand sich gerade im Grenzbereich zwischen zwei Zuständen befindet, ständig Meldungen verschickt, ohne damit wesentliche neue Informationen zu erbringen. Die erhaltenen Lastinformationen ersetzen bei den Empfängern die alten Informationen nicht sofort, sondern werden in einem ersten Schritt nur

gespeichert. In regelmäßigen Abständen wird aus den erhaltenen Werten eine Wahrscheinlichkeitsverteilung für den Lastzustand eines Prozessors berechnet und mit der bisherigen Verteilung gewichtet summiert. Über die Gewichte dieser Summation kann eingestellt werden, wie schnell sich das Ergebnis dem Augenblickswert der Prozessorlasten anpaßt bzw. wie stark kurzzeitige Schwankungen durch Mittelung unterdrückt werden. Um nun den möglichen Gewinn einer Migrationsentscheidung zu ermitteln, muß man die Gewinne für alle einzelnen möglichen Zustände entsprechend ihrer Wahrscheinlichkeit aufsummieren. Aus allen möglichen Entscheidungen (nämlich der Migration eines beliebigen lokalen Jobs zu einem beliebigen anderen Prozessor) kann eine optimale Entscheidung ermittelt werden. Damit die Gesamtzahl zu speichernder Lastverteilungen und abzutestender Entscheidungen nicht zu stark wächst, tauscht ein Prozessor Lastinformationen und Prozesse nicht mit allen anderen Prozessoren, sondern nur mit einer Gruppe ihm nahe liegender Prozessoren, dem sogenannten *buddy set* aus. Kritisch ist naturgemäß die Wahl einer Bewertungsfunktion für Migrationen. Da in dieser Arbeit Echtzeitsysteme im Vordergrund standen, berechnet sich der Gewinn einer Migration daraus, inwiefern die Austauschpartner mit der neuen Verteilung die Zeitschranken (dead lines) für die Ausführung ihrer Jobs besser einhalten können. In einem Nicht-Echtzeitsystem müßten andere Kriterien zum Zuge kommen, wie z.B. die Nähe kooperierender Tasks oder die Speicherauslastung. In Simulationen haben die Autoren gezeigt, daß bis zu etwa 4 Auslastungsstufen der Nachrichtenverkehr nicht nennenswert ansteigt, und daß die Wahl der anfänglichen Auslastungsschätzung keine Bedeutung hat, d.h. die Auslastungsschätzungen passen sich hinreichend schnell an. Zur Wahl des Gewichtungsfaktors und der Update-Periode schlagen die Autoren eine schrittweise Optimierung vor: Ausgehend von einer initialen Wahl wird ein Parameter variiert, bis optimale Ergebnisse erreicht werden. Danach wird dieser erste Parameter auf seinem Optimalwert gehalten und der zweite variiert, bis wieder ein Optimalwert gefunden ist. Ausgehend von diesem Parametersatz könnte das Verfahren dann iterativ wiederholt werden. Dabei besteht natürlich prinzipiell die Gefahr, in einem lokalen Minimum zu landen. Das Verfahren ist aber wesentlich weniger aufwendig, als ein ganzes Raster von Parameterkombinationen zu durchsuchen und erreicht gute Ergebnisse, sofern die Oberfläche der Funktion nicht zu unregelmäßig ist.

Bei den bisher vorgestellten Verfahren handelt es sich um theoretisch entwickelte Verfahren, die nicht für ein bestimmtes, real existierendes Parallelrechnersystem geschaffen wurden. Sie nehmen daher alle bis zu einem gewissen Grade Universalität für sich in Anspruch, bedürfen aber für eine praktische Nutzung einer Anpassung ihrer Parameter an die Bedürfnisse eines realen Rechnersystems. Es ist auch denkbar, daß ein realer Lastbalancierer mehrere Algorithmen kombinieren muß, um allen auftretenden Situationen gewachsen zu sein. Im folgenden sollen daher reale Implementierungen von Lastbalancierungssystemen vorgestellt werden.

Für Cluster von Workstations wurden einige Lastbalancierungssysteme ent-

wickelt. Bei einem Workstation-Cluster handelt es sich um eine lose gekoppelte Konfiguration von Rechnern, die zunächst für den Standalone-Einsatz konzipiert sind und im Normalbetrieb auch nur Programme ausführen, die lokal gestartet werden. Da Workstations üblicherweise im interaktiven Betrieb arbeiten, ist die Prozessorauslastung auf den einzelnen Rechnern eher gering, so daß es sich anbietet, die Leerlaufzeiten zu nutzen, indem Jobs von momentan überlasteten Rechnern auf weniger ausgelastete Prozessoren migriert werden. Bei dieser Umverteilung ist aber auch zu berücksichtigen, daß lokal gestartete Jobs Vorrang in der Ausführung haben sollten, da die Rechner im wesentlichen immer noch als Workstations genutzt werden. Diese Randbedingungen bedürfen einer Erweiterung des Schedulers, der auf den einzelnen Workstations bereits vorhanden ist und zur Abwicklung des Vordergrundbetriebs auch weiterhin benötigt wird.

In [Nit94] wird die Implementierung eines Lastausgleichsverfahrens für Workstation-Cluster beschrieben, auf denen neben dem normalen interaktiven Betrieb parallele Programme unter PVM [BDGM94] ausgeführt werden sollen. Zu diesem Zweck ist es in einem ersten Schritt erforderlich, vom vorhandenen UNIX-Scheduler Daten über die momentane Auslastung der einzelnen Workstations zu sammeln. Der von diesen Durchschnittszahlen überstrichene Zeitbereich liegt in der Größenordnung einiger Minuten. Da die Umverteilung von Jobs in einem Workstation-Cluster mit üblicher Ethernet-Vernetzung und Kommunikation über die normalen Systemschnittstellen (TCP/IP) einen relativ starken Overhead bedeutet, ist kein allzu häufiger Austausch von Prozessen vorgesehen, so daß die langen Sampling-Perioden voll ausreichend sind. Die Maschinenauslastungen werden aufgeschlüsselt nach dem Rechenzeitanteil für interaktive und Vordergrund-Jobs. Aus diesen Größen kann dann berechnet werden, welchen Rechenzeitanteil ein neuer PVM-Prozeß auf diesem Prozessor erhalten würde. Dieser Anteil wird noch mit der relativen Leistungsfähigkeit eines Prozessors multipliziert, um heterogene Cluster korrekt zu berücksichtigen.

Der Lastbalancierer speichert unabhängig vom verwendeten Algorithmus zu jedem Prozeß eine Reihe von Flags, die bei der Erzeugung des Prozesses vom Anwendungsprogrammierer gesetzt werden können. Mit diesen Flags kann z.B. dem Lastbalancierer die voraussichtliche Lebenszeit des Prozesses mitgeteilt werden; als kurzlebig gekennzeichnete Prozesse werden dabei bevorzugt nicht migriert, da sich deren Laufzeit durch eine Migration im Verhältnis stärker verlängern würde. Weiterhin kann die Migration einzelner Prozesse komplett unterbunden werden, wenn die Präsenz eines bestimmten Prozesses auf einem bestimmten Prozessor für die Funktion einer Anwendung zwingend erforderlich ist. In der Arbeit wurden drei verschiedene Lastbalancierungsalgorithmen implementiert und untersucht. Bei dem ersten handelt es sich um eine Variation des *Drafting*-Algorithmus, bei den beiden letzteren um selbstentwickelte Verfahren.

Beim *Drafting*-Verfahren wurde als Maß für das *age* von Prozessen das Verhältnis von CPU-Zeit zur Wallclock-Zeit eines Prozesses gewählt, also der Anteil an

der Gesamtrechenleistung, den ein Prozessor erhält. Auf diese Weise wird derjenige Prozeß migriert, der momentan vom Prozessor am schlechtesten bedient wird und damit die besten Aussichten hat, von einer Migration zu profitieren.

Das zweite Verfahren, das aufgrund seiner Arbeitsweise als *reaktiv* bezeichnet wird, geht wie das Standard-Verteilungsschema von PVM von einer zyklischen Aufteilung der Prozesse auf die vorhandenen Prozessoren aus. Es existieren nur die drei Lastzustände „Normal", „Unterlast" und „Überlast", deren Berechnung durch eine Verknüpfung verschiedener Lastmaße (Anzahl PVM-Prozesse, Vordergrundlast, PVM-Anteil) mit gewählten Schranken erfolgt. In der Berechnung dieser Werte wird die Last der anderen Prozessoren jedoch *nicht* mit einbezogen. Befindet sich ein Prozessor in unter- oder überlasteten Zustand, so sendet er an alle anderen Prozessoren entsprechende Nachrichten. Diese Nachrichten werden jedoch nicht sofort weiterverarbeitet, sondern in Empfangspuffern zwischengespeichert. Wenn ein Prozessor in den Zustand „Überlast" geht, so wählt er aus dem Puffer anhand von vorhandenen Unterlastnachrichten einen passenden Partner zur Migration aus. Die Auswahl eines Prozesses erfolgt zum einen anhand der vom Programmierer vorgegebenen Flags, zum anderen anhand eines Timeout-Zählers, der Ping-Pong-Effekte vermeidet. Kann kein passender Migrationspartner ausfindig gemacht werden, so verschickt der Prozessor selber eine Überlast-Nachricht. Zusätzlich wird für den Fall, daß weder eine Unterlast noch eine Überlast vorliegt, überprüft, ob eine sehr dringende Überlastmeldung vorliegt, da in diesem Fall eine Migration ebenfalls sinnvoll sein kann.

Ebenfalls neu entwickelt wurde das dritte Verfahren der sogenannten *Prozeßbalancierung*, bei dem Wert auf die Kooperativität gelegt wurde, d.h. eine Migrationsentscheidung wird nicht von einem einzelnen Prozessor getroffen, sondern von einem Master, der durch Einsammeln von Lastinformationen einen Überblick über die Lastsituation im gesamten Netz besitzt. Dieses Verfahren verwendet die bereits oben angesprochenen Parameter *drive* und *overdrive*, die ausdrücken, welche Rechenleistung ein Prozeß momentan erhält bzw. welche Rechenleistung ein migrierter Prozeß erhalten würde. Da das Ziel ist, allen Prozessoren einen möglichst gleichen Rechenleistungsanteil zur Verfügung zu stellen, wird der Prozeß mit dem niedrigsten *drive* und der Prozessor mit dem höchsten *overdrive* ermittelt. Ob der Master einen Migrationsvorschlag verschickt, ist vom Auslastungsverhältnis und der Anzahl der Prozesse auf dem überlasteten Prozessor abhängig. Ergeht ein solcher Vorschlag, liegt es in der Hand des überlasteten Prozessors, einen Prozeß auszuwählen. Priorität haben bei dieser Auswahl die vom Programmierer vorgegebenen Flags; sollte sich daraus keine eindeutige Entscheidung ergeben, so wird der Prozeß gewählt, für den der erwartete Zeitgewinn am höchsten ist. Dieser ist prozeßabhängig, da der Zeitverlust durch die Migration selber von der Prozeßgröße abhängt. Neu ins System kommende Prozesse werden vom Master auf den Prozessor geschickt, der

momentan den größten *overdrive* bietet. Sowohl im Fall der Migration als auch der initialen Verteilung führt der Master eine Vorkorrektur der Auslastungsparameter durch, d.h. er schätzt, wie sich *drive*, *overdrive* und Anzahl von Prozessen durch die Veränderungen verschieben werden.

Da dieses Verfahren mit einem zentralen Master zumindest theoretisch Skalierungsprobleme hat, wurde in einem weiteren Versuch das *drafting* mit den Lastmaßen der Prozeßbalancierung kombiniert. Das *drafting* ist allerdings auch nicht beliebig skalierbar, da jeder Prozessor Lastinformationen über alle anderen Prozessoren benötigt; neben dem Speicheraufwand wächst so auch das Nachrichtenaufkommen für die Übermittlung von Lastinformationen quadratisch.

In den praktischen Versuchen haben sich nur die beiden letzten Verfahren als geeignet erwiesen, gegenüber einer statischen Verteilung durch das PVM-Laufzeitsystem eine bessere Lastverteilung zu erzielen und damit letztendlich die Gesamtausführungszeit zu verringern. Als entscheidend hat sich dabei die nicht vorhandene Kooperativität der beiden ersten Verfahren erwiesen, d.h. die Nichtbeachtung der umliegenden Systemzustände bei der Einschätzung der eigenen Situation führt zu Fehlentscheidungen. Diese fallen bei einem System mit vergleichsweise „teuren", d.h. zeitaufwendigen Migrationen, wie es mit Ethernet verbundene und unter PVM laufende Workstations darstellen, stärker ins Gewicht und sind daher unbedingt zu vermeiden. Skalierungsprobleme durch den zentralen Lastbalancierer des Prozeßbalancierungsverfahrens haben sich nicht gezeigt; rechnet man den Aufwand, der in den Experimenten mit 6 Workstations ermittelt wurde, auf realistische Größen von Workstation-Clustern (bis zu etwa 100 Maschinen) hoch, so wird festgestellt, daß eine zentrale Instanz keine wesentliche Behinderung darstellt. Diese Abschätzung kann in massiv-parallelen Rechnern mit ihrer höheren internen Kommunikationsleistung (und damit niedrigerem Migrationsaufwand) sowie der höheren Prozessorzahl anders ausfallen, so daß man teilweise dezentralen Ansätzen den Vorzug geben würde.

3.1.2 Task-Scheduling

Neben dem Job-Scheduler, dessen Aufgabe die Überwachung des Gesamtsystems ist, sind auf den einzelnen Prozessorknoten eines Parallelrechners weitere Einheiten zur Ausführung von Aufgaben notwendig, die autonom auf dem jeweiligen Prozessor abgewickelt werden können. Im Bereich des Scheduling können dabei als Aufgabenbereiche die Steuerung und Überwachung der diesem Prozessor vom Job-Scheduler zugeordneten Prozesse identifiziert werden. Da massiv-parallele Systeme in erster Linie zur Beschleunigung von Anwendungen durch Parallelverarbeitung benutzt werden, und das Multitasking mehrerer Prozesse auf einem Prozessor die für den einzelnen Prozeß zur Verfügung stehende Rechenleistung wieder reduzieren würde, lassen viele

massiv-parallele Betriebssysteme kein Multitasking auf den Prozessoren zu, so daß diese Aufgaben sich auf die Überwachung beschränken.

Eine solche Beschränkung vereinfacht zwar die Implementierung der Betriebssystemteile, die lokal auf den Rechenprozessoren liegen und reduziert auch den Ressourcenbedarf, so daß den Anwendungen z.B. ein größerer Anteil des lokalen Speichers zur Verfügung steht. Die Option, mehrere Prozesse im Multitasking auf einem Prozessor mit einem entsprechenden *Task-Scheduling* auszuführen, eröffnet aber eine Reihe von Optionen, die sich in einer erweiterten Funktionalität des Gesamtsystems äußern.

Zum einen gibt es auch bei parallelen numerischen Anwendungen Situationen, in denen ein Prozeß nicht rechnet, sondern I/O-Operationen durchführt. Dazu rechnen bei parallelen Programmen nicht nur explizite I/O-Operationen, sondern auch durch Kommunikation und Synchronisation der Prozesse verursachte Verzögerungen. Diese Zeiten lassen sich nutzen, um andere Benutzer-Prozesse auszuführen und so den Systemdurchsatz zu erhöhen. Zum anderen ist es im interaktiven Betrieb eines Parallelrechners nicht akzeptabel, daß einzelne, länger laufende Programme große Teile des Rechners belegen und ihn so für andere Benutzer unzugänglich machen. Mit einem möglichen Multitasking auf den Prozessoren kann ein Timesharing zwischen den Anwendungen stattfinden, so daß jeder Benutzer einen (möglichst gerechten) Anteil an der Gesamt-Rechenleistung erhält. Multitasking auf den Prozessoren würde weiterhin erlauben, Programme auszuführen, die mehr Prozessoren anfordern als physikalisch vorhanden sind; ein physikalischer Prozessor kann mehrere „logische" Prozessoren emulieren.

Dieses Maß an Flexibilität erkauft man sich naturgemäß durch einen Overhead, der in dem Aufwand für die Prozeßverwaltung und die Zeit für Kontextumschaltungen zu suchen ist. Eine Kontextumschaltung wird insbesondere dann teuer, wenn die quasi-parallel ausgeführten Anwendungen nicht gemeinsam in den Hauptspeicher des Prozessors passen, so daß bei jeder Kontextumschaltung nennenswerte Datenmengen auf Sekundärspeicher ausgelagert werden müssen. Der Durchsatzgewinn kann sich auf diese Weise in einen Verlust umkehren.

Ein weiteres Problem beim Task-Scheduling paralleler Programme ist die erforderliche Koordination der Scheduling-Entscheidungen. Da die einzelnen Prozesse einer parallelen Anwendung während des Programmlaufes üblicherweise Informationen austauschen und ihr Fortschritt vom unbehinderten Informationsfluß abhängt, ist es für die Vermeidung von Wartezeiten einzelner Prozesse erforderlich, daß alle Prozesse sich gleichzeitig im aktiven Zustand befinden. Ist dies nicht der Fall, so können „schlafende" Prozesse nicht auf Nachrichten anderer, laufender Partner reagieren, deren Fortschritt damit ebenfalls behindert wird. Scheduling-Verfahren, die

diesen Punkt berücksichtigen, werden allgemein als *Gang Scheduling*-Verfahren bezeichnet [GTU91]. Die dafür erforderliche Abstimmung der lokalen Scheduler untereinander erfordert eine Kommunikation, die das Multitasking paralleler Programme auf einem Parallelrechner weiter verteuert.

Es ist aus zwei Gründen möglich, daß diese Abwägung auf SVM-basierten Systemen günstiger ausfällt: Zum einen dauern bei heutigen Systemen die Unterbrechungen durch Kommunikation (also Seitenfehler) bei den heute üblichen, softwarebasierten SVM-Implementationen vergleichsweise lange, und zum anderen handelt es sich bei der Kommunikation durch SVM um eine sogenannte *einseitige Kommunikation*, d.h. eine Anfrage eines Rechenprozesses (=Seitenfehler) wird in eine unmittelbare Reaktion auf der Empfängerseite (=Seitentransfer durch das Betriebssystem) umgesetzt, ohne daß ein möglicherweise inaktiver Benutzerprozeß aktiv werden und explizit antworten muß. Ein nicht vollständiges *Gang Scheduling* wirkt sich also nur an Synchronisationspunkten im Programm aus.

Die Frage, inwieweit Kontextumschaltungen sich generell bei Seitenfehlern in parallelen Programmen auswirken, wurde in [BHMW94] für eine Reihe realer Programme in einer Simulationsumgebung untersucht. Den Vergleichen gingen zwei Untersuchungen voraus, in denen die Datenlokalität (und damit die Effizienz von Demand Paging) sowie die Wichtigkeit von *Gang-Scheduling* untersucht wurden. Das Ergebnis der ersten Untersuchung ist, daß numerische Anwendungen nur eine vergleichsweise geringe Datenlokalität besitzen, ein Demand Paging also nach Möglichkeit vermieden werden sollte: Sofern der einer Anwendung zur Verfügung stehende Realspeicher nur um 10 bis 20 Prozent kleiner ist als der von der Anwendung benötigte, steigt deren Ausführungszeit schon auf ein mehrfaches. An einen Scheduler ergibt sich damit die Forderung, die Anzahl aktiver Tasks auf einem Prozessor so zu begrenzen, daß ihre Datenanforderungen durch den Hauptspeicher befriedigt werden können. Die zweite Voruntersuchung zeigte, daß ein mögliches Timesharing mehrerer paralleler Programme aber nicht unkoordiniert erfolgen darf, da normale, parallele Anwendungen regelmäßige Synchronisationspunkte beinhalten und nur dann „gut laufen", wenn alle Tasks gleichzeitig aktiv sind.

Für den Fall, daß Paging in geringem Umfang auftritt, wurden aus diesen Ergebnissen folgende zwei Strategien untersucht:

- Der Prozessor wartet aktiv auf den Empfang der Seite.
- Es findet eine parallele Kontextumschaltung statt, d.h. ein Seitenfehler auf einem Prozessor hält sämtliche Threads des parallelen Programmes an.

Die Wahl von koordinierten Umschaltungen durch *Gang Scheduling* anstelle nurlokaler Umschaltungen wird durch die Vermutung gestützt, daß bei der Unterbrechung einer Task das Fortkommen der gesamten Anwendung fraglich ist und es

daher günstiger ist, das ganze Programm anzuhalten. Die Ergebnisse zeigen jedoch, daß parallele Kontextumschaltungen eine so aufwendige Operation darstellen, daß ihr Overhead den Zeitgewinn durch Ausnutzung der Wartezeiten überwiegt; es ist günstiger, einen Prozessor auf seine Seite aktiv warten zu lassen. Die Autoren weisen allerdings ausdrücklich darauf hin, daß dieses Ergebnis allein durch den Kommunikationsoverhead des *Gang Scheduling* verursacht wird; rein lokale Umschaltungen könnten eine durchaus interessante Alternative darstellen, wenn auf den Prozessoren sequentielle „Hintergrund-Jobs" vorhanden sind, auf die mit wesentlich geringerer Verzögerung umgeschaltet werden kann. Solche Hintergrundlasten standen allerdings im Rahmen dieser Untersuchung nicht zur Verfügung. Insgesamt werden parallele Kontextumschaltungen nur unter dem Gesichtspunkt einer gerechteren Rechenzeitverteilung als sinnvoll angesehen, aufgrund des hohen Overheads durch die Synchronisation der lokalen Scheduler und eventueller Ein-/Auslagerungsoperationen auf Sekundärspeicher sollten die Scheduling-Perioden aber vergleichsweise lang sein (im Bereich von Sekunden).

Ein genau konträrer, d.h. extrem feingranularer Ansatz zum Verstecken und Ausnutzen von Wartezeiten durch Speicherfehler wird in [GrUn95] mit dem Konzept einer *vielfädigen Prozessorarchitektur* vorgeschlagen. Die Idee dieses Konzeptes ist es, den Prozessor mit einem zusätzlichen Speicher zur Speicherung von mehreren Prozeßkontexten zu versehen. Dieser Speicher soll es erlauben, sehr schnell (im Extremfall zwischen einzelnen Maschineninstruktionen) zwischen verschiedenen Prozessen umzuschalten. Ein Round-Robin-Betrieb zwischen diesen Prozessen wäre praktisch ohne Overhead möglich, und auch sehr kurze Verzögerungen, die auf normalen Architekturen keine Umschaltungen rechtfertigen würden, können durch Rechnungen anderer Prozesse ausgenutzt werden. Nachteilig an diesem Konzept ist, daß es zum einen eine zugeschnittene Hardware in Form erweiterter Prozessoren benötigt, um seine Effizienz zu erreichen, und zum anderen bedingt es, daß ein Programm in so viele Kontrollfäden aufgeteilt werden muß, daß immer rechenbereite Prozesse übrigbleiben. Diese Aufteilung könnte bei numerischen Anwendungen mit regulärer Struktur durch einen Compiler erfolgen. Es bleibt aber das Problem bei der Anwendung auf SVM-Systeme, daß Speicherfehler durch die Aufteilung des Speichers in Seiten sehr gebündelt auftreten können: Ordnet man z.B. jeder Iteration einer Schleife einen eigenen Kontrollfaden zu, so wird ein Seitenfehler bei vorzugsweise sequentiellen Speicherzugriffen zur gleichzeitigen Blockade von sehr vielen Prozessen führen. Damit der Prozessor in dieser Situation nicht leerläuft, müßte sein „Kontextspeicher" extrem groß werden. Eine solchermaßen vielfädige Architektur eignet sich daher eher zum Überbrücken vergleichsweise kurzer Verzögerungen, wie sie z.B. durch Cache-Misses auftreten können.

3.1.3 Thread-Scheduling

Das Thread-Scheduling stellt die unterste und am feinsten verteilende Komponente eines Scheduling-Systemes dar. Aufgabe des Thread-Schedulings ist es allgemein, die in einem parallelen Programm vorhandene Arbeit auf die vorhandenen Kontrollflüsse (die, im Sinne einer Geschwindigkeitssteigerung, auf unterschiedlichen Prozessoren laufen) aufzuteilen. Bei numerischen Anwendungen findet diese Aufteilung üblicherweise auf Basis eines sogenannten *Datenparallelismus* statt, d.h. es sind gleichartige Operationen auf große Datenfelder anzuwenden, typischerweise in Form von Schleifen, deren Iterationen parallel ausgeführt werden können. Aufgrund der Feingranularität solcher Verteilungen wäre ein Thread-Scheduling durch das Betriebssystem selbst meist zu aufwendig und wird eigenständig im Laufzeitsystem unter Kontrolle der Anwendung vorgenommen; das Betriebssystem stellt nur die Kontrollflüsse zur Verfügung sowie Hilfsmittel, um eventuell erforderliche Kommunikation der Prozesse untereinander zu ermöglichen.

Es existieren eine Reihe von Spracherweiterungen für Hochsprachen, die es erlauben, durch zusätzliche Annotationen eines Programmes die Parallelisierung und Arbeitsverteilung zu steuern. Stellvertretend für solche Spracherweiterungen soll an dieser Stelle *SVM-Fortran* [BeGe95] vorgestellt werden, da diese Erweiterung, wie der Name bereits erkennen läßt, besondere Sprachkonstrukte für das effiziente Thread-Scheduling von SVM-basierten Programmen zur Verfügung stellt.

SVM-Fortran ist eine Spracherweiterung für Fortran-77, die zusätzliche Konstrukte zur Parallelisierung von Programmen auf Rechnern mit einem globalen Adreßraum zur Verfügung stellt. Die Realisierung dieses globalen Adreßraumes ist nicht Teil der Sprachdefinition bzw. -implementierung, sondern wird zur Laufzeit durch das Betriebssystem in Zusammenarbeit mit einer Laufzeitbibliothek vorgenommen. Dies ermöglicht es prinzipiell, SVM-Fortran-Programme auch auf Rechnern auszuführen, die ihren globalen Adreßraum nicht per SVM realisieren (die Befehle, die zur Kommunikation mit dem SVM-Laufzeitsystem dienen, haben in diesem Falle keine Wirkung). Alle Konstrukte sind als Kommentare ausgeführt, so daß ein normaler Fortran-Compiler für einen sequentiellen Rechner diese überliest und ein SVM-Fortran-Programm ohne Veränderung sequentiell ausgeführt werden könnte. Vom SVM-Fortran-Compiler werden diese Kommentare jedoch ausgewertet und in Aufrufe des Laufzeitsystems übersetzt.

Das prinzipielle Ausführungsmodell von SVM-Fortran ist das SPMD-Modell, d.h. alle Prozessoren erhalten den gleichen Programm-Code zur Ausführung, können zur Laufzeit jedoch durch unsynchronisierten Ablauf oder entsprechende parallele Konstrukte zur gleichen Zeit unterschiedliche Teile dieses Codes ausführen. Im Gegensatz zum High-Performance-Fortran werden nicht die Daten des Programmes

explizit über die Prozessoren verteilt, sondern es wird die im Programm vorhandene Rechenarbeit aufgeteilt. Eine dieser Arbeitsverteilung angepaßte Datenverteilung ergibt sich durch das SVM-Management dann in vielen Fällen automatisch. Die Organisation dieses Parallelismus wird durch *Prozessor-Sets* erreicht. *Prozessor-Sets* stellen Untermengen aus der Gesamtzahl aller vorhandenen Prozessoren oder einem anderen *Prozessor-Set* dar, die Teile eines SVM-Fortran-Programms gemeinsam ausführen. *Prozessor-Sets* können entweder explizit durch Anweisungen erzeugt werden oder entstehen implizit durch parallele Konstrukte. Bei diesen parallelen Konstrukten handelt es sich entweder um parallele Regionen (deren Anfang und Ende durch SVM-Fortran-Direktiven festgelegt werden) oder um parallele Schleifen, die durch das Konzept der schachtelbaren Prozessor-Sets auch mehrdimensionalen Parallelismus zulassen.

SVM-Fortran bietet eine Reihe von Scheduling-Strategien für parallele Schleifen an, die vom Programmierer durch entsprechende Parameter des Schleifenbefehls gewählt werden können. Die vorhandenen Strategien zerfallen in eine Reihe von Kategorien:

- Statische Verteilungen
- Dynamische Verteilungen
- Align-Scheduling
- Scheduling mittels Templates

Statische Verteilungen teilen die vorhandenen Iterationen anhand eines vor der Ausführung der Schleife festliegenden Schemas auf die Prozessoren eines *Prozessor-Sets* auf. Der Vorteil an diesen Verteilungen ist ihr geringer Overhead, da die Verteilungsvorschrift zur Laufzeit global bekannt ist und jeder Prozessor seinen Iterationsraum lokal bestimmen kann, ohne mit seinen Partnern kommunizieren zu müssen. Im einzelnen existieren folgende Verteilungsstrategien:

- `BLOCK` bzw. `REPBLOCK`: Der Gesamtiterationsraum wird in eine Zahl zusammenhängender Blöcke unterteilt, die der Anzahl beteiligter Prozessoren entspricht. Die beiden Varianten unterscheiden sich nur in der Aufteilung des „Restes", falls die Zahl der Iterationen nicht glatt durch die Zahl beteiligter Prozessoren teilbar ist oder weniger Iterationen als Prozessoren vorhanden sind.
- `CYCLIC` bzw. `REPCYCLIC`: Iterationen werden zyklisch auf die Prozessoren verteilt. Bei Schleifen, deren Rechenaufwand pro Iteration dreiecksförmig steigt oder fällt, kann auf diese Weise eine gleichmäßige Lastverteilung mit geringem

Overhead erreicht werden. Die zweite Variante erlaubt replizierte Ausführung einer Iteration, falls mehr Prozessoren als Iterationen zur Verfügung stehen. Optional kann die Anzahl zusammenhängend verteilter Iterationen von eins auf einen anderen Wert erhöht werden (diese Variante wird an anderen Stellen auch häufig als *blockzyklische* Verteilung bezeichnet).

Dynamische Verteilungen [Ott95] sind dann angebracht, wenn der Rechenaufwand pro Schleifeniteration in einer nicht im voraus abschätzbaren Weise schwankt. Alle dynamischen Verfahren basieren auf einem Pool noch nicht abgearbeiteter Iterationen, aus dem auf Anfrage Pakete von Iterationen ausgegeben werden. Die Anzahl ausgegebener Iterationen ist vom Algorithmus im einzelnen abhängig:

- `SELF`: Pro Anfrage wird lediglich eine Iteration vergeben. Bei vernachlässigbarem Overhead für das Aushändigen würde dieses Verfahren durch seine Feingranularität eine optimale Arbeitslastverteilung erreichen.
- `CHUNK`: Pro Anfrage wird eine feste, vorgebbare Zahl von Iterationen ausgegeben. Dieses Verfahren reduziert den Scheduling-Overhead auf Kosten der Genauigkeit der Lastbalance und ist eine Verallgemeinerung des `SELF`-Algorithmus.
- `GUIDED` bzw. `BLOCK_GUIDED` geben gemäß [PoKu87] immer kleinere Pakete aus und versuchen auf diese Weise, gleichzeitig niedrigen Overhead und eine nahezu perfekte Lastbalance zu erreichen. Die geblockte Variante erlaubt es zusätzlich, Paketgrößen auf ein mehrfaches einer vorgegebenen Zahl aufzurunden, um potentielle *false-sharing*-Effekte zu vermeiden.
- `FACTORING` bzw. `BLOCK_FACTORING` arbeiten nach einem ähnlichen Verfahren wie die `GUIDED`-Algorithmen, geben jedoch gemäß [HSF91] zu Beginn kleinere Pakete aus. Diese Verfahren haben Vorteile, wenn Gefahr besteht, daß einzelne Prozesse über einen längeren Zeitraum den Prozessor entzogen bekommen und ihre anfänglich erhaltenen, großen Pakete nicht bearbeiten können.

Wo und *wie* die noch unbearbeiteten Iterationen gespeichert werden, ist nicht Teil der Sprachdefinition; im Sinne der Vermeidung eines Flaschenhalses wird eine dezentralisierte Implementation bevorzugt, wie in [Ott95] beschrieben.

Align-Scheduling erlaubt es, die Arbeitsverteilung einer momentan vorhandenen Datenverteilung auf den Prozessoren anzupassen. Zu diesem Zweck wird mit der Direktive `ON HOME(`*Variable*) ein (von der Schleifenvariablen abhängiger) Variablenausdruck angegeben. Jeder Prozessor evaluiert diesen Ausdruck für den gegebenen Iterationsraum und führt die Iterationen aus, deren Datenelemente er lokal

verfügbar hat. Zusätzliche Synchronisationen verhindern, daß Iterationen mehrfach oder überhaupt nicht ausgeführt werden, falls SVM-Daten auf den Sekundärspeicher ausgelagert waren. Dieses Verfahren ist direkt auf SVM zugeschnitten und erlaubt eine Minimierung von Seitenfehlern (bezüglich eines Feldes), kann potentiell aber sehr große Lastimbalancen erzeugen, falls die vorherige Datenverteilung entsprechend unausgeglichen war.

Templates bieten die Möglichkeit, beliebige Verteilungen zu erzeugen, zu speichern und bei Bedarf abzurufen. Logisch betrachtet stellt ein Template ein Feld dar, in dem für jede Iteration der Prozessor gespeichert ist, der diese Iteration ausführen soll. Die Dimension eines Templates muß dabei nicht der Dimension der auszuführenden Schleife entsprechen; einzelne Dimensionen einer mehrdimensionalen Schleife, die nicht parallel ausgeführt werden sollen, können sequentialisiert oder repliziert werden. Templates ermöglichen es zum Beispiel, eine einmal in einer parallelen Schleife benutzte Arbeitsverteilung zu speichern und wiederzuverwenden, wenn die gleiche Schleife nochmals ausgeführt wird. Auf diese Weise kann zum Beispiel das in [Li94] beschriebene *Affinity-Scheduling* realisiert werden.

SVM-Fortran unterstützt die Definition von verschiedenen Speicherklassen: Variablen dürfen die Klassen `PRIVATE`, `SHARED` und `PSHARED`. Private Variablen existieren auf jedem Prozessor als eigene, nur von diesem Prozessor ansprechbare Instanz, während die beiden anderen Typen durch das SVM-Management global bzw. im momentan aktiven Prozessor-Set zugreifbar sind.

Zur Steuerung des unterliegenden SVM-Managements erlaubt SVM-Fortran das explizite Locking von Speicherregionen, d.h. ein Prozessor darf einen durch `LOCK`-Direktiven begrenzten Bereich nur bearbeiten, wenn er zuvor exklusiven Zugriff auf eine bestimmte Menge von Variablen erlangt hat. Es ist ebenso möglich, Datenelemente mittels *Prefetch*-Anweisungen vorausschauend anzufordern, um die durch Seitenfehler verursachten Latenzzeiten zu verdecken. Bei beiden Konstrukten hat jedoch der Programmierer selbst zu beachten, daß diese Operationen nur auf Basis der kleinsten Kohärenzeinheit, nämlich einer Seite erfolgen können; SVM-Fortran erlaubt zu diesem Zweck, die Seitengröße zur Laufzeit abzufragen.

3.2 Varianten der Grundstruktur

Die in den vorherigen Abschnitten anhand von Beispielen vorgestellte Grundstruktur eines Scheduling-Systems für Parallelrechner werden in einer Reihe von Arbeiten abgewandelt, um ihre Leistungsfähigkeit für bestimmte Hardware-Strukturen oder Nutzungsformen zu verbessern. Ein häufiger Ansatzpunkt ist der Job-Scheduler. Da dieser nur einmal vorhanden ist und die Programmverteilung und -überwachung

für das Gesamtsystem zu leisten hat, kann er bei größeren Konfigurationen mit vielen Prozessoren und/oder Programmen Skalierungsprobleme bereiten. Um dieses Problem zu umgehen, werden von einer Reihe von Arbeiten *Baumstrukturen* vorgeschlagen, d.h. zwischen einem einzelnen, global arbeitenden Job-Scheduler sowie den lokalen Task-Schedulern werden Schichten aus „Zwischen-Schedulern" gelegt, die jeweils den Teil des Job-Scheduling für ihren Unterbaum übernehmen und Informationen verdichtet nach oben weiterreichen (Bild 3.2).

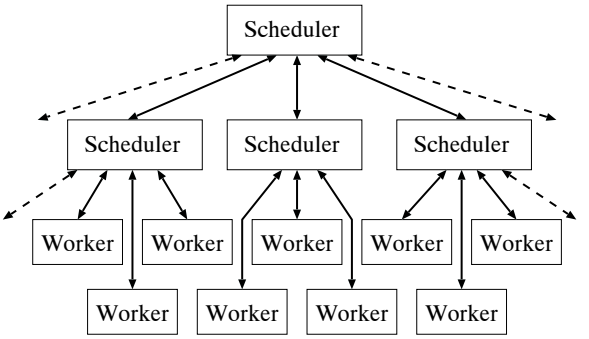


Abbildung 3.2
baumstrukturiertes Scheduling-System

Eine solche Struktur wurde vor einigen Jahren von Tilborg und Wittie unter dem Namen *Wave Scheduling* in [TiWi84] vorgestellt. Dabei bilden die eigentlichen Rechenknoten die unterste Schicht des Baumes und der darüberliegende Teilbaum realisiert den verteilten Scheduler. Trifft nun an einer beliebigen Stelle des Netzes ein Job ein, so vergleicht der Manager-Knoten die angeforderte Zahl von Prozessoren mit der Zahl momentan freier Prozessoren in dem von ihm verwalteten Unterbaum. Überschreitet die Anforderung einen vorher festgelegten Prozentsatz der freien Prozessoren, so reicht der lokale Manager die Anforderung an seinen Vater im Baum weiter, der selbst wieder die gleiche Prüfung durchführt. Auf diese Weise pflanzt sich die Anfrage wie eine Welle evtl. bis zur Wurzel fort, wo sie entweder sofort befriedigt, in eine Warteschlange eingereiht oder zurückgewiesen wird. Analog wandert eine Anfrage im Baum nach unten, bis ein Manager gefunden ist, der die Anfrage gerade eben noch befriedigen kann. Der Algorithmus setzt voraus, daß die unteren Manager-Knoten hinreichend oft (die benötigte Rate ist abhängig von der Ankunftsrate von Jobs) an ihren Vaterknoten Informationen über die momentane Auslastung schicken. Da diese Informationen asynchron zum Eintreffen von Jobs versandt werden sollen, sind sie naturgemäß nie ganz exakt. Die weiter oben bereits erwähnte Strategie, nie alle noch freien Knoten an einen Job zu vergeben, soll verhindern, daß aufgrund veralteter Informationen ein Job zu tief in dem Baum herunterwandert und wieder zurückgereicht werden muß. Diese Reserve darf naturgemäß einerseits nicht zu hoch angesetzt werden, um die Systemauslastung nicht unnötig zu verschlechtern,

andererseits nicht zu niedrig, da sonst der Fall wahrscheinlich wird, daß ein Manager mitten in der Anforderung von Rechenknoten von seinen Sub-Managern feststellt, daß in Wirklichkeit gar nicht mehr genug Rechenknoten vorhanden sind und bereits gemachte Reservierungen wieder zurückgenommen werden müssen. Rechnungen ergeben, daß der Optimalwert von der momentanen Systemauslastung (d.h. von der Wahrscheinlichkeit, eine bestimmte Rechenknotenzahl reservieren zu können) abhängt. In [TiWi84] wird das Problem so weit vereinfacht, daß jeder Managerknoten nur noch eine Tabelle benötigt, aus der der von der Auslastung abhängige „Zuschlagsfaktor" ausgelesen werden kann. Die Effizienz von Wave-Scheduling ist abhängig von der Geschwindigkeit, mit der sich die einzelnen Anfragen und Reservierungen im Netzwerk ausbreiten. Bei Systemauslastungen nahe 100 Prozent wäre ein zentraler Scheduler effizienter, da dieser alle Entscheidungen ohne Nachrichtenaustausch fällen kann.

Feitelson und Rudolph greifen dieses System hierarchischer Controller in dem von ihnen vorgeschlagenen *Distributed Hierarchical Control*-Konzept wieder auf [FeRu90-1, FeRu90-2]. Sie verwenden es aber nicht nur zur Zuordnung von Prozessen zu Prozessoren, sondern auch zum Laufzeit-Scheduling der einzelnen rechenbereiten Jobs. Es ist dabei möglich, einzelnen Rechenknoten mehrere Prozesse zuzuordnen und diese in einem Zeitscheibenverfahren ablaufen zu lassen. Zur Laufzeit wird dann vom „Betreuer" eines Jobs (d.h. von dem Managerknoten, der gerade so weit oben in der Hierarchie steht, daß alle betroffenen Rechenknoten in seinem Unterbaum liegen) ein Befehl an seine Unterknoten ausgesandt, wenn dessen Zeitscheibe beginnt. Da sich die Anweisungen gleichzeitig nach unten in alle Teilbäume ausbreiten, ist gewährleistet, daß die Ausführung aller Prozesse nahezu zeitgleich auf den Rechenknoten beginnt. Damit ist die für die Ausführung paralleler Prozesse sehr erwünschte Eigenschaft des *gang scheduling* erreicht, da so Zeitverluste durch das Warten auf noch nicht laufende Partnerprozesse minimiert werden. Für die Festlegung der Zeitscheibenlänge wird eine Gewichtung entsprechend der Prozeßzahl vorgeschlagen, so daß größere Jobs entsprechend längere Zeitscheiben erhalten. Eine scheinbar unfaire Behandlung kleinerer Jobs tritt dabei nicht auf, solange kleinere Jobs in entsprechend höherer Zahl auftreten, eine für reale Systeme durchaus sinnvolle Annahme.

Da in dem ganzen Modell untergeordnete Manager autark über die Verteilung von Jobs entscheiden können, die ihre Kapazität nicht überschreiten, wird die normalerweise schlechte Ausfallsicherheit von Baumstrukturen (Ausfall des Wurzelknotens = Ausfall des Gesamtsystemes) abgemildert. Selbst der Ausfall eines Managers während des laufenden Betriebs bedeutet nicht, daß von ihm behandelte Jobs verloren sind: Seine nachgeordneten Controller sind am Ausbleiben von Anweisungen in der Lage, Störungen zu erkennen und auf diese Weise das Scheduling „seiner" Jobs mit zu übernehmen. Aufgrund der fehlenden Synchronisation kann jetzt nur kein vollständiges Gang-Scheduling mehr erfolgen. Lediglich der Ausfall von Rechenknoten führt zum Abbruch von Jobs, die in Teilen auf ihnen ausgeführt wurden.

3.3 Existierende parallele Betriebssysteme

Aufgrund der - gerade auch im Vergleich zu sequentiellen Rechnern - sehr hohen Komplexität von Betriebssystemen für parallele Rechnersysteme gibt es neben einer Reihe von proprietären Implementationen auch einige universelle, in gewissem Maß portable Betriebssysteme, die durch ihre Grundstruktur einem möglichst breiten Spektrum von Architekturen gerecht zu werden suchen. Häufig sind diese Betriebssysteme mit akzeptablem Aufwand auf weitere Parallelrechner anpaßbar. Als Beispiel solcher portabler Betriebssysteme, die in der Realität eine gewisse Verbreitung gefunden haben, sollen im folgenden die Betriebssystemkerne *Mach* und *Chorus* besprochen werden. Als Beispiel eines proprietären Betriebssystems wird das *UNICOS* in seiner Implementation für Vektorrechner der Firma Cray Research vorgestellt.

3.3.1 Mach

Mach ist ein an der Carnegie Mellon University entwickelter Betriebssystemkern [RJOS89, RBFG89], der zur universellen Verwendung sowohl auf sequentiellen als auch auf parallelen Rechnern entwickelt wurde. Mach selbst bildet kein vollständiges, lauffähiges Betriebssystem, sondern stellt die Basis für die Entwicklung von Systemen dar, wie z.B. OSF/1 der Open Software Foundation, das wiederum für eine Reihe existierender sequentieller und paralleler Rechnerarchitekturen als Betriebssystembasis dient. Ein Beispiel für eine Implementierung für sequentielle oder moderat parallele Rechner mit gemeinsamem Speicher ist Digital Unix der Firma Digital Equipment; Implementationen für massiv-parallele Rechner existieren z.B. auf der Intel Paragon, der KSR-1 oder der Convex SPP. Vom Mach-Kernel existieren momentan zwei Varianten: Frühere Versionen (bis 2.x) waren von vornherein als Kernels für Unix-artige Betriebssysteme konzipiert und beinhalten daher Teile, die die für Unix-Anwendungen benötigte Funktionalität bereitstellen. Kernels dieser Generation sind *monolithisch* konzipiert, d.h. der Kernel stellt eine logische Einheit dar, die nicht auf mehrere Prozessoren verteilt werden kann. In Version 3 von Mach wurde dieses Konzept durch einen *Microkernel* ersetzt, der lediglich Basisfunktionen wie Interprozeßkommunikation zur Verfügung stellt und auf dem beliebige Betriebssystemschnittstellen implementiert werden können. Diese können als voneinander unabhängige, miteinander kommunizierende Tasks prinzipiell beliebig über ein paralleles System verteilt werden.

Während in einem normalen Unix-Betriebssystem der Prozeß die einzige Einheit bildet, auf dessen Basis Scheduling, Betriebsmittelvergabe und Schutzmechanismen arbeiten, unterscheidet Mach zwischen *Tasks* und *Threads* [Bla90]. Betriebsmittelvergabe und Schutzmechanismen arbeiten auf der Ebene von Tasks, so daß die in

einer Task arbeitenden Threads gleichberechtigt auf alle Ressourcen einer Task zugreifen können. Die Tatsache, daß alle Threads einer Task im gleichen Kontext ablaufen, erlaubt es, zwischen diesen Threads mit einem wesentlich geringeren Overhead umzuschalten, als es mit normalen Unix-Prozessen möglich wäre; daraus resultiert eine effizientere Bearbeitung paralleler Programme.

Mach 2.x wurde in erster Linie für Multiprozessorsysteme mit einem physikalisch gemeinsamen Speicher konzipiert. Demzufolge ist die zentrale Datenstruktur des Schedulers eine *run queue*, die alle rechenbereiten Threads, nach Priorität in Listen sortiert, beinhaltet. Der Scheduler arbeitet nur auf Thread-Ebene und hat keine Kenntnis über die Zugehörigkeit von Threads zu bestimmten Tasks. Wenn ein Prozessor einen Thread beendet oder blockiert hat, sucht er sich aus dieser Liste den rechenbereiten Thread mit der höchsten Priorität heraus. Spezielle „Hint"-Zeiger, die auf die höchstpriorisierte Warteschlange zeigen, in der sich rechenbereite Threads befinden, ermöglichen ein schnelleres Absuchen der Warteschlange. Zusätzlich zu der globalen *run queue* existiert pro Prozessor eine lokale *run queue*, in der sich Threads befinden, die an einen bestimmten Prozessor gebunden werden müssen. Dies ist z.B. der Fall, wenn Interrupts auf einem bestimmten Prozessor auftreten und nur bedient werden können, wenn die Interrupt-Service-Routine auch auf diesem Prozessor ausgeführt wird. Die in lokalen Listen liegenden Threads haben im Scheduling Vorrang.

Das Scheduling-Konzept von Mach in der bisher vorgestellten Form beinhaltet keine besonderen Mechanismen, um die Ausführung paralleler, aus mehreren Treads bestehender Programme zu unterstützen. Da der Scheduler keinerlei Kenntnis über die Zugehörigkeit von Threads zu Tasks besitzt, kann er von sich aus weder eine abgestimmte Verteilung auf die zur Verfügung stehenden Prozessoren vornehmen noch eine koordinierte Ausführung paralleler Threads erreichen. Um den Scheduler in solchen Situationen zu unterstützen, stellt Mach zwei Mechanismen zur Verfügung: *Hints* und *Processor Allocation*. Mittels *Hints* können Benutzer-Threads dem Scheduler Hinweise geben, daß ihre Ausführung unterbrochen werden sollte, möglicherweise zu Gunsten eines anderen Threads. Diese Möglichkeit erlaubt es zum Beispiel, bei kooperierenden Prozessen mit Erzeuger-Verbraucher-Verhalten das Scheduling dem Datenfluß anzupassen und so Leerlaufzeiten zu vermeiden. Hints können unterschiedlich stark sein (bis zu absoluten *discouragements*, die dem Aufrufer den Prozessor auf jeden Fall entziehen), es ist jedoch nicht möglich, daß ein Thread sich selbst bevorzugt oder seine Zeitscheibe eigenmächtig verlängert; eine Monopolisierung von Prozessoren durch einzelne Threads ist damit nicht möglich.

Der Mechanismus der *Prozessor-Allokation* erlaubt es Anwendungen, Gruppen von Threads Gruppen von physikalischen Prozessoren zuzuordnen. Solange die Zahl von Threads in einer Gruppe kleiner oder gleich der Zahl von Prozessoren in der zugeordneten Gruppe ist, kann dann jeder Thread exklusiv auf einem Prozessor

bearbeitet werden, die Anwendung läuft im echten Parallelbetrieb ab. Je nach Strategie der Prozessor-Zuordnung kann das System aber auch entscheiden, der Prozessorgruppe weniger Prozessoren zuzuordnen; zwischen den Threads der Prozeßgruppe findet dann ein Timesharing mit dem Ziel statt, allen Threads einen möglichst gleichen Anteil an der vorhandenen Rechenleistung zuzuteilen. Die Entscheidungen über die Allokation von Prozessoren fallen dabei nicht im Kernel selbst, sondern in einem externen Server, an den Programme über definierte Ports Anfragen stellen können. Auf diese Weise ist es möglich, ohne Neustart oder Neuübersetzung des Kernels wechselnde Allokationsstrategien zu implementieren. Der Nachteil ist der höhere Overhead durch die Kommunikation zwischen drei statt zwei Komponenten; da der Allokationsmechanismus jedoch bevorzugt zur längerdauernden Allokation (für Zeiten im Sekunden- oder sogar Minutenbereich) konzipiert wurde, fällt dies in der Praxis nicht weiter ins Gewicht.

In Mach 3.0 wurde das Konzept eines monolithischen Kernels zugunsten eines *Microkernels* aufgegeben. Ein Microkernel stellt nur die minimal auf jedem Prozessor notwendigen Systemdienste zur Verfügung, die direkt auf der Hardware aufsetzen und daher mit der höchsten Privilegstufe laufen müssen. Zu diesen Diensten gehören der Thread-Scheduler, Funktionen zur Interprozeßkommunikation sowie die Speicherverwaltung, von der allerdings nur die Seitenverwaltung im Kernel realisiert ist - das Ein- und Auslagern von Seiten erfolgt über außerhalb des Kernels liegende Server. Ein Microkernel hat für Parallelrechner mit verteiltem Speicher den Vorteil, daß die dort auf jedem Prozessor erforderliche Kopie des Kernels kleiner ist und die restlichen, in einzelne Server gegliederten Prozesse beliebig auf die vorhandenen Prozessoren verteilt werden können. Die vom Microkernel zur Verfügung gestellte Interprozeßkommunikation stellt sicher, daß die zwischen den Threads laufenden Nachrichten an die korrekten Zielprozessoren weitergeleitet werden. Weiterhin ermöglicht die Auslagerung jeglicher Unix-Funktionalität aus dem Kernel, auf einem Mach 3.0-basierten System völlig neu strukturierte Betriebssysteme mit anderen Systemschnittstellen aufzusetzen, ohne zusätzlichen Overhead für die Umsetzung eines Betriebssystem-Interfaces in ein anderes einzuführen.

3.3.2 Chorus

Chorus ist das Ergebnis eines Forschungsprojektes bei INRIA[1], Betriebssysteme für verteilte Systeme zu entwickeln. Die momentane Version 3 von Chorus ist das Ergebnis mehrerer evolutionärer Stufen [RAAB91], die ausgehend von Experimenten mit Elementen verteilter Betriebssysteme zu einem vollständigen, kommerziell einsetzbaren System mit optionaler Unix-kompatibler Systemschnittstelle geführt haben.

[1]Institut National de Recherche en Informatique et Automatique

Chorus bildet z.B. die Grundlage für das Betriebssystem des massiv-parallelen Rechnersystems T3E der Firma Cray Research. Bei der Konzeption von Chorus bestand das Ziel von vornherein darin, dieses System auf einer sehr breiten Klasse von Rechnerarchitekturen einzusetzen, die sich sowohl in ihrer Speicherarchitektur als auch in der Leistungsfähigkeit des Kopplungsnetzwerks unterscheiden. Das Konzept, das bei Chorus daher von Anfang an verfolgt wurde, ähnelt daher dem von Mach 3, d.h. die einzige Betriebssystemkomponente, die zwingend auf jedem Prozessor vorhanden sein muß, ist der *Nucleus*, d.h. der Systemkern, der für die Funktion von Chorus auf jedem Prozessor resident vorhanden sein muß und Systemdienste zur Verfügung stellt, auf denen alle anderen Komponenten von Chorus aufbauen:

- Der *supervisor* nimmt die von der Hardware ausgelöste Ereignisse (z.B. Interrupts) entgegen und stößt die zugeordneten Reaktionen an;
- der *real-time executive* steuert Prozessor-Scheduling und -Allokation; außerdem stellt er Synchronisationsprimitive zur Verfügung;
- der *virtual memory manager* übernimmt die Speicherverwaltung;
- der *inter-process communication manager* stellt Mittel für die Kommunikation (möglicherweise verteilter) Prozesse zur Verfügung.

Obwohl einige dieser Funktionen (speziell die Teile der Interprozeßkommunikation) auch außerhalb der Kernels realisierbar wären, wurde aus Effizienzgründen entschieden, sie in den Kernel zu verlagern.

Die Prozeßstruktur ähnelt der von Mach: Ein Speicherkontext, Kommunikationskanäle und Ausführungskontext werden in einem *Aktor* zusammengefaßt, der wiederum mehrere Threads beinhalten kann. Alle Threads eines Aktors haben gleichberechtigten Zugriff auf die Ressourcen des Aktors. Da ein Thread auf diese Weise an seinen Aktor gebunden ist, ist es nicht möglich, Threads eines Aktors auf verteilten Prozessoren auszuführen. Eine Ausnahme bilden symmetrische Multiprozessorsysteme, bei denen alle Prozessoren gleichberechtigten Zugriff auf den Hauptspeicher besitzen. Das Scheduling ist funktionell nahezu identisch mit dem von Mach: Es arbeitet nur auf der Ebene von Threads, und der Basis-Scheduler berücksichtigt keine Zugehörigkeiten von Threads zu Aktoren. Er ist in der Lage, Threads zu unterbrechen und erlaubt die Zuordnung von Prioritäten; weiterhin unterstützt er wie Mach die Senkung von Prioritäten auf der Basis verbrauchter Rechenzeit. Zusammen mit der ebenfalls vorhandenen Möglichkeit der Prozessorallokation kann der Scheduler von Chorus einen weiten Anwendungsbereich abdecken, der von Timesharing-Systemen bis zu massiv-parallelen Rechnern mit dedizierter Ausführung auf jedem Prozessor reicht.

Ein Beispiel eines auf Chorus basierenden Betriebssystemes ist Chorus/MiX, das Anwendungen eine Unix-kompatible Systemschnittstelle zur Verfügung stellt. Durch die Abbildung eines Unix-Prozesses auf einen Aktor mit defaultmäßig nur einem Thread können die unter Unix üblichen Schutzmechanismen und Kommunikationskanäle mit wenig Aufwand auf vom Chorus-Nucleus zur Verfügung gestellte Dienste abgebildet werden. Weitere Unix-Dienste, die nicht im Kernel enthalten sind, werden über außerhalb des Kernels liegende Server realisiert. Zu diesen Diensten gehören z.B. Prozeßmanagement, Dateisystem oder I/O-Steuerung. Im Gegensatz zu „klassischen" Unix-Systemen mit einem monolithischen Kernel sind alle Server *multithreaded*, d.h. es können mehrere Anfragen verschiedener Prozesse an einen Server gleichzeitig aktiv sein, und es können keine Prozeßblockaden auftreten, weil ein Systemdienst momentan von einem anderen Prozeß genutzt wird. Auf diese Weise profitieren auch Programme, die nicht speziell auf Chorus/MiX angepaßt wurden, von seinen Vorteilen. In Programmen, die neu- oder umgeschrieben werden, können dann auch die durch den Nucleus zur Verfügung gestellten erweiterten Möglichkeiten genutzt werden. Beispiele dafür sind das Multithreading oder die transparente Verteilung von Prozessen auf verschiedene Prozessoren.

3.3.3 Cray UNICOS

UNICOS ist ein von der Firma Cray Research entwickeltes Betriebssystem, welches ursprünglich für die Vektorrechner des gleichen Herstellers implementiert wurde, später aber auch unter Zuhilfenahme eines Chorus-Microkernels auf den massivparallelen Rechner T3E portiert wurde. Bei den zuerst genannten Rechnern handelt es sich um schwach bis moderat parallele Rechnersysteme (bis zu 32 Prozessoren) mit einem physikalisch gemeinsamem Speicher. Eine für den Scheduler wichtige Vorgabe ist, daß die Hardware dieser Maschinen aus Performance-Gründen *keine* Speicherverwaltungseinheit besitzt, die ein *Demand Paging*, d.h. das Ein-und Auslagern momentan nicht in Gebrauch befindlicher Speicherbereiche eines Programmes erlauben würde. Die Möglichkeiten des Systems beschränken sich auf das *Swapping*, d.h. ein Job kann nur komplett aus- oder eingelagert werden. Da dies eine ungleich zeitaufwendigere Operation ist, muß der Scheduler bei seinen Entscheidungen den Speicherbedarf von Jobs stärker berücksichtigen, als dies bei Systemen mit *Demand Paging* nötig ist. Im Extremfall kann ein einzelner sequentieller Job, der zur Ausführung gelangt, soviel Speicher belegen, daß keine anderen Jobs mehr geladen werden können und eventuell Prozessoren leerstehen müssen.

Das Scheduling-System besteht aus den folgenden Komponenten ([Cr2099]):

- Der Unified Resource Manager (URM);
- der Memory Scheduler;
- der Fair-Share-Scheduler.

Dem URM obliegt die Entscheidung, welche Jobs aus einer Warteschlange herausgenommen werden und zur Ausführung gelangen. Die Entscheidung, welcher Job wann gestartet wird, versucht URM so zu treffen, daß die Ressourcenbelastung durch die gerade laufenden Jobs möglichst nahe an den vom Systemadministrator vorgegebenen Zielen liegt. Zu diesen Zielen gehört die Gesamtzahl laufender Jobs, der Gesamtspeicherverbrauch aller laufenden Jobs sowie bei eingeschaltetem Fair-Share-Scheduler eine Rechenzeitverteilung auf die Jobs einzelner Benutzer. URM startet keinen Job, der zu einer Überschreitung der Grenzen für Jobzahl oder Ressourcenverbrauch führen würde. Bei der Ermittlung des Ressourcenverbrauchs laufender Jobs berücksichtigt URM nicht nur den aktuellen Wert, sondern auch den beim Start des Jobs vorgegeben Maximalwert. Da Jobs üblicherweise ihre Ressourcen nicht direkt am Anfang, sondern erst begleitend während des Programmlaufes belegen, benutzt URM eine Glättungsfunktion, die aus Maximalwert und aktuellem Verbrauch einen Erwartungswert für die nahe Zukunft berechnet. Der Glättungsfaktor ist für jede Ressource individuell einstellbar, so daß der Erwartungswert sich, vom Maximalverbrauch ausgehend, mehr oder weniger schnell an die aktuellen Verhältnisse anpaßt. Auf diese Weise kann verhindert werden, daß Grenzen für die Nutzung von Ressourcen überschritten werden, weil bereits laufende Jobs ihren Ressourcenbedarf sprunghaft erhöhen, andererseits können sich die Schätzwerte trotzdem den realen Verhältnissen anpassen, wenn ein Job seine Grenzwerte nicht ausnutzt. Der Zyklus, in dem URM diese Werte neu ermittelt, läßt sich ebenso frei einstellen wie der Zyklus, in dem URM die Rangfolge der wartenden Jobs ermittelt und Jobs zum Start freigibt. Dabei kann es sich naturgemäß nur um Jobs handeln, deren Start nicht zu einer Überschreitung der oben angesprochenen Auslastungsgrenzen führt. Die Rangfolge der verbleibenden Jobs berechnet sich aus einer gewichteten Summe verschiedener Eigenschaften der Jobs. Zu diesen Eigenschaften gehören sowohl der Ressourcenbedarf der Jobs (Speicher, CPU-Zeit, Bandkapazität), die bisherige Wartezeit des Jobs, seine Priorität als auch die Menge an Ressourcen, die durch laufende Jobs des gleichen Benutzers bereits belegt werden. Durch Variation der Gewichtungsfaktoren läßt sich die Bedeutung einzelner Komponenten in weiten Grenzen einstellen. Um zu verhindern, daß Jobs, die entsprechend dieser Bewertung eine niedrige Einstufung erhalten, bei starker Maschinenauslastung nicht zum Zuge kommen, erlaubt es URM, die klassifizierten Jobs in 4 Klassen einzuteilen. Während im Normalfall nur die Jobs mit der höchsten Wertung gestartet werden, wählt URM bei einer

Klasseneinteilung mit einem geringen Prozentsatz auch Jobs aus den Klassen mit niedrigerer Wertung aus.

URM bearbeitet sowohl interaktive als auch Batch-Jobs; während letztere im Rhythmus der Neubewertung immer wieder neu auf Startbarkeit untersucht und neu bewertet werden, wird bei interaktiven Jobs sofort und einmalig über eine Annahme oder Ablehnung entschieden.

Dem *Fair-Share-Scheduler* obliegt es, allen Benutzern einen den Vorgaben des Systemadministrators entsprechenden Anteil an der Gesamtrechenleistung zur Verfügung zu stellen. In einem System ohne Fair-Share-Scheduler wird das Scheduling der laufenden Prozesse unabhängig davon vorgenommen, zu welchem Benutzer sie gehören. Dies führt im allgemeinen dazu, daß ein Benutzer mit einer größeren Zahl laufender Prozesse auch einen größeren Gesamtanteil an den Systemressourcen erhält. Der Fair-Share-Scheduler versucht, diesem Effekt entgegenzuwirken, indem er den Ressourcenbedarf der laufenden Prozesse beobachtet und für jeden Job einen „Nutzungsfaktor" berechnet, der sich aus einer gewichteten Summe von CPU-Zeitverbrauch, Speicherbedarf sowie der Anzahl der durchgeführten Systemaufrufe und I/O-Operationen berechnet. Die Gewichtungsfaktoren dieser einzelnen Anteile sind ähnlich wie bei URM frei einstellbar, so daß sich je nach Einsatzprofil und Konfiguration eines Systems die Entscheidungen des Fair-Share-Schedulers anpassen lassen. Dazu gehört auch das „Gedächtnis", d.h. ein Wert, der angibt, wie schnell der Scheduler Ressourcenverbräuche in der Vergangenheit vergessen soll. Die Wirkung ist ähnlich wie eine dynamische Variation der Prioritäten einzelner Prozesse, d.h. Anwender, die ihren vorgegebenen Anteil überschreiten, erhalten für ihre Prozesse eine niedrigere Priorität und damit vom eigentlichen Scheduler seltener Prozessorzeit, während umgekehrt Prozesse, die einen zu niedrigen Anteil erhalten, höhere Prioritäten bekommen. Im Extremfall kann der Fair-Share-Scheduler einen Prozeß ganz anhalten, wenn sein Ressourcenbedarf eine einstellbare Obergrenze erreicht.

URM kooperiert mit einem eventuell laufenden Fair-Share-Scheduler, indem die vom Fair-Share-Scheduler ermittelten Ressourcenverbräuche in die Entscheidung mit einbezogen werden, ob weitere Jobs eines Benutzers gestartet werden sollten.

Der Memory-Scheduler entscheidet, welche Jobs momentan im Hauptspeicher gehalten und welche Jobs auf den Sekundärspeicher ausgelagert werden. Für diese Entscheidung berechnet der Memory-Scheduler, analog zu den beiden eben beschriebenen Komponenten, für jeden Prozeß unabhängig davon, ob er momentan im Hauptspeicher oder auf ein externes Speichermedium ausgelagert ist, eine Priorität, die sich als Summe einer Reihe von Parametern darstellt, deren Gewichtung einstellbar ist. Zu diesen Parametern gehören:

- der Status eines Prozesses (ein- oder ausgelagert);
- der Speicherbedarf des Prozesses;
- die Priorität eines Prozesses und damit die Wahrscheinlichkeit, daß er CPU-Zeit erhält;
- die Zeitspanne, für die ein Prozeß ein- oder ausgelagert ist;
- die Jobklasse (Batch oder interaktiv).

Sortiert man alle Prozesse nach ihren Prioritäten, so läßt sich mit den Wissen um den verfügbaren Speicher einfach ermitteln, welche Prozesse momentan „wichtig genug" sind, im Hauptspeicher zu sein. Diese Entscheidung kann aber durch eine Reihe weiterer Parameter eingegrenzt werden:

Zum einen definiert der Memory-Scheduler den Begriff eines *hog*-Prozesses, d.h. eines Prozesses, der durch das Überschreiten eines bestimmten Speicher- oder CPU-Bedarfs eine besondere Belastung für das System darstellt. Man kann die Menge an Hauptspeicher begrenzen, die für solche Prozesse insgesamt zur Verfügung stehen soll, und Prozesse, deren Größe diese Menge überschreitet, werden überhaupt nicht mehr eingelagert. Umgekehrt kann man mittels des *big_proc*-Parameters festlegen, ab welcher Grenze ein Prozeß nicht mehr ausgelagert wird, da sein Auslagern eine zu große Verzögerung darstellen würde. Dieser Grenzwert liegt üblicherweise oberhalb der halben Hauptspeichergröße.

Um ein wiederholtes Ein- und Auslagern einzelner Jobs (sog. *trashing*) zu vermeiden, kann vorgegeben werden, für welche Zeitspanne ein Prozeß mindestens ein- oder ausgelagert bleiben muß, bevor er seinen Zustand verändern darf. Umgekehrt ist es auch möglich, die Zeit, für die ein Prozeß ausgelagert ist, zu begrenzen: Bei den gegebenen Einstellmöglichkeiten des Memory-Schedulers führt das Optimierungsziel, einen ausgelagerten Job im Sinne der Vermeidung von *trashing* auch bevorzugt ausgelagert zu belassen, dazu, daß ein Job mit steigender Zeit im ausgelagerten Zustand immer unwahrscheinlicher wieder eingelagert wird. Dieser Parameter kann helfen, zu verhindern, daß Prozesse im ausgelagerten Zustand „gefangen" bleiben.

Wie am Anfang erwähnt, ist auf Cray-Systemen unter UNICOS kein Demand Paging möglich, und der Speicher tendiert, durch wiederholte Aus-und Einlagerungsaktionen dazu, zu fragmentieren, d.h. freier Speicher ist nur noch in kleinen Stücken verfügbar, von denen keines mehr ausreicht, einen ausgelagerten Job aufzunehmen. Aus diesem Grund werden in regelmäßigen Abständen *repacks* durchgeführt, bei denen alle Prozeßsegmente zusammengeschoben werden und ein einziger, zusammenhängender, freier Bereich entsteht. Die Frequenz, mit der solche Operationen ablaufen, ist von der Häufigkeit der Ein-und Auslagerungsoperationen sowie der

Hauptspeichergröße abhängig; zu niedrige Werte verhindern das Einlagern von Prozessen und verschlechtern damit die Prozessorauslastung, während zu hohe Werte den Systemoverhead in die Höhe treiben.

Wie man aus dieser Beschreibung erkennen kann, läßt sich das Scheduling in weiten Teilen an die jeweilige Betriebsart eines Rechnersystems anpassen und ist wesentlich genauer auf die Architektur der Zielmaschine angepaßt als die in den vorigen Abschnitten beschriebenen „Universalbetriebssysteme", die auf keine spezielle Architektur abzielen. Es ist aber auch anzumerken, daß sich immer noch relativ wenige Automatismen finden, die einzelne Scheduler-Parameter an die jeweilige Lastsituation automatisch anpassen könnten; bei Änderungen des Betriebs sind also manuelle Eingriffe durch den Systembetreuer häufig zwingend.

3.4 Wertung der vorgestellten Verfahren

In diesem Kapitel wurde ein breites Spektrum an Scheduling-Verfahren und Betriebssystemen für parallele Rechnersysteme vorgestellt. Auch wenn diese in vielen Fällen von der Speicherarchitektur eines Rechnersystems unabhängig und damit auch auf SVM-basierte Systeme anwendbar sind, so ist doch festzustellen, daß entweder die mit SVM zwangsläufig entstehenden Randbedingungen nicht berücksichtigt werden oder SVM nur als zusätzliche Betriebssystemkomponente eingeführt wurde. Sobald SVM aber primäres Programmiermodell eines parallelen Rechnersystemes wird, ist eine Kooperation des SVM-Managements mit den restlichen Systemkomponenten für eine effiziente Programmabarbeitung ähnlich wichtig wie die in [Nag93] behandelte Kooperation der Scheduling-Komponenten untereinander. Eine solche Kooperation läßt sich aber, wo es nötig ist, punktuell in ein bestehendes paralleles Betriebssystem einfügen. Aus diesem Grund wird im folgenden Kapitel kein vollständig neues Scheduling-System entwickelt, sondern es werden einzelne Komponenten vorgestellt, die einer verbesserten Wechselwirkung von SVM-Verwaltung und Scheduling-System dienen.

Kapitel 4

Untersuchte Scheduling-Algorithmen

Untersuchungen des Einflusses verschiedener Thread-Scheduling-Algorithmen auf das Laufzeitverhalten SVM-basierter paralleler Programme wurden bereits in [Li94] ausführlich vorgenommen; Ziel von [Li94] war es, Algorithmen zu entwickeln, die eine möglichst hohe Performance eines einzelnen Programmes im dedizierten Betrieb erreichen. Dort war als Randbedingung vorgegeben, daß jeweils nur ein Programm auf dem Parallelrechner aktiv war. In dieser Arbeit steht jedoch eine andere Problemstellung für das Task- und Job-Scheduling im Mittelpunkt. Ein generelles Ergebnis der im vorigen Kapitel angesprochenen Arbeiten ist, daß sich durch Scheduling auf dieser Ebene keine Verbesserung der Performance eines parallelen Programmes erzielen läßt: Ein SVM-basiertes Programm, das ständig Seitenfehler erzeugt, läßt sich weder durch Umordnung der Prozesse noch durch Bevorzugung im Scheduling in ein effizientes, d.h. aus der Sicht des Benutzers mit der vollen Geschwindigkeit aller belegten Prozessoren ablaufendes umwandeln. Umgekehrt ist es aber sehr wohl möglich, ein im dedizierten Betrieb effizient laufendes Programm durch schlechtes Job- und Task-Scheduling im Mehrprogrammbetrieb sehr ineffizient zu machen (z.B. durch fehlendes *Gang-Scheduling*). Der Schwerpunkt beim Task-und Job-Scheduling liegt daher nicht in erster Linie auf der *Erzielung* von möglichst guter Performance, sondern auf deren *Erhaltung*. Einem einzelnen, parallelen Programm sollte eine Ablaufumgebung zur Verfügung gestellt werden, die der eines dedizierten Betriebs möglichst ähnlich ist. Ist dies aufgrund von Ressourcenbeschränkungen nicht möglich, so sollte die Verteilung der vorhandenen Kapazitäten möglichst fair erfolgen, d.h. jedes Programm sollte einen Anteil an der Gesamtrechenleistung erhalten, der seinen Prioritäten entspricht. Aus dem Multiprogramming-Betrieb ergibt sich weiterhin die Forderung, die Programme möglichst weitgehend gegeneinander abzuschirmen. Darunter ist nicht nur der von Betriebssystemen für sequentielle Rechner her bekannte Speicherschutz zu verstehen, sondern auch eine Behandlung von Programmen, die Systemressourcen in einer Weise beanspruchen, daß eine effiziente Abarbeitung anderer Programme

ebenfalls nicht mehr möglich ist. Im Fall von SVM handelt es sich dabei z.B. um Programme, die durch ständige Seitenfehler einen erheblichen Datenverkehr auf dem Verbindungsnetz verursachen und damit auch die Kommunikation anderer Programme signifikant behindern.

In diesem Kapitel sollen die im Rahmen dieser Arbeit entwickelten Verfahren zur Lösung dieser Problemstellungen vorgestellt werden, die in den folgenden Kapiteln mit Hilfe eines Simulationssystems auf ihre Tauglichkeit überprüft werden.

4.1 Ausnutzung der Wartezeiten bei Seitenfehlern

Wie in [BHMW94] ermittelt wurde, ist eine gleichzeitige Kontextumschaltung auf allen Prozessoren immer dann, wenn ein einzelner Prozeß einen Seitenfehler auslöst, nicht effizient, der Overhead für das Gang-Scheduling den erwarteten Durchsatzgewinn übersteigt. Es wurde allerdings die Frage offengelassen, inwieweit sich lokale Kontextumschaltungen rentieren könnten. Bei einer lokalen Kontextumschaltung nimmt ein Prozessor selbstständig und ohne Koordination mit anderen Task-Schedulern eine Kontextumschaltung zu einem rechenbereiten Prozeß vor. Eine solche Strategie verspricht z.B. dann Gewinn, wenn auf den Prozessoren zusätzliche, sequentielle Prozesse vorhanden sind, die keine Koordination erfordern. Auf diese Weise könnten die in parallelen Programmen immer vorhandenen Leerlaufzeiten im Sinne einer Durchsatzsteigerung ausgenutzt werden. Eine Bedingung für einen solchen Betrieb ist allerdings, daß der durch das Multitasking entstehende Overhead nicht so groß wird, daß Vordergrundanwendungen langsamer werden oder im Extremfall die effektiv aus dem Rechnersystem gezogene Rechenzeit sogar niedriger wird. Da im SVM-Programmiermodell Programmunterbrechungen bevorzugt durch Seitenfehler entstehen, deren Auflösung eine längere Zeit benötigt, darf aber erwartet werden, daß die für Hintergrundprogramme abfallende Rechenzeit in hinreichend großen ,,Portionen" zur Verfügung steht, so daß das Auffüllen dieser Zeiten mit Hintergrund-Jobs sinnvoll ist.

4.2 Behandlung von Programmen mit extrem vielen Seitenfehlern

Während die Ausnutzung von Leerlaufzeiten blockierter Prozesse zur Durchsatzsteigerung eines parallelen Systems beitragen kann, gehört die Behandlung von Programmen mit einer hohen Seitenfehlerrate zu den Maßnahmen, die die Stabilität

und Verfügbarkeit verbessern. Die Implementierung einer solchen Behandlung teilt sich in zwei Problemkreise auf: Die *Erkennung* einer solchen Situation und deren *Behebung*.

4.2.1 Erkennungsverfahren

Für die Erkennung bieten sich prinzipiell zwei Ansätze an, nämlich eine *zentrale* oder eine *dezentrale* Erkennung. Bei einer zentralen Erkennung wird der vollständige Systemzustand periodisch von einer systemglobalen Einheit abgefragt, die aufgrund dieser Daten dann entscheiden kann, ob und wo exzessive Seitenfehlerraten auftreten. Der Vorteil einer solchen Lösung ist der globale Überblick im Moment der Entscheidung; nachteilig ist der Zeitaufwand, alle benötigten Daten zu einer zentralen Einheit zu übermitteln. Dieser potentielle Flaschenhals existiert nicht bei einer dezentralen Erkennung, bei der ein Prozessor selbstständig entscheiden würde, daß die Seitenfehlerrate zu hoch ist, und Gegenmaßnahmen einleitet. Da Situationen extrem hoher Seitenfehlerrate wie bei *false sharing* jedoch immer zwischen mehreren, auf verschiedene Prozessoren verteilten Threads auftreten, ist die Gefahr sehr hoch, daß mehrere Prozessoren gleichzeitig Maßnahmen einleiten, die sich gegenseitig behindern oder sogar aufheben können. Als Kompromißlösung würde sich eine Mischform anbieten, bei der die Erkennung lokal erfolgt, die Einleitung von Maßnahmen jedoch zentral koordiniert wird.

4.2.2 Verfahren zur Unterdrückung

Als Maßnahmen zur Unterdrückung der aufgrund von *false* oder *true sharing* entstehenden Seitenfehler bieten sich verschiedene Verfahren an. Diese unterscheiden sich in ihrem Overhead, dem Aufwand, sie rückgängig zu machen sowie der für das Gesamtsystem entstehenden Zusatzarbeit.

4.2.2.1 Migration von Threads

Eine denkbare Variante, Kommunikation zwischen Prozessoren zu reduzieren, ist die Migration von Threads auf andere Prozessoren. Migriert man z.B. alle auf eine Speicherseite zugreifenden Threads auf *einen* Prozessor, so sind überhaupt keine Seitentransfers mehr nötig, da alle nunmehr in Multitasking auf einem Prozessor laufenden Threads direkten Zugang zu dieser Seite besitzen. Die Threads erhalten nun zwar nur noch einen Anteil der Rechenleistung eines Prozessors, was aber durch die nunmehr wegfallenden Blockaden durch Seitenfehler die Ausführungszeit effektiv reduziert. Daß Maß der Kompensation ist abhängig vom Overhead-Verhältnis

von Seitenfehler-Behandlung zum Multitasking und der Zahl der Threads, die auf einem Prozessor ausgeführt werden; bei einer sehr hohen Seitenfehlerrate ist eine erhebliche Beschleunigung möglich. Nachteilig an dieser Variante ist aber, daß nach Beendigung der *false sharing*-Situation wieder eine Rückmigration erfolgen muß, um echte Parallelität zu erhalten; es kann daher günstiger sein, nicht alle Threads auf einen Prozessor zu legen, sondern ihre Verteilung so umzustrukturieren, daß die hohe Kommunikationsrate keine Verbindungen belastet, die auch andere, von dieser Verklemmung nicht betroffene Threads benutzen. Auf diese Weise wird auch die Behinderung anderer Programme behoben, die problembehafteten Threads selbst werden aber nicht beschleunigt. Als Vorteil ist dagegen anzumerken, daß nach Ende der Verklemmungssituation sofort wieder echter Parallelismus eintritt und eine Defragmentierung der Thread-Verteilung generell sinnvoll ist, da nach Beendigung des Programmes eine besser zusammenhängende Partition entsteht.

Diesen Möglichkeiten steht jedoch der in Parallelrechnern mit verteiltem Speicher üblicherweise enorm hohe Aufwand für die Migration von Threads entgegen. *False sharing* tritt üblicherweise nur stellenweise in einem Programm auf, und es ist nicht unwahrscheinlich, daß durch die Verzögerung während der Migration (ein Thread, der gerade migriert wird, kann nicht auf Nachrichten anderer Threads reagieren und blockiert daher auch seine Partner in gewissem Sinne) mehr Zeit verloren wird, als im folgenden gewonnen werden könnte. Auch die Netzbelastung durch die Migration selbst ist nicht zu unterschätzen, da hierbei der komplette Code und private Daten eines Threads transferiert werden. Es macht daher Sinn, nach Verfahren zu suchen, die ohne Transfer großer Datenmengen auskommen.

4.2.2.2 Verriegeln von Seiten

Ein Verfahren, das auch bisweilen direkt von parallelen Hochsprachen unterstützt wird, ist das Verriegeln (*locking*) von Seiten. Bei diesem Verfahren wird eine Seite, nachdem sie zum anfragenden Prozessor transferiert wurde, von weiteren Transfers ausgeschlossen, bis dieser Zustand durch eine entsprechende Entriegelungsanweisung aufgehoben wird. Verriegelungen können neben den auch von Systemen mit physikalisch gemeinsamem Speicher bekannten atomaren Update-Operationen auch zur Reduktion von *false* oder *true sharing* eingesetzt werden. Wenn mehrere Threads eine Reihe von Zugriffen auf eine Speicherseite durchführen müssen, reduziert die Einrahmung dieser Zugriffe mit entsprechenden Lock-Primitiven die Anzahl von Seitentransfers auf ein erträgliches Maß, mit dem Nachteil, daß andere Threads möglicherweise lange auf den Erhalt der Seite warten müssen. Die Abhängigkeit der Entriegelung vom Erreichen eines bestimmten Punktes im Programmablauf birgt aber auch noch weitere Gefahren, falls mehrere Seiten ver- und entriegelt werden: es kann zu *Deadlocks* kommen, wenn gegenseitige Abhängigkeiten zwischen dem

Erhalt einer Seite und dem Entriegeln einer anderen bestehen. Da es nicht akzeptabel wäre, ein funktionsfähiges Programm durch Scheduling-Maßnahmen in ein nicht funktionsfähiges zu überführen, muß sich die betriebssystemgesteuerte Ver- und Entriegelung von Seiten an anderen Marken als dem Ende von Instruktionssequenzen orientieren (das Setzen solcher Marken wäre ohne Zusatzinformationen über die Programmstruktur ohnehin für das Betriebssystem schwer zu lösen). Eine Möglichkeit wäre, die Verriegelung einer Seite durch das Ablaufen eines Timers auszulösen. Die Festlegung der Verriegelungszeit ist aber ein nicht exakt lösbares Problem, da das Betriebssystem keine Kenntnis darüber hat, über was für eine Zeit die Threads noch in Zukunft Zugriffe auf die Seiten ausführen: Ist die Zeit zu kurz, treten nach der Freigabe erneut *Sharing*-Effekte auf, ist sie jedoch zu lang, werden andere Threads unnötig aufgehalten.

4.2.2.3 Anhalten von Threads

Eine dem Verriegeln von Seiten in diesem Zusammenhang ähnliche Methode ist, einen Thread im Fall von *Sharing*-Effekten für einen gewissen Zeitraum anzuhalten. Der Effekt ist identisch zum Verriegeln von Seiten (ein Thread führt keine Zugriffe auf eine bestimmte Seite mehr durch, wodurch andere Threads nicht mehr behindert werden), jedoch wird ein Thread hier nicht gegenüber anderen bevorteilt, sondern gegen die Konkurrenten um die Seite zurückgesetzt. Wichtig ist, daß beim Erkennen einer Konkurrenzsituation das Anhalten soweit koordiniert erfolgt, daß genau ein Thread weiterrechnet; sind es mehrere, so treten weiterhin *Sharing*-Effekte zwischen ihnen auf; werden alle blockiert, so geht Zeit verloren, in der kein Programmfortschritt stattfindet und nach Ablauf der Sperren setzen sich die *Sharing*-Probleme fort.

4.2.3 Untersuchte Strategien

Im Rahmen dieser Arbeit wurden zwei Strategien untersucht, die auf dem zeitweisen Anhalten von Threads basieren: ein Verfahren mit zentraler Steuerung sowie eines mit dezentraler Steuerung:

4.2.3.1 Thread-Unterbrechung mit zentraler Steuerung

Bei der Variante mit zentraler Steuerung versenden Prozessoren eine Nachricht an den globalen Scheduler, wenn sie erkennen, daß ein Thread in eine *Sharing*-Situation geraten ist. Diese läßt sich dadurch identifizieren, daß ein Thread innerhalb einer bestimmten Zeitspanne eine Seite immer wieder anfordern muß, d.h. die lokale Thread-Verwaltung muß die aufgetretenen Seitenfehler protokollieren. Eine in einer solchen

Situation verschickte Nachricht beinhaltet Programm- und Thread-Identifikation sowie die Nummer der betroffenen Seite. Erhält der globale Scheduler diese Nachricht, so durchsucht er eine Liste, ob bereits Nachrichten anderer Threads für die gleiche Seite vorliegen. Sind keine derartigen Einträge vorhanden, so wird der betroffene Thread *nicht* angehalten, ansonsten wird an den anfragenden Prozessor eine Nachricht zurückgeschickt, um den Thread für eine vom globalen Scheduler festgelegte Zeit anzuhalten. Da *false* bzw. *true sharing* dadurch gekennzeichnet sind, daß mehrere Threads gleichzeitig auf eine Seite schreibend zugreifen wollen, ist es bei gleichen ,,Schwellwerten" für die Nachrichten an den globalen Scheduler wahrscheinlich, daß diese Nachrichten zur ungefähr gleichen Zeit auftreten (streng gilt dies nur für eine Konkurrenzsituation mit zwei Threads, bei mehreren Threads ist es möglich, daß weitere Threads später hinzukommen) und die Anhaltezeit weitgehend vom ersten Thread genutzt werden kann. Weiterhin ,,vergißt" der globale Scheduler einen Eintrag in der Liste nach einer Zeit, die der Anhaltezeit entspricht. Auf diese Weise haben nach Ende der Anhaltephase andere Threads eine Chance, ihre Meldungen als erste zum globalen Scheduler durchzubringen und damit effektiv die Seite zu erhalten.

Kritisch ist bei diesem Verfahren die Wahl der Zeitspanne, für die die Threads angehalten werden: bei zu kurzer Wahl treten die Seitenfehler erneut auf und der Zyklus wiederholt sich unnötig häufig, bei zu langer Wahl wird der Programmablauf unnötig verlängert und durch den übergroßen Versatz im Programmfortschritt einzelner Threads kommt es an Synchronisationsstellen zu unnötigen Verzögerungen. Da eine Zeitspanne ideal wäre, die der Zeit entspricht, die der erste Thread benötigt, um alle Zugriffe auf die fragliche Seite durchzuführen, diese aber von den neben den Zugriffen noch auszuführenden Operationen abhängig ist, ist es dem globalen Scheduler nicht möglich, diese Idealzeit von vornherein zu benutzen; man müßte bei einer festen Wahl der Verzögerungszeit einen Kompromißwert finden, der einer möglichst großen Zahl ,,typischer" Fälle gerecht wird. Da parallele Programme aber üblicherweise in Schleifen mit sich wiederholenden Operatioen ablaufen, ist es wahrscheinlich, daß sich *Sharing*-Situationen wiederholen und das System aus vorherigen Durchläufen ,,lernen" kann, ob eine gewählte Verzögerung ausreichend war oder nicht.

Es wurde daher auch eine Variante dieses Algorithmus untersucht, die die Anhaltezeit in Abhängigkeit von der Vorgeschichte festlegen und sozusagen über die Laufzeit eines Programmes seine Charakteristiken ,,lernen" kann. Diese Variante soll im folgenden vorgestellt werden.

Bei festen Anhaltezeiten verwirft der globale Scheduler die Informationen über eine *Sharing*-Situation, nachdem die Anhaltezeit abgelaufen ist. Behält man jedoch diese Daten, so kann man beim Wiederauftritt von *Sharing* auf die gleiche Seite beurteilen, ob die vorherige Maßnahme erfolgreich war oder die Zeit variiert werden

muß. Als Anhaltspunkt kann der zeitliche Abstand der beiden Ereignisse genommen werden: tritt die neue *Sharing*-Situation unmittelbar nach der vorigen auf, so war die Anhaltezeit offenbar zu kurz bemessen und muß im nächsten Anlauf verlängert werden; war der Abstand hingegen deutlich größer als der Zeitraum, über den Seitenfehler nachgehalten werden, so handelt es sich vermutlich um eine neue *Sharing*-Situation (z.B. bei Programmen, bei denen sich Teile wiederholen, um den nächsten Durchlauf) und die Zeit kann beibehalten oder eventuell sogar reduziert werden. Für diese Strategie muß der globale Scheduler pro Seite einen Zeitstempel sowie die letztmalig verwendete Anhaltezeit dauerhaft speichern; ist die Größe der vom Programm verwendeten Datenstrukturen bekannt, so wäre es zusätzlich möglich, für eine Seite gewonnene Erfahrungen auf eine andere in dem gleichen Datenfeld zu übertragen und so den Lernvorgang abzukürzen. Auf jeden Fall erforderlich wäre es für eine praktische Implementation, bei der Festlegung von Anhaltezeiten mitzuspeichern, für welche Teile des Programmes die ermittelten Anhaltezeiten Gültigkeit haben, da sich die Zugriffsfrequenz auf eine bestimmte Datenstruktur in den einzelnen parallelen Strukturen stark unterscheiden kann. Eine dafür erforderliche Aufteilung der Anwendung könnte entweder vom Compiler oder automatisch zur Laufzeit ermittelt werden: Da parallele Konstrukte wie z.B. Schleifen meist durch Synchronisationsprimitive abgeschlossen werden, kann ein Betriebssystem anhand der Adressen im Programmcode, von der solche Aufrufe erfolgen, ein Programm selbständig aufteilen.

Da eine *Sharing*-Situation aber auch zwischen mehr als zwei Threads auftreten kann, muß bei einer Erhöhung der Anhaltezeit berücksichtigt werden, wie viele Prozessoren beim vorigen Auftreten beteiligt waren: waren z.B. beim initialen Auftreten drei Threads beteiligt, so scheidet durch das Anhalten von zwei Threads nur ein Thread aus der Konkurrenzsituation aus und es ist völlig normal, daß die beiden anderen Threads nach Ablauf der Anhaltezeit erneut eine *Sharing*-Situation melden. Dies stellt also kein Indiz für eine zu kurze Anhaltezeit dar. Erst dann, wenn auch der beim ersten Mal nicht angehaltene Thread sich in diesem Zyklus wieder meldet, ist es sinnvoll, die Anhaltezeit zu verlängern, da die Anhaltezeit dann offensichtlich für ihn zu kurz war, um alle Zugriffe durchführen zu können.

Für die Formel, nach der Anhaltezeiten verlängert werden sollen, bieten sich verschiedene Möglichkeiten an. Im Rahmen dieser Arbeit wurde eine lineare Erhöhung untersucht, d.h. ausgehend von einem festen Minimalwert wird die Anhaltezeit in festen Inkrementen verlängert. Denkbar wäre aber z.B. auch eine exponentielle Steigerung, d.h. ausgehend von einem vergleichsweise kleinen Wert wird die Anhaltezeit mit einem festen Faktor multipliziert. Eine solche Formel wäre in der Lage, sich in weniger Schritten an eine optimale Zeit anzunähern, würde bei langen Zeiten diese aber weniger genau einstellen.

4.2.3.2 Thread-Unterbrechung mit dezentraler Steuerung

Für den Fall, daß keine globale Koordination erwünscht ist, böte sich ein Verfahren an, bei dem die Threads für eine zufällige Zeit angehalten werden, so daß einer der Threads früher wieder beginnt und seine Zugriffe abwickeln kann, bevor die Konkurrenten ,,aufwachen". Ein solches Prinzip wird z.B. beim CSMA/CD-Zugriffsverfahren (*C*arrier *S*ense *M*ultiple *A*ccess/*C*ollision *D*etection) auf Ethernet-basierte Netze [HeLa92] angewandt, bei dem Stationen voneinander unkoordiniert auf ein gemeinsames Medium zugreifen und sich bei der Erkennung von Kollisionen für eine zufällige Zeit zurückziehen. Die Berechnung dieser Zeit erfolgt nach dem sog. *Truncated Binary Exponential Backoff*-Verfahren:

$$t_n = i * t_{Slot}, i = rand(2 * 2^k), k = min(n, 10)$$

Hierbei bezeichnet n, wie oft eine Station beim Sendeversuch bereits eine Kollision festgestellt hat. Die angegebene Formel bewirkt, daß bei anhaltenden Kollisionen die Wartezeiten immer länger werden; auf diese Weise bekommen die Wartezeiten der einzelnen Sender eine immer größere Streubreite, wodurch wahrscheinlicher wird, daß die Wartezeiten sich so weit unterscheiden, daß ein Transfer ohne Überlappungen möglich wird. Nach 10 Fehlversuchen steigen die Wartezeiten allerdings nicht weiter an, um die Antwortzeiten nicht allzu schlecht abschätzbar werden zu lassen; nach 16 Fehlversuchen wird ein Sendeversuch abgebrochen, so daß sich eine obere Schranke für die Zeit angeben läßt, nach der der Versand entweder stattgefunden hat oder aufgegeben wurde.

Die Wahl einer zufälligen Wartezeit aus einem bestimmten Zeitbereich ist der Kernpunkt der Kollisionsauflösung; sie stellt allerdings auch den Vorteil von Simulationen, auf denen die Ergebnisse dieser Arbeit beruhen, gegenüber Tests auf realen Rechnern in Frage, daß Läufe beliebig reproduzierbar sind. Auch wenn die Zufallswerte mit Algorithmen ermittelt werden, deren Startwert setzbar ist und die somit für verschiedene Läufe gleiche Folgen von Zufallszahlen produzieren können, so ist nicht von der Hand zu weisen, daß sich bei einer ,,zufällig anderen" Folge von Zahlen ein gänzlich anderer Verlauf der Simulation ergibt. Dieser Effekt ist nur durch mehrfache Ausführung und Mittelung zu berücksichtigen.

Das dezentrale Verfahren stellt eine auf SVM angepaßte Variation des oben beschriebenen Kollisionsbeseitigungsverfahrens von Ethernet dar. Die Erkennung von *Sharing*-Situationen ist identisch zu der Variante mit zentraler Steuerung: Eine solche Situation wird angenommen, wenn innerhalb einer vorgegebenen Zeit für einen Thread eine vorgegebene Zahl von Seitenfehlern für eine bestimmte Seite erreicht wird. Im Gegensatz zur zentralen Steuerung jedoch entscheidet der lokale Scheduler selbstständig, den betroffenen Thread für eine bestimmte Zeit anzuhalten. Diese Zeit ist wie bei Ethernet eine zufällige Zeit, deren möglicher Bereich durch die Anzahl

gegeben ist, wie oft eine solche Situation „hintereinander" erkannt wird, und exponentiell mit der Anzahl der „Fehlversuche" steigt. Die Erkennung, daß es sich um eine wiederholte *Sharing*-Situation handelt, verläuft analog zur adaptiven Variante des zentralisierten Algorithmus: Eine Wiederholung liegt vor, wenn die erneute *Sharing*-Situation in einem Abstand zur vorherigen liegt, der kleiner oder gleich der Anhaltezeit zuzüglich der Zeit ist, die benötigt wird, um eine für eine Erkennung notwendige Historie aufzubauen. Der für die Ermittlung der Zufallszahlen benutzte Generator ist ein nach [Kn81] aufgebauter, linear kongruenter Generator mit Gleichverteilung der erzeugten Zahlen. Der Startwert des Zufallszahlengenerators ist einstellbar, so daß sich Läufe reproduzieren lassen.

Kapitel 5

Beschreibung des entwickelten Simulationssystems

Für den Vergleich von Scheduling-Algorithmen lassen sich die verläßlichsten Ergebnisse selbstverständlich auf einem realen Rechnersystem erzielen. In der Praxis ist dieser Ansatz jedoch mit einer Reihe vom Problemen verbunden:

Zum einen greift die Modifikation von Scheduling-Algorithmen tief in das Betriebssystem eines parallelen Rechners ein. Diese Eingriffe sind daher nur bei einem Rechner möglich, der sich nicht im Produktionsbetrieb befindet. Weiterhin müssen vom Hersteller des Rechners die Eingriffsmöglichkeiten in den Systemkern zur Verfügung gestellt werden. Dies ist meist nicht der Fall, da der Hersteller dafür Teile seines Betriebssystem-Codes in Quellform offenlegen muß.

Ein weiteres Problem ist die Spezialisierung auf eine bestimmte Plattform. Eine Reihe von Parametern, die die Vergleichsergebnisse beeinflussen, sind durch die Hardware festgelegt. Es ist daher nicht möglich, durch weitere Vergleiche mit variierten Parametern den Einfluß dieser Parameter auf die Ergebnisse zu prüfen und damit die Stabilität der vorgeschlagenen Strategien sicherzustellen. Zu den Parametern, die auf einem realen Parallelrechner im allgemeinen nicht variierbar sind, gehören z.B. Seitengröße, Topologie und Leistungsfähigkeit des Verbindungsnetzwerkes sowie die verwendete Variante des SVM-Protokolles.

Problematisch ist zum anderen die Reproduzierbarkeit eines bestimmten Maschinenzustandes: Während auf einer realen Maschine Messungen von einer Reihe äußerer Faktoren (wie z.B. Zusatzlasten durch andere Programme oder unregelmäßig auftretende Eigenaktivitäten des Betriebssystems) beeinflußt werden können, ist ein Systemzustand in einer Simulation, die keine Zufallsfunktionen verwendet, nur von den Eingabeparametern abhängig und daher beliebig reproduzierbar. Diese Reproduzierbarkeit ist enorm wichtig für das Detailverständnis von Effekten, deren Ursachen ähnlich dem Debugging von Programmen nur durch schrittweise Betrachtung mit immer höherer Detaillierung gefunden werden können.

Der entscheidende Grund dafür, daß für diese Arbeit die Wahl auf ein Simulationssystem und nicht auf eine reale Hardware fiel, ist aber insbesondere die Tatsache, daß momentan kein verfügbares paralleles Rechnersystem existiert, welches SVM als primäres Programmiermodell anbietet. Kendall Square Research war bisher die einzige Firma, die mit der KSR-1 und KSR-2 ein solches System anbot. Diese Alternative konnte jedoch nach dem Konkurs des Herstellers nicht mehr genutzt werden. Es deutet sich aber in jüngster Zeit an, daß SVM-basierte Systeme in Zukunft wieder an Boden gewinnen werden; ein Beispiel dafür ist die von Silicon Graphics vorgestellte Origin2000-Baureihe [SGI96], die bis zu 128 Prozessoren mit einer SVM-artigen Speicherarchitektur bietet.

Aus diesen Gründen wurde ein Simulationssystem entwickelt, das eine flexible und erweiterbare Basis für den Vergleich von Scheduling-Verfahren bildet. Als Implementationssprache wurde aufgrund der weiten Verbreitung und ihrer Leistungsfähigkeit C++ [Sto92] gewählt. Die Verwendung des objektorientierten Konzeptes von C++ erlaubte eine klare Gliederung des zu simulierenden Systems in einzelne Komponenten, deren Unabhängigkeit sich bereits in der Datenkapselung der einzelnen Objekte ausdrückt. Diese klare Trennung ist eine Voraussetzung, um den Anspruch einer leichten Austauschbarkeit von Komponenten gerecht zu werden.

5.1 Basisarchitektur

Bei dem Simulator handelt es sich um einen *ereignisgesteuerten* Simulator. Dies bedeutet, daß kein Bezug zwischen der im simulierten Rechnersystem herrschenden Zeit und der bisher vom Simulator verbrauchten realen Rechenzeit besteht. Die Simulation besteht darin, das simulierte Rechnersystem durch eine Folge von Ereignissen zu beschreiben, durch die sich der Zustand des Rechnersystems ändert. Zentrale Datenstruktur einer solchen Simulation ist dabei eine Liste, in die alle in Zukunft auftretenden Ereignisse chronologisch einsortiert werden (Abb. 5.1). Dadurch, daß alle Typen von Ereignissen von einem Basistyp `TEvent` abgeleitet werden, ist es möglich, in der Ereignisliste beliebige Typen zu verwalten, ohne in der Bearbeitung der Liste darauf Rücksicht nehmen zu müssen. Folgende Komponenten sind dabei allen Ereignissen gemeinsam:

- ein Zeitstempel, der den Zeitpunkt des Auftretens markiert;
- eine Methode `Execute`, deren Inhalt vom Ereignistyp abhängt;
- eine Kennzeichung, die das Auffinden dieses Ereignisses in der Liste ermöglicht.

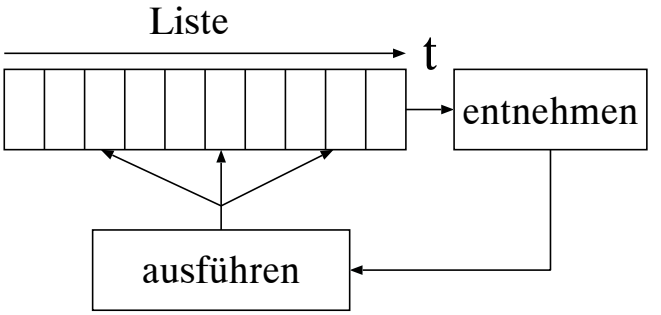


Abbildung 5.1
Ausführungsprinzip des Simulators

Der sich im Simulator wiederholende Zyklus besteht darin, das Ereignis mit dem kleinsten Zeitstempel (durch die Sortierung liegt dieses Element am Kopf der Liste) aus der Liste zu entnehmen, die Simulationszeit auf dessen Zeitstempel zu setzen und die notwendigen Aktionen über die `Execute`-Methode vorzunehmen. Bei der Ausführung eines Ereignisses ist es nun möglich, daß neue Ereignisse entstehen, die wiederum in die Liste einsortiert werden. Auf diese Weise läuft die Liste nicht leer.

Bei diesem Verfahren ist zu berücksichtigen, daß die Erzeugung eines neuen Ereignisses unter Umständen eine Vermutung in die Zukunft darstellt. Da zwischen dem Einsortieren und Ausführen eines Ereignisses beliebig viele andere Ereignisse stattfinden können, kann es nötig werden, dieses Ereignis wieder aus der Liste zu entfernen. Da erzeugendes und löschendes Ereignis aber unabhängig voneinander sind (das erzeugende Ereignis existiert zum Zeitpunkt der Löschung üblicherweise nicht mehr), ist es notwendig, das potentiell zu löschende Ereignis zu kennzeichnen. Über die Kennzeichnung muß dabei eine Übereinkunft zwischen Erzeuger und Entnehmer getroffen werden; die Listenverwaltung wird davon nur insoweit berührt, als daß sie Funktionen zum Löschen bereitstellen muß.

Ein Beispiel für eine Situation, in der Löschen erforderlich ist, ist der blockierende Empfang einer Nachricht durch einen Prozeß. Da der Prozeß bis zum Eintreffen der Nachricht nicht weiterarbeiten kann, ist es wünschenswert, den Prozessor während der Wartezeit für andere Aufgaben nutzen zu können. Andererseits benötigt die dafür erforderliche Kontextumschaltung eine nicht unerhebliche Zeit, weshalb es sinnvoll wird, den Prozessor eine gewisse Zeit aktiv auf die Nachricht warten zu lassen. Das Ereignis, das den Prozessor in den Wartezustand überführt, erzeugt deshalb ein weiteres Ereignis, das nach einer vorgegebenen Zeit einen *Timeout* auslöst und dem wartenden Prozeß den Prozessor entzieht. Trifft nun die erwartete Nachricht vor dem *Timeout* ein, so wird dieses *Timeout*-Ereignis vor seiner Ausführung wieder entfernt. Bleibt die Nachricht dagegen aus, so kommt der Timeout zur Ausführung.

Das vorgestellte Konzept beinhaltet bis jetzt noch keinerlei Abbruchkriterien. Da auf dem simulierten Rechnersystem bestimmte Hintergrundprozesse kontinuierlich und unabhängig von der Auslastung des Gesamtsystems ablaufen, würde die Simulation nie zu einem Ende gelangen. Als Nebenbedingung für einen Abbruch existiert daher eine Abfrage, ob noch abzuarbeitende Programme vorhanden sind. Haben alle vorgegebenen Arbeitslasten das simulierte Rechnersystem wieder verlassen, wird die Simulation daher auf jeden Fall beendet.

Umgekehrt bedarf es natürlich auch einer Initialzündung dieses Kreislaufes, d.h. die Eingabe des ersten Ereignisses kann nicht aus der Schleife selbst heraus erfolgen. Nach dem Einlesen der Konfiguration und vor dem Einstieg in die Schleife wird deshalb pro auszuführendem Job ein Ereignis mit der Eintrittszeit dieses Jobs in die Liste eingespeist. Die Ausführung beginnt demzufolge mit dem Simulationszeitpunkt, an dem das simulierte Rechnersystem seinen ersten Auftrag zu Bearbeitung erhält.

5.2 Struktur der Prozessoren

Die Prozessoren, aus denen das simulierte parallele Rechnersystem aufgebaut ist, bilden den aufwendigsten Teil des Simulators. Der Grund für diese Komplexität ist darin zu suchen, daß für die Untersuchung von SVM-basierten Systemen die Speicherzugriffsmuster der Anwendungsprogramme eine entscheidende Bedeutung haben. Jeder einzelne Speicherzugriff kann potentiell einen Seitenfehler auslösen und den folgenden Ablauf verändern. Ein Prozeß kann daher nicht mehr wie in der Simulation einfacherer Architekturen vereinfacht durch eine Abfolge von Prozessor- und I/O-Nutzung dargestellt werden, sondern muß auf der Ebene einzelner Anweisungen modelliert werden. Dies erschwert die Simulation natürlich erheblich und treibt auch den Rechenaufwand in die Höhe, ist aber für die Ermittlung aussagekräftiger Resultate unerläßlich.

Ein simulierter Prozessor besteht aus einer Reihe voneinander unterscheidbarer Komponenten, die sich in ähnlicher Form als Hardware- oder Software-Komponenten auf jedem Knoten eines realen parallelen Rechnersystems wiederfinden. Abbildung 5.2 zeigt eine Übersicht über diese Komponenten und den Datenfluß zwischen ihnen.

Der Prozessorkern ist dabei die weitaus aufwendigste Komponente. Der Grund dafür ist, daß neben der Implementation des Kerns selbst auch die Erstellung eines „Compilers" notwendig ist, der die in einer hochsprachenähnlichen Programmiersprache geschriebenen Testprogramme in den zur Ausführung benötigten Maschinencode transformiert. Diese Sprache soll im folgenden vorgestellt werden.

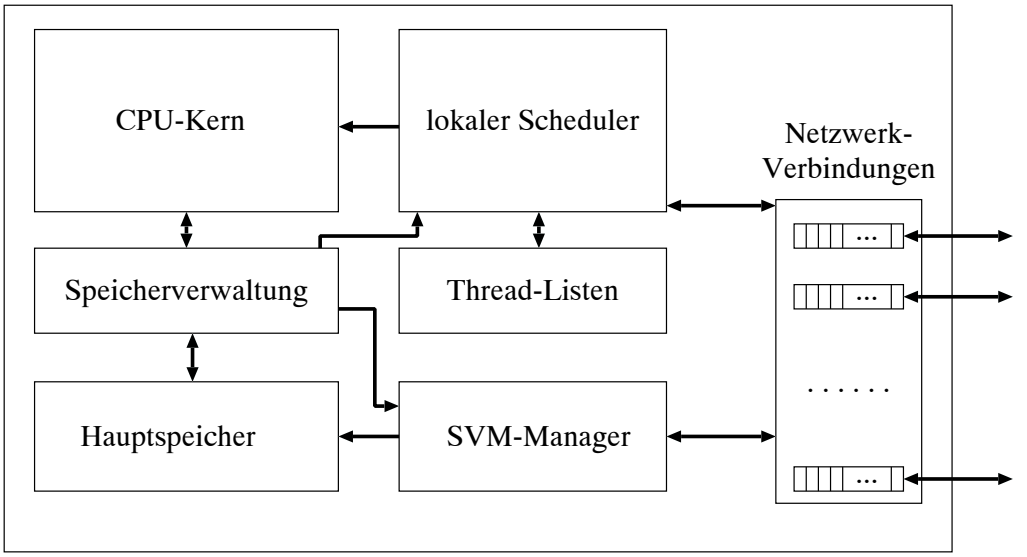


Abbildung 5.2
Blockdiagramm der simulierten Prozessorelemente

5.2.1 Programmiersprache

Bei der Programmiersprache zur Formulierung von Testanwendungen für das simulierte Rechnersystem ist zwischen der Hochsprache, die man als Anwender des Simulators benutzt, und der Sprache, die auf Maschinenebene abläuft, zu unterscheiden.

Da numerische Anwendungen auch heute üblicherweise noch in Fortran programmiert werden, lag es nahe, die Hochsprache in ihrer Syntax an Fortran anzulehnen. Dies hat den Vorteil, daß existierende Anwendungen mit relativ geringem Aufwand zur Untersuchung auf den Simulator portiert werden können. Es wurde hier allerdings nur eine vergleichsweise geringe Untermenge des Fortran-Sprachumfangs implementiert, die für die im Rahmen dieser Arbeit durchgeführten Tests notwendig war. Die Sprache ist aber bei Bedarf leicht um weitere Konstrukte zu erweitern.

Als Ausführungsmodell wurde das sogenannte SPMD-Modell gewählt. SPMD steht dabei für **S**ingle **P**rogram **M**ultiple **D**ata und bedeutet, daß alle parallelen Prozesse eines Programmes (im folgenden als *Threads* bezeichnet) den gleichen Code abarbeiten und Zugriff auf das gleiche Datensegment mit geteilten Daten haben, jedoch zusätzlich jeweils ein eigenes, privates Datensegment besitzen und zu einem Zeitpunkt unterschiedliche Teile des Programmes ausführen können. Wird ein unterschiedlicher Kontrollfluß benötigt (z.B. in Master-Slave-Anwendungen), so kann dies durch Abfrage der eigenen Thread-Nummer erfolgen. Durch die gleiche Technik ist es möglich, eine exklusive Ausführung bestimmter Programmteile durch einen

Prozessor zu erreichen; der Default-Ausführungsmodus ist repliziert, d.h. alle Prozessoren führen das Programm in gleicher Weise aus.

Die Grundstruktur eines Programmes für den Simulator soll hier anhand eines Beispieles erklärt werden. Abbildung 5.3 zeigt als einfache, parallele Berechnung eine Matrixmultiplikation.

Ein Programm besteht prinzipiell aus drei Sektionen, die durch die in eckige Klammern eingeschlossenen Schlüsselwörter `Shared, Private` und `Code` eingeleitet werden. Die beiden ersten Sektionen dienen der Definition der vom Programm genutzten Datenstrukturen. Im Gegensatz zu Fortran müssen alle benutzten Datenstrukturen explizit definiert werden. Unterschieden wird dabei zwischen Variablen, von denen jeder Thread eine private Version besitzt (`Private`), und Variablen, die für alle Threads gleichermaßen zugreifbar sind (`Shared`). Die Kohärenz der `Shared`-Variablen wird dabei vom SVM-Laufzeitsystem einer Anwendung garantiert.

Als Datenstrukturen stehen Felder beliebiger Dimension sowie Skalare zur Verfügung. Während bei Skalaren die Angabe ihres Datentyps (Ganz- oder Gleitkommazahl) ausreichend ist, müssen bei Feldern zusätzlich mindestens Dimension und Größe spezifiziert werden. Die optionale `Alignment`-Direktive erlaubt eine Ausrichtung von Feldkomponenten auf Seitengrenzen. Ohne diesen Parameter (oder mit der Angabe `Alignment 0`) wird eine dichtestmögliche Anordnung im Adreßraum gewählt, d.h. das erste Feldelement liegt hinter dem letzten Element der davor definierten Variablen und alle Elemente des Feldes folgen direkt aufeinander. Eine `1` als Parameter würde dagegen bewirken, daß jedes Feldelement auf einer eigenen Seite abgelegt wird. Dieses Schema setzt sich bis zur Dimension des Feldes fort; so bedeutet z.B. bei den im Beispiel benutzten zweidimensionalen Feldern eine `2`, daß jede Zeile eines Feldes auf einer Seitengrenze beginnen soll. Als Sonderwert für die Ausrichtung ist noch die Dimension des Feldes plus 1 zugelassen; dies bewirkt, daß nur das Feld selber auf eine Seitengrenze ausgerichtet wird, innerhalb des Feldes jedoch wieder eine dichte Packung erfolgt.

Ein Unterschied in der Behandlung von Feldern gegenüber Fortran ist, daß der links stehende Index die Zeile und nicht die Spalte angibt. Diese Spiegelung der Array-Indices erleichterte einerseits die Implementation, stellt andererseits bei der Umsetzung von Fortran-Programmen aber auch keine unüberwindliche Hürde dar, da alle Index-Ausdrücke lediglich gespiegelt werden müssen.

Das hier implementierte Schema des dimensionsweisen Ausrichtens stellt einen Kompromiß zwischen Laufzeiteffizienz und Erfordernissen von SVM-Programmen dar; so werden in [Li94] zwar noch erheblich komplexere Feldstrukturierungen vorgeschlagen, mit denen eine vollständige Anpassung der Datenverteilung an die Arbeitsverteilung möglich ist; jedoch ist zu berücksichtigen, wie kompliziert die dabei entstehenden Indexberechnungsfunktionen sind, die zur Laufzeit für jeden Feldzugriff

```
[Shared]
Array A,B,C
  Contents Real
  Rank 2
  Dimension $1,$1
  Alignment 2
End

[Private]
Scalar I,J,K
  Contents Integer
End
Scalar T
  Contents Real
End

[Code]

InitialThreads $2

PDO(BLOCK) I=1,$1
  DO K=1,$1
    A(I,K)=I+K
    B(I,K)=I+K
  ENDDO
ENDDO

PDO($3) I=1,$1
   DO J=1,$1
     T=0.0
     DO K=1,$1
       T=T+A(I,K)*B(K,J)
     ENDDO
     C(I,J)=T
   ENDDO
ENDDO
```

Abbildung 5.3
Beispielprogramm für die Matrixmultiplikation

ausgewertet werden müssen. Speziell für Verteilungen, die zyklisch oder mehrdimensional verteilen, lassen sich Divisionen durch Zahlen, die keine Zweierpotenz sind, nicht vermeiden. Die Integer-Division ist jedoch auf modernen RISC-Prozessoren, wie sie in massiv-parallelen Rechnern eingesetzt werden, eine der zeitaufwendigsten Operationen. Der Vorteil, einige Seitenfehler einzusparen, kann also sehr schnell durch einen höheren Rechenaufwand verloren gehen. Im Gegensatz dazu wird für das hier benutzte Schema der Ausrichtung auf Seitengrenzen lediglich eine Modulodivision durch die Seitengröße verwandt, und da die Seitengröße in der Praxis immer eine Zweierpotenz ist, läßt sich diese Operation zur Laufzeit sehr effizient durch eine UND-Maskierung realisieren. In einer Pascal-artigen Hochsprache läßt sich die Adreßberechnung wie folgt formulieren:

```
Acc=<start address of array>;
Multiplier=<size of element>;
FOR z=1 TO <rank of array> DO
  IF z=AlignmentDim
    Rest=Multiplier AND (PageSize-1);
    IF Rest<>0 THEN Multiplier:=Multiplier+Rest;
  ENDIF
  Acc=Acc+<z-th index>*Multiplier;
  Multiplier=Multiplier*<z+1-th dimension>;
ENDDO
Speicheradresse=Acc
```

Neben den unvermeidlichen Multiplikationen sind lediglich Additionen und logische Verknüpfungen erforderlich, die auf Maschinenebene mit wenig Rechenaufwand realisierbar sind.

Die Menge der implementierten Programm-Anweisungen ist gering, jedoch so gewählt, daß sich damit alle in dieser Arbeit untersuchten numerischen Algorithmen nachbilden lassen. Die erste Anweisung eines Programmes lautet üblicherweise `InitialThreads`. Mit diesem Befehl wird dem Programmlader mitgeteilt, wieviele parallele Threads dieser Job direkt nach dem Start aufbauen soll. Man kann diese Threads als die „virtuellen Prozessoren" betrachten, die ein paralleles Programm ausführen sollen. Im Idealfall wird auch jeder dieser Threads auf einem separaten physikalischen Prozessor ausgeführt, so daß eine optimale Beschleunigung erreicht werden kann; der globale Scheduler kann jedoch auch entscheiden, daß ein Prozessor mehrere Threads im Multitasking bearbeiten soll. Da das Verhältnis von logischen zu physikalischen Prozessoren durch Migration geändert werden kann, ist es dem System möglich, die Anzahl der von einer Anwendung physikalisch belegten Prozessoren der Lastsituation im System anzupassen, ohne daß eine Anwendung auf den

Verlust oder die Hinzunahme von Prozessoren reagieren müßte. Diese Abstraktion erlaubt auch die Bearbeitung von Anwendungen, deren logische Prozessorzahl höher ist als die Zahl der physikalisch zur Verfügung stehenden Prozessoren.

Das erste Schleifennest des Beispielprogrammes wird sequentiell vom ersten Thread ausgeführt. Diese Schleife dient nur zur Vorbelegung der beiden Quellfelder und ist nötig, da der Simulator keine I/O-Operationen unterstützt, mit denen man die Felder aus einer externen Datei laden könnte. Da der Ausführungsmodus repliziert ist (alle Prozessoren durchlaufen den gleichen Code parallel), muß die Sequentialisierung durch ein `IF-ENDIF`-Konstrukt und die Abfrage der eigenen Thread-Nummer durchgeführt werden; diese Nummer steht in der vordefinierten Variablen `MYPROC` zur Verfügung. Die `IF`-Abfrage führt dazu, daß alle anderen Prozessoren außer dem ersten dieses Schleifennest vollständig überspringen (es wäre möglich, mit einem `ELSE`-Zweig andere Operationen parallel auszuführen) und direkt zur Ausführung des `GSYNC`-Befehls kommen. Dieser Befehl realisiert eine Barriere, d.h. die Prozessoren können den hinter `GSYNC` liegenden Code erst ausführen, wenn alle Prozessoren den davor liegenden Code fertig bearbeitet haben. In diesem Fall bewirkt `GSYNC`, daß alle Threads auf die Initialisierung des Feldes durch Thread 1 warten. Dies ist wichtig, damit nicht andere Threads in der folgenden Schleife auf uninitialisierte Elemente zugreifen.

Die zweite Schleife realisiert die Matrixmultiplikation, bei der über die Zeilen des Ergebnisfeldes parallelisiert wird (mehrdimensionaler Parallelismus ist nicht realisiert). Eine parallele Schleife unterscheidet sich von einer sequentiellen Schleife lediglich durch das andere Schlüsselwort sowie die in Klammern angegebene Verteilungsstrategie, die darüber entscheidet, wie die Schleifeniterationen auf die einzelnen Threads verteilt werden sollen. Es stehen eine Reihe von Strategien zur Verfügung, die sich in statische und dynamische Verteilungsstrategien unterteilen lassen:

Statische Verteilungen sind dadurch gekennzeichnet, daß die Menge der auszuführenden Iterationen von jedem Thread lokal aus der Anzahl der Threads und der Gesamtzahl der Iterationen berechnet wird; sie werden als statisch bezeichnet, weil diese Menge a priori berechnet werden kann und sich während der Ausführung der Schleife nicht mehr ändert. Im einzelnen werden die folgenden statischen Strategien angeboten:

`BLOCK`: Der gesamte Iterationsraum wird in $n_{Threads}$ gleiche Stücke unterteilt, von denen jeweils eines von einem Thread ausgeführt wird.

`CYCLIC`: Aufeinanderfolgende Iterationen werden von aufeinanderfolgenden Threads ausgeführt.

`BLOCKCYCLIC`: Hier wird ähnlich wie bei `CYCLIC` verfahren, jedoch kann die Größe der zyklisch verteilten Pakete vom Benutzer festgelegt werden.

Ein Beispiel für die sich ergebende Verteilung von 16 Iterationen auf 4 Threads zeigt Abbildung 5.4.

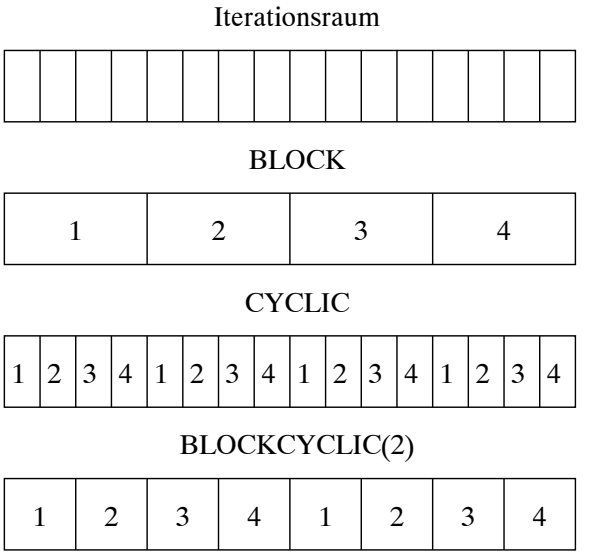


Abbildung 5.4
Statische Schleifenverteilung

Im Gegensatz zu statischen Verteilungsstrategien erlauben dynamische Varianten, die Lastaufteilung erst während der Schleifenabarbeitung festzulegen. Dazu existiert eine globale Instanz (hier bei Thread 1 eines parallelen Programmes), die alle noch zu bearbeitenden Schleifeniterationen speichert. Jeder Thread muß sich nun von dieser Instanz ein Iterationspaket abholen, bevor er mit der Schleifenbearbeitung beginnen kann. Ist dieses Paket abgearbeitet, so muß sich der Thread ein neues Paket holen. Dieser Zyklus setzt sich fort, bis ein Thread keine Iterationen mehr erhalten kann.

Dynamische Verfahren haben den Vorteil, sich besser an Situationen anzupassen, in denen in die Bearbeitungszeit in nicht vorhersehbarer Weise von Iteration zu Iteration schwankt. Solche Schwankungen können einerseits durch einen unterschiedlichen Rechenaufwand an sich bedingt sein, andererseits aber auch ihre Ursache in einer unterschiedlichen Zahl von Seitenfehlern pro Iteration haben. Dynamische Verfahren

haben weiterhin Vorteile, wenn in einer Multitasking-Umgebung damit gerechnet werden muß, daß einzelne Threads zwischenzeitlich den Prozessor verlieren. In beiden Fällen verhindert ein Aushändigen von Arbeit nach Bedarf, daß ein Thread a priori zuviel Rechenarbeit erhält, was dazu führen würde, daß alle anderen Threads auf ihn warten müßten. Nachteilig ist der Overhead durch die zusätzliche Kommunikation sowie in dieser Implementation die zentrale Vergabestelle für Iterationen, die sich als Flaschenhals erweisen kann.

Die vom Simulator angebotenen dynamischen Verteilungsstrategien unterscheiden sich in der Größe der Pakete, die bei jeder Anfrage ausgehändigt werden:

- `SINGLE`:
  Bei jeder Aushändigung wird nur eine Iteration vergeben.
- `CHUNKS(n)`:
  Bei jeder Aushändigung werden `n` Iterationen vergeben.
- `GUIDED(b)`:
  Bei jeder Aushändigung werden $i/n_{Threads}$ Iterationen vergeben, wobei i die Anzahl der Restiterationen ist.
- `FACTORING(b)`:
  In einem ersten Schritt werden $n_{Threads}$-mal $i/(2 * n_{Threads})$ Iterationen vergeben, danach $n_{Threads}$-mal $i/(4 * n_{Threads})$ Iterationen usf.

Die beiden letzten Algorithmen versuchen, durch die Vergabe immer kleinerer Pakete einen möglichst geringen Overhead mit optimaler Lastbalancierung zu erreichen; detailliertere Beschreibungen dieser Algorithmen können in [PoKu87] bzw. [HSF91] nachgelesen werden. Der Parameter `b` ermöglicht ein sogenanntes *Loop-Blocking*, mit dem festgelegt werden kann, daß ein ausgehändigtes Paket immer ein mehrfaches von `b` Iterationen beinhalten soll. Dies ist eine in [Li94] beschriebene Erweiterung, die genutzt werden kann, um das Aushändigen sehr kleiner Pakete und damit verbundenes *false sharing* am Ende der Schleife zu vermeiden.

Neben den im Beispiel gezeigten Befehlen steht weiterhin ein `PRINT`-Statement zur Verfügung, das z.B. zum Debugging von Programmen genutzt werden kann, sowie ein `STOP`-Befehl zum vorzeitigen Programmabbruch. Wie das Beispiel weiterhin zeigt, besteht die Möglichkeit, Programme zu parametrieren. Dazu können in der Lastbeschreibung (siehe Abschnitt 5.7) bei der Spezifikation eines Programmes Parameter angegeben werden, die an den mit `$1,$2,...$n` gekennzeichneten Stellen textuell eingesetzt werden. Dies ermöglicht es, Programme mit verschiedenen Problemgrößen, Schleifenalgorithmen oder Prozessorzahlen zu testen, ohne dafür das Programm umschreiben zu müssen.

5.2.2 Abarbeitung auf Maschinenebene

Die im vorigen Abschnitt beschriebene „Hochsprache" bedarf einiger Transformationen, um zur Laufzeit auf einem Simulator ausgeführt zu werden. Ein wichtiger Grund ist eine bessere Laufzeiteffizienz des Simulators selbst, da bei einem einmaligen Parsing die nachfolgende, mehrfache Abarbeitung der einzelnen Befehle deutlich einfacher und schneller erfolgen kann. Ein Grund, der eine „Vorkompilation" sogar zwingend notwendig macht, ist die Zerlegung von komplizierten Formel- und Indizierungsausdrücken in einfache Lade-, Speicher- und Rechenbefehle: Potentiell kann bei jedem Speicherzugriff ein Speicherfehler auftreten, der dazu führt, daß die momentan bearbeitete Instruktion abgebrochen und dem betroffenen Thread der Prozessor entzogen wird. Um bei Verfügbarkeit des Datums den Thread wieder korrekt anlaufen lassen zu können, ist es erforderlich, nach dem Abbruch einer Instruktion den Thread in den Zustand zurückzuversetzen, den er *vor* der Ausführung des Speicherzugriffes hatte. Dies wird mit steigender Komplexität einer Anweisung immer schwieriger und kann sogar unmöglich werden, wenn ein Befehl mehrere Schreibzugriffe beinhaltet. Aus diesem Grund werden Programme beim Ladevorgang in den Simulator in eine Folge einfacherer Anweisungen transformiert. Weitere Aufgaben des Programmladers sind die Linearisierung geschachtelter Konstrukte zum Programmfluß sowie die Zuordnung von Variablen auf bestimmte Adressen im geteilten bzw. privaten Datensegment der Threads. Zusammenfassend übernimmt der Programmlader in diesem Simulator die Aufgaben eines Compilers in einem realen System. Die Transformation des Programm-Codes ist natürlich abhängig von der Architektur des unterliegenden „Prozessors". Hier fiel aus zwei Gründen die Wahl auf eine Stack-orientierte Maschine:

- Formelausdrücke lassen sich sehr einfach in eine UPN-Notation übertragen, die dann direkt von einer Stack-orientierten Maschine abgearbeitet werden kann. Ebenso gestaltet sich die Übersetzung geschachtelter Konstrukte recht einfach.
- Obwohl in der Realität keine Prozessoren mit einem beliebig tiefen Stack existieren, ist das Verhalten eines Stack-orientierten Prozessors bzgl. der Speicherzugriffe identisch zu einem registerbasierten Prozessor. Im Zusammenhang mit SVM interessieren aber nur diese, ein Prozessor wird eigentlich nur gebraucht, um diese Zugriffe im passenden Zeitraster zu erzeugen.

Abbildung 5.5 zeigt die transformierte Form des Beispielprogrammes. Die Befehlsgruppen, die durch einen einzelnen Hochsprachen-Befehl erzeugt wurden, sind dabei durch dickere Linien voneinander getrennt. Pfeile deuten die möglichen Sprungwege an, auf denen der normale, sequentielle Kontrollfluß verlassen werden kann. Die dafür notwendigen Sprungadressen werden beim Transformationsvorgang erzeugt.

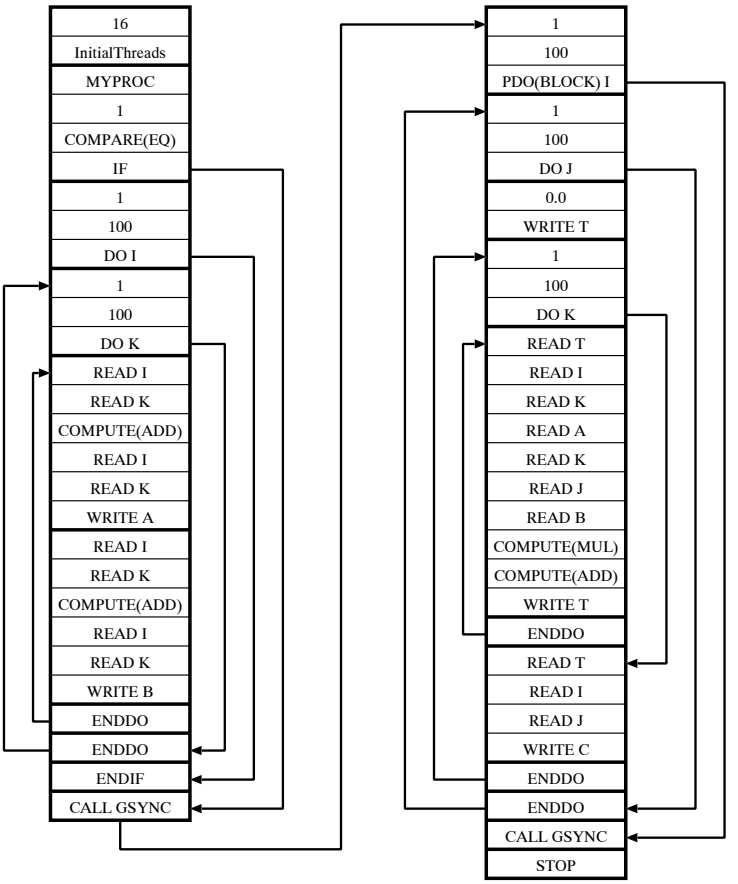


Abbildung 5.5
Transformiertes Beispielprogramm für die Matrixmultiplikation

Wie man sehen kann, werden nur die notwendigsten Transformationen vorgenommen; Konstrukte, die noch weiter in einfachere Befehle zerlegt werden könnten (wie z.B. Schleifenbefehle), bleiben unberührt. Besonders zu bemerken ist die automatische Ausführung einer globalen Synchronisation am Ende der parallelen Schleife. Dies garantiert, daß bei aufeinanderfolgenden, voneinander datenabhängigen Anweisungsblöcken alle Prozessoren mit korrekten Daten arbeiten.

5.2.3 Lokale Prozeßverwaltung

Da Multitasking auf den Prozessoren zugelassen ist, ist eine Instanz erforderlich, die alle auf einem Prozessor momentan laufenden Threads verwaltet und ein lokales Scheduling vornimmt. Zur korrekten Verwaltung ist es erforderlich, jedem Thread einen der vier Zustände zuzuordnen, zwischen denen er hin- und herwechseln kann. Die vier Zustände mit den möglichen Übergängen sind in Abbildung 5.6 dargestellt.

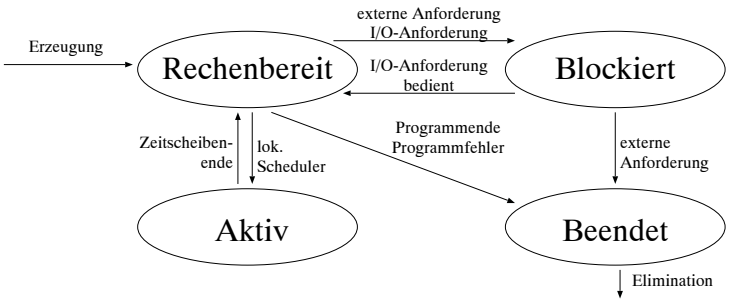


Abbildung 5.6
mögliche Zustände eines Threads

Initial ist ein Thread im Zustand „Rechenbereit", d.h. er ist bereit zur Ausführung auf dem Prozessor. Ein Scheduling findet nur zwischen rechenbereiten Threads statt, und der lokale Scheduler wählt nach einem einstellbaren Algorithmus einen Thread zur Ausführung aus und überführt ihn in den Zustand „Aktiv". Da ein Thread aber üblicherweise nicht nur lokal rechnet, sondern auch I/O-Operationen durchführt, während deren Bearbeitung der Thread nicht weiterrechnen kann, kann ein Thread in den Zustand „Blockiert" überführt werden, in dem er bis zur Fertigstellung der I/O-Operation verbleibt. Unter I/O-Operationen sind hier alle Formen der Kommunikation zu anderen Prozessoren zu verstehen, wie z.B. SVM-Seitentransfers oder Nachrichtenverkehr für dynamisches Schleifen-Scheduling. Wichtig ist die Feststellung, daß die Überführung zwischen den Zuständen „Rechenbereit" und „Blockiert" niemals selbstständig vom Scheduler vorgenommen wird, sondern von anderen Systemkomponenten ausgelöst wird, wenn diese erkennen, daß eine I/O-Operation nicht sofort bedient werden kann. Der lokale Scheduler kann lediglich Überführungen zwischen den Zuständen „Rechenbereit" und „Aktiv" durchführen; soll der momentan aktive Thread in den Zustand „Blockiert" überführt werden, so erfolgt dies über den Zwischenzustand „Rechenbereit".

Während die ersten drei Zustände auf praktisch allen Multitasking-Systemen vorhanden sind, ist der vierte Zustand „Beendet" ein Resultat der speziellen Bedürfnisse der SVM-Umgebung: Ein Thread hat neben seinen eigentlichen Rechenaufgaben auch Teile des SVM-Seitenmanagements zu übernehmen. Da die einzelnen Threads eines parallelen Programmes grundsätzlich asynchron laufen, ist es möglich, daß nach dem Ende eines Threads noch Anfragen für von ihm verwaltete Seiten eintreffen. Aus diesem Grund wird ein beendeter Thread nicht sofort vollständig entfernt, es wird lediglich von ihm belegter Speicher für private Daten und Code freigegeben und er wird auch nicht mehr im Scheduling berücksichtigt. Erst, wenn alle Threads eines Programmes diesen Zustand erreicht haben, werden diese vollständig von den Prozessoren entfernt.

Für jeden Thread werden darüber hinaus noch eine Reihe statistischer Daten angelegt, die der Steuerung des Schedulings dienen. Dazu gehören z.B. die variable Priorität eines Threads sowie eine Protokollierung der bisher vom Thread verbrauchten Rechenzeit. Diese Daten werden benötigt, um den lokalen Scheduler mit Informationen zu versorgen. Die lokalen Scheduling-Strategien entsprechen weitgehend denen, die auch auf sequentiellen Maschinen Verwendung finden:

- First Come, First Served
- Round-Robin
- reine Prioritätensteuerung
- Round-Robin mit Prioritätensteuerung
- minimale absolute CPU-Zeit
- minimale CPU-Zeit in der letzten Vergangenheit

Die letzte Strategie basiert auf einem CPU-Zeit-Zähler, der bei jedem Scheduling-Vorgang um einen bestimmten Faktor reduziert wird. Auf diese Weise „vergißt" das System frühere Phasen hoher CPU-Ausnutzung.

5.2.4 Speicherverwaltung

Alle Speicherzugriffe von Programmen durchlaufen eine Speicherverwaltungseinheit (MMU). Diese Einheit hat folgende Aufgaben zu erfüllen:

- Überprüfung der Speicherzugriffe auf Korrektheit;
- Übersetzung logischer in physikalische Adressen;
- Auslösung von Seitentransfers.

Unter den ersten Punkt fällt eine Überwachung, ob Speicherzugriffe innerhalb der Segmentgrenzen eines Threads liegen. Überschreitet ein Zugriff diese Grenzen, wird ein Signal ausgelöst und der fragliche Thread beendet. Da die Verwaltung des physikalischen Speichers aber nur in mehrfachen der Seitengröße erfolgt, können Segmentüberschreitungen nicht mit letzter Sicherheit erkannt werden. Dieser Mechanismus schützt ein Programm natürlich auch nicht davor, sich seine eigenen Daten zu überschreiben; er ist als Schutz für andere Threads gedacht. Die Übersetzung logischer in physikalische Adressen ist in Abbildung 5.7 schematisch dargestellt.

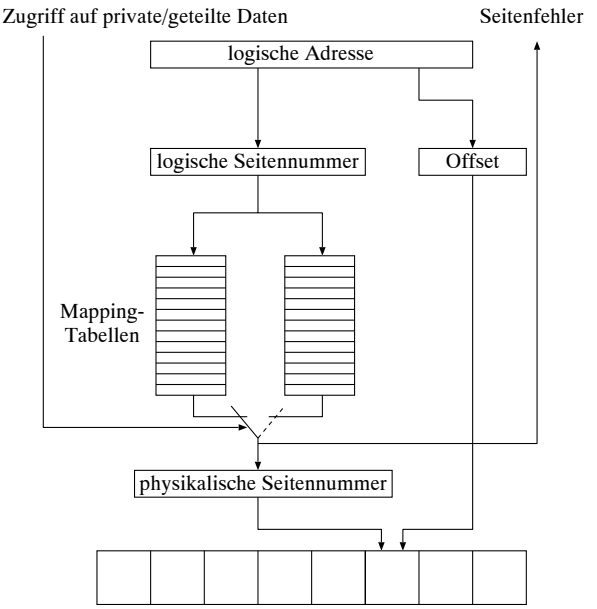


Abbildung 5.7
Adreßübersetzung in der MMU

Zunächst berechnet die MMU aus der logischen Adresse eine logische Seitennummer sowie den Offset in dieser Seite. Anhand zweier Tabellen (jeweils eine für Zugriffe in das private bzw. in das geteilte Datensegment) kann dann eine physikalische Seitennummer ermittelt werden. Ist dieser Eintrag leer, so ist die Speicherseite nicht vorhanden und muß geholt werden.

Da Demand Paging zwar vorgesehen, aber im Rahmen der in dieser Arbeit untersuchten Programme und Algorithmen nicht benötigt wurde, kann der Fall einer nicht vorhandenen Seite nur für Seiten im geteilten Segment auftreten. Zusätzlich kann es hier aber auch noch sein, daß eine Seite zwar vorhanden ist, aber kein Schreibrecht für diese Seite existiert. In beiden Fällen löst die MMU zwei Aktionen aus:

- Der Thread, der den Speicherzugriff ausgelöst hatte, wird in den Zustand „Blockiert" versetzt.
- An den SVM-Manager des zugehörigen Threads wird eine Aufforderung abgesetzt, die fragliche Seite zu beschaffen bzw. das Recht dieses Threads auf Lesen/Schreiben zu erhöhen.

Wie im vorigen Abschnitt schon erwähnt wurde, wird der betroffene Thread in den Zustand zurückversetzt, den er vor der Ausführung des Speicherzugriffes hatte. Trifft nun die Seite beim SVM-Manager ein, so reicht er diese an die MMU weiter. Anhand

einer Liste noch ausstehender Seitentransfers kann diese dann entscheiden, welcher Thread wieder in den Zustand „Rechenbereit" zu versetzen ist. Dieser kann dann wieder am normalen Scheduling teilnehmen und wiederholt den Speicherzugriff, sobald er die CPU erhält.

Umgekehrt können auch die SVM-Manager mit der MMU kommunizieren. Dies ist erforderlich, wenn von anderen Prozessoren Anfragen für Seiten im geteilten Datensegment ankommen. Da die SVM-Manager nur logische Seitennummern kennen und die Verwaltung des physikalischen Speichers allein der MMU obliegt, müssen diese die Seite von der MMU holen.

5.2.5 Steuerung des Nachrichtenverkehrs

Obwohl das Programmiermodell für Anwendungsprogramme SVM ist und auch auf Anwendungsebene kein explizites Message-Passing angeboten wird, werden naturgemäß auf unterer Ebene alle Kommunikationsvorgänge durch den Austausch von Nachrichten über das Verbindungsnetzwerk realisiert. Dabei handelt es sich einmal um Nachrichten, die die „Betriebssysteme" der Knoten untereinander austauschen und die direkte Reaktionen auf den Knoten auslösen. Zu dieser Klasse gehören z.B. Nachrichten, mit denen der globale Scheduler die Programmausführung auf den Rechenknoten beeinflussen kann, andererseits aber auch Statusrückmeldungen der Rechenknoten.

Nachrichten, die von Anwendungsprogrammen ausgelöst werden, bedürfen einer zusätzlichen Verwaltung. Zwar könnten die Anwendungen selber nur über den virtuell gemeinsamen Speicher kommunizieren, es gibt aber bestimmte anwendungsnahe Systemdienste, deren Realisierung mittels Message-Passing wesentlich natürlicher und effizienter ist als mit Hilfe des SVM-Programmiermodells. Zu diesen Diensten zählen die globale Synchronisation aller Threads eines Programmes sowie die Beschaffung von Iterationspaketen beim dynamischen Schleifen-Scheduling. Da ein Thread nicht unbedingt zu dem Zeitpunkt aktiv sein muß, zu dem eine Nachricht für ihn eintrifft, ist es notwendig, Warteschlangen für Nachrichten einzurichten. Des weiteren kann es sein, daß ein Thread eine Nachricht empfangen will, diese aber noch nicht verfügbar ist. Für die Handhabung solcher Wartezustände und die Verwaltung noch nicht abgeholter Nachrichten existiert daher auf den Prozessoren eine eigene Systemkomponente. Auf der Sendeseite werden alle Nachrichten mit einem zusätzlichen Kopf versehen, der Programmnummer sowie Quell- und Ziel-Thread beinhaltet. Jeder Prozessor besitzt weiterhin Tabellen, in denen die Zuordnung von Threads zu physikalischen Prozessoren für alle Programme verzeichnet ist, an deren Ausführung er beteiligt ist. Ohne diese Information hätte der Nachrichten-Manager keine Informationen, wohin er eine Nachricht schicken soll, da das Netzwerk selbst nur mit physikalischen Prozessornummern arbeitet.

Der Mechanismus auf der Empfangsseite ist mehrstufig: Will ein Thread eine Nachricht empfangen, so schaut der Nachrichten-Manager zuerst in seinen Listen nach, ob bereits eine Nachricht verfügbar ist, auf die die vom Thread bereitgestellte Maske paßt. Diese Maske besteht aus dem numerischen Typ einer Nachricht sowie der Nummer des Quell-Threads. Beide Parameter sind optional, so daß es auch möglich ist, Nachrichten beliebigen Typs oder von beliebigen Threads eines Programmes zu empfangen. Findet der Manager eine passende Nachricht, so wird diese dem anfragenden Thread übergeben und die Programmausführung kann ohne Unterbrechung fortgesetzt werden. Ist dagegen keine Nachricht vorhanden, so hat dies zweierlei Aktionen zur Folge:

- Der Manager trägt den wartenden Thread mit der Maske in eine besondere Liste ein.
- Es wird ein Timer gestartet, der dem Thread nach einer gewissen Zeit den Prozessor entzieht.

Da ein Thread während des Wartens auf eine Nachricht nicht fortgeführt werden kann (die Systemfunktionen, die explizites Message-Passing nutzen, benötigen keinen asynchronen Empfang), ist es sinnvoll, ihm während der Wartezeit den Prozessor zu entziehen, damit diese Zeit von anderen Threads genutzt werden kann. Andererseits benötigt die doppelte Kontextumschaltung aber auch eine nicht zu vernachlässigende Zeit, so daß eine Umschaltung in Situationen, in denen die Nachricht zwar noch nicht da ist, aber in nächster Zukunft eintreffen wird, den Rechenzeitgewinn in einen Verlust umkehren würde. Deshalb erlaubt der Timer dem Thread, eine bestimmte Zeit aktiv auf die Nachricht zu warten. Der Wert der Zeitspanne ist einstellbar und von den restlichen Systemparametern abhängig. Er sollte entsprechend den vorher gemachten Ausführungen größer sein als eine doppelte Kontext-Umschaltzeit.

Trifft eine Nachricht ein und findet der Manager einen wartenden Thread in seiner Liste, so ist die weitere Reaktion davon abhängig, ob der zugehörige Timer bereits abgelaufen war. Falls nicht, wird die Nachricht an den noch aktiven Thread übergeben, der daraufhin normal weiterläuft, und der Timer wird eliminiert. War der Timer hingegen schon abgelaufen, so muß der Thread wieder in den Zustand „Rechenbereit" zurückversetzt werden und der Manager sortiert die Nachricht in seine Listen ein, als ob gar keine Empfangsnachfrage stattgefunden hätte. Sobald der unterbrochene Thread nämlich wieder zur Ausführung gelangt, wird er den abgebrochenen Befehl wiederholen und damit dem Manager wieder nach einer Nachricht fragen. Diese Nachfrage kann der Manager jetzt direkt beantworten.

5.3 SVM-Management

Der im Simulator implementierte SVM-Algorithmus verwendet das im zweiten Kapitel beschriebene „klassische" SVM-Protokoll mit getrenntem Manager und Eigentümer für eine Seite. Da die SVM-Verwaltung aber als Objekt mit klar definierten Schnittstellen zu MMU und Prozessorkern realisiert ist, ist ein Austausch gegen eine andere Verwaltung jederzeit möglich.

Zu jeden Thread existiert eine Instanz eines SVM-Managers. Diese ist untrennbar mit dem Thread verbunden und wird erst gelöscht, wenn der Thread endet. Nach außen unterhält der SVM-Manager zwei bidirektionale Kommunikationskanäle:

- eine Schnittstelle zum Message-Passing-System, um über Nachrichten mit den SVM-Managern anderer Threads Daten austauschen zu können;
- eine Schnittstelle zur MMU, um mit ihr Seiten austauschen zu können.

Der Nachrichtenverkehr läuft dabei nicht über den im letzten Abschnitt besprochenen Nachrichten-Manager, da Anfragen von außen völlig asynchron zum Ablauf des zugehörigen Threads auftreten, der Nachrichten-Manager jedoch diese Nachrichten nur auf Anfrage weiterreichen würde. Außerdem könnte immer nur der Thread Seitenanfragen bedienen, der momentan auf dem Prozessor ausgeführt wird. Der „SVM-Thread" ist deshalb immer ausführbar, d.h. Anfragen werden sofort und ohne Verzögerung unabhängig vom CPU-Zustand bearbeitet. Dieses Modell entspricht zwar nicht ganz der Realität, vereinfacht aber die Simulation, da sonst der laufende Vordergrundprozeß für die SVM-Bearbeitung unterbrochen werden müßte. Man kann sich außerden vorstellen, daß auf einem für SVM konzipierten Rechnersystem die Problematik der asynchronen Unterbrechungen berücksichtigt wird und das SVM-Protokoll von einer dedizierten Hardware abgewickelt wird.

Da ein SVM-Manager genauso wie der Rechen-Thread nur logische Thread-Nummern, aber keine physikalischen Prozessornummern kennt, werden zum Verschicken die selben Tabellen mit der Thread↔Prozessor-Zuordnung wie vom Nachrichten-Manager genutzt. Im einzelnen verschickt ein SVM-Manager folgende Typen von Nachrichten:

- `GETREAD GETWRITE UPDRIGHT`
  Diese drei Nachrichten beginnen den Zyklus der Behandlung eines Seitenfehlers. Mit ihnen erbittet ein Thread beim Manager einer Seite, die Seite zum Lesen bzw. Schreiben zur Verfügung zu stellen bzw. das Leserecht auf die Seite um das Schreibrecht zu erweitern. Als Argumente enthalten sie die Seitennummer sowie die Nummer des anfragenden Threads.

- `SENDPAGE`
  Mit dieser Nachricht fordert der Manager den Owner einer Seite auf, eine Kopie dieser Seite an den anfragenden Thread zu schicken. Neben der Seitennummer und der Nummer des Ziel-Threads beinhaltet die Nachricht auch die neuen Rechte des Owners an dieser Seite; eine Reduktion der Rechte oder das vollständige Invalidieren können also implizit erfolgen, ohne dafür eine separate Nachricht verschicken zu müssen.

- `CHANGERIGHT`
  Diese Nachricht wird vom Manager versandt, um die Schreibrechte eines Threads auf eine Seite auf Lesen zu reduzieren oder eine Seite komplett zu invalidieren. Der Versand dieser Nachricht ist z.B. notwendig, wenn andere Lesekopien invalidiert werden müssen, bevor ein Knoten das Schreibrecht erhalten kann.

- `CHANGEDRIGHT`
  Mit dieser Nachricht antwortet der Manager auf eine `UPDRIGHT`-Anfrage, sobald alle eventuellen Lesekopien der Seite invalidiert wurden.

- `GOTPAGE`
  Mit dieser Nachricht liefert der (bisherige) Owner einer Seite die Seite an den nachfragenden Thread ab. Der SVM-Manager gibt die Seite an seine MMU weiter, die die Daten in eine freie physikalische Seite kopiert und den unterbrochenen Thread wieder entsperrt.

- `GOTINITPAGE`
  Diese Nachricht ist eine effizientere Variante von `GOTPAGE` und wird vom Manager genutzt, wenn zum ersten Mal auf eine Seite zugegriffen wird, die uninitialisierte Daten enthält. Da der Inhalt einer solchen Seite vor dem ersten Zugriff beliebig ist, wird zu Beginn für solche Seiten auch kein physikalischer Speicher reserviert, dies erfolgt beim ersten Zugriff. Da die initialen Daten einer solchen Seite beliebig sind, kann der Manager mit einer solchen Antwort dem Anfrager mitteilen, daß er die Seite selbst erzeugen kann. Auf diese Weise erspart man sich die zeitintensive Übertragung einer Seite zum anfragenden Prozessor.

- `ACKRCV`
  Mit dieser Nachricht bestätigt der anfragende Thread beim Manager den Erhalt einer Seitenanfrage. Da der Thread diese Nachricht abschickt und sofort mit der Programmabarbeitung fortfährt, behindert diese zusätzliche Nachricht ihn nicht. Sie dient vielmehr dazu, dem Manager den Abschluß des Seitentransfers anzuzeigen. In der momentanen Implementierung ist es so, daß der Manager nicht mehrere Nachfragen pro Seite gleichzeitig bearbeiten kann. Dieser

Nachteil macht sich nur bemerkbar, wenn sehr viele Prozessoren gleichzeitig um eine Seite bewerben, also in einer Situation, in der ohnehin keine gute Performance zu erreichen wäre.

Löst ein Thread einen Seitenfehler aus, so wird diesem normalerweise der Prozessor entzogen und der lokale Scheduler angestoßen, um eine sinnvolle Nutzung der Wartezeit zu ermöglichen; zu Vergleichszwecken ist jedoch auch *Spin-Waiting* einschaltbar, bei dem der Thread aktiv auf die Seite wartet. Dieses Verfahren erlaubt zwar kürzestmögliche Reaktionszeiten auf das Eintreffen der Seite, geht aber zu Lasten der Effizienz.

Das SVM-Management unterstützt die Anwendung von *Page-Locks*, d.h. ein Rechen-Thread kann seinen SVM-Manager anweisen, bestimmte Seiten, die sich in seinem Besitz befinden, solange nicht abzugeben, bis er eine dazu passende *Unlock*-Anweisung erhalten hat. Diese Eigenschaft wird von der MMU direkt nach Erhalt einer Seite genutzt, um die Seite auf dem Prozessor zu halten, bis der Thread, der den Seitenfehler ausgelöst hat, mindestens einen Zugriff auf diese Seite durchführen konnte. Ohne diese Verriegelung kann es vorkommen, daß ein Prozeß eine Seite verliert, bevor der Scheduler auf diesen Prozeß umschalten konnte. Dieses *Page-Trashing* kann im Extremfall dazu führen, daß ein Programm mit vielen Seitenfehlern *überhaupt nicht* terminiert. Eine Seite wird allerdings nur verriegelt, wenn der Thread, der diese Seite angefordert hatte, durch die Verfügbarkeit der Seite wieder rechenbereit wird. Ansonsten könnte es passieren, daß eine Seite für einen Thread unnötigerweise festgehalten wird, der zwischenzeitlich auch noch aus anderen Gründen suspendiert wurde (z.B. weil erkannt wurde, daß er an einer *Sharing*-Situation beteiligt war und er deshalb für einen Zeitraum „schlafen gelegt" wurde).

Beim Laden eines Programms wird anhand der Größe des gemeinsamen Segmentes sowie der initialen Zahl von Threads eine Verteilung der Management-Aufgaben vorgenommen; dies ist eine zyklische Verteilung. Eine zyklische Verteilung hat bei Programmen, die über gewisse Zeiträume nur mit einem Array arbeiten, den Vorteil, daß im Gegensatz zu einer Blockverteilung kein Prozessor übermäßig mit Management-Aufgaben belastet wird. Über zusätzliche Nachrichten wäre jedoch auch eine Migration von Management-Aufgaben während der Laufzeit möglich.

5.4 Verbindungsnetzwerk

Das Verbindungsnetzwerk zwischen den Prozessoren ist als eine Anordnung von Punkt-zu-Punkt-Verbindungen ausgelegt, die wahlweise uni- oder bidirektional sein können. Da jede Verbindung einzeln spezifiziert wird, lassen sich auf diese Weise

beliebige Strukturen wie z.B. Gitter, Hypercubes oder Ringe simulieren. Die Herausnahme einzelner Verbindungen würde es auch erlauben, Störungen im Netzwerk nachzuempfinden.

Da auf einer physikalischen Leitung zu einem Zeitpunkt nur eine Nachricht übertragen werden kann, existiert für jede Verbindung auf der Senderseite eine Warteschlange, die mit einer FIFO-Strategie abgearbeitet wird. Andere Strategien wie *Shortest Job First*, die im Mittel kürzere Antwortzeiten ergeben, wären zwar denkbar, haben aber für andere Systemkomponenten den Nachteil, daß beim Verschicken mehrerer Nachrichten zwischen zwei Prozessoren die Nachrichten nicht mehr notwendigerweise in der Sendereihenfolge ankommen, was den Aufbau dieser Komponenten deutlich komplizierter gestalten würde. Um Blockaden von einzelnen Leitungen durch lange Nachrichten zu verhindern, ist es aber möglich, eine maximale Nachrichtenlänge anzugeben. Nachrichten, die länger sind, werden vom Router dann in eine Folge kürzerer Pakete zerlegt.

Wenn zwischen zwei Prozessoren keine direkte Verbindung besteht, so muß die Nachricht über Zwischenstationen geroutet werden. Die Festlegung der Routen für jedes Sender↔Empfänger-Paar erfolgt beim Start des Simulators nach dem Einlesen der Verbindungsliste. Die dabei ermittelte Tabelle ändert sich während des Programmlaufes nicht, es findet also kein dynamisches Routing statt. Obwohl dies die Latenzzeiten in Einzelfällen verbessern könnte, hat es den Nachteil, daß die Anlieferung von Nachrichten in der Sendereihenfolge dann auch nicht mehr garantiert werden könnte.

Bei der Übertragung mit Zwischenstationen kann zwischen zwei Protokollen gewählt werden, die auch in der Realität Anwendung finden (Abbildung 5.8).

Beim *Store-and-Forward* wird das Nachrichtenpaket von einer Zwischenstation empfangen, danach wird die Leitung freigegeben und das Paket in die Sendewarteschlange der Verbindung zur nächsten Station einsortiert. Der Vorteil dieses Verfahrens ist, daß die Nachricht zu einem Zeitpunkt immer nur eine Verbindung im System belegt. Nachteilig ist jedoch, daß Speicher zur Zwischenspeicherung benötigt wird und eine Nachricht gegenüber dem nachfolgend beschriebenen *Wormhole-Routing* länger zur Übertragung braucht. Der Nachteil der längeren Übertragungszeit läßt sich aber abmildern, indem man, wie im Bild gezeigt, die Nachricht in kleinere Pakete aufspaltet und diese nacheinander abschickt. Auf diese Weise ist es möglich, daß in einer Pipeline-artigen Verarbeitung eine schnellere Übertragung erreicht wird. Diese Technik findet naturgemäß ihre Grenze, sobald die Übertragungszeit einer Nachricht von der Startup-Zeit dominiert wird, und sie bedeutet eine stärkere Belastung des Netzwerkes.

Noch bessere Durchsätze lassen sich mit dem *Wormhole-Routing* [NiKi93] erzielen. Bei diesem Verfahren wird in einem ersten Schritt der gesamte Pfad vom

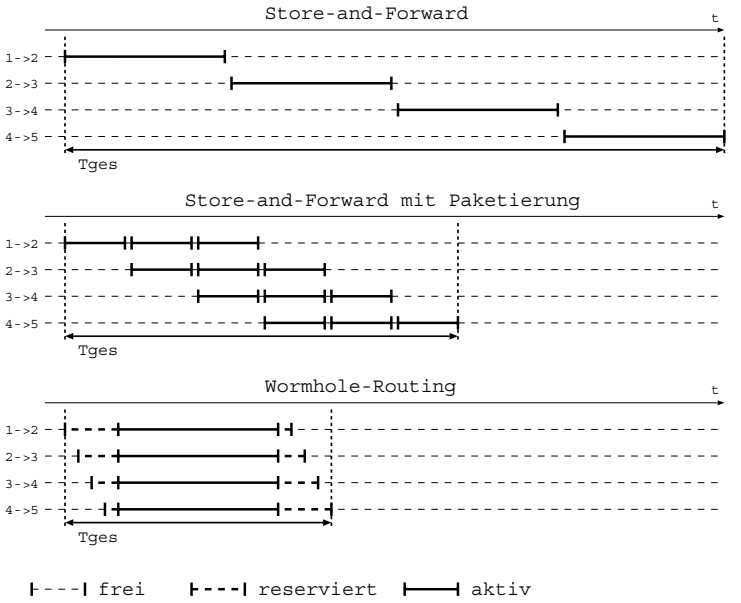


Abbildung 5.8
Routing-Prinzipien

Sender zum Empfänger reserviert. Die einzelnen Nachrichtenstrecken werden dann zu einer einzigen zusammengeschaltet, über die die Nachricht in einem Schritt übertragen werden kann. Die Hardware des Netzwerkes muß das Zusammenschalten aber unterstützen, und die Übertragungsstrecke bleibt während der gesamten Übertragungszeit für andere Nachrichten unzugänglich.

5.5 Globales Scheduling

Der globale Scheduler ist eine eigenständige Instanz, der die globale Überwachung der Programmausführung auf dem parallelen System obliegt. Programme, die zur Ausführung gelangen sollen, müssen den globalen Scheduler durchlaufen. Dieser erzeugt die parallelen Threads und ermittelt nach einer vorgebbaren Strategie die Knoten, auf die das Programm initial verteilt wird. Sind nicht genügend aufnahmebereite Knoten vorhanden, so kann der globale Scheduler aber auch die Annahme verweigern oder das Programm in eine Warteschlange einreihen.

Sofern das Programm aus mehr als einem Thread besteht, verschickt der globale Scheduler die Threads mit der Anweisung, sie initial in den Zustand „Blockiert" zu versetzen. Auf diese Weise wird verhindert, daß ein loslaufender Thread versucht, mit anderen Threads Kontakt aufzunehmen, die möglicherweise noch gar nicht geladen sind. Für die Rechenknoten wäre es dann sehr schwierig zu entscheiden, ob es sich

bei der empfangenen Nachricht um eine Nachricht handelt, die für eine spätere Behandlung zu speichern ist, oder um einen schlichten Fehlläufer, der verworfen werden kann. Sobald der letzte Thread beim Empfänger angekommen ist, schickt der globale Scheduler ein Signal an alle betroffenen Rechenknoten, ihre Threads in den Zustand „Rechenbereit" zu überführen.

Die Rechenknoten benachrichtigen umgekehrt den globalen Scheduler, sobald ein Thread beendet wurde. War dies ein normales Programmende (durch einen `STOP`-Befehl), so dekrementiert der globale Scheduler in seiner Tabelle der laufenden Programme den Thread-Zähler. Erreicht der Zähler den Wert Null, so ist das parallele Programm beendet, und der globale Scheduler verschickt eine Nachricht, die zugehörigen Threads zu löschen. Wurde der Thread jedoch durch einen Fehler beendet (Segmentüberschreitung, Stack-Unterlauf, Division durch Null oder Zeitüberschreitung), so führt dies zur sofortigen Beendigung aller Threads und einer entsprechenden Meldung auf der Konsole.

Im Rahmen einer Systemüberwachung ist es auch Aufgabe des globalen Schedulers, einen globalen Lastausgleich zwischen den Rechenknoten herzustellen, d.h. die Arbeitslast möglichst gleichmäßig auf alle Prozessoren zu verteilen. Dies versucht er zum einen durch eine möglichst „gute" initiale Verteilung zu erreichen, zum anderen durch die Migration von Threads während der Laufzeit. Die Informationen über den Lastzustand des Systems erhält er durch regelmäßige Nachrichten der Rechenknoten. Da diese Nachrichten von den Rechenknoten asynchron verschickt werden (es wird nicht auf den Empfang gewartet), und ein Versand auch nur erfolgt, wenn der lokale Scheduler ohnehin aktiv ist, bedeuten diese Nachrichten keine große Belastung für das System. Sie sind auch recht klein, da sie nur summarische Angaben enthalten:

- Anzahl von Threads in den Zuständen „Rechenbereit", „Blockiert" sowie „Beendet";
- prozentuale Auslastung des Prozessors über den Zeitraum seit der letzten Meldung;
- belegter und freier physikalischer Speicher in Seiten.

Diese Angaben ermöglichen es dem globalen Scheduler zum einen, beim Laden eines Programmes einen Satz möglichst wenig belasteter Prozessoren zu finden, zum anderen kann er bei deutlichen Lastungleichgewichten die Migration einzelner Threads initiieren.

5.6 Migration

Unter der *Migration* eines Threads ist die vollständige Verschiebung eines Threads von einem Rechenknoten zu einem anderen zu verstehen. Diese Operation wird vom globalen Scheduler angestoßen, um einerseits eine bessere Lastbalance im Gesamtsystem zu erreichen, andererseits aber auch die Verteilung der Threads über das Prozessornetz besser an ihre Kommunikationsstruktur anzupassen. Da die Migration eines möglicherweise laufenden Threads eine verhältnismäßig aufwendige Operation ist, soll ihr Ablauf hier in einem eigenen Abschnitt beschrieben werden.

Eine Migration wird mit einer speziellen Nachricht an den Prozessor, auf dem der fraglichen Thread bisher plaziert war, begonnen. Diese Nachricht wird üblicherweise vom globalen Scheduler verschickt, könnte bei dezentralen Lastbalancierungsalgorithmen aber auch von anderen Rechenprozessoren stammen. Als erste Reaktion prüft der Prozessor, ob der Thread momentan ausgeführt wird oder nicht. Falls nicht, kann die Migration sofort begonnen werden, ansonsten wird ein Flag gesetzt, welches die Ausführung nach Ende der laufenden Maschineninstruktion unterbricht und den lokalen Scheduler des Prozessors aufruft. Dies ist einerseits sinnvoll, da die Migration eines Threads, der sich mitten in der Ausführung einer Maschineninstruktion befindet, wegen der unter Umständen laufenden Aktivitäten in anderen Baugruppen des Prozessors deutlich schwieriger durchzuführen wäre, andererseits ist die dadurch entstehende Verzögerung im Vergleich zur gesamten Migrationszeit vernachlässigbar: Die größte Verzögerung tritt auf, wenn der Thread gerade auf den Empfang einer Nachricht wartet und diese nicht innerhalb der Timeout-Zeit eintrifft. Der Aufruf der lokalen Schedulers ist ohnehin nötig, da ein neuer Thread zur Ausführung auszuwählen ist.

Zur eigentlichen Migration wird der Thread inklusive aller benötigten Statusinformationen in eine Nachricht verpackt, da diese einzelne, große Nachricht im allgemeinen mit weniger Overhead übertragen werden kann. Der Datensatz der Nachricht beinhaltet folgende Daten:

- Das private Datensegment des Threads;
- eine Kopie des Code-Segmentes des Threads;
- die Seiten des geteilten Segmentes, von denen der Thread eine Kopie besitzt;
- der momentane Status des Threads („Rechenbereit" oder „Aktiv", „Blockiert", „Beendet");
- der SVM-Manager des Threads mit seinen Daten;

- der Registersatz des Threads, d.h. bei dem hier implementierten Prozessormodell der Operanden-Stack;
- der aktuelle Stand des Programmzählers.

Auf dem empfangenden Prozessor werden umgekehrt die Komponenten des Threads in die entsprechenden Datenstrukturen wieder einsortiert; ist der Thread im Zustand „Rechenbereit", so wird gleichzeitig der lokale Scheduler angestoßen, da es sein kann, daß dieser neue Thread eine höhere Scheduling-Priorität als der momentan ausgeführte hat.

Neben der eigentlichen Migration ist sicherzustellen, daß alle Prozessoren, die an der Ausführung des parallelen Programmes beteiligt sind, ihre Thread-Mapping-Tabellen korrigieren. Um dies sicherzustellen, verschickt der Sender neben dem zu migrierenden Thread eine Mapping-Tabelle, die alle Partner-Threads des zu migrierenden Threads beinhaltet. Dies ist notwendig, da es sein kann, daß der Zielprozessor bisher überhaupt keine Threads des betroffenen Programmes ausgeführt hat; in diesem Fall besitzt er auch keine Mapping-Informationen über Threads dieses Programmes. In einem zweiten Schritt korrigiert der Sendeprozessor in seiner Tabelle den Eintrag für den migrierten Thread; sofern er nach dem Absenden noch Nachrichten für diesen Thread erhält, kann er sie mit dieser Information weiterleiten. Sobald der Empfangsprozessor den Thread erhalten hat, verschickt er an den globalen Scheduler und mit Ausnahme des Senders alle Prozessoren, die Threads dieses Programmes bearbeiten, eine kurze Nachricht, mit der die Wanderung des Threads bekanntgemacht wird. Diese Operation darf erst nach dem Empfang erfolgen, da es sonst passieren könnte, daß andere Prozessoren dem Empfangsprozessor Nachrichten für einen Thread schicken, den er weder besitzt noch kennt. Die FIFO-Strategie des Netzes garantiert, daß vom Sender weitergeleitete Nachrichten die Nachricht mit dem zu migrierenden Thread nicht überholen können.

5.7 Lastspezifikation

Mit Hilfe der Lastspezifikation kann dem Simulator vorgegeben werden, welche Programme zu welchem Zeitpunkt auf dem simulierten Parallelrechner zur Ausführung gelangen sollen. Die Spezifikation erfolgt in einer Sektion der Simulator-Konfigurationsdatei, wobei pro Programm eine Zeile anzugeben ist. Das Beispielprogramm aus Abbildung 5.3 könnte z.B. mit der folgenden Zeile initiiert werden:

```
Load=  1,     0,   0.000, , 5.0, , mxm.prg 100 16 CHUNKS(5)
```

Die erste Spalte hinter dem Schlüsselwort `Load` ist eine frei vergebbare ganze Zahl, die zur Identifikation des Jobs genutzt wird. Die einzige Bedingung für diese Zahl ist, daß eine Zahl nicht mehrfach für verschiedene Jobs benutzt werden darf. An zweiter Stelle steht die Default-Priorität der Threads des Jobs, die von den lokalen Schedulern bei prioritätsgesteuerten Algorithmen genutzt wird.

Die nächsten beiden Spalten bezeichnen den Zeitpunkt, zu dem der Job dem globalen Scheduler zur Ausführung angeboten wird. Wahlweise kann der Zeitpunkt absolut oder relativ zum Ende eines anderen Jobs angegeben werden; zur relativen Angabe muß in die zweite Spalte die Nummer des Jobs geschrieben werden, nach dessen Ende mit der Ausführung begonnen wird, ansonsten bleibt dieses Feld leer. Die relative Angabe erlaubt z.B. die Simulation interaktiver Nutzung, bei der ein Anwender sequentiell ein Programm nach dem anderen ausführt; man kann mit diesem Mechanismus aber auch Batch-Scheduler simulieren, die es nicht zulassen, daß ein Anwender zu einem Zeitpunkt mehrere Batch-Jobs ausführt.

Mit den optionalen Zeitangaben in den folgenden beiden Spalten ist es möglich, maximale Laufzeiten für das Programm anzugeben. Der erste Wert bezieht sich dabei auf die „Wallclock"-Zeit, d.h. die Zeit vom Abschicken des Jobs bis zum Ende des letzten Threads inklusive aller Wartezeiten. Eine solche Begrenzung ist z.B. sinnvoll, um „Endlosläufer", d.h. Programme, die z.B. aufgrund extremen *false sharings* nicht in sinnvoller Zeit ablaufen würden, nach endlicher Zeit abzubrechen. Die zweite Zeitangabe bezieht sich hingegen auf die von den einzelnen Threads verbrauchte CPU-Zeit. Sobald einer der Threads des Jobs diese Zeit verbraucht hat, wird er beendet, und es erfolgt eine Meldung an den globalen Scheduler, der daraufhin den vollständigen Job abbricht.

Als letzter Eintrag steht in der Zeile ein Verweis auf die Datei, die den Quelltext des Programmes beinhaltet. Optional kann dieser Angabe eine beliebige Zahl von durch Leerzeichen getrennten Parametern folgen, die dann beim Laden des Programmes an den mit `$n` gekennzeichneten Stellen eingesetzt werden. In diesem Beispiel bedeuten die Parameter, daß zwei Matrizen der Größe 100 auf 16 Prozessoren multipliziert werden sollen; als Scheduling-Strategie sollen Pakete von 5 Iterationen dynamisch verteilt werden.

5.8 Simulator-Konfiguration

Alle in der Simulation variierbaren Parameter werden dem Simulator in einer ASCII-Datei mitgeteilt, die in einzelne Sektionen eingeteilt ist. Als Sektionen sind dabei definiert:

General	Anzahl der Prozessoren und Zykluszeit, mit der Zeiten aufgelöst werden sollen;
Network	Geschwindigkeit der Punkt-zu-Punkt-Verbindungen, Topologie sowie Routing-Strategie;
Memory	Seitengröße sowie Bandbreite der lokalen Speicher;
SVM	Sperren oder Freigabe der Optimierung für initiale Seitenerzeugung, Scheduler-Reaktion bei Seitenfehlern;
CPU	Ausführungszeiten für einzelne Maschineninstruktionen sowie Zeitbedarf für Kontextumschaltungen;
Threads	
GlobalScheduler	physikalische Position im Prozessornetz sowie Strategie;
LocalScheduler	Zeitscheibengröße, Strategie sowie Timeout-Zeit für das Warten auf Nachrichten;
Load	Spezifikation der auszuführenden Programme.

Der eigentliche Quelltext der auszuführenden Programme befindet sich in getrennten Dateien, auf die in der [**Load**]-Sektion verwiesen wird; damit ist es möglich, ein Programm mehrfach auszuführen, ohne dies mehrfach in die Konfigurationsdatei aufnehmen zu müssen.

Weiterhin beinhaltet jeder Abschnitt Schalter, mit denen festgelegt werden kann, ob die entsprechenden Simulatorkomponenten Ereignisse in die Trace-Datei (s. folgender Abschnitt) schreiben sollen; auf diese Weise ist es möglich, die Größe der Trace-Datei auf den interessierenden Teil zu beschränken.

5.9 Ergebnisauswertung

Als Ergebnis eines Simulationslaufes wird von Simulator eine Trace-Datei geschrieben, die die Ereignisse während der Simulation mit Zeitstempeln protokolliert. Als zu protokollierende Ereignisse kommen dabei Zustandsänderungen aller Systemkomponenten in Fragen, wie z.B. Netzwerk, Prozessor und Seitenverteilung in den Programmen. Um den Umfang der Trace-Datei begrenzen zu können, können in der Konfigurationsdatei einzelne, besonders umfangreiche Protokollanteile ausgeschaltet werden. Werden z.B. alle optionalen Teile ausgeschaltet, so ergibt sich für das Beispielprogramm aus Abbildung 5.3 eine Trace-Datei von ca. einem Megabyte. Die optionalen Teile können diesen Wert jedoch drastisch vergrößern:

- Ausgabe der Routing-Schritte einer Nachricht: ca. 1,2 MByte;
- Ausgabe aller lokaler Speicherzugriffe mit Startadresse und Länge: ca. 600 MByte;
- Protokollierung aller ausgeführten Maschineninstruktionen: ca. 380 MByte;
- Protokollierung der SVM-Seitenverteilung: ca. 150 KByte.

Von diesen Erweiterungen wird lediglich letztere vom Analyse-Tool *PARvis* ausgewertet. *PARvis* ist eine X-basierte Visualisierungsumgebung, die die Trace-Datei einlesen und in eine Reihe graphischer Darstellungen transformieren kann. Diese Darstellungen ermöglichen eine wesentlich einfachere Beurteilung der Simulationsläufe als die Trace-Datei selber oder aus ihr erzeugte, summarische Statistiken. Die Darstellungsmöglichkeiten von PARvis umfassen die Visualisierung von Prozessorzuständen, Nachrichtenaustausch bzw. Netzwerkaktivität sowie Seitentransfers des SVM-Systems. In den folgenden Kapiteln werden wiederholt Visualisierungen von Simulationsläufen gezeigt werden, die mit PARvis erstellt wurden. Die Funktionalität von PARvis an dieser Stelle zu beschreiben, würde den Rahmen dieses Abschnittes bei weitem sprengen; diese kann in [Arn93], [Mu94] bzw. [ABNR95] nachgelesen werden.

Kapitel 6

Ergebnisse der Untersuchungen

6.1 Benutzte Maschinenkonfiguration

Wie im vorangegangenen Kapitel gezeigt wurde, kann das im Rahmen dieser Arbeit implementierte Simulationssystem in sehr weiten Bereichen konfiguriert werden. Bei den folgenden Simulationen wurde allerdings nicht für jeden Lauf ein komplett neuer Parametersatz definiert, sondern es wurden, ausgehend von einem „Standard-Satz", nur die für die jeweilige Fragestellung relevanten Größen variiert. Dies ist einerseits sinnvoll, um ein möglichst hohes Maß an Vergleichbarkeit der Ergebnisse zu erhalten, und andererseits auch nötig, um die Anzahl zu testender Kombinationen zu begrenzen. Der Standard-Parametersatz beinhaltet folgende Einstellungen und ist bezüglich Netzwerk und Speicherbandbreite an eine Intel Paragon angelehnt:

- Netzwerk:
  - 16 Prozessoren, angeordnet als 2-dimensionales Gitter mit bidirektionalen Verbindungen;
  - asymptotische Datenrate 90 MByte/s, 30 μs Startup-Zeit;
  - max. Nachrichten-Paketgröße 8 kByte (wird in der hier untersuchten Konfiguration nicht überschritten);
  - Wormhole-Routing mit statischem Routing, d.h. der Pfad für eine Sender/Empfänger-Kombination ist fest.
- Speicher:
  - Seitengröße: 4 KByte;
  - lokale Speicherbandbreite: 400 MByte/s;
  - *lazy allocation* für shared pages, d.h. physikalischer Speicher wird erst beim ersten Ansprechen einer Seite belegt;

  - lokale Verriegelung von angeforderten SVM-Seiten für einen Zugriff durch den anfragenden Thread;

- Prozessoren:
  - Additions-/Multiplikations-Befehl: 20 ns;
  - Divisionsbefehl: 2 μs;
  - Lade-/Speicherbefehl (ohne Zugriffszeit auf den Speicher selbst): 30 ns;
  - Schleifen-Overhead: 300 μs.

- Scheduling-System:
  - kein Spin-Waiting bei Seitenfehlern;
  - Rückschaltung in den gleichen Thread: 10 μs;
  - Kontext-Umschaltung in einen anderen Thread des gleichen Jobs: 20 μs;
  - Kontext-Umschaltung in einen Thread eines anderen Jobs: 60 μs;
  - Zeitscheibenlänge: 20 ms;
  - lokale Scheduling-Strategie: Round-Robin zwischen den höchstpriorisierten Threads;
  - globaler Scheduler auf Knoten 16 (d.h. in einer Ecke des Netzes).

Die Zeit zur Auflösung eines Seitenfehlers beträgt bei Wahl dieser Parameter ca. 150 μs, sofern keine anderen Nachrichten in dieser Zeit das Netz blockieren.

6.2 Verwendete Testprogramme

Für die in dieser Arbeit durchgeführten Untersuchungen wurden zwei verschiedene Programmkerne verwendet. Bei dem einen handelt es sich um die in Abbildung 5.3 bereits gezeigte Matrixmultiplikation mit blockweiser oder zyklischer Verteilung, bei dem anderen um eine LU-Zerlegung.

6.2.1 Matrixmultiplikation

Die in Abbildung 5.3 gezeigte Matrixmultiplikation wird in den folgenden Untersuchungen mit zwei verschiedenen Verteilungen benutzt: zum einen mit blockweiser, zum anderen mit zyklischer Verteilung. Bei einer blockweisen Verteilung liegen die

von einem Prozessor erzeugten Zeilen im Zielfeld direkt nebeneinander, d.h. mögliches *false sharing* ist nur auf Seiten möglich, auf denen sowohl das Ende des Datenbereiches eines Prozessors als auch der Anfang des Datenbereiches des nächsten Prozessors liegen; da aber alle Prozessoren zur gleichen Zeit mit der Bearbeitung beginnen, ist es wahrscheinlich, daß Prozessor $n+1$ die Bearbeitung seiner Daten auf der fraglichen Seite bereits beendet hat, sobald Prozessor n diese Seite anfordert.

Läßt man diesen Effekt außer acht, so kann man abschätzen, daß die Matrixmultiplikation effizient läuft, wenn das Ergebnisfeld mindestens so viele Speicherseiten im Hauptspeicher belegt wie Prozessoren eingesetzt werden. Bei einer Seitengröße von 4 KByte, 16 Prozessoren und einer Feldelementgröße von 8 Byte muß das Feld also mindestens $16 * 512 = 8192$ Elemente besitzen, was einer Matrix von mindestens $91 * 91$ Elementen entspricht. Allgemein sind Dimensionen besonders günstig, bei denen die von einem Prozessor bearbeitete Datenmenge ohne Rest durch die Seitengröße teilbar ist, d.h. die Bedingung

$$\frac{n^2}{p} = k * 512, k = 1, 2, 3..., p = Prozessorzahl, n = Felddimension$$

ist erfüllt. Für 16 Prozessoren ist diese Bedingung z.B. für $n = 128$ gegeben.

Diese theoretische Überlegung wurde anhand einer Reihe von Simulationen überprüft, in denen die Feldgröße variiert und die Laufzeit gemessen wurde. Die Ausführungszeit ist dabei auf die dritte Potenz der Felddimension n bezogen: Da der Rechenaufwand der Matrixmultiplikation und damit auch die Laufzeit einer sequentiellen Version mit n^3 wächst, ist dieses Verhältnis umgekehrt proportional zum erzielten Speedup und bei konstanter Prozessorzahl auch umgekehrt proportional zur Effizienz; höhere Werte bedeuten hierbei eine schlechtere Effizienz. Abbildung 6.1 zeigt das Ergebnis.

Es zeigt sich, daß die Effizienz ab einer Feldgröße von 100x100 nur noch geringfügigen Schwankungen unterworfen ist, die Matrixmultiplikation bei kleineren Größen jedoch eine deutlich schlechtere Effizienz zeigt. In den folgenden Untersuchungen wurde daher eine Matrixmultiplikation der Feldgröße 100 als Beispiel eines Programmes mit guter Effizienz, d.h. niedriger Seitenfehlerrate gewählt.

Im Gegensatz zur blockweisen Verteilung bearbeitet bei der zyklischen Verteilung ein Prozessor jeweils nur eine Zeile des Zielfeldes; es ist daher zu erwarten, daß diese Variante erst dann ohne exzessives *false sharing* arbeitet, wenn eine Zeile des Ergebnisfeldes mindestens eine Speicherseite belegt; dies ist ab Feldgrößen von 512 der Fall. Für die Untersuchungen wurde ebenfalls eine Feldgröße von 100x100 gewählt; dies garantiert das erwünschte *false sharing*, andererseits erleichtert es aufgrund gleicher Problemgröße Vergleiche mit der ersten Variante.

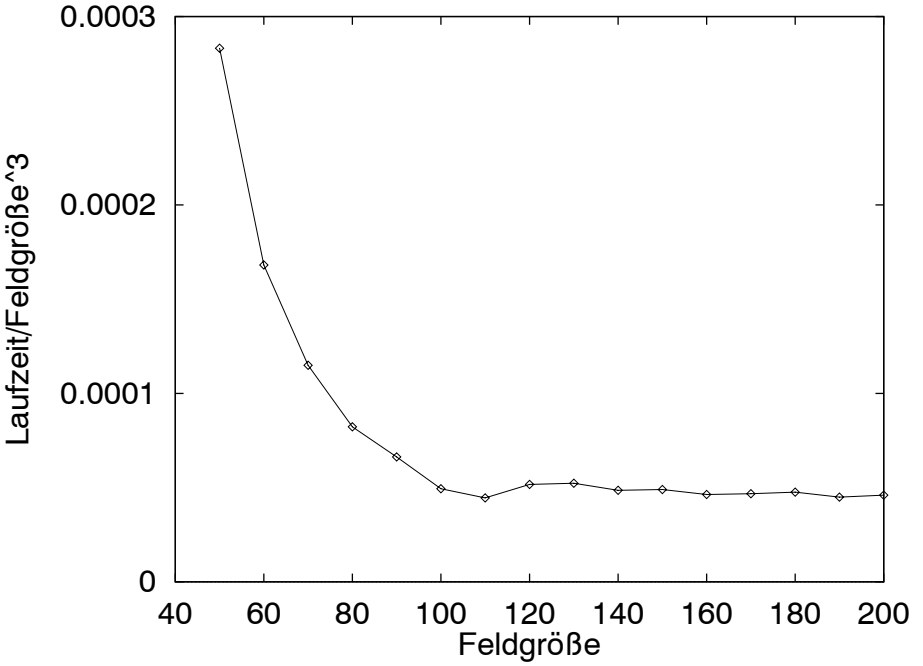


Abbildung 6.1
Effizienz der Matrixmultiplikation bei variierender Feldgröße

6.2.2 LU-Zerlegung

Neben diesen beiden Varianten der Matrixmultiplikation wurde ebenfalls das Laufzeitverhalten einer LU-Zerlegung untersucht, die in Bild 6.2 dargestellt ist.

Kennzeichen der LU-Zerlegung ist, daß mit jedem Durchlauf der äußeren, sequentiellen `K`-Schleife die Anzahl parallel ausführbarer Durchläufe in der mittleren `I`-Schleife um eins sinkt; gleichzeitig wird der Bereich des Feldes, auf dem gearbeitet wird, immer kleiner. Für eine Blockverteilung bedeutet dies, daß ab einem von Matrixgröße und Prozessorzahl abhängigen Wert *false sharing* auftreten kann. Für den hier untersuchten Fall von 16 Prozessoren und eine Matrixgröße von 200x200 Elementen ergibt sich ein Speicherbedarf von 1600 Byte pro Seite, es passen also ca. 3 Zeilen auf eine Speicherseite. Sobald die an die Prozessoren ausgegebenen Arbeitspakete in den Bereich dieses Wertes kommen, tritt *false sharing* über das durch fehlende Ausrichtung verursachte Maß hinaus auf. Bei einer Prozessorzahl von 16 ist dies also etwa ab der 150. Iteration gegeben. Allerdings sinkt auch mit steigendem `K` die Anzahl der zu bearbeitenden Elemente in der Matrix, was die *False Sharing*-Effekte begrenzt. Über die ganze Zerlegung hinweg sind jedoch Lese-Seitenfehler gegeben: Im Durchlauf k benötigt jeder Prozessor die Zeile k zur Modifizierung der von ihm zu bearbeitenden Zeilen; diese Zeile ist jedoch gerade im vorigen Durchlauf von einem Prozessor schreibend bearbeitet worden, der demzufolge zu Beginn

des k-ten Durchlaufes alleinige Rechte auf die zugehörige Seite besitzt. Dies bedeutet, daß die betroffene Seite zu Beginn des Durchlaufs an alle anderen Prozessoren verschickt werden muß. Da es sich dabei nur um Lesezugriffe handelt, wäre zwar prinzipiell ein überlappendes Versenden möglich (was die in diesen Untersuchungen verwendete Version des Simulationssystems nicht unterstützt), dessen Effizienz würde aber durch zwei Punkte deutlich begrenzt:

- Die Anzahl gleichzeitig verschickbarer Seiten wird durch die Anzahl der Kommunikationsverbindungen eines Prozessors begrenzt. In einem 2-dimensionalen Gitter könnten damit unter Idealbedingungen 4 Kopien gleichzeitig verschickt werden. Dieser Wert reduziert sich, wenn der Prozessor sich am Rand des Gitters befindet oder alle Empfänger sich „in der gleichen Richtung" befinden, so daß einzelne Verbindungen nicht genutzt werden können.

- Die betreffende Zeile befindet sich mit anderen Zeilen auf einer Seite, die möglicherweise noch bearbeitet werden müssen. Diese Schreibzugriffe verursachen ein dauerndes Invalidieren der Lesekopien. Dieses Problem läßt sich durch die Einführung eines privaten, temporären Vektors (im Listing mit `T` bezeichnet) mildern, in den alle Prozessoren vor der `I`-Schleife die Zeile umkopieren. Es läßt sich aufgrund des nicht-synchronisierten Ankommens vor der Kopierschleife allerdings nicht ganz ausschließen.

Ist die k-te Zeile allerdings einmal übertragen und treten keine Invalidierungen auf, so läuft der Rest eines Durchlaufs mit ähnlich guter Effizienz ab wie die Matrixmultiplikation mit Blockverteilung. Der Effekt, daß durch die Nichtausrichtung von Zeilenstartadressen auf Seitengrenzen in dieser Phase sporadisch Seitenwanderungen auftreten, wird um so geringer, je weiter der Rechenvorgang fortschreitet, da ein immer kleinerer Teil der Zeilen bearbeitet wird. Insgesamt stellt die LU-Zerlegung also ein Programm dar, in dem sich Phasen mit für SVM günstigem und ungünstigem Verhalten abwechseln. Sie kommt damit dem Verhalten eines realen Programmes deutlich näher als die beiden Versionen der Matrixmultiplikation.

Betrachtet man die Laufzeiten bei dedizierter Ausführung unter Variation von Prozessorzahl und Problemgröße, so ergeben sich die in Abbildung 6.3 gezeigten Verhältnisse.

Es zeigt sich, daß die LU-Zelegung für Prozessorzahlen oberhalb von 4 nur schlecht skaliert, und die Laufzeiten zum Teil auch wieder ansteigen. Dieser Anstieg ist zum einen begründet durch den höheren Aufwand, zu Beginn einer Iteration die k-te Zeile an alle Prozessoren zu verteilen, zum anderen durch eine höhere Wahrscheinlichkeit von *false sharing*. Diese beiden Faktoren überwiegen ab einer von der Problemgröße abhängigen Grenze die Vorteile des möglichen höheren Parallelismus. Alle folgenden Simulationen wurden mit einer Problemgöße von $200 * 200$

durchgeführt; dies ist noch eine Größe, bei der die Laufzeit bei voller Prozessorzahl ansteigt, die Verfahren zur Seitenfehlerreduktion also einen Effekt haben könnten, andererseits ist der Simulationsaufwand noch vertretbar.

6.3 Ausnutzung von Leerlaufzeiten durch sequentielle Hintergrundlasten

Zur Untersuchung der Frage, inwieweit sich durch Seitenfehler bedingte Wartezeiten in parallelen SVM-Anwendungen durch sequentielle, im Hintergrund laufende Anwendungen nutzen lassen, wurden ein Programm mit geringer Seitenfehlerrate sowie eines mit hoher Seitenfehlerrate ausgeführt, wobei auf jedem Prozessor zusätzlich ein sequentielles Programm lief. Dieses Programm ist in Abbildung 6.4 gezeigt und berechnet fortwährend Skalarprodukte. Da das Programm nur auf einem Prozessor läuft und damit keine Unterbrechungen durch Seitenfehler oder Kommunikation auftreten können, ist es immer rechenbereit und somit geeignet, alle verfügbaren Rechenpausen aufzufüllen. Die Simulationen wurden nach Beendigung der parallelen Anwendung abgebrochen, d.h. Statistiken über die den sequentiellen Programmen zugeteilte Rechenzeit beziehen sich nur auf die Ausführungszeit des parallelen Programmes. Bei den parallelen Anwendungen handelt es sich um die im vorigen Abschnitt beschriebene Matrixmultiplikation (Abbildung 5.3), zum einen mit einer Block- und zum anderen mit einer zyklischen Verteilung in der zweiten parallelen Schleife. Während die zyklische Verteilung eine SVM-Anwendung mit extrem hoher Seitenfehlerrate darstellt, erzeugt die Blockverteilung nach einer Anlaufzeit nur noch sporadisch Seitenfehler, die durch die Nichtausrichtung von Zeilen auf Seitenanfänge bedingt sind.

Es liegt auf der Hand, daß ein Hintergrundbetrieb nur dann effizient sein kann, wenn die Umschaltzeit zwischen zwei Threads nicht zu groß ist. Zur Quantifizierung dieser Überlegung wurde die Kontext-Umschaltzeit über einen weiten Bereich variiert. Der gewählte Bereich bis zu 2 Millisekunden mag auf den ersten Blick unrealistisch hoch erscheinen, es ist jedoch zu überlegen, daß durch zwischenzeitliches Umschalten auf einen anderen Thread zusätzliche Verzögerungen durch Cache-Flushes auftreten, d.h. die Hintergrundanwendung verdrängt die Daten des parallelen Programmes aus den Prozessor-Daten-Caches. Da praktisch alle modernen Mikroprozessoren mit Caches arbeiten, im Simulationssystem jedoch keine Simulation von Caches vorgesehen ist, wurde versucht, diese Effekte in Form längerer Kontextumschaltzeiten zu berücksichtigen.

```
[Shared]

Array A
 Contents Real
 Rank 2
 Dimension $1,$1
 Alignment 3
End


[Private]

Scalar I,J,K
 Contents Integer
End

Scalar L
 Contents Real
End

Array T
 Contents Real
 Rank 1
 Dimension $1
End
```

```
[Code]

InitialThreads $2

PDO(BLOCK) I=1,$1
 DO K=1,$1
  A(I,K)=0
 ENDDO
 A(I,I)=1
ENDDO


DO K=1,$1-1
 DO J=K,$1
  T(J)=A(K,J)
 ENDDO
 PDO($3) I=K+1,$1
  L=A(I,K)/T(K)
  A(I,K)=L
  DO J=K+1,$1
   A(I,J)=A(I,J)-L*T(J)
  ENDDO
 ENDDO
ENDDO
```

Abbildung 6.2
Programm zur LU-Zerlegung

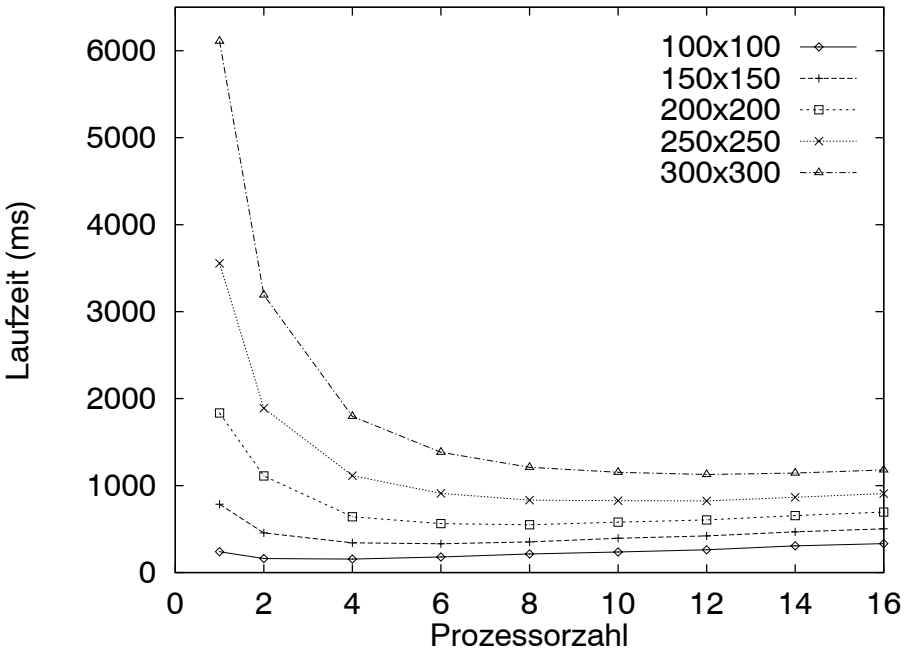


Abbildung 6.3
Laufzeit der LU-Zerlegung bei variierender Problemgröße und Prozessorzahl

```
[Private]                    [Code]

Array A,B                    InitialThreads 1
 Contents Real
 Rank 1                      DO I=1,$1
 Dimension $1                 A(I)=I
 Alignment 0                  B(I)=I
End                          ENDDO

Scalar I,Z                   DO Z=1,1000000
 Contents Integer             SUM=0.0
End                           DO I=1,$1
                               SUM=SUM+A(I)*B(I)
Scalar SUM                    ENDDO
 Contents Real                PRINT SUM
End                          ENDDO
```

Abbildung 6.4
Skalarprodukt als sequentielle Hintergrundlast

Um eine untere Grenze für die Laufzeit des parallelen Programmes zu erhalten, wurde jeweils zu Vergleichszwecken das parallele Programm ohne Hintergrundlast gemessen, und zwar jeweils mit und ohne Spin-Waiting bei Seitenfehlern.

6.3.1 Matrixmultiplikation

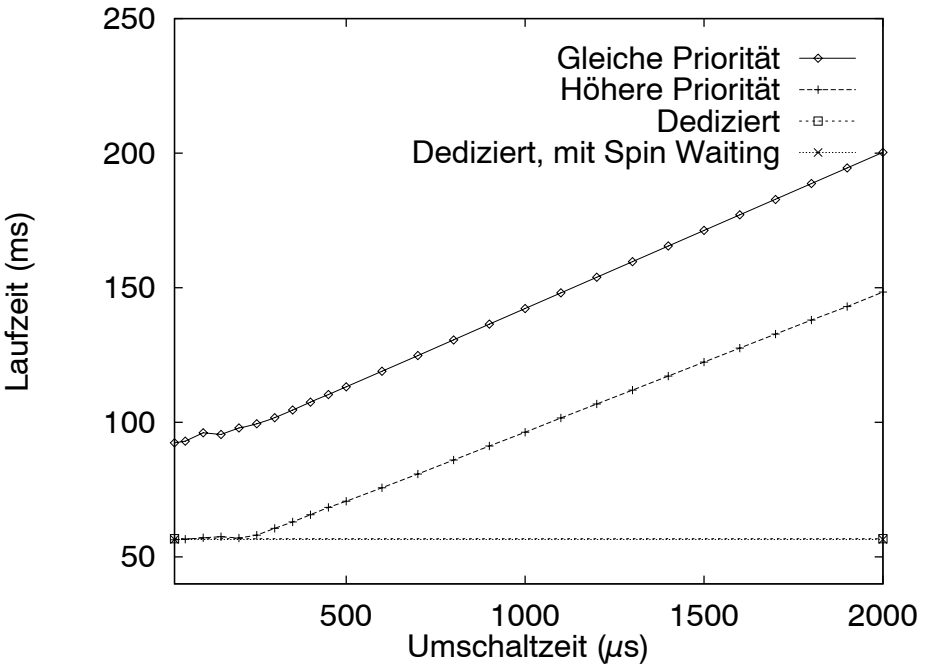


Abbildung 6.5
Paralleler Job mit sequentieller Hintergrundlast - niedrige Paging-Rate

Abbildung 6.5 zeigt die Laufzeitergebnisse für die Variante mit Blockverteilung, also relativ geringer Seitenfehlerrate. Zu Vergleichszwecken wurde ebenfalls eine Reihe mitgemessen, in der das parallele Vordergrundprogramm keine höhere Priorität erhalten hatte. Wie erwartet, erhöht sich die Gesamtlaufzeit des parallelen Programmes um den Betrag der von den sequentiellen Programmen verbrauchten Rechenzeit, die Kurve zeigt qualitativ aber den gleichen Verlauf: Bis zu einer Kontext-Umschaltzeit von ca. 200μs zeigen sich keine nennenswerten Veränderungen in der Laufzeit, sie bleibt auf einem Wert, der auch im Einzelprogrammbetrieb (mit oder ohne Spin-Waiting bei Seitenfehlern) erreicht wird. Dieses Verhalten ist qualitativ dadurch erklärbar, daß bis zu einer Umschaltzeit in der Größenordnung eines Seitenfehlers längere Umschaltzeiten dadurch kompensiert werden, daß die Zeitscheibe des Hintergrundprogrammes entsprechend verkürzt wird; hat das parallele Programm dagegen keine höhere Priorität, so nutzt die sequentielle Last nach Ankunft der

Seite ihre Zeitscheibe aus und verzögert das parallele Programm entsprechend; dies erklärt die bereits im Bereich bis 200 μs ansteigende obere Kurve.

Abbildung 6.6 zeigt die den Hintergrundprogrammen zur Verfügung gestellte Rechenzeit, summiert über alle 16 Prozessoren. Berücksichtigt wurde dabei nur die Startphase der Multiplikation, in der eine große Zahl von Lese-Seitenfehlern auf die Quellfelder auftritt: Da die Blockverteilung während der Berechnung nur minimal Schreibfehler verursacht, ist die einzige weitere Phase, in der den Hintergrundprogrammen nennenswert Rechenzeit zugeteilt wird, die Auslaufphase, in der einzelne Prozessoren früher fertigwerden und an der Synchronisationsstelle der parallelen Schleife warten müssen. Dies sind aber keine Leerlaufzeiten, die durch Seitenfehler verursacht wurden.

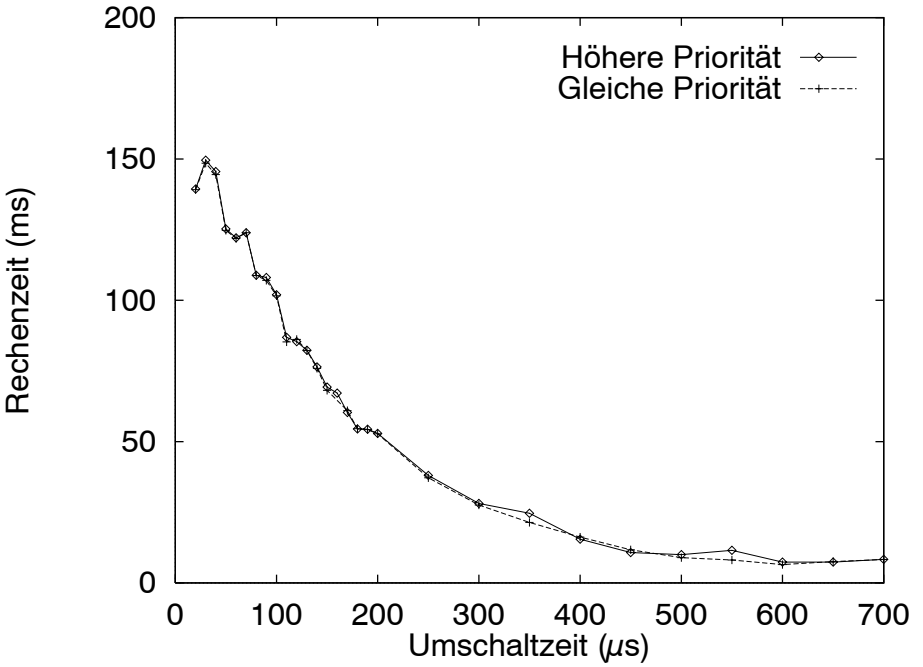


Abbildung 6.6
Rechenzeit für Hintergrundlasten - niedrige Paging-Rate

Die Kurve zeigt einen zuerst steileren, dann zunehmend flacheren Abfall bei höheren Kontextumschaltzeiten. Im Gegensatz zu der Ausführungszeit ist hier aber kein scharfer Knick bei einem bestimmten Zeitwert zu erkennen. Diese „Unschärfe" ist zum einen der Summierung über alle Prozessoren zuzuschreiben, zum anderen geht der Multiplikation eine globale Synchronisation voraus, die nicht alle Prozessoren zum gleichen Zeitpunkt verlassen, d.h. es ist schwierig, einen globalen Startpunkt festzulegen, an dem die Multiplikation beginnt. Interessant ist ferner, daß sich kein Unterschied einstellt, wenn dem parallelen Programm keine höhere Priorität gegeben

wird. Dieser Effekt ist einer Eigenschaft der Round-Robin-Komponente des verwendeten lokalen Schedulers zuzuschreiben: Trifft eine angeforderte Seite an, so wird unabhängig von der Scheduling-Strategie zuerst das laufende (Hintergrund-)Programm unterbrochen und der Scheduler angestoßen, da nur dieser seine eigene Strategie kennt und entscheiden kann, ob dieser neue Thread den Prozessor sofort erhalten muß. Nach der Round-Robin-Regel wird dann der bisher laufende Thread höchster Priorität in der Warteschlange nach hinten gestellt und der nachfolgende aktiviert; dies ist in diesem Fall aber der parallele Thread, so daß die Threads des parallelen Programmes trotz ihrer nicht höheren Priorität den Prozessor schnellstmöglich nach Erhalt einer Seite zugeteilt bekommen; der Programmverlauf in der seitenfehlerintensiven Anlaufphase ist identisch und verändert sich erst in der folgenden, seitenfehlerarmen Phase: Dort sorgt das Round-Robin-Prinzip dafür, daß das parallele Programm nur noch die Hälfte der Prozessorzeit erhält und bewirkt die konstante Erhöhung der Laufzeit in Abbildung 6.5.

Wäre das Round-Robin-Verfahren so implementiert, daß ein Thread, der eine Scheduling-Entscheidung nicht zu verantworten hat (also in diesem Fall der sequentielle Hintergrundprozeß), seine Zeitscheibe aufbrauchen darf, bis die Umschaltung stattfindet, so wäre der Unterschied zwischen den Kurven in Abbildung 6.5 deutlich, da dann jeder Thread pro Seitenfehler eine Verzögerung in Größenordnung der Zeitscheibenlänge erfahren und diese Zeit sich auch in Abbildung 6.6 entsprechend niederschlagen würde. Ein solches Verhalten wäre aber sicher nicht erwünscht, da es den Ablauf des parallelen Programmes in einem unakzeptablen Maße verzögern würde; ein gänzliches Entfernen der Round-Robin-Komponente wäre katastrophal, da ein paralleler Thread nach dem ersten Seitenfehler den Prozessor bis zum Ende des Hintergrundprogrammes nicht mehr zurückerhalten würde.

Diese Beobachtung führt zu der Erkenntnis, daß einem Thread, der eine angeforderte Seite erhalten hat, der Prozessor möglichst schnell zur Verfügung gestellt werden sollte, idealerweise durch eine temporäre Anhebung seiner Priorität: Solange ein Thread auf diese Seite nicht zugreifen kann, bleibt diese für ihn verriegelt und kann auch nicht zu anderen Prozessen, die diese Seite benötigen, weiterwandern; damit werden weitere Prozesse eines parallelen Programmes ausgebremst!

Es bleibt zu klären, weshalb der Kurvenknick in der Ausführungszeit erst bei einer Umschaltzeit liegt, die deutlich über der Zeit zur Auflösung eines Seitenfehlers liegt: Wie bereits im vorigen Abschnitt angesprochen, treten zu Anfang der Matrixmultiplikation sehr viele Lese-Seitenfehler auf den beiden Quellmatrizen auf. Diese hohe Seitenfehlerrate führt einerseits zu einem höheren Nachrichtenaufkommen auf dem Netz mit zeitweisen Blockierungen, andererseits sequentialisiert der SVM-Manager gehäuft auftretende Anfragen nach der gleichen Seite. Beide Faktoren führen dazu, daß sich die mittlere Bearbeitungszeit für einen Seitenfehler in

der Startphase verlängert. Abbildung 6.7 zeigt einen mit PARvis visualisierten Ausschnitt aus der Anlaufphase mit vielen Seitenfehlern. Die dick eingezeichneten Linien bezeichnen Seitentransfers, die dünnen Linien sonstige Steuernachrichten des SVM-Managements. Es ist zu erkennen, daß die Übertragung von Seiten inklusive der dafür nötigen Steuernachrichten zum Teil deutlich mehr als 150 μs benötigt, was auf Blockierungen im Netzwerk zurückzuführen ist. Als Beispiel markiert der weiße Pfeil auf Höhe vom Prozessor 8 die Gesamtzeit zur Bearbeitung eines Seitenfehlers auf selbigem Prozessor. Die sich in diesem Fall ergebende, durch die Länge des Pfeils verdeutlichte Zeit von ca. 200 μs ist im kleinen Fenster dargestellt.

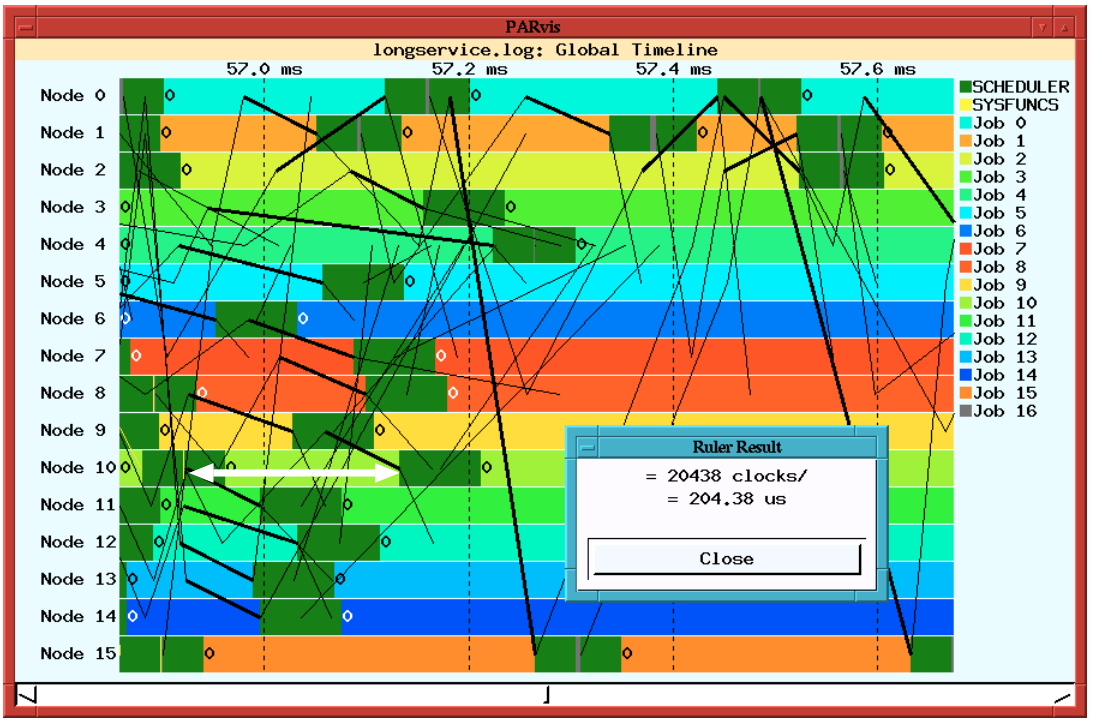


Abbildung 6.7
Verlängerte Seitentransferzeiten durch zeitweise Überlastung des Netzwerks

Abbildung 6.8 zeigt die durchschnittlichen Zeiten zur Auflösung eines Seitenfehlers, die mit einer Version des Simulators mit erweiterter Protokollierung für die gegebene Konfiguration ermittelt wurden. Der Bruch bei ca. 100 μs wird durch die in den anderen Kurven nicht berücksichtigte Initialisierung der beiden Quellfelder verursacht: Bis zu diesem Punkt wird die Initialisierung so schnell durchlaufen, daß ein Prozessor beginnt, Variablen auf einer Seite zu initialisieren, die ein anderer Prozessor noch nicht verlassen hat. Dies verursacht *false sharing* zwischen den beiden Prozessoren und eine dementsprechend starke Verzögerung des Programmablaufes. Dies bedeutet aber, daß alle anderen Prozessoren, die unbehindert von *false sharing*

laufen, wesentlich schneller ihren Teil der Initialisierung abschließen: sie warten an einer Synchronisationsstelle und führen während dieser Wartezeit ihre Hintergrundprogramme aus. Da die Hintergrundprogramme jedoch keinen Nachrichtenverkehr verursachen, ist das Netzwerk in dieser Phase für die Auflösung der Seitenfehler frei, und die betreffenden Nachrichten können ohne Verzögerung laufen. Da die Anzahl dieser Seitenfehler erheblich ist, wird der Durchschnittswert durch sie nach unten verschoben. Dies bedeutet, daß erst die Werte jenseits des Sprunges einen realistischeren Eindruck für die Verhältnisse während der Startphase der Matrixmultiplikation vermitteln. Anhand dieser Werte in einer Größenordnung von 300 μs ergibt sich, daß der Laufzeitanstieg in Bild 6.5 merklich *vor* den Kontextumschaltzeiten in der Größenordnung der effektiven Bearbeitungszeit eines Seitenfehlers liegt.

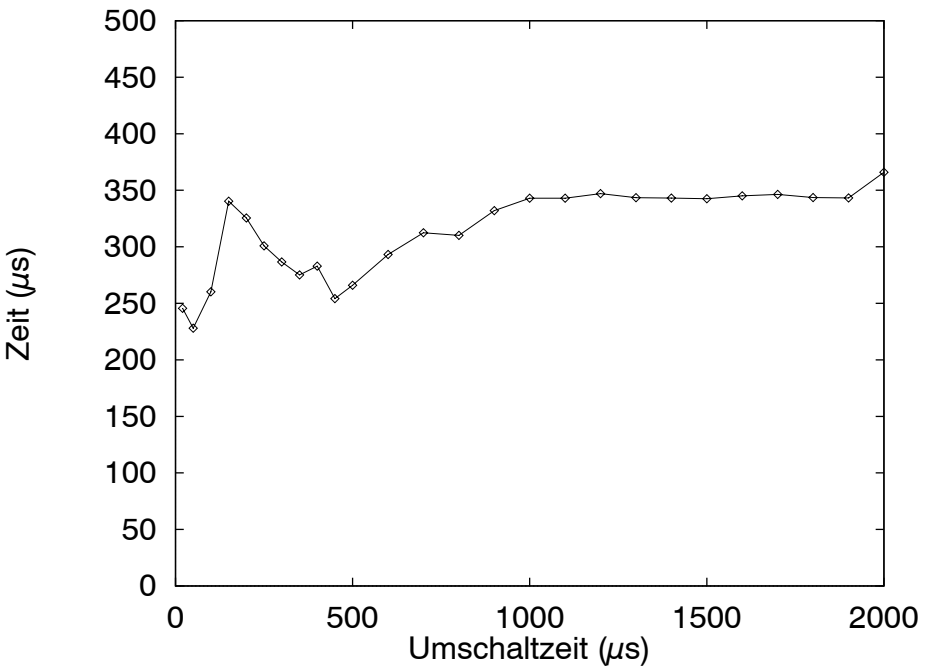


Abbildung 6.8
Mittlere Zeit zur Behebung eines Seitenfehlers in Abhängigkeit von der Kontext-Umschaltzeit

Um zu ermitteln, inwieweit sich die Kurven für Netzwerktopologien mit höherem Verbindungsgrad und damit geringerer Wahrscheinlichkeit von Verklemmungen verändern, wurde die gleiche Simulationsreihe mit einer Reihe anderer Vernetzungstopologien durchgeführt. Das Ergebnis zeigt Abbildung 6.9, in dem die Ausführungszeiten der Gitter-Topologie mit einem Hypercube, einem 2-dimensionalen Torus sowie dem (unrealistischen) Extremfall einer vollständigen Vernetzung verglichen werden.

Abbildung 6.9 zeigt, daß bei einer vollständigen Vernetzung, bei der keine

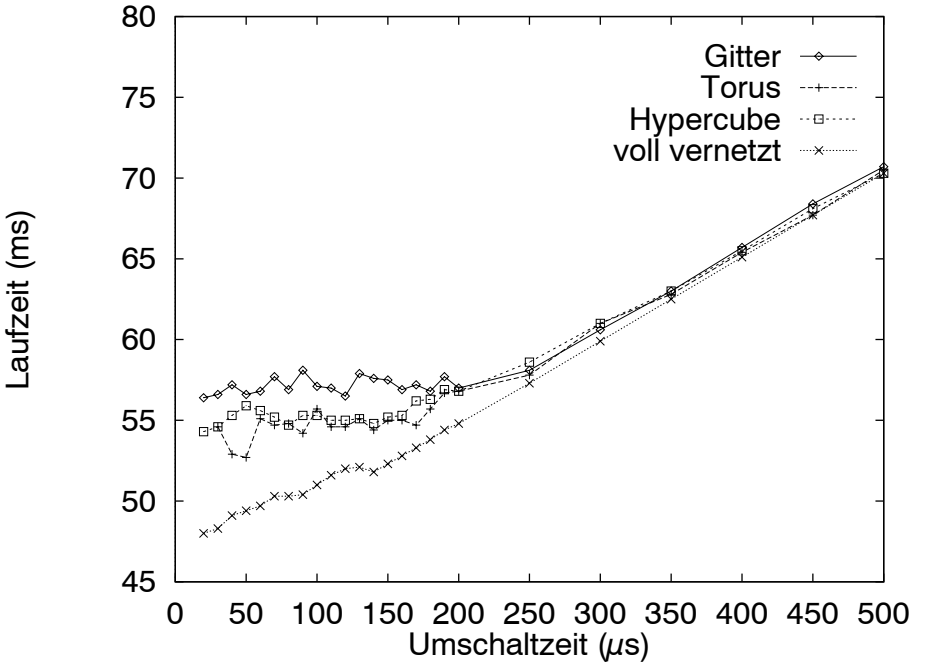


Abbildung 6.9
Ausführungszeiten mit niedriger Seitenfehlerrate bei verschiedenen Netzwerktopologien

Blockierungen auf dem Netzwerk auftreten können und die Auflösungszeiten für Seitenfehler immer ihre Minimalwerte annehmen, die Ausführungszeiten bereits bei kleinen Kontextumschaltzeiten sich mit einer ähnlichen Steigung wie später bei allen anderen Kurven erhöhen. Torus und Hypercube zeigen ein ähnliches Verhalten wie das Gitter, erreichen im Bereich näherungsweise konstanter Ausführungszeiten jedoch ein etwas geringeres Niveau als das Gitter. Der Grund für dieses Verhalten dürfte in der geringeren Kommunikationsbreite der Gitter-Topologie zu suchen sein: während bei Torus und Hypercube *jeder* Prozessor vier Direktverbindungen zu anderen Prozessoren besitzt, sind im Gitter die Prozessoren am Rand (drei Verbindungen) oder gar in den Ecken (nur zwei Verbindungen) benachteiligt; sie müssen ihre Nachrichten, die z.B. beim Torus über Direktverbindungen laufen, über Routen mit einer Reihe von Zwischenstationen schicken, was einerseits die Transferzeiten erhöht, andererseits auch die Gefahr von Blockierungen auf dem Netz erhöht.

Für sehr große Kontextwechselzeiten zeigen alle Kurven näherungsweise die gleichen Ausführungswerte. In diesem Bereich ist die Verzögerung eines Threads durch einen Seitenfehler so hoch, daß die einzelnen Seitenfehler auch weiter gestreut über der Zeitachse auftreten, was die Wahrscheinlichkeit von Blockierungen reduziert.

Allen Kurven mit einem Teil näherungsweise konstanter Ausführungszeit ist eine

gewisse Streubreite in diesem Teil gemeinsam, d.h. das Verhalten ist unsystematisch. Der Grund hierfür ist, daß unterschiedlich hohe Kontext-Umschaltzeiten die Ausführung einzelner Threads zeitlich unterschiedlich gegeneinander versetzen, so daß sich möglicherweise ein anderes, mit mehr oder weniger Kollisionen auf dem Netz verbundenes Kommunikationsmuster ergibt. Die Unterschiede in der Ausführungszeit sind aber so marginal, daß sie in der Praxis keine Rolle spielen sollten, wenn das Ziel nicht die absolut bestmögliche Performance eines einzelnen Programmes ist (in einem solchen Falle wäre ohnehin ein dedizierter Betrieb angebracht).

Deutlich anders stellt sich die Situation im Falle zyklischer Verteilung, also sehr hoher Seitenfehlerrate dar (Abbildung 6.10). Ausgehend von der Ausführungszeit im Einzelprogrammbetrieb steigt die Kurve näherungsweise linear an, wobei kein nennenswerter Unterschied besteht, ob das parallele Programm mit gleicher oder höherer Priorität ausgeführt wird (die beiden Kurven überdecken sich exakt im Bild). Als Grund für das Fehlen eines Abschnittes mit konstanter Ausführungszeit ist anzusehen, daß die Variante mit hoher Seitenfehlerrate bereits im dedizierten Modus eine Ausführungszeit besitzt, die ein Mehrfaches der Ausführungszeit der Variante mit niedriger Seitenfehlerrate beträgt. Dies bedeutet, daß die Ausführungszeit dieser Variante allein durch die Zeit zur Auflösung eines Seitenfehlers (in die die Zeit zur Kontext-Rückumschaltung einbezogen werden kann) bestimmt wird; der eigentliche Rechenzeitanteil an der Gesamtausführungszeit (ca. 40 ms auf jedem Prozessor) ist zwar absolut genauso hoch wie bei niedriger Seitenfehlerrate (es wird die gleiche Menge an Berechnungen durchgeführt, nur mit einer anderen Verteilung), jedoch relativ gesehen vernachlässigbar; dieses Programm wird durch die Einführung von Hintergrundlasten also deutlich langsamer. Der Effekt sollte allerdings nicht überbewertet werden, da dieses Programm ohnehin keine gute Performance erzielt und daher nicht als Bewertungsmaßstab genommen werden kann, ob eine Ausführung sequentieller Hintergrundlasten ein paralleles, im Vordergund laufendes Programm behindert.

Die für Hintergrundprogramme zur Verfügung stehende Rechenzeit wurde für den Fall einer hohen Seitenfehlerrate nicht untersucht. Dies ist zum einen nicht erfolgt, weil für die Kurven qualitativ kein anderer Verlauf zu erwarten ist als bei niedriger Seitenfehlerrate (aufgrund der höheren Seitenfehlerrate wären die absoluten Zeitwerte naturgemäß höher), und es zum anderen bei dieser Variante erheblich schwieriger ist, die durch Seitenfehler und durch unterschiedliche Ankunftszeiten der Threads am Schleifenende entstehenden Zeiträume zu trennen: Die hohe Seitenfehlerrate führt in Zusammenhang mit den Netzwerklatenzen zu einer ,,Trennung" der Datenzugriffsbereiche der einzelnen Threads, auf die im folgenden Abschnitt noch genauer eingegangen wird. Dies wiederum hat zur Folge, daß einzelne Threads ihren Arbeitsanteil bereits nach einem Bruchteil der Gesamtlaufzeit abgeschlossen haben. Die diesen Threads zugeordneten Prozessoren können bis zu dem Zeitpunkt, an dem

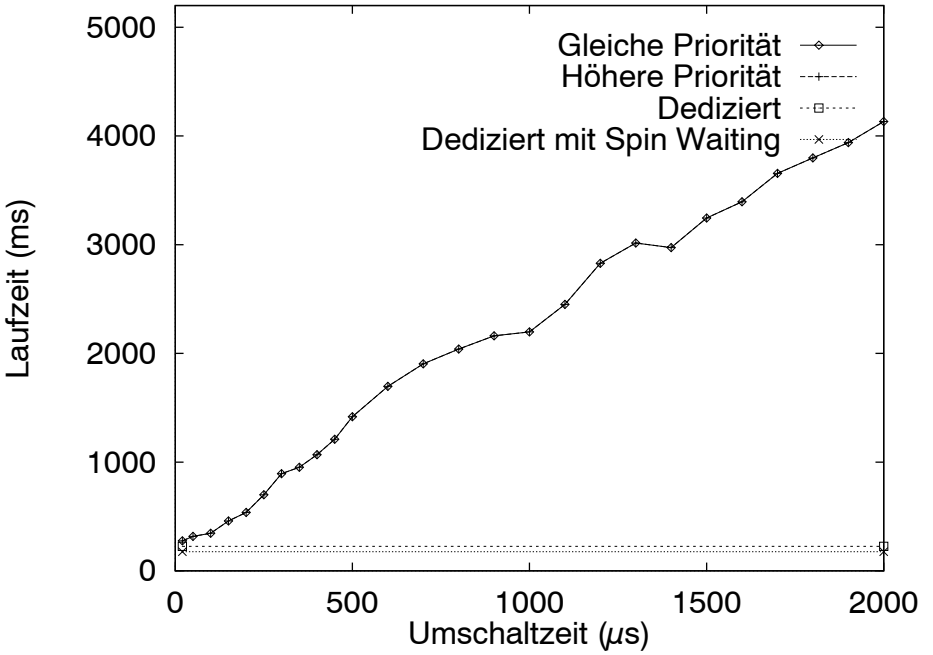


Abbildung 6.10
Paralleler Job mit sequentieller Hintergrundlast - hohe Paging-Rate

auch die letzten Threads die Synchronisationsstelle am Schleifenende erreicht haben, ihre Zeit den sequentiellen Hintergrundprozessen vollständig zur Verfügung stellen, und dieser Anteil überlagert die eigentlich interessierenden Werte deutlich. Für eine Kontextumschaltzeit von 20 μs ergibt sich z.B. ein Zeitanteil von 2 Sekunden durch Seitenfehler, jedoch 1,5 Sekunden durch Wartezeiten an der Synchronisation.

6.3.2 LU-Zerlegung

Wie in der Beschreibung der LU-Zerlegung bereits festgestellt wurde, kann sich das Speicherzugriffsverhalten und damit auch die Seitenfehlerrate der LU-Zerlegung mit fortschreitender Laufzeit verändern. Aus diesem Grund macht es wenig Sinn, die Programmlaufzeit nur summarisch zu betrachten; stattdessen zeigen Abbildung 6.11 bis 6.13 die Laufzeiten auf die Durchläufe der `K`-Schleife aufgeschlüsselt für verschiedene Kontextumschaltzeiten; Abbildung 6.14 zeigt noch einmal die Gesamtlaufzeiten.

Es zeigt sich anhand der unruhigen Verläufe, daß schon bei vergleichsweise kleinen Umschaltzeiten die Laufzeiten für einzelne Iterationen massiv anwachsen, die Laufzeiten für andere Iterationen jedoch gleichbleiben oder erst später anzusteigen beginnen. Offensichtlich handelt es sich bei diesen Iterationen um Iterationen, in denen durch die Blockverteilung und die fehlende Ausrichtung besonders starkes *false*

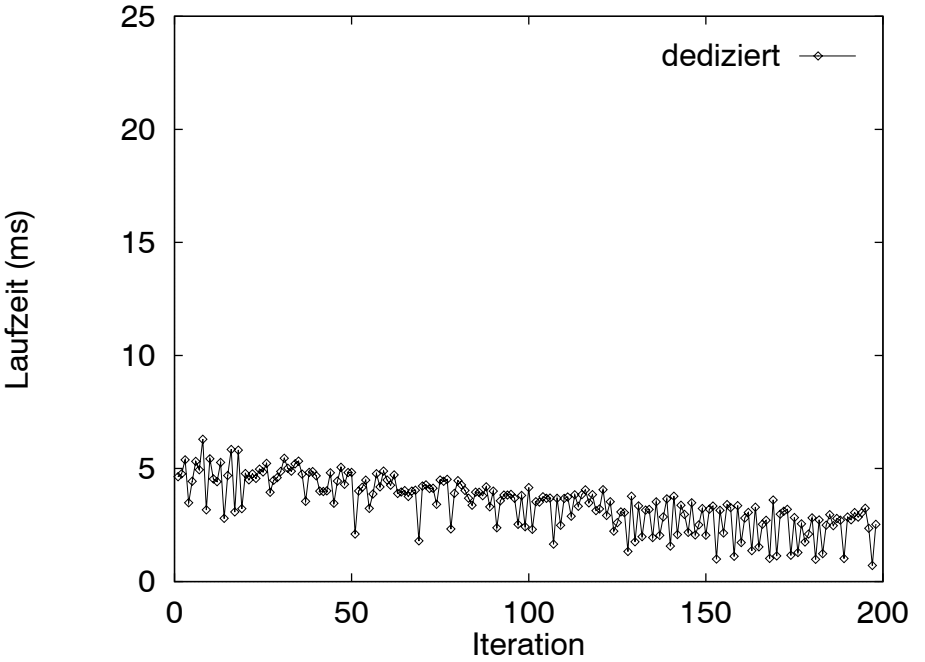


Abbildung 6.11
Laufzeit der LU-Zerlegung pro Iteration (Bild 1 von 3)

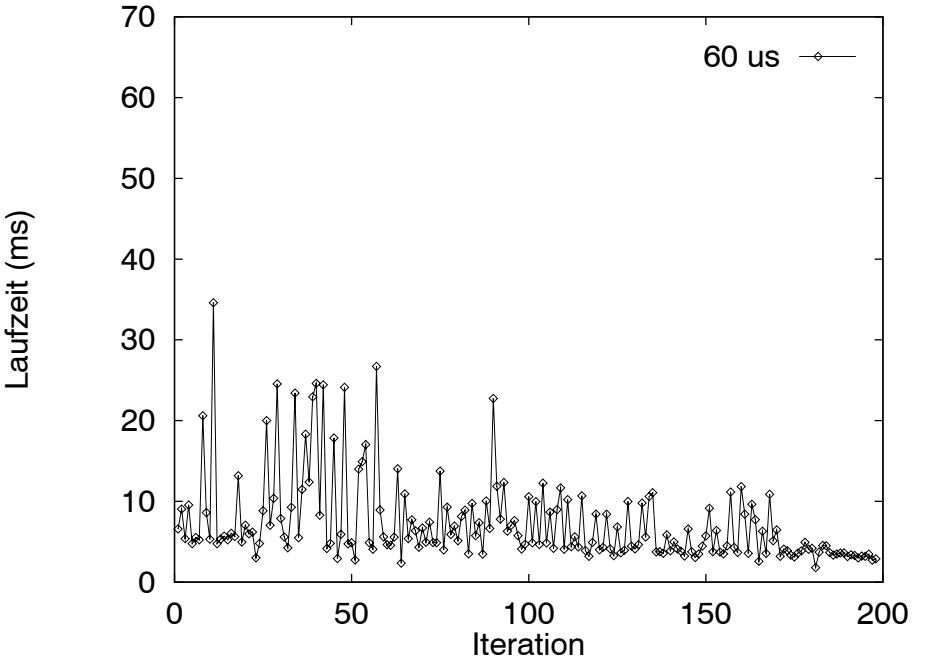


Abbildung 6.12
Laufzeit der LU-Zerlegung pro Iteration (Bild 2 von 3)

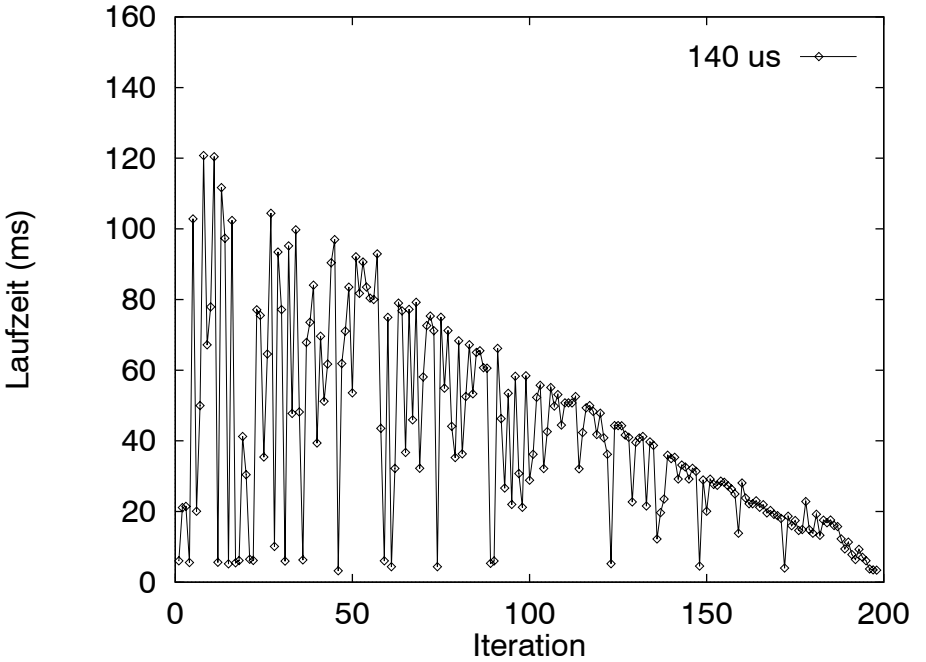


Abbildung 6.13
Laufzeit der LU-Zerlegung pro Iteration (Bild 3 von 3)

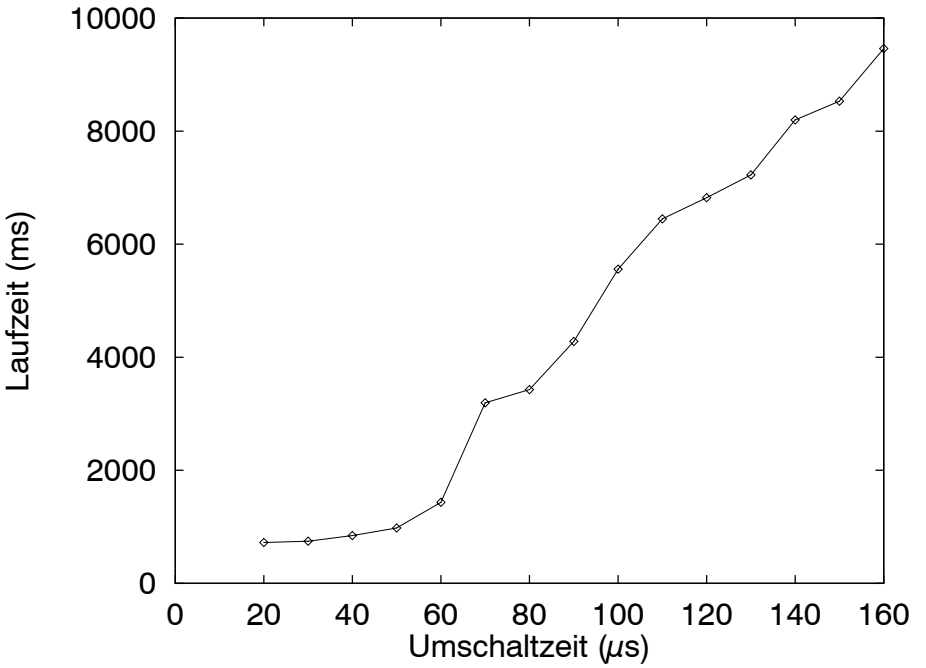


Abbildung 6.14
Gesamtlaufzeit der LU-Zerlegung

sharing auftritt; da die Anzahl der internen Rechenoperationen pro Zugriff auf das Feld hier ungleich niedriger ist als bei der Matrixmultiplikation (2 Rechenoperationen gegenüber einem kompletten Skalarprodukt), wird die Laufzeit nur noch durch die Geschwindigkeit bestimmt, mit der die betroffenen Seiten hin- und hergereicht werden können; es tritt paktisch immer ein Seitenfehler pro Schreib- oder Lesezugriff auf, was auch den linearen Abfall der Spitzenwerte zu höheren Iterationen hin erklärt. Da weiterhin eine Seite nach Erhalt für einen Zugriff verriegelt ist, der Zugriff aber erst nach der Kontextumschaltung erfolgen kann, geht die Kontextumschaltzeit direkt in die Zykluszeit ein, was den stetigen Anstieg der Spitzen- und der Gesamtlaufzeit (die von den Spitzenwerten dominiert wird) erklärt. In der Abbildung, die die Gesamtlaufzeit zeigt, ist aber auch noch ein kleiner Bereich (bis ca. 60 μs) zu erkennen, in dem die Programmlaufzeit fast nicht ansteigt. Die LU-Zerlegung könnte also vermutlich ein ähnlich gutes Verhalten wie die Matrixmultiplikation zeigen, wenn die Laufzeit nicht bereits ab 60 μs von den Iterationen mit starkem *false sharing* dominiert werden würde. Die Laufzeitkurve der LU-Zerlegung zeigt mithin sowohl Charakteristiken der Matrixmultiplikation mit hoher als auch mit niedriger Seitenfehlerrate und rechtfertigt damit in gewisser Weise die Annahme dieser Programme als „Extrembeispiele". Insgesamt ist aber festzustellen, daß sich ohne Maßnahmen zur Reduzierung der *False-Sharing*-Effekte die LU-Zerlegung nur sehr begrenzt als für die Hintergrundausführung anderer Programme eignet. Diese Maßnahmen könnten sowohl auf der Seite der Anwendung als auch (mit vermutlich etwas geringerer Effizienz, da das System erst eingreifen kann, wenn die *Sharing*-Situation bereits entstanden ist, der Programmierer sie durch Umstrukturierung jedoch von vorherein verhindern kann) auf der Systemseite durch geeignetes Scheduling getroffen werden. Solche Maßnahmen werden im nächsten Abschnitt untersucht.

Als Zusammenfassung kann gesagt werden, daß sequentielle Hintergrundlasten ein Programm um so stärker beeinflussen, je höher die Seitenfehlerrate dieses Programmes ist. Für auf SVM optimierte Programme, die so wenig Paging verursachen, daß man durch Parallelverarbeitung eine wirkliche Beschleunigung erreicht, sind sequentielle Hintergundlasten sinnvoll, solange der Overhead zum Umschalten zwischen zwei Programmen deutlich unter dem Zeitaufwand zur Auflösung eines Seitenfehlers bleibt. Dies bedingt zum Beispiel, daß bei der Umschaltung kein zusätzlicher Overhead durch Ein-oder Auslagern von Programmdaten in den virtuellen Speicher erfolgen darf, da bereits der Transfer einer einzigen Seite von einem Sekundärspeicher (der bei üblichen, massiv-parallelen Rechnersystemen nur auf wenigen, ausgezeichneten Knoten vorhanden ist) eine Verzögerung in der gleichen Größenordnung wie der primäre Seitenfehler selbst bedeuten würde. Der lokale Speicher der Prozessoren muß also groß genug sein, den *working set* beider Programme aufzunehmen.

6.4 Behandlung von Programmen mit extrem vielen Seitenfehlern

In diesem Abschnitt wird untersucht, wie sich die in Kapitel 4 beschriebenen Verfahren zur Behandlung von Programmen mit hoher Seitenfehlerrate (also mit einem großen Anteil von *false* bzw. *true sharing*) verhalten.

6.4.1 Matrixmultiplikation

Als erstes Beispielprogramm soll hier wieder die relativ einfach zu analysiernde Matrixmultiplikation untersucht werden, und zwar in der Variante mit hoher Seitenfehlerrate. Im Gegensatz zum vorherigen Abschnitt wird hier auch die Initialisierung der Quellfelder mitberücksichtigt, da dieser Teil ein deutlich anderes Seitenfehlerprofil aufweist und so Gesichtspunkte für die Parametrisierung der in Kapitel 4 angesprochenen Algorithmus-Variante mit variablen Anhaltezeiten liefert.

Der zentralisierte und der dezentrale Algorithmus benutzen das gleiche Verfahren zur *Erkennung* von *Sharing*-Situationen. Die Erkennung beruht auf einer Speicherung aller Seitenfehler, die ein Thread in der kürzeren Vergangenheit ausgelöst hat. Überschreitet die Zahl der Seitenfehler auf eine bestimmte Seite eine bestimmte Grenze, so wird eine *Sharing*-Situation erkannt. Die einstellbaren Parameter dieses Verfahrens sind die Zeitspanne, nach der ein aufgetretener Seitenfehler ,,vergessen" wird, sowie die Zahl der in dieser Zeitspanne aufgetretenen Fehler bzw. die sich daraus ergebende Seitenfehlerrate. Die Wahl dieser Parameter ist kritisch für den Erfolg der nun zu entwickelnden Strategien zur *Behebung* des Problems: Spricht die Erkennung zu schnell an, so werden Threads unnötig angehalten, was einerseits den Parallelitätsgrad verschlechtert und andererseits durch die Sequentialisierung zu unnötigen Wartezeiten an Synchronisationsstellen führt. Spricht die Erkennung dagegen zu spät an, so können die Behebungsverfahren nur einen kleineren Zeitraum des Gesamtzeitraumes, in dem die *Sharing*-Situation auftritt, behandeln und verzögern den parallelen Programmablauf eventuell auch dadurch, daß die Warteperiode unnötig lange anhält.

Die entscheidende Größe zur Wahl der Parameter ist die Zeit zur Bearbeitung eines Seitenfehlers: Wenn zwei Threads in eine *Sharing*-Situation geraten und neben dem Warten auf die hin- und hergehende Seite keine weitere Operation ausführen, so beträgt der Rhythmus, in dem für einen Thread Seitenfehler auftreten, das *doppelte* der Seitenfehlerbearbeitungszeit. Verlangen dagegen n Threads nach einer Seite und sorgt der Manager dieser Seite (wie in dem benutzten Simulationssystem) dafür, daß alle Threads die Seite in einem Round-Robin-Schema erhalten, so steigt die Zykluszeit auf das n-fache der Seitenfehlerbearbeitungszeit. Da in der Realität durchaus

Situationen auftreten können, in denen ein erheblicher Teil aller Threads sich um eine Seite bemüht, sollte die als Grenzwert eingestellte Seitenfehlerrate in etwa dem Produkt aus Seitenfehlerbearbeitungszeit und der typischen Prozessorzahl eines Jobs entsprechen. Dabei ist auch zu berücksichtigen, daß aufgrund von Netzwerkblockaden sich die Seitenfehlerbearbeitungszeit in *Sharing*-Situationen etwas nach oben verschieben kann (s. vorheriger Abschnitt).

Entsprechend der gewählten Parameter im Simulator ergibt sich in Situationen mit hoher Seitenfehlerrate eine Seitenfehlerbearbeitungszeit in der Größenordnung von 300 μs. Multipliziert man dies mit einer maximalen Prozessorzahl von 16, so ergibt sich eine worst-case-Zykluszeit von ca. 5 ms. In einer ersten Meßreihe wurde festgelegt, daß für ein Eingreifen mindestens 10 Seitenfehler auftreten mußten, was einen ,,Gedächtnis" von 50 ms entspricht. Dieser Wert wurde in weiteren Meßreihen auf 5, 3 und 2 Seitenfehler reduziert.

Ohne Eingriffe benötigt die Matrixmultiplikation eine Laufzeit von 209 ms und erzeugt 4713 Seitenfehler (316,3 ms bzw. 5731 Seitenfehler mit Hintergrundlast); aufgrund des an allen Stellen auftretenden *false sharing* und der hohen Seitenfehlerrate steigt die mittlere Auflösungszeit auf 408 μs an. Einen Ausschnitt aus dem Lauf dieses Programmes ohne Hintergrundlasten zeigt Abbildung 6.15.

Das Bild zeigt den Zustand der einzelnen Prozessoren und den zwischen ihnen stattfindenden Nachrichtenverkehr. Es ist zu erkennen, daß kurzen Zeiten von Rechenaktivität lange *idle*-Zeiten ohne Programmfortschritt gegenüberstehen, deren Dauer durch Nachrichtenketten bestimmt wird. Diese Nachrichtenketten sind das Ergebnis von *false sharing* im Programm, bei dem mehrere Prozessoren auf eine Seite schreibend zugreifen, die dann zwischen den Prozessoren herumgereicht werden muß. Während die dickeren Linien den eigentlichen, längerdauernden Seitentransfer anzeigen, handelt es sich bei den dünneren Linien um Steuernachrichten vom und zum Manager der jeweiligen Seite (Prozessor 8, 9 bzw. 10). Da auf die beiden Quellfelder nur lesend zugegriffen wird, können die Zugriffskonflikte nur beim Schreiben der Ergebnisse in das Zielfeld auftreten. Da das Feld eine Größe von 100 Zahlen pro Zeile aufweist und jede Zahl 8 Byte Speicher belegt, passen bei einer Seitengröße von 4 KByte ca. 5 Zeilen auf jede Seite. Dies korreliert mit dem Bild, in dem zu erkennen ist, daß jeweils zwischen 5 Prozessoren eine Seite herumgereicht wird. Da die Seitengröße kein exaktes Mehrfaches der Zeilengröße ist, ist an dem unteren Zyklus momentan ein weiterer Prozessor beteiligt. Daraus könnte man ableiten, daß bei idealer Sequentialisierung der einzelnen Gruppen durch gezieltes Anhalten von Threads ein maximaler Parallelitätsgrad von 3 erreichbar wäre, d.h. die Programmlaufzeit wäre etwa 3-mal so hoch wie bei einem Programm mit optimaler Arbeitsverteilung. Als eine Näherung für ein solches Programm kann z.B. die Blockverteilung aus dem vorigen Kapitel verwendet werden, aus deren Laufzeit im dedizierten Betrieb (ca. 57 ms, siehe Abbildung 6.6) sich eine theoretische Zeit von

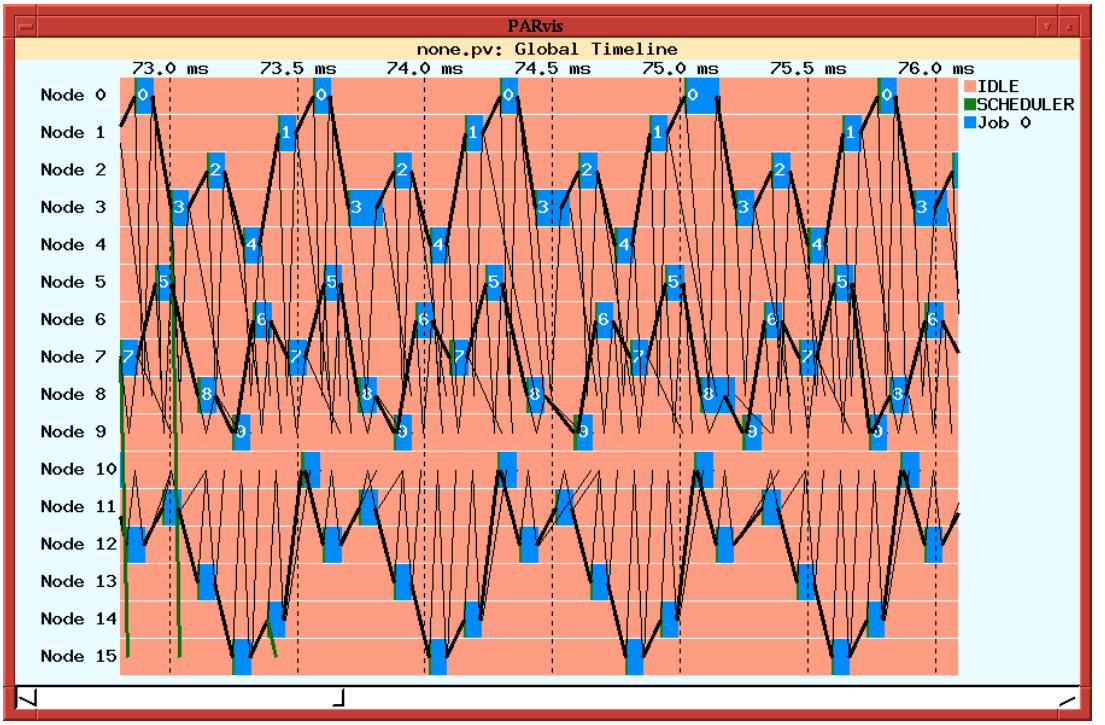


Abbildung 6.15
false sharing in der Matrixmultiplikation ohne Eingriffe: Anhand der dickeren Linien, die Seitentransfers bezeichnen, ist zu erkennen, welche Prozessoren miteinander eine Seite teilen müssen

ca. 170 ms ergibt. Daß bereits ohne Eingriffe die Laufzeit nicht bedeutend höher als dieser Wert ist, liegt an einer ,,automatischen" Synchronisation durch die Netzwerklatenzen: Im Zyklus des Herumreichens einer Seite werden bestimmte Prozessoren gegenüber anderen Prozessoren bevorteilt, da der Weg im Netzwerk vom Seiten-Manager zu ihnen länger ist als zu den anderen Prozessoren. Dies bedeutet, daß die Nachricht zum Weiterschicken der Seite an den nächsten Prozessor länger unterwegs ist, und während dieser Zeitspanne hat ein Prozessor Zeit, mit der Seite zu arbeiten. Auf diese Weise wandern die Adressen innerhalb der Seite, auf denen die Prozessoren arbeiten, immer weiter auseinander, und irgendwann ist ein Punkt erreicht, an dem der ,,schnellste" Prozessor die fragliche Seite verläßt; er kann auf der nächsten Seite solange unbehindert rechnen, bis der nächste Prozessor mit seinen Zugriffen auf diese neue Seite wechselt (siehe Abbildung 6.16).

In dieser Phase ist der erreichbare Parallelitätsgrad um eins höher, und der Zyklus, in dem die restlichen Prozessoren die Seite erhalten, verkürzt sich ebenfalls. Durch das zeitweise Anhalten einzelner Threads wird dieser Effekt natürlich noch unterstützt, so daß sich für die Laufzeiten mit Eingriff z.T. niedrigere Werte ergeben

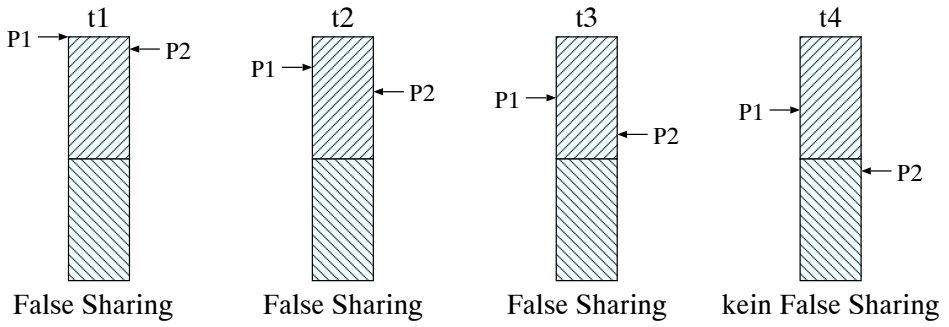


Abbildung 6.16
Beendigung von *false sharing* durch unterschiedliche Geschwindigkeiten in der Bearbeitung

als der genannte Wert von 170 ms.

6.4.1.1 Variante mit zentraler Steuerung

Abbildung 6.17 zeigt die sich beim Einsatz des Algorithmus mit zentraler Steuerung (Abschnitt 4.2.3.1) und fixen Anhaltezeiten ergebenden Programmlaufzeiten.

Unabhängig von der ,,Reaktionsgeschwindigkeit" des Eingriffsmechanismus zeigen alle Kurven einen qualitativ gleichen Verlauf: Mit wachsender Anhaltezeit fällt die Laufzeit zunächst steiler ab, um bei Werten um 6..8 ms ihren Minimalwert zu erhalten, und steigt danach annähernd linear wieder an. Es ist aber auch zu erkennen, daß bei einer schnelleren Erkennung von *Sharing*-Situationen die Minimalwerte der Laufzeit wesentlich schneller erreicht werden; bei einer langsameren Erkennung ist der Bereich minimaler Laufzeit schmaler, und der Verlauf bis zu diesem Punkt ist deutlich unruhiger. Berücksichtigt man, daß ein Prozessor zur Berechnung einer vollständigen Zeile im Zielfeld ca. 9 ms benötigt (dies läßt sich näherungsweise aus der Laufzeit der Variante mit Block-Scheduling ableiten), so wird verständlich, daß das Minimum der Laufzeit in diesem Bereich liegt: die Anhaltezeit der anderen Threads reicht dem übrigbleibenden Thread gerade aus, seine Berechnungen und Zugriffe auf dieser Seite abzuarbeiten. Daß dieses Minimum bereits etwas vor der oben genannten Zeit zur Bearbeitung einer Zeile erreicht wird, hat folgende Gründe:

- Die Erkennung greift erst, wenn die *Sharing*-Situation bereits für einen gewissen Zeitraum bestanden hat. Während dieser Zeit konnten die Threads bereits einen Teil ihrer Zugriffe durchführen, so daß die benötigte Restzeit niedriger ausfällt als der oben genannte Wert.

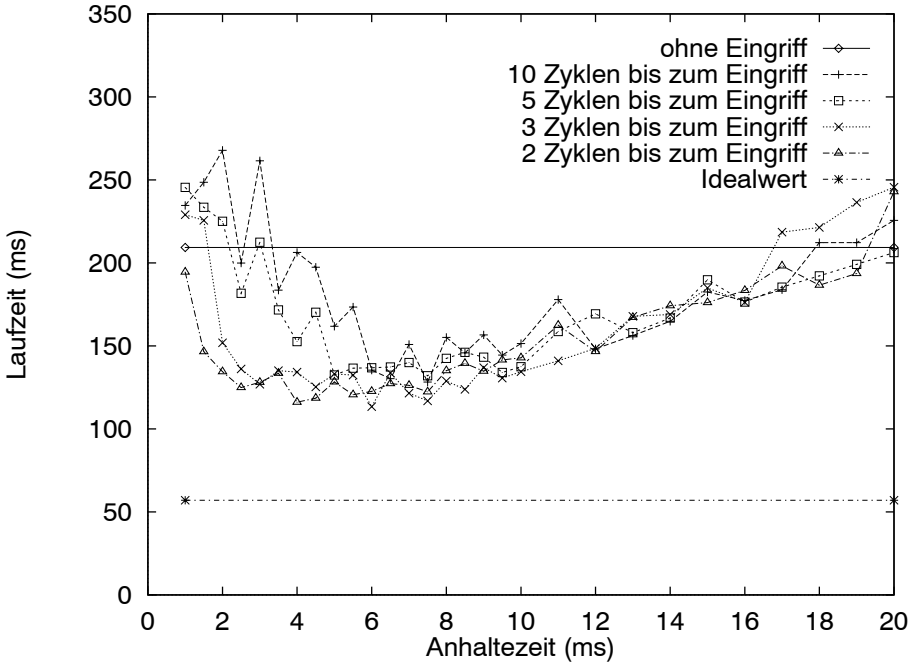


Abbildung 6.17
Laufzeiten mit Eingriff bei *false sharing*

- Da das Anhalten der $n-1$ anderen Threads zeitlich nach der Erkennung des nicht anzuhaltenden Threads erfolgt, laufen diese entsprechend zeitversetzt auch wieder an, was dem ersten Thread etwas mehr Zeit für seine Zugriffe verschafft, als durch die Anhaltezeit eigentlich gegeben ist.

Ist die Anhaltezeit zu niedrig gewählt, so kann der nicht angehaltene Thread seine Zugriffe nicht ungestört zu Ende führen, und die *Sharing*-Situation stellt sich erneut ein; ist die Zeit dagegen zu lang, so werden Threads angehalten, die bereits hätten weiterarbeiten können, und die Laufzeit verlängert sich unnötig (Anstieg der Kurven bei Werten über ca. 10 ms).

Daß Varianten mit längerer Reaktionszeit im Bereich zwischen 2 und 5 ms Anhaltezeit deutlich schlechtere Ergebnisse erbringen, jedoch bei Werten darüber keinen nennenswerten Unterschied zu Varianten mit kürzerer Reaktionszeit zeigen, kann darauf zurückgeführt werden, daß in dem Bereich von Anhaltezeiten zwischen 2 und 5 ms nach dem Wiederanlauf weiter *false sharing* auftritt. Wenn n angibt, wie oft eingegriffen wurde, und k die Anzahl Seitenfehler, ab der Maßnahmen ausgelöst werden, so muß jeder Thread $k * n$ Seitenfehler erleiden, d.h. der Programmablauf wird stärker verzögert. Das relative Verhältnis der auftretenden Seitenfehler ist also zwar nur abhängig von k, bei größerem n steigen aber die absoluten Differenzen.

Auch wenn die beschriebenen Maßnahmen eine merkliche Verringerung der Laufzeit erbringen, so ist deren Senkung nicht als primäres Ziel dieser Strategie zu sehen; durch das Anhalten einzelner Threads werden die Threads in ihrem Programmfortschritt so verzögert, daß sich erhebliche Wartezeiten an Synchronisationsstellen ergeben (die bei einer einfachen Matrixmultiplikation nicht vorhanden sind). Entscheidender ist die Reduzierung der auftretenden Seitenfehler. Abbildung 6.18 zeigt die Seitenfehlerzahlen in Abhängigkeit von der Geschwindigkeit der Erkennung und der Anhaltezeit.

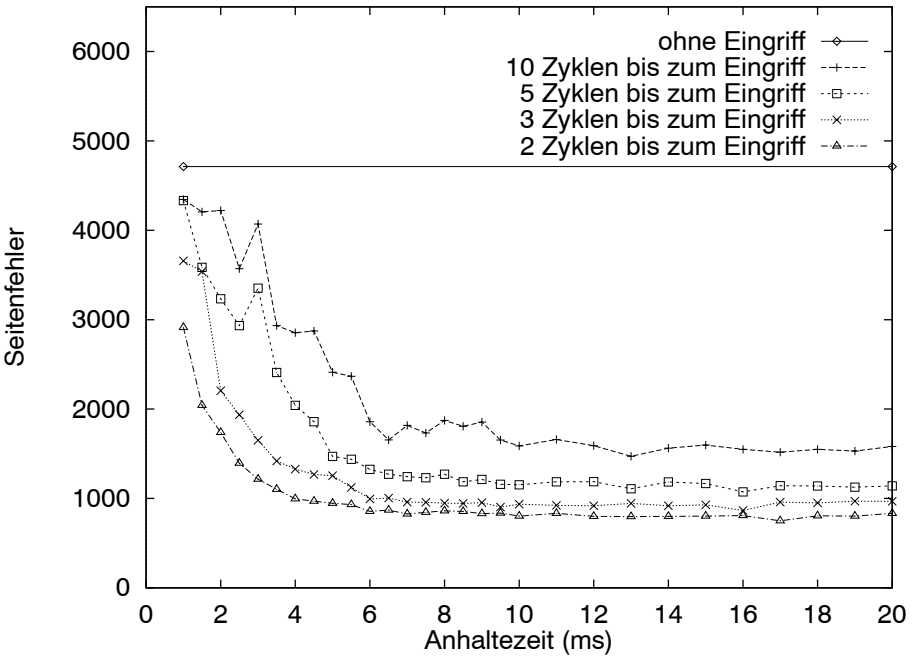


Abbildung 6.18
Seitenfehlerzahlen mit Eingriff bei *false sharing*

Es zeigt sich eine kontinuierliche Verringerung der Seitenfehler in Abhängigkeit von der Anhaltezeit, wobei bei schnellerer Reaktion naturgemäß weniger Seitenfehler auftreten: Es muß weniger *false sharing* stattfinden, bis der Mechanismus greift. Ein schnelleres Eingreifen führt gleichzeitig zu einem kontinuierlicheren Verlauf. Für Zeiten ab ca. 6 ms stellt sich keine wesentliche Verbesserung mehr ein. Da in der Matrixmultiplikation nicht nur Schreibfehler auf das Zielfeld auftreten, sondern auch Lesefehler auf die beiden Quellfelder, kann die Zahl der Seitenfehler nicht unter 600 sinken. Diese Zahl berechnet sich aus der Gesamtseitenzahl der Quellfelder (40), multipliziert mit der Zahl der Prozessoren minus eins (ein Prozessor besitzt eine Seite nach der Initialisierung und muß sie deswegen nicht laden).

Der Vollständigkeit halber sollen an dieser Stelle auch die sich ergebenden Durchschnittszeiten zur Seitenfehlerauflösung gezeigt werden (Abbildung 6.19). Für sehr

lange Anhaltezeiten wird dieser Wert von den in der Initialisierung auftretenden Lese-Seitenfehlern dominiert. Interessant ist lediglich der Zwischenbereich, in dem der Durchschnittswert deutlich niedriger ist: In diesem Bereich treten noch nennenswert Seitenfehler durch *false sharing* auf, die Entzerrung durch das zeitweise Anhalten von Threads beginnt jedoch bereits zu greifen, so daß die noch auftretenden Fehler weniger durch Netzwerkblockierungen behindert werden.

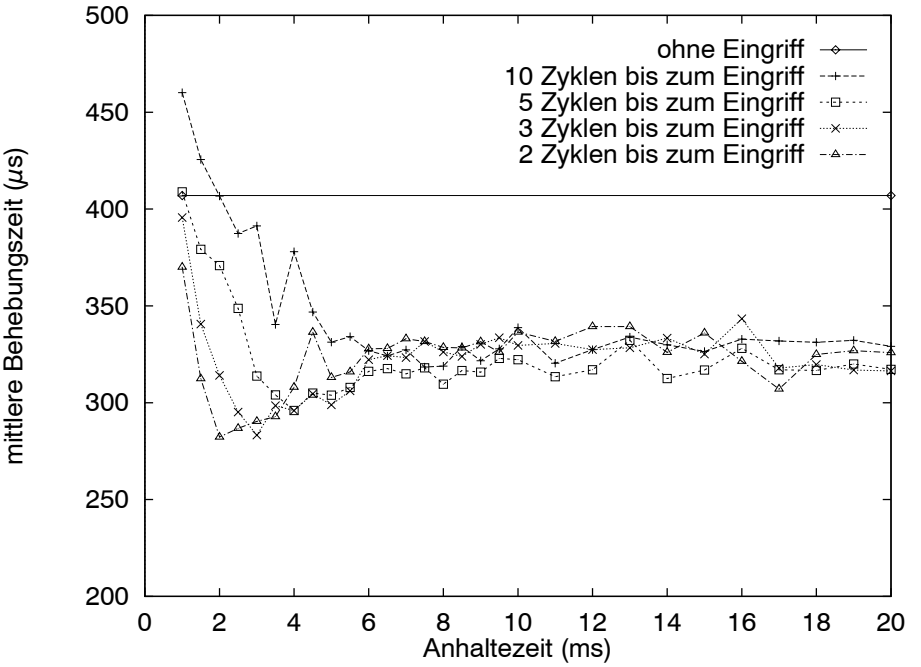


Abbildung 6.19
Seitenfehlerbehebungszeiten mit Eingriff bei *false sharing*

Die bisher in diesem Abschnitt gezeigten Ergebnisse beziehen sich auf ein *dediziert* laufendes, paralleles Programm, d.h. es existieren keine sequentiellen Hintergrundlasten. Nimmt man diese wie im vorigen Abschnitt dazu, so ändern sich die Ergebnisse für Laufzeit und durchschnittliche Seitenfehlerbehebungszeit deutlich (Abbildungen 6.20, 6.21 und 6.22):

Das Laufzeitverhalten zeigt zwar grundsätzlich noch das vom dedizierten Betrieb her bekannte Bild eines starken Abfalls und eines kontinuierlichen Anstieges danach, ist aber von einer Reihe von extremen Spitzen überlagert. Bei den Konstellationen, an denen die Spitzen auftreten, kann die Programmausführung entweder durch zu langes Anhalten oder erhöhtes *false sharing* verlängert worden sein. Ein Blick auf die Seitenfehlerzahlen zeigt, daß diese zwar zeitweilig ansteigen, aber nicht in einem Maße, das den Anstieg der Laufzeit erklären würde. Es kann also nur sein, daß einzelne Seitenfehler extrem langsam aufgelöst wurden. Eine Analyse eines solchen

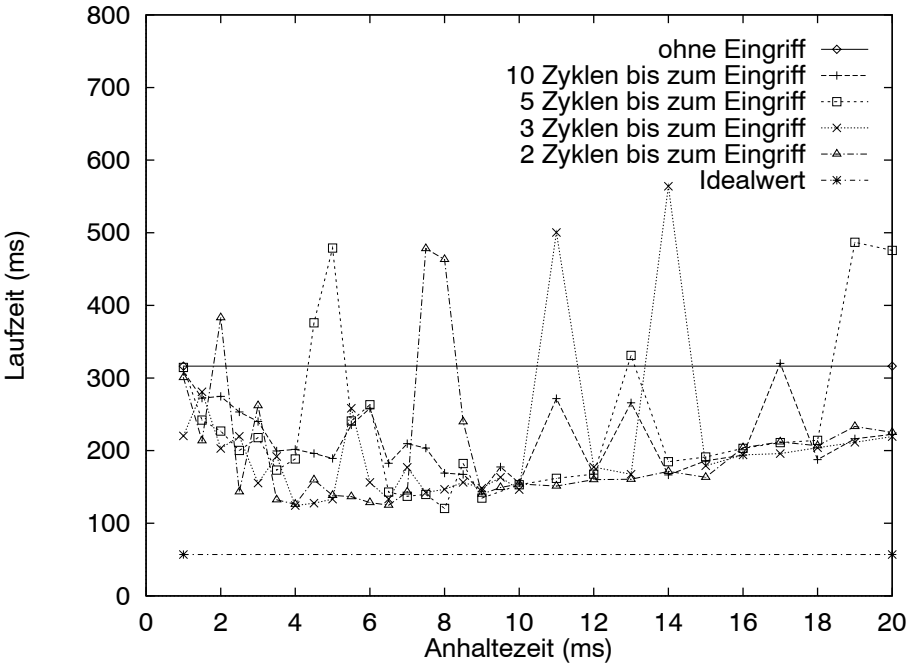


Abbildung 6.20
Laufzeiten mit Hintergrundlast und Eingriff bei *false sharing*

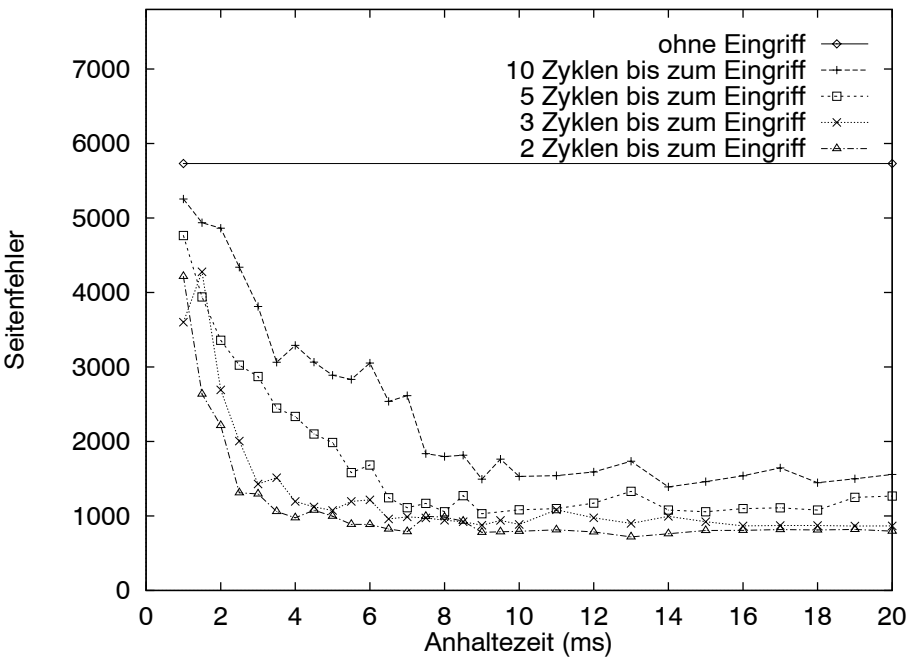


Abbildung 6.21
Seitenfehlerzahlen mit Hintergrundlast und Eingriff bei *false sharing*

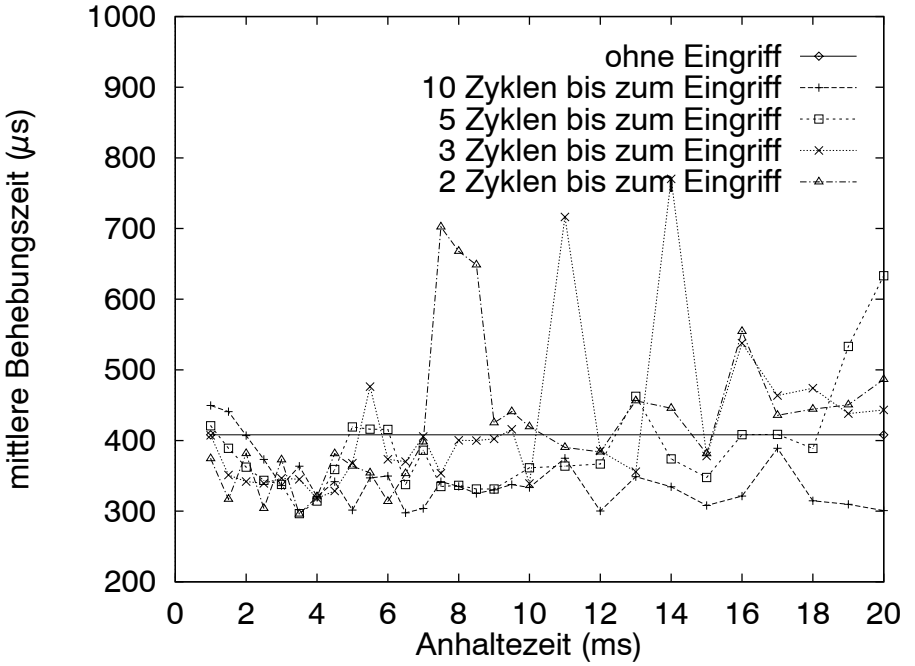


Abbildung 6.22
Seitenfehlerbehebungszeiten mit Hintergrundlast und Eingriff bei *false sharing*

„Ausreißers" mittels PARvis zeigt die Ursache (Abbildung 6.23): Zwei Prozessoren geraten in eine *False-Sharing*-Situation und erkennen diese nach der vorgegebenen Zahl von Zyklen auch. Die Anweisung des globalen Schedulers (auf Prozessor 15), einen Thread anzuhalten, trifft aber in einem Moment ein, in dem gerade einer der beiden konkurrierenden Threads wieder aktiv geschaltet wird, da er die Seite erhalten hat (im Bild auf Prozessor 9). Um dem Thread einen ungestörten Zugriff auf die Seite zu ermöglichen, muß diese für einen Zugriff verriegelt werden (siehe Abschnitt 5.3). Da der Thread jedoch durch die Anweisung des globalen Schedulers kurz danach wieder suspendiert wird, führt er diesen Zugriff nicht aus und die Verriegelung wird *nicht* entfernt. Der um die Seite konkurrierende Thread auf Prozessor 10 kann dadurch die angeforderte Seite erst erhalten, wenn die Anhaltezeit auf Prozessor 9 abgelaufen ist. Nach Ablauf dieser Zeit sind wieder beide Threads rechenbereit, und das *false sharing* beginnt von neuem. Im Ergebnis wird der Programmablauf ohne positiven Effekt verzögert, und ein Prozessor sieht wiederholt Seitenfehlerbehebungszeiten in der Größenordnung der Anhaltezeit, was die Durchschnittszeiten in die Höhe treibt. Die Lösung für dieses Problem ist, daß beim Anhalten eines Threads alle für ihn vorgenommenen Seitenverriegelungen wieder entfernt werden *müssen*. Auf diese Weise kann die Seite zu dem Thread wandern, der vom globalen Scheduler nicht angehalten wurde und die Strategie arbeitet wieder mit Erfolg, wie die Bilder 6.24, 6.26 und 6.25 zeigen. Die Laufzeiten erreichen wieder die gleiche

Größenordnung wie im dedizierten Betrieb, d.h. die Hintergrundprogramme wirken sich nicht störend aus.

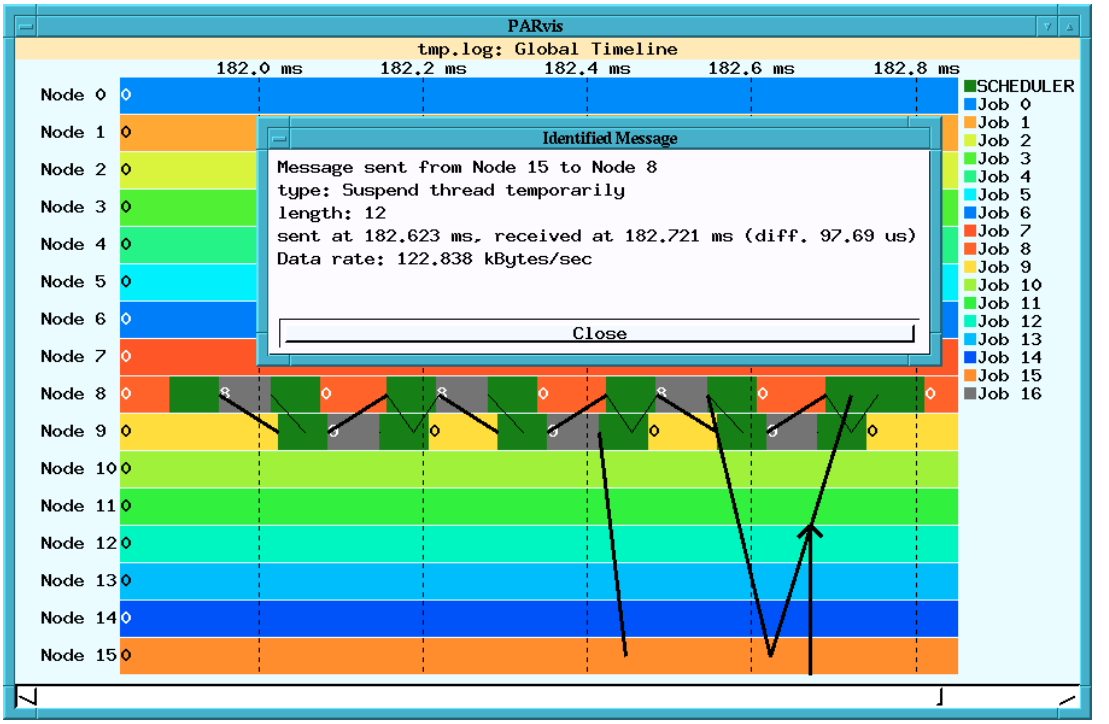


Abbildung 6.23
Verzögerung des Programmablaufes durch Nichtfreigabe von Seiten: Der schwarze Pfeil kennzeichnet die kritische Nachricht, die den anzuhaltenden Thread kurz nach Seitenempfang erreicht.

Die eben geschilderte Situation hätte natürlich auch ohne weiteres im dedizierten Betrieb ohne Hintergrundlast auftreten können; daß dies nicht der Fall war, ist darin begründet, daß beim Vorhandensein von Hintergrundlasten Kontextumschaltungen zwischen verschiedenen Programmen auftreten, die als zeitaufwendiger angenommen wurden als das bloße Reaktivieren des einzigen vorhandenen Threads aus einem Idle-Zustand heraus; die Wahrscheinlichkeit, daß eine Anhaltenachricht während der Rückschaltung auftritt, steigt mit deren Länge.

6.4.1.2 Variante ohne zentrale Steuerung

Abbildungen 6.27 bzw. 6.28 zeigen die mit der Variante ohne zentrale Steuerung erreichten Laufzeiten bzw. Seitenfehlerzahlen für die Matrixmultiplikation. Als Erkennungschwelle für *sharing*-Situationen wurden hierbei drei aufeinanderfolgende Seitenfehler benutzt; dieser Wert hat sich bei der zentralen Variante als sinnvoller

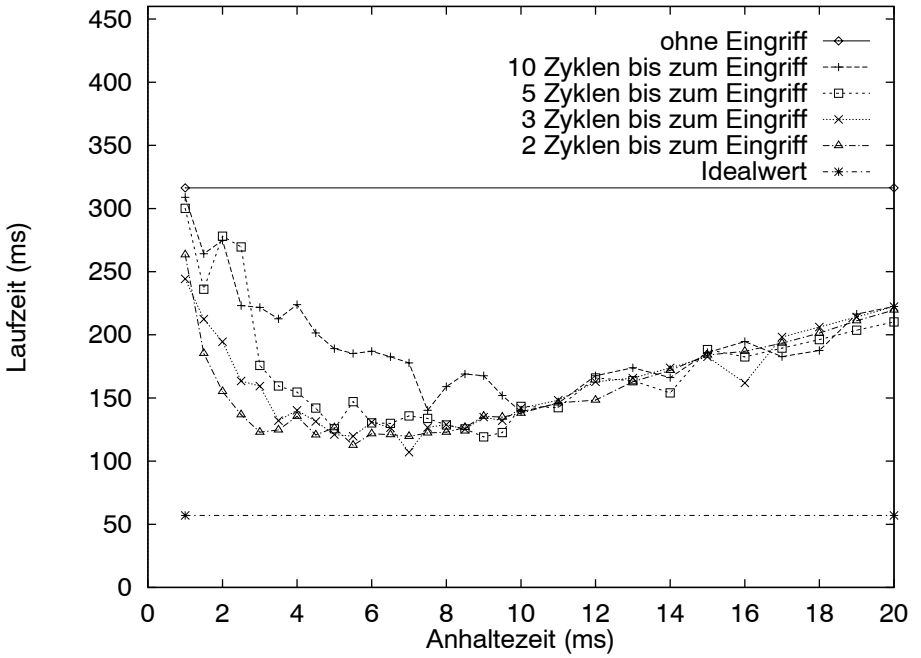


Abbildung 6.24
Laufzeiten mit Hintergrundlast und Eingriff bei *false sharing*-korrigierte Variante

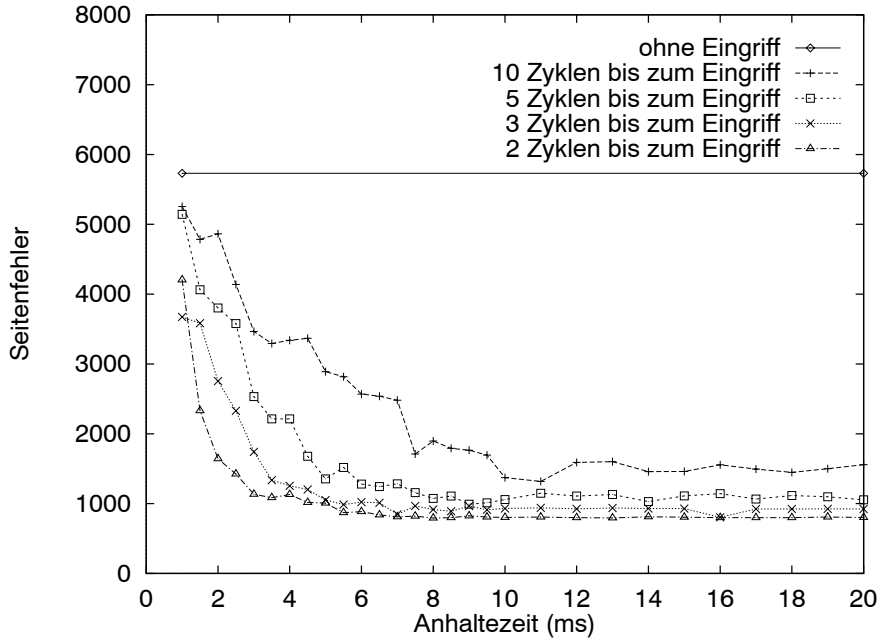


Abbildung 6.25
Seitenfehlerzahlen mit Hintergrundlast und Eingriff bei *false sharing*-korrigierte Variante

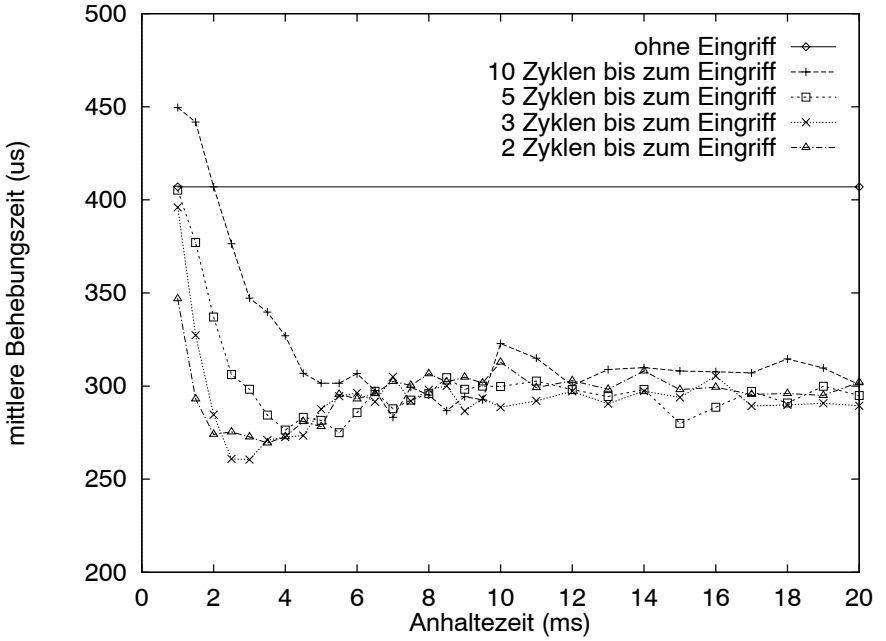


Abbildung 6.26
Seitenfehlerbehebungszeiten mit Hintergrundlast und Eingriff bei *false sharing*-korrigierte Variante

Kompromiß zwischen genügend schneller Erkennung und hinreichendem Sicherheitsabstand gegenüber Fehlerkennungen ergeben. Bei dem variierten Parameter handelt es sich um die *Slot-Zeit*, d.h. die Einheit, in der alle möglichen Anhaltezeiten gequantelt sind. Sie stellt also gleichzeitig auch die minimal mögliche Anhaltezeit dar.

Es ist sofort erkennbar, daß auch geringe Variationen der Slot-Zeit drastische Veränderungen in der Laufzeit nach sich ziehen; die zufällige Komponente des CSMA/CD-Verfahrens führt also ähnlich wie bei Ethernet zu einer schlechten Voraussagbarkeit von Ausführungszeiten. Es werden aber in keinem Fall Laufzeiten wie bei einer gut eingestellten zentralen Steuerung erreicht. Ein generelles Problem einer dezentralen Steuerung ist nämlich, daß bei Konkurrenzsituationen zwischen n Prozessoren, die $n-1$ Anhalteschritte erfordern, die dezentrale Variante die Anhaltezeiten unnötig anhebt, da die lokale Steuerung nicht überblickt, daß mehr als ein weiterer Mitbewerber existiert. Dies führt dazu, daß Prozessoren unnötig angehalten werden, die Seitenfehlerzahlen aber nicht ansteigen, was auch Abbildung 6.28 zeigt. Neben den starken Schwankungen ist noch erkennbar, daß für zwei Bereiche der Slot-Zeit Minima erreicht werden: zum einen für sehr kleine Zeiten, bei denen der exponentielle *Backoff*-Algorithmus Gelegenheit hat, sich an die von den Prozessoren benötigte Zeit anzupassen, zum anderen bei ca. 6 ms, wo bereits die einfache Slot-Zeit ausreicht. Im Zwischenbereich reicht der erste Schritt nicht aus, und der zweite

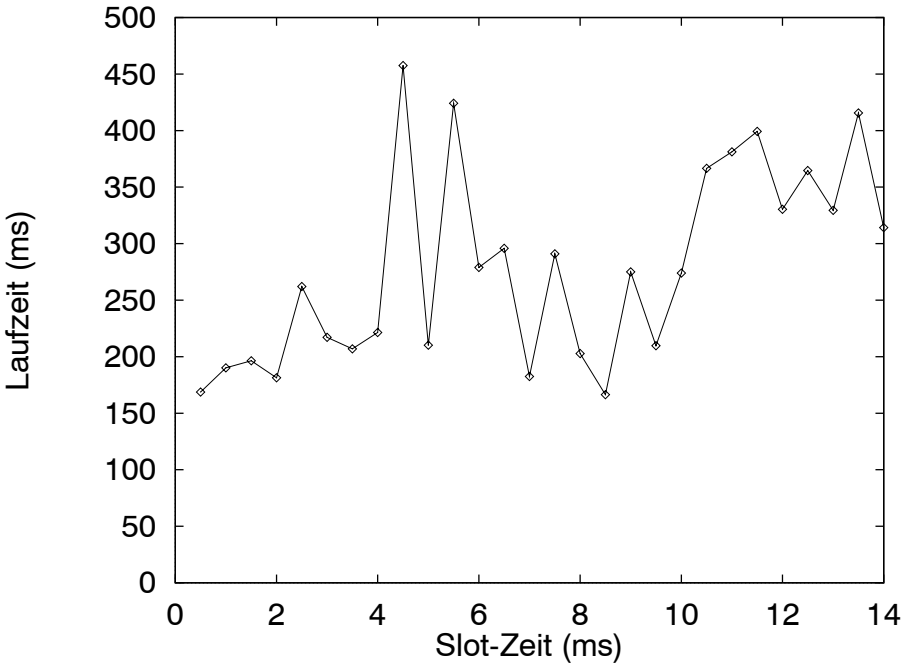


Abbildung 6.27
Laufzeiten mit Hintergrundlast und dezentralem Eingriff bei *false sharing*

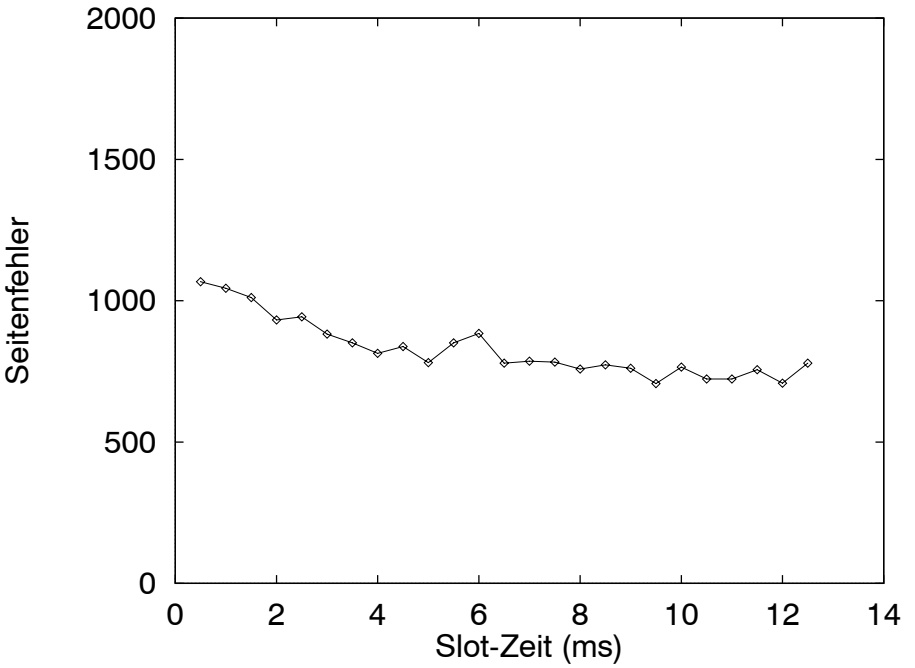


Abbildung 6.28
Seitenfehlerzahlen mit Hintergrundlast und dezentralem Eingriff bei *false sharing*

Schritt ist bereits zu lang; im Bereich jenseits 6 ms ist bereits der erste Schritt zu lang.

Die Seitenfehlerzahlen sehen deutlich günstiger als das Laufzeitverhalten aus: es werden etwa die gleichen Werte wie bei der Variante mit zentraler Steuerung erreicht, und die Werte für kleine Slot-Zeiten sind niedriger als bei der zentralen Steuerung mit festen Anhaltezeiten. Dies ist der Tatsache zuzuschreiben, daß bei kleinen Slot-Zeiten zwar auch erneutes *sharing* möglich ist, die dezentrale Variante aber dann automatisch längere Zeiten einstellt.

Insgesamt kann gesagt werden, daß auch mit einer dezentralen Steuerung eine Reduktion der Seitenfehlerzahlen möglich ist, die Programmlaufzeiten jedoch sehr empfindlich mit der Wahl der Parameter schwanken. Da der zusätzliche Kommunikationsaufwand für eine zentrale Steuerung gering ist (es kommen nur kurze Nachrichten zum Einsatz, die um zwei Größenordnungen kürzer sind als ein Seitentransfer), und die dafür erforderliche Verwaltung ähnlich wie das SVM-Seitenmanagement verteilbar ist, wird im folgenden nur noch die Variante mit zentraler Steuerung betrachtet.

6.4.2 LU-Zerlegung

Nach der Matrixmultiplikation soll an dieser Stelle ebenfalls das Verhalten der LU-Zerlegung mit Eingriffen bei *false sharing* beschrieben werden. Diese zeigt bei der Ausführung mit sequentiellen Hintergrundlasten und Kontextumschaltzeiten über einer bestimmten Grenze deutliches *false sharing*, was die Laufzeit einzelner Iterationen auf ein Mehrfaches ansteigen läßt. Im Gegensatz zur Matrixmultiplikation treten die Seitenfehler hier allerdings nur zwischen zwei Prozessoren auf, und die Synchronisation nach jeder Iteration verhindert ein Auseinanderlaufen der Speicherzugriffe, so daß das *false sharing* bei jeder Iteration erneut auftritt. Aufgrund der geringeren Zahl an Rechenoperationen pro Zugriff auf den gemeinsamen Speicher ist zu erwarten, daß in diesem Falle kürzere Anhaltezeiten ausreichend sind, weshalb der untersuchte und in Abbildung 6.29 dargestelle Bereich der Anhaltezeiten gegenüber der Matrixmultiplikation verschoben ist. Als Kontextumschaltzeit wurde eine Zeit von 100 μs gewählt, bei dem sich der Einfluß des *false sharing* bereits deutlich auf die Laufzeit auswirkt (Abbildung 6.14).

Die Abhängigkeit der Laufzeit von der Anhaltezeit zeigt qualitativ den gleichen Verlauf wie bei der Matrixmultiplikation, allerdings wird das Minimum bereits bei einer niedrigeren Anhaltezeit erreicht. Die Abbildungen 6.30 bis 6.31 zeigen für eine zu kurze, optimale oder zu lange Anhaltezeit die sich ergebenden Ausführungszeiten pro Iteration.

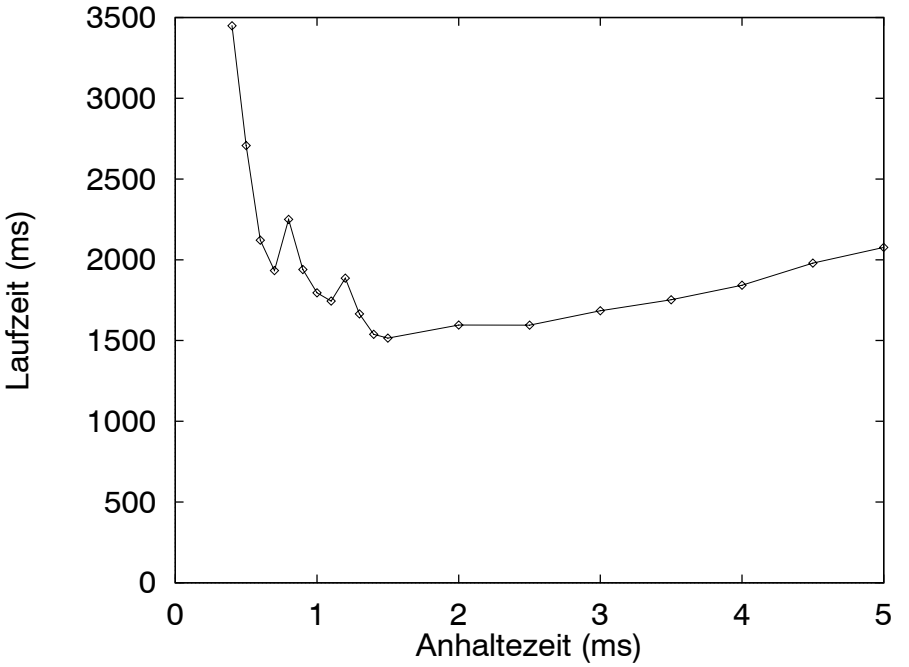


Abbildung 6.29
Laufzeit der LU-Zerlegung bei fixer Anhaltezeit

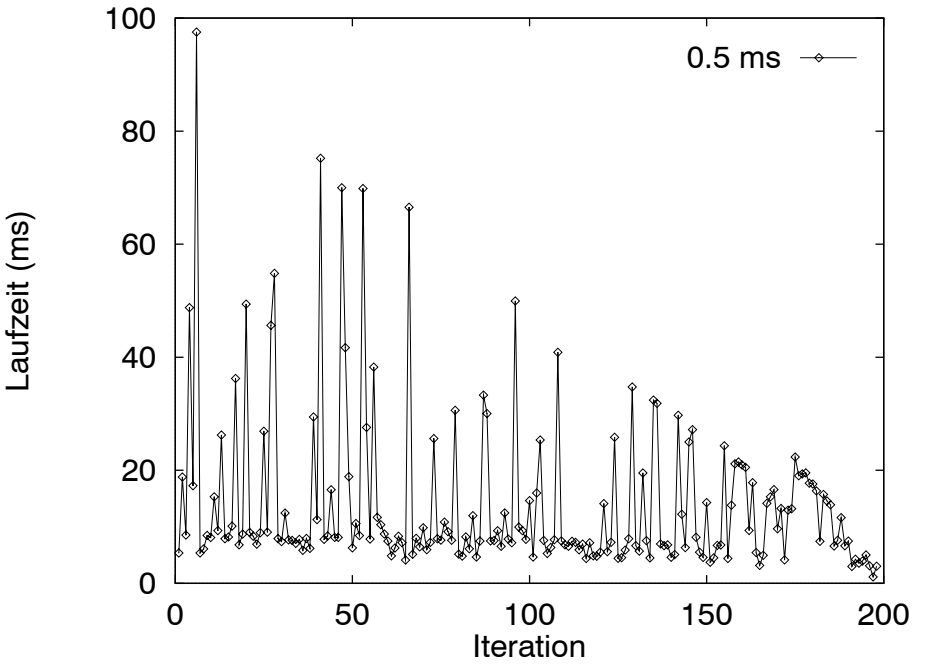


Abbildung 6.30
Schleifenlaufzeit der LU-Zerlegung - zu kurze Anhaltezeit

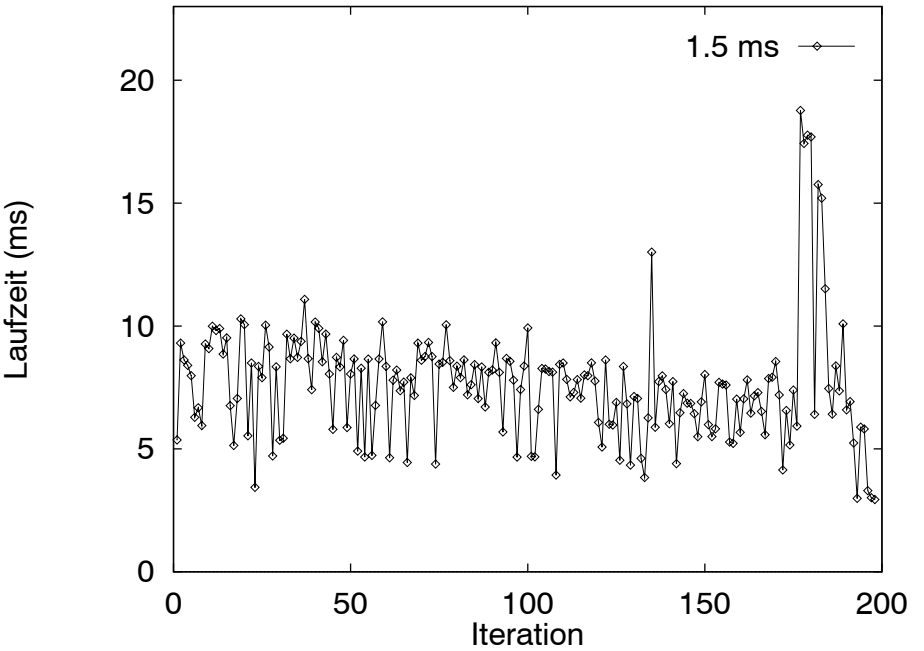


Abbildung 6.31
Schleifenlaufzeit der LU-Zerlegung - optimale Anhaltezeit

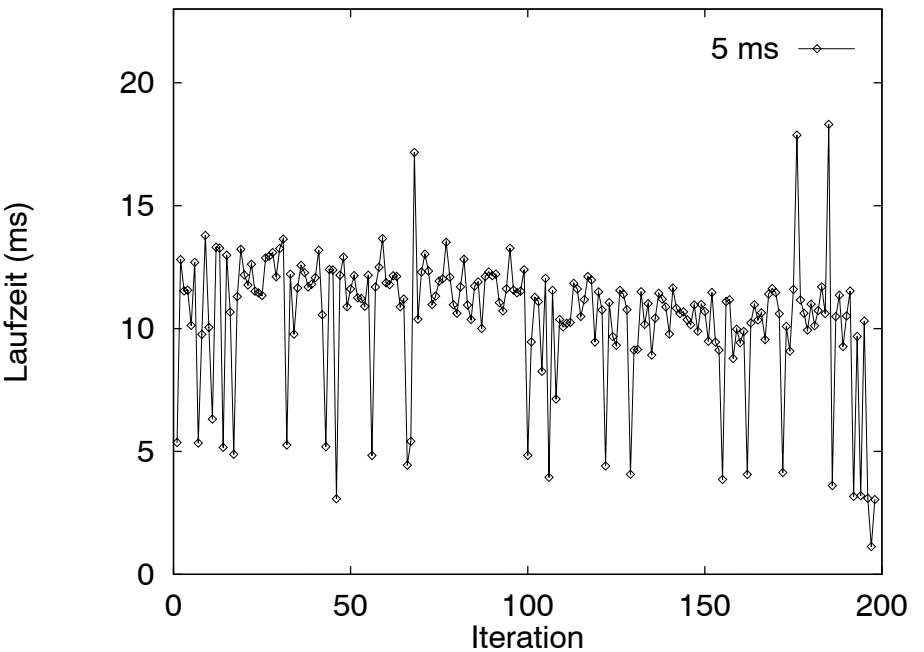


Abbildung 6.32
Schleifenlaufzeit der LU-Zerlegung - zu lange Anhaltezeit

In Abbildung 6.30 ist der Fall einer zu kurzen Anhaltezeit (0.5 ms) dargestellt. Es ergibt sich weiterhin das von der Variante ohne Beeinflussung her bekannte Bild einiger Iterationen, deren Laufzeit durch *false sharing* enorm ansteigt. Die Höhe der Spitzen und damit die Gesamtlaufzeit hat zwar abgenommen, die Anhaltezeiten waren aber nicht lang genug, um einem Prozessor Gelegenheit zu geben, seine Zugriffe auf eine Speicherseite durchzuführen. Im Vergleich dazu ergibt der aus Abbildung 6.29 ablesbare Idealwert von 1,5 ms eine nahezu vollständige Unterdrückung aller Spitzen. Lediglich in der Endphase der LU-Zerlegung um die 180. Iteration ergeben sich noch Spitzen. Demgegenüber zeigt Abbildung 6.32 den Effekt einer zu langen Anhaltezeit (5 ms): Alle Iterationen werden gegenüber dem Idealwert um eine bestimmte Dauer verlängert, die durch den Zeitraum bestimmt ist, in dem Prozessoren noch angehalten sind, obwohl die von ihnen benötigte Seite zur Benutzung frei wäre.

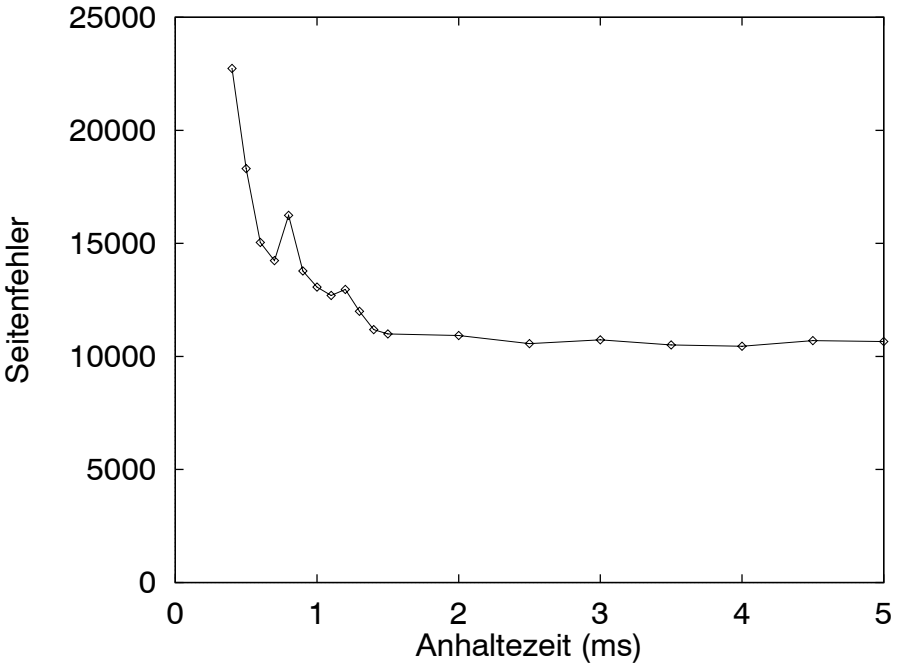


Abbildung 6.33
Seitenfehlerzahlen der LU-Zerlegung bei fixer Anhaltezeit

Abbildung 6.33 zeigt die Anzahl der von der LU-Zerlegung erzeugten Seitenfehler in Abhängigkeit von der Anhaltezeit. Im Unterschied zur Matrixmultiplikation ist der algorithmisch bedingte und nicht durch zeitweises Anhalten entfernbare Anteil deutlich höher und macht fast die Hälfte aller Seitenfehler aus; es ist aber immer noch erkennbar, daß Anhaltezeiten oberhalb eines für die Laufzeit idealen Wertes keine nennenswerte Reduzierung der Seitenfehlerzahlen mehr erbringen.

6.4.3 Adaptive Festlegung von Anhaltezeiten

Wie aus diesen Ergebnissen deutlich wird, kann das zeitweise Anhalten von Threads in *Sharing*-Situationen durchaus zur Verringerung der Laufzeit eines einzelnen Programmes führen und die Netzwerkbelastung signifikant reduzieren, es wurde jedoch auch klar, daß die Wahl einer günstigen Anhaltezeit entscheidend für den Erfolg einer solchen Strategie ist. Die optimale Zeit ist jedoch abhängig von der Charakteristik der aktuellen Datenzugriffsmuster, so daß die Wahl eines festen Wertes als in der Praxis wenig erfolgversprechend angesehen werden muß. Bereits an dem einfachen Programmbeispiel der Matrixmultiplikation (Abbildung 5.3) wird dies erkennbar: Innerhalb der bisher nicht betrachteten Initialisierung der Quellfelder treten ebenfalls *False-Sharing*-Effekte auf, die in der nicht auf Seitengrenzen ausgerichteten Lage der Zeilen in den Quellfeldern begründet sind. Die Situation wird zwar vom System erkannt und auch mit deutlichem Erfolg behandelt, da die Initialisierung von Zeilen jedoch eine sehr schnelle Operation ist, ist bereits eine Anhaltezeit von vier Millisekunden (die in der eigentlichen Multiplikation noch keine optimalen Ergebnisse erzielt) zu lang und führt zu einer Phase, in der *kein* Prozessor mehr arbeitet (Abbildung 6.34).

Um solche unnötigen Verzögerungen durch zu lange Anhaltezeiten zu vermeiden, wurde in Kapitel 4 bereits eine Variante des zentralisierten Ansatzes vorgeschlagen, die mit einer dynamischen Anpassung der Anhaltezeiten arbeitet. Die Ergebnisse dieser Erweiterung sollen im folgenden vorgestellt werden.

Es ist einsichtig, daß ein anpassungsfähiger Algorithmus seine Vorteile erst bei sich wiederholenden Programmteilen ausspielen kann. Für einen ersten Test wurde daher die Matrixmultiplikation *mehrfach* ausgeführt. Dabei wurde den einzelnen Durchläufen noch eine Schleife vorangeschaltet, die dafür sorgt, daß alle Prozessoren Lesekopien der Quellfelder besitzen und somit der unter dem Gesichtspunkt der ,,Lernfähigkeit" wichtige, erste Durchlauf nicht durch Lesefehler überlagert wird. Als Anzahl der Seitenfehler, nach denen ein Eingriff erfolgt, wurden 3 Seitenfehler gewählt; dieser Wert hat sich in den vorigen Untersuchungen als deutlich effektiver als 5 oder 10 erwiesen, bietet aber trotzdem noch einen gewissen ,,Sicherheitsabstand" gegenüber verfrüht ausgelösten Aktionen. Tabelle 6.1 zeigt die sich für jeden Durchlauf ergebenden Durchlaufszeiten und Seitenfehlerzahlen, Tabelle 6.2 die Verzögerungszeiten, die sich für die einzelnen Seiten eingestellt haben. Es wurde dabei von einer initialen Verzögerung von einer Millisekunde ausgegangen, die in Schritten von einer Millisekunde erhöht werden konnte; da es sich bei dem untersuchten Programm lediglich um einen einzelnen Kern handelt, konnte für diesen Fall auf eine Berücksichtigung des momentan aktiven Programmkonstruktes verzichtet werden.

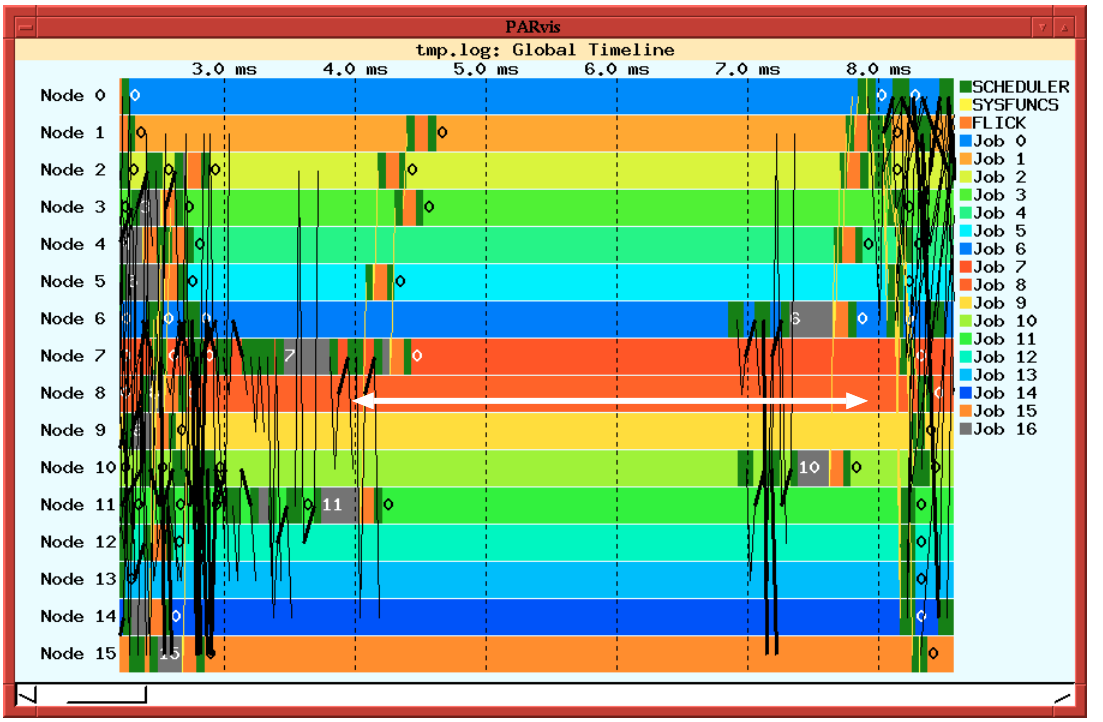


Abbildung 6.34
Unnötige Verzögerung der Initialisierung bei fester Anhaltezeit: der durch den weißen Pfeil gekennzeichnete Zeitbereich, in dem die Anwendung blockiert ist, ist auch an dem deutlich reduzierten Kommunikationsaufkommen zu erkennen (am linken und rechten Rand Bereiche normalen Verhaltens)

Die Zahlen zeigen, daß die Anhaltezeiten sich bereits nach den ersten beiden Iterationen nicht mehr wesentlich verändern und sich damit auch die Ausführungszeiten stabilisieren; der Algorithmus kann sich also zumindest in diesem Beispiel hinreichend schnell auf das Laufzeitverhalten des Programmes einjustieren, so daß bereits für kleine Wiederholungszahlen Verbesserungen zu erzielen sind. Bei den eingestellten Verzögerungszeiten ist zu erkennen, daß für Seiten, die im Zielfeld „weiter vorn liegende" Zeilen beinhalten, längere Verzögerungen eingestellt wurden. Dieser (sinnvolle) Effekt ist darauf zurückzuführen, daß auf diese Zeilen im Programmablauf direkt am Programmanfang zugegriffen wird, d.h. in einer Phase, in der die Zugriffe der einzelnen Prozessoren noch nicht auseinandergelaufen sind. Bei den Seiten, die erst im späteren Verlauf benötigt werden, wird es immer wahrscheinlicher, daß auf sie von den Prozessoren derart zeitversetzt zugegriffen wird, daß entweder gar keine Überlappungen mehr auftreten oder zumindest kürzere Anhaltezeiten ausreichend sind.

Da - wie oben bereits erwähnt - Verzögerungen oder Seitenfehler durch das Lesen

Lauf	Laufzeit	Seitenfehler	Lauf	Laufzeit	Seitenfehler
1	141.3 ms	1110	6	103.9 ms	398
2	107.8 ms	455	7	104.0 ms	407
3	104.0 ms	394	8	104.1 ms	395
4	104.0 ms	404	9	104.1 ms	436
5	104.0 ms	406	10	104.0 ms	402

Tabelle 6.1
Durchlaufzeiten und Seitenfehlerzahlen für adaptive Anhaltezeiten

der Quellfelder nicht enthalten sind, lassen sich diese Zahlen nicht mit den Werten für feste Anhaltezeiten aus vorherigen Abschnitten vergleichen; die hier verwendete Programmvariante wurde deshalb auch einmal mit der nicht-adaptiven Variante und den Parametern ausgeführt, die das beste Ergebnis erzielt hatten (3 Seitenfehler bis zum Ansprechen, 7 ms Anhaltezeit in Abbildung 6.24). Dabei ergaben sich eine Ausführungszeit von 110.7 ms und 551 Seitenfehler; die adaptive Variante erzielt also bereits nach dem ersten Durchlauf bessere Ergebnisse als jede Kombination fixer Parameter.

Zu erkennen ist allerdings auch, daß die Anhaltezeiten sich für einige Seiten nicht stabilisieren (in Tabelle 6.2 fett gesetzt). Dies ist darauf zurückzuführen, daß der adaptive Algorithmus in seiner hier untersuchten Form die Anhaltezeiten lediglich *anheben* kann; er kann z.B. nicht darauf reagieren, daß sich aufgrund einer besseren Anpassung der Anhaltezeit für eine Seite eine kürzere Überlappungszeit auf einer anderen Seite ergibt und die Anhaltezeit für diese Seite wieder reduziert werden könnte; dies kann dazu führen, daß asymptotisch eine Einstellung gewählt wird, die oberhalb des Idealwertes liegen kann. Eine Reduktion würde einen Mechanismus erfordern, der erkennt, ob der nicht angehaltene Prozessor nicht mehr auf die Seite zugreift und dann die Suspendierung der anderen Prozesse vorzeitig abbricht. Ein solcher Mechanismus erfordert einen zusätzlichen Speicherzugriffsmonitor, der in der Lage ist, Zugriffe auf Speicherseiten zu überwachen und bei Veränderung der Zugriffsfrequenz ein Signal abzugeben.

6.5 Strukturelle Einordnung der Ergebnisse

Die in dieser Arbeit vorgestellten Untersuchungen wurden so durchgeführt, daß alle für die Arbeit wichtigen und relevanten Effekte systematisch quantifiziert werden konnten. Um die in Kapitel 4 aufgezeigten Problembereiche vollständig erfassen zu können, hat es ausgereicht, zwei in technisch-wissenschaftlichen Programmen typische Programmkerne (in unterschiedlichen Ausprägungen) zu verwenden. Zur Einordnung der Ergebnisse entsprechend der vorgegebenen Aufgabenstellungen sollen

	Lauf									
Seite	1	2	3	4	5	6	7	8	9	10
40	9	9	9	9	9	9	9	9	9	9
41	6	8	8	8	8	8	8	8	8	8
42	11	11	11	11	11	11	11	11	11	11
43	6	6	6	6	6	6	6	6	6	6
44	5	5	5	5	5	5	5	5	5	5
45	1	1	3	3	4	4	4	5	5	5
46	-	-	-	-	-	-	-	-	-	-
47	4	4	4	4	4	4	4	4	4	4
48	2	5	5	5	5	5	5	5	5	5
49	4	4	4	4	4	4	4	4	4	4
50	3	3	3	3	3	3	3	3	3	3
51	2	3	3	3	3	3	3	3	3	3
52	2	4	5	5	5	6	7	7	7	7
53	5	5	5	5	5	5	5	5	5	5
54	6	6	6	6	6	6	6	6	6	6
55	2	3	3	3	3	3	4	4	5	5
56	3	3	3	3	3	3	3	3	4	4
57	1	1	1	2	3	3	3	3	3	4
58	4	4	4	4	4	4	4	4	4	4
59	3	4	4	4	4	4	4	4	4	4

Tabelle 6.2
Anhaltezeiten (ms) pro Seite je Iteration

in diesem Abschnitt die untersuchten Varianten noch einmal strukturell gegenübergestellt werden. Zur Erläuterung sind in den Abbildungen dieses Abschnitts die Fragestellungen noch einmal orthogonal - d.h. nach dem untersuchten Programmkern - aufgeführt.

Alle Programmkerne wurden sowohl dediziert als auch nicht dediziert, d.h. mit zusätzlichen sequentiellen Hintergrundlasten untersucht. Während die dedizierten Läufe in erster Linie der Untersuchung der grundsätzlichen Eigenschaften eines Kerns und der Festlegung einer für die folgenden Untersuchungen adäquaten Problemgröße dienten, lieferten die nicht-dedizierten Läufe die relevanten Ergebnisse der Arbeit, da sie den angestrebten Multiprogramming-Betrieb in einer Produktionsumgebung auf massiv-parallelen Systemen mit virtuell gemeinsamem Speicher simulieren.

Nach der Bestimmung der Problemgröße haben sich die Untersuchungen zunächst auf die Matrixmultiplikation mit Blockverteilung konzentriert, wobei als

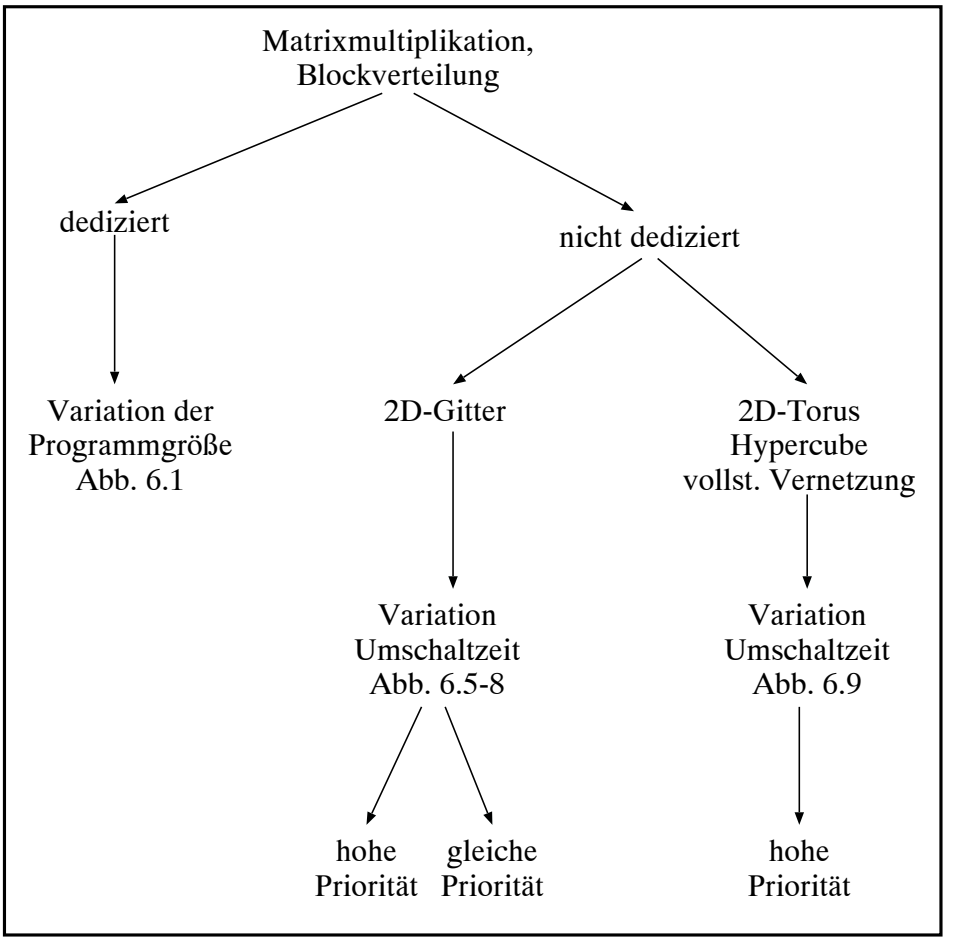


Abbildung 6.35
Durchgeführte Untersuchungen mit der Matrixmultiplikation (Blockverteilung)

Einflußparameter die Kontext-Umschaltzeit, die Nutzung verschiedener Netzwerk-Topologien sowie eine Variation der Prioritätsverhältnisse zwischen paralleler Anwendung und der Hintergrundlast im Detail untersucht wurden.

Für die Matrixmultiplikation mit zyklischer Verteilung (Abbildung 6.36) waren erheblich mehr Untersuchungen nötig, da hier neben der Variation der Kontext-Umschaltzeiten zusätzlich die in dieser Arbeit entwickelten Verfahren zur Reduzierung des *false sharing* durch Senkung des Parallelitätsgrades an Bedeutung gewinnen. Bei diesen Verfahren handelt es sich um Verfahren mit dezentraler bzw. zentraler Steuerung. Das Verfahren mit zentraler Steuerung wurde nicht nur in einer Standardversion mit festen Anhaltezeiten und Lock-Optimierung analysiert, sondern es wurden auch Simulationen ohne Lock-Optimierung und mit adaptiver Anpassung der Anhaltezeiten durchgeführt.

Als weiterer Problemkreis wurde die LU-Zerlegung stellvertretend für eine Grup-

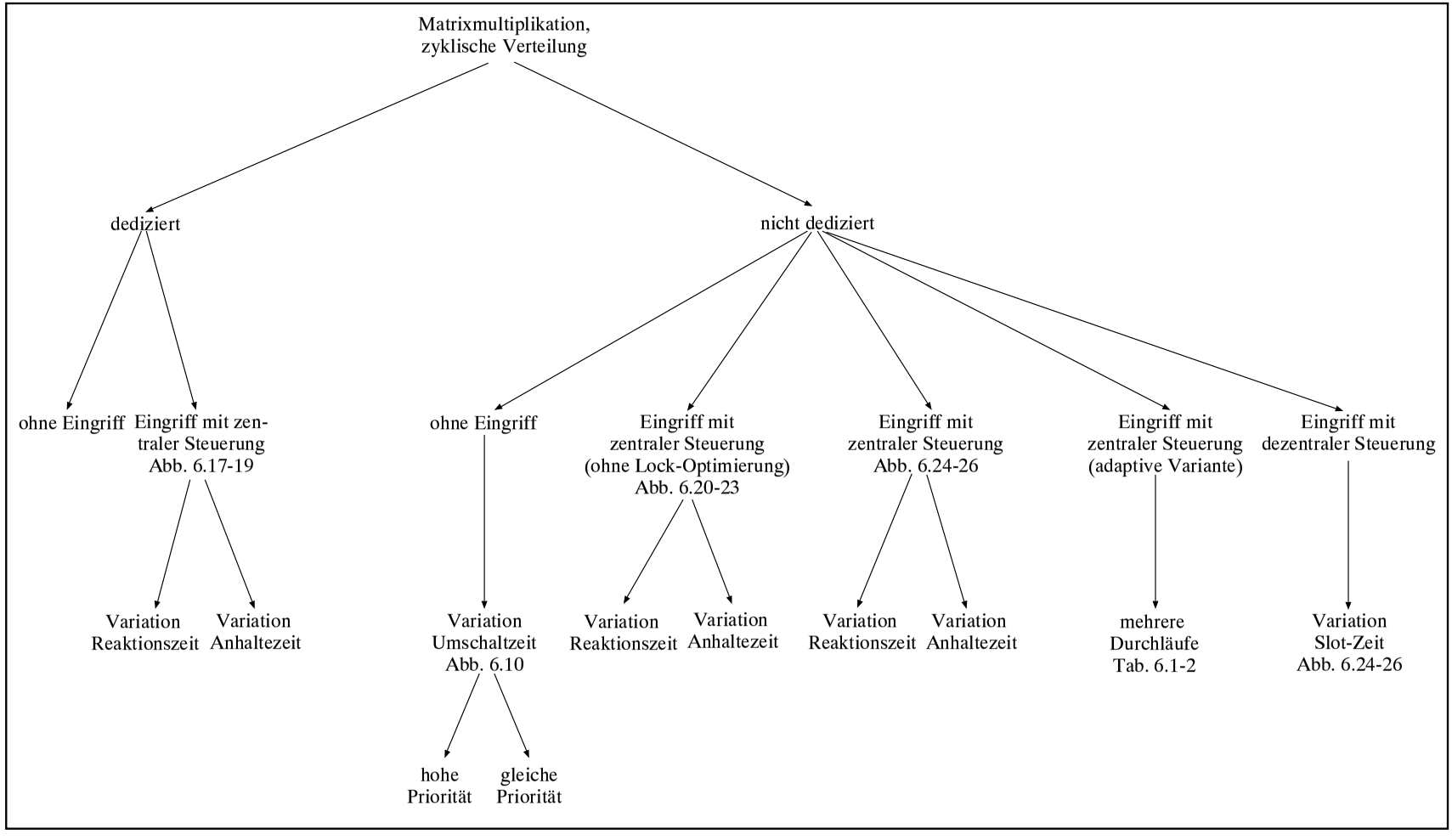


Abbildung 6.36
Durchgeführte Untersuchungen mit der Matrixmultiplikation (zyklische Verteilung)

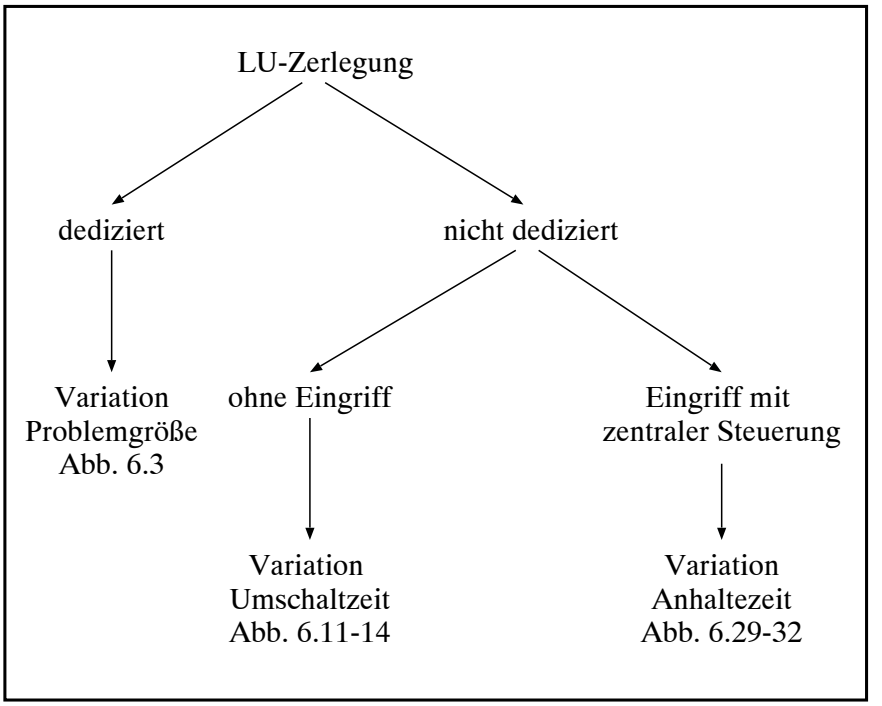


Abbildung 6.37
Durchgeführte Untersuchungen mit der LU-Zerlegung

pe häufig in technisch-wissenschaftlichen Anwendungen auftretender Programmkerne untersucht, deren Rechenaufwand und möglicher Parallelitätsgrad während der Laufzeit stark variiert und die durch die beiden Varianten der Matrixmultiplikation nicht abgedeckt werden konnten (Abbildung 6.37). Dabei wurde nur die Strategie zur Reduzierung von *false sharing* untersucht, die sich aufgrund der vorherigen Untersuchungen als günstigste Variante herausgestellt hatte.

Mit den in diesem Kapitel gemachten Untersuchungen sind die wesentlichen, für das Nutzungsprofil in technisch-wisenschaftlichen Umgebungen typischen Effekte untersucht und Lösungsstrategien entwickelt worden, die unter den beschriebenen Randbedingungen einen akzeptablen Multiprogramming-Betrieb auch auf SVM-Systemen ermöglichen.

Kapitel 7

Zusammenfassung

Massiv-parallele Rechnersysteme mit physikalisch verteiltem Hauptspeicher stellen zwar einen kostengünstigen Weg dar, hohe Rechenleistungen mit niedrigen Realisierungskosten zu erzielen, erfordern aber auf der anderen Seite neue Betriebssystemkonzepte, Programmiermodelle und -sprachen, um praktisch einsetzbar zu sein. Der virtuell gemeinsame Speicher (SVM) ist ein Programmiermodell, daß diesen Anforderungen nach vereinfachter Programmierung von Rechnersystemen mit physikalisch verteiltem Speicher gerecht zu werden versucht. Für einen erfolgreichen Einsatz von SVM sind allerdings Modifikationen von Anwendungs- und Betriebs-Software erforderlich. Während die Optimierung paralleler Anwendungen auf SVM bereits in einer Reihe von Arbeiten behandelt wurde, wurden bisher noch vergleichsweise wenig Anstrengungen unternommen, parallele Betriebssysteme und insbesondere deren Scheduling-System auf eine effiziente Kooperation mit SVM abzustimmen. In der vorliegenden Arbeit wurde daher untersucht, wie sich bestehende, parallele Betriebssysteme um Komponenten erweitern lassen, die eine effizientere Abarbeitung SVM-basierter Anwendungen ermöglichen.

Eine allgemein in parallelen Rechnersystemen wichtige Funktion zur Durchsatzsteigerung ist die Möglichkeit des Multitasking auf den Rechenprozessoren, da dies die Ausnutzung von durch I/O-Operationen verursachten Leerlaufzeiten erlaubt. Die Ergebnisse dieser Arbeit zeigen, daß unkoordiniertes, d.h. lokal initiiertes Umschalten auf andere Prozesse bei Seitenfehlern in der Lage ist, die freiwerdenden Rechenkapazitäten auszunutzen, ohne ein für SVM optimiertes Vordergrundprogramm in seinem Ablauf nennenswert zu behindern. Es müssen aber durch das Scheduling folgende Randbedingungen erfüllt sein:

- Der lokale Speicher eines Prozessors muß groß genug sein, daß sowohl Vordergrund- als auch Hintergrundanwendungen gemeinsam in ihn hineinpassen, so daß beim Umschalten kein Overhead durch Paging-Operationen entsteht.

- Der lokale Scheduler muß eine Umschaltzeit zwischen zwei Prozessen besitzen, die deutlich kleiner ist als die mittlere Zeit zur Auflösung eines Seitenfehlers.
- Ein unterbrochener Vordergrundprozeß muß beim Erhalt einer Seite sofort wieder den Prozessor erhalten; ist dies nicht gegeben, so besteht die Gefahr, daß aufgrund von Seitenverriegelungen der Fortschritt des parallelen Programmes insgesamt behindert wird.

Allgemein erfordert das Multitasking auf parallelen Rechnersystemen aber auch Kontrollinstanzen, die dem Betriebssystem einen Eingriff ermöglichen, sofern Anwendungsprogramme sich in einer Weise verhalten, die den Ablauf anderer Anwendungen negativ beeinflussen kann. Speziell bei SVM-basierten Programmen trifft dies zu, wenn diese durch ungünstige Arbeits- oder Datenverteilung *false* oder *true sharing* auslösen, d.h. einzelne Seiten werden wiederholt zwischen verschiedenen Prozessoren hin- und herkopiert. Das dadurch gesteigerte Datenaufkommen im Verbindungsnetzwerk kann auch negative Auswirkungen auf die Kommunikation anderer, paralleler Anwendungen haben und ist daher nach Möglichkeit zu vermeiden. In dieser Arbeit wurden Verfahren entwickelt und untersucht, mit deren Hilfe ein Betriebssystem derartige Situationen erkennen und behandeln kann. Die Verfahren basieren darauf, daß ab einer vorgegebenen Rate von Fehlern auf eine bestimmte Seite einzelne Threads einer parallelen Anwendung zeitweise angehalten werden. Der nicht angehaltene Thread kann dann ungehindert seine Zugriffe durchführen, und nach Ablauf der Anhaltezeit kommen die anderen Threads wieder zum Zuge.

Das Anhalten einzelner Threads darf aber nicht unkoordiniert erfolgen, da es sonst seine Wirkung nicht entfalten oder ein Programm sogar verlangsamen kann. Es wurden in der Arbeit zwei Verfahren entwickelt, deren Ziel es ist, das Anhalten einzelner Threads so zu steuern, daß im Ergebnis ein Thread nicht angehalten wird und seine Speicherzugriffe ungehindert durchführen kann. Das eine Verfahren benutzt eine zentrale Steuereinheit, die das Anhalten von Threads koordiniert, das andere Verfahren arbeitet vollständig dezentral nach dem Muster der Kollisionserkennung in CSMA/CD-basierten Netzwerkprotokollen. Durch Simulationen von Programmkernen hat sich ergeben, daß beide Verfahren geeignet sind, die Seitenfehlerraten bei geeigneter Parameterwahl drastisch zu reduzieren, die Variante mit zentralisierter Steuerung hat jedoch günstigere und deterministischere Auswirkungen auf die Programmlaufzeiten. Da der zusätzliche Kommunikationsaufwand sich in Grenzen hält und das Verfahren skalierbar ist, ist es als das im allgemeinen günstigere Verfahren anzusehen.

Es hat sich in den Simulationen aber auch erwiesen, daß die zu wählenden Zeitparameter auf von der Anwendung abhängige Werte eingestellt werden müssen, um optimale Ergebnisse zu erhalten. Da die Programmcharakteristiken einem Betriebssystem vor dem Programmstart im allgemeinen nicht bekannt sind, müssen diese

vom Betriebssystem während des Laufes selbständig ermittelt werden. Es wurde dazu eine Erweiterung der zentralisierten Steuerung entwickelt und implementiert, deren grundsätzliche Brauchbarkeit an einem der untersuchten Programmkerne demonstriert werden konnte.

Literaturverzeichnis

[AAO92] V. Abrossimov, F. Armand, M.I. Ortega: *A Distributed Consistency Server for the CHORUS System.*
CS/TR-91-91, Chorus systèmes, March 1992

[Arn93] A. Arnold: *PARvis: Eine X-basierte Umgebung zur Visualisierung von parallelen Programmen in Multiprozessorsystemen.*
Jül-2848, Forschungszentrum Jülich, 1993

[ABNR95] A. Arnold, J. Bernert, W. E. Nagel, M. Röth: *Performance-Analyse paralleler Programme: Die PARvis-Visualisierungsumgebung.*
ITG-Mitteilungen, Pars-Workshop 1995, Nr.14, S.191-200

[BDGM94] A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, V. Sunderam: *A Users Guide to Parallel Virtual Machine.*
MIT Press, 1994

[BeGe95] R. Berrendorf, M. Gerndt: *SVM-Fortran Reference Manual Version 1.4.*
1995

[Bla90] D.L. Black: *Scheduling Support for Concurrency and Parallelism in the Mach Operating System.*
IEEE Computer, No. 5, pp. 35-43, 1990

[BHMW94] D.C. Burger, R.S. Hyder, B.P. Miller, D.A. Wood: *Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors.*
Proc. Supercomputing 1994, Washington D.C., pp. 590-599

[Cr2099] Cray Research: *UNICOS Tuning Guide.*
SR-2099 8.0

[Cr93] Cray Research, *CRAY T3D System Architecture Overview Manual*, 1993, available via WWW: `http://www.cray.com/PUBLIC/product-info/mpp/T3D_Architecture_Over/T3D.overview.html`

[CuTz91] P.-J. Cuang, N.-F. Tzeng: *An Efficient Submesh Allocation Strategy for Mesh Computer Systems.*
Proc. 1991 International Conference on Distributed Computer Systems, pp. 256-263

[FeRu90-1] D.G. Feitelson, L. Rudolph: *Distributed Hierarchical Control for Parallel Processing.*
IEEE Computer, No. 5, pp. 65-76, 1990

[FeRu90-2] D.G. Feitelson, L. Rudolph: *Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control.*
Proc. 1990 International Conference on Parallel Processing, pp. I.1-I.8

[GaJo79] M.R. Garey, D.S. Johnson: *Computers and Intractability: A Guide to the Theory of NP–Completeness.*
Freeman, San Francisco, 1979

[GeBe95] M. Gerndt, R. Berrendorf: *Parallelizing Applications with SVM- Fortran.*
Proc. HPCN Europe 1995, Mailand, 2. - 5. Mai 1995, Lecture Notes in Computer Science

[GrUn95] W. Grünewald, T. Ungerer: *Simulation einer vielfädigen Prozessorarchitektur.*
PARS-Workshop 1995, ITG-Mitteilungen Nr.14, Dezember 1995, S. 219-226

[GTU91] A. Gupta, A. Tucker, S. Urushiba: *The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications.*
Proc. ACM SIGMETRICS 1991, San Diego, pp. 120-132

[HeLa92] H.-G. Hegering, A. Läpple: *Ethernet-Basis für Kommunikationsstrukturen.*
DATACOM-Verlag, 1992

[HoNa95] F. Hoßfeld, W.E. Nagel: *Per Aspera ad Astra: On the Way to Parallel Processing.*
IB-9507, Forschungszentrum Jülich, 1995

[HPF94] High Performance Fortran Forum: *High Performance Fortran Language Specification.*
Version 1.1, November 1994

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

[Kn81] D.E. Knuth: *The Art of Computer Programming; Volume 2: Seminumerical Algorithms.*
Second Edition, Addison-Wesley Publishing Company, 1981

[Li86] K. Li: *Shared Virtual Memory System on Loosely Coupled Multiprocessors*
Ph.D. Thesis, Yale University, Technical Report YALEU-RR-492, 1986

[Li94] M.A. Linn: *Untersuchungen zur Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamem Speicher.*
Jül-2931, Forschungszentrum Jülich, 1994

[LiCh91] K. Li, K.-H. Cheng: *A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System.*
Journal of Parallel and Distributed Computing, pp. 12:79-83, 1991

[LiWi94] W. Liu, V. Lo, K. Windisch, B. Nitzberg: *Non-Contiguous Processor Allocation Algorithms for Distributed Memory Multicomputers.*
Proc. Supercomputing '94, Washington D.C., pp. 227-236

[Mu94] Ch. Müllender: *Visualisierung der Speicheraktivitäten von parallelen Programmen in Systemen mit virtuell gemeinsamem Speicher.*
Jül-2911, Forschungszentrum Jülich, 1994

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

[Nit94] R. Nitzinger: *Dynamische Lastbalancierung paralleler Prozesse unter Berücksichtigung lokaler interaktiver Last.*
Diplomarbeit, RWTH Aachen, 1994

[NiKi93] L.M. Ni, P.K. McKinley: *A Survey of Wormhole Routing Techniques in Direct Networks.*
IEEE Computer, No. 2, pp. 62-76, 1993

[NXG85] L.M. Ni, Ch.-W. Xu, T.B. Gendreau: *A Distributed Drafting Algorithm for Load Balancing.*
IEEE Transactions on Software Engineering, Vol. SE-11, No. 10, pp.1153-1161, 1985

[Oed85] W. Oed: *Untersuchungen von Zugriffen auf den Arbeitsspeicher in Vektorrechnersystemen.*
Jül-1994, Forschungszentrum Jülich, 1985

[OSF92] Open Software Foundation and Carnegie Mellon University: *Mach 3 Kernel Principles: Norma-MK14.*
OSF, November 1992

[Ott95] H. Otto: *Entwicklung und Analyse von dynamischen Loop-Scheduling-Algorithmen für SVM-Fortran.*
Jül-3156, Forschungszentrum Jülich, 1995

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

[RJOS89] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones: *Mach: A System Software kernel.*
Proc. 34th Computer Society International Conference COMPCON 89, 1989

[RBFG89] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, R. Sanzi: *Mach: A Foundation for Open Systems.*
Proc. Second Workshop on Workstation Operating Systems (WWOS2), 1989

[RAAB91] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser: *Overview of the CHORUS Distributed Operating Systems.*
CS/TR-90-25.1, Chorus systèmes, 1991

[SLWT] U. Schwiegelshohn, W. Ludwig, J.L. Wolf, J. Turek, P.S. Yu: *Smart SMART Bounds for Weighted Response Time Scheduling.*

[SGI96] Cray Research, *CRAY Origin2000 Supercomputing Series - Preliminary Draft.*
Silicon Graphics Inc., 1996

[ShHo94] K.G. Shin, C.-J. Hou: *Design and Evaluation of Effective Load Sharing in Distributed Real-Time Systems.*
IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 7, 1994

[Smi80] R.G. Smith: *The Contact Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.*
IEEE Transactions on Computer Science, Vol. C-29, No.12, pp. 1104-1113, 1980

[Sta85] J.A. Stankovic: *Stability and Distributed Scheduling Algorithms.*
IEEE Transactions on Software Engineering, Vol. SE-11, No.10, pp.1141-1152, 1985

[Sto92] B. Stoustrup: *Die C++-Programmiersprache.*
Verlag Addison-Wesley, 1992

[TiWi84] A.M. van Tilborg, L.D. Wittie: *Wave Scheduling – Decentralized Scheduling of Task Forces in Multicomputers.*
IEEE Transactions on Computers, Vol. C-33, No.9, pp. 835-844, 1984

[TLWF94] J. Turek, W. Ludwig, J.L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, P.S. Yu: *Scheduling Parallelizable Tasks to Minimize Average Response Time.*
Proc. SPAA '94, 6th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 200-209

[TSWY94] J. Turek, U. Schwiegelshohn, J.L. Wolf, P.S. Yu: *Scheduling Parallel Tasks to Minimize Average Response Time.*
Proc. Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 112-121, 1994

[Zei93] S. Zeisset: *Evaluation and Enhancement of the Paragon's Multiprocessor Shared Virtual Memory System.*
Masterthesis, TU München, 1993

[Zhu92] Y. Zhu: *Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers.*
Journal of Parallel and Distributed Computing, pp. 16:328-337, 1992

Dank

Die vorliegende Arbeit ist im Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich entstanden. Ich möchte dem Institutsleiter Herrn Prof. Dr. rer. nat. F. Hoßfeld, Inhaber des Lehrstuhls für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen, herzlich für sein Interesse und die mir gewährte Unterstützung danken.

Herrn Prof. Dr. rer. nat. D. Haupt, Leiter des Rechenzentrums der RWTH Aachen, danke ich für die Übernahme des Korreferats und fruchtbare Diskussionen.

Mein besonderer Dank gilt Herrn Prof. Dr. rer. nat. W. E. Nagel, seinerzeit Leiter der Abteilung Programmiertechniken im ZAM und nunmehr Leiter des Zentrums für Hochleistungsrechnen (ZHR) an der Technischen Universität Dresden. Seine Bemerkungen und wertvollen Anregungen haben die Arbeit positiv beeinflußt.

Für konstruktive Diskussionen möchte ich mich bei den Herren H. Bast, R. Berrendorf, Dr. M. Gerndt und A. Krumme bedanken; ihre Anmerkungen und Vorschläge waren bei der Entwicklung der Systemkonzepte sehr hilfreich.

Die freundliche und sehr gute Arbeitsatmosphäre und die Hilfsbereitschaft meiner Kollegen im ZAM, insbesondere die Unterstützung bei der schriftlichen Ausarbeitung durch Herrn W. Frings, haben wesentlich zum Gelingen dieser Dissertation beigetragen. Ihnen allen gilt an dieser Stelle mein herzlicher Dank.