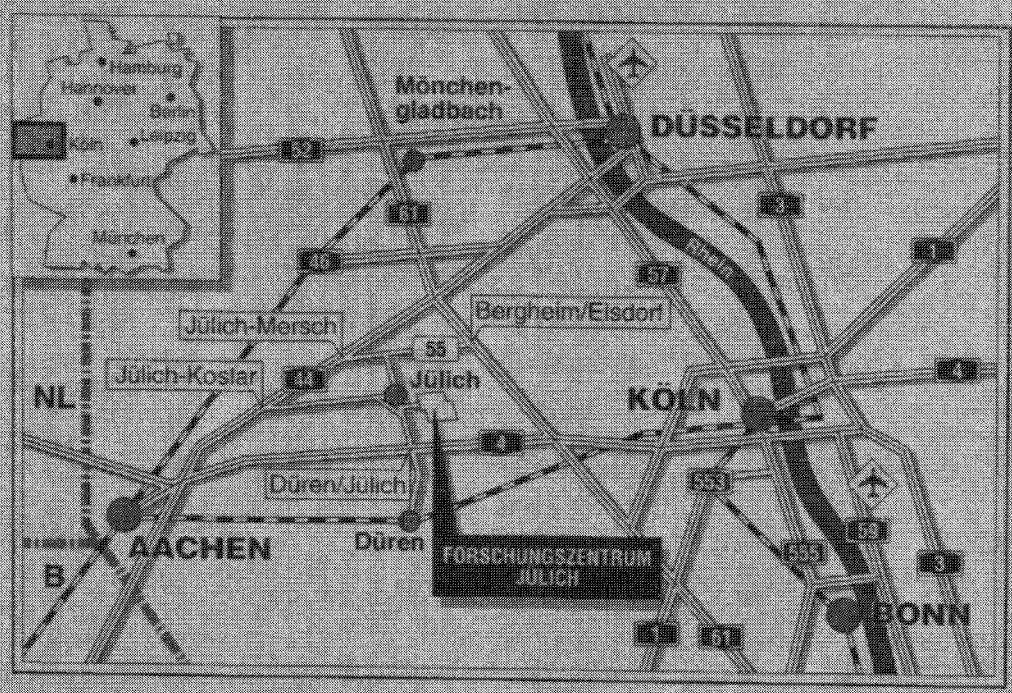
FORSCHUNGSZENTRUM JÜLICH GmbH
KFA

Zentralinstitut für Angewandte Mathematik

Untersuchungen zur Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamem Speicher

Markus Andreas Linn



Berichte des Forschungszentrums Jülich ; 2931
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-2931
D 82 (Diss. RWTH Aachen)

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

Untersuchungen zur Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamem Speicher

Markus Andreas Linn

Kurzfassung

Die Ablaufplanung von parallelen Programmen bei Rechnern mit virtuell gemeinsamem, real jedoch auf die Prozessoren verteiltem Speicher wird maßgeblich von zwei Problembereichen beeinflußt: Auf der einen Seite hängt der effiziente Betrieb dieser Rechner von der Lokalität der Daten ab; die Daten müssen am Ort des Zugriffs weitgehend lokal verfügbar sein, sonst ergeben sich permanent Datentransfers, die die Programmausführung verzögern. Auf der anderen Seite können unbalancierte Arbeitslasten die effiziente Ausführung behindern. Die Berücksichtigung beider Aspekte durch eine explizite Datenabbildung zur Maximierung der Datenlokalität bei gleichzeitiger Optimierung der *thread*-Scheduler-Algorithmen kann zu Effizienzvorteilen bei Systemen mit virtuell gemeinsamem Speicher führen.

Die vorgestellten Ergebnisse basieren auf Untersuchungen, die mithilfe eines Simulationssystems durchgeführt wurden. Dazu wurden unabhängige Modelle für das Parallelrechnersystem, für den virtuell gemeinsamen Speicher und für die Arbeitslast entwickelt und implementiert. Die Scheduler-Komponente des Simulators umfaßt sowohl die bereits von den Systemen mit global gemeinsamem Speicher her bekannten *thread*-Scheduler-Algorithmen als auch neue algorithmische Varianten, die auf die besonderen Belange bei Systemen mit virtuell gemeinsamem Speicher geeignet angepaßt sind. Darüber hinaus ist eine explizite Datenabbildung entwickelt und in das Simulationssystem integriert worden. Die Untersuchungen zeigen, daß eine geeignete Datenabbildung von der Art der Arbeitslast abhängt und explizit wählbar sein muß. Ebenso muß auch das *thread*-Scheduling in Abhängigkeit von der Arbeitslast, deren Datenzugriffsstruktur und den daraus resultierenden Speicherreferenzen geeignet wählbar sein. Alle durchgeführten Untersuchungen zeigen, daß der Einfluß der Datenzugriffsmuster auf die Ausführungszeit erheblich ist: Der bei ungeeigneter Ablaufplanung entstehende Overhead kann in ungünstigen Fällen um Größenordnungen über dem eigentlichen CPU-Zeitbedarf einer Anwendung liegen. Dies zeigt die strategische Bedeutung einer effizienten Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamem Speicher.

Inhaltsverzeichnis

1. Einleitung

Die Historie der Entwicklung von elektronischen Rechnern ist seit Anbeginn geprägt durch eine immense Steigerung ihrer Leistungsfähigkeit. Dennoch stellt die maximale Rechenleistung heutiger Supercomputer für eine Vielzahl von Anwendern eine Grenze dar, die weit unterhalb der von ihnen benötigten Leistung liegt. Forschungsgegenstand in Naturwissenschaften und Technik sind heute zunehmend komplexe, nicht-lineare Systeme, für die sich das Werkzeug der numerischen Simulation als ein unverzichtbares Instrument entwickelt [CPME93, Hos91, Wil89]. In vielen Fällen können Lösungen aufgrund der Rechenzeitanforderungen nur näherungsweise oder aber nur für kleine Problemgrößen erhalten werden. Eine befriedigende Behandlung dieser Aufgabenstellungen erfordert weitere Leistungssteigerungen im Supercomputing; dazu können grundsätzlich die folgenden vier Bereiche beitragen:

- Technologie
- Architektur
- Algorithmik
- Software-Technik

Algorithmik und Software-Technik sind wesentlich an die Vorgaben durch die Architektur gebunden; Technologie und Architektur bestimmen ihrerseits die eigentliche Rechengeschwindigkeit bezogen auf Operationen pro Zeiteinheit. An der Steigerung der Leistungsfähigkeit von Supercomputern hat die Erhöhung der Rechengeschwindigkeit einen erheblichen Anteil [Led90b]; am meisten haben technologische Neuerungen beigetragen: Der Übergang von elektromechanischen Relais über Elektronenröhren und Transistoren zu integrierten Halbleiterbauelementen (ICs) mit immer höherer Integrationsdichte und Schaltgeschwindigkeit kennzeichnet die Verkürzung der Rechnertaktzeiten von Zehntelsekunden auf Nanosekunden. Da elektrische Signale in einer Nanosekunde nur etwa dreißig Zentimeter Laufstrecke zurücklegen, müssen für eine weitere Verkürzung der Taktzeit die räumlichen Abstände innerhalb der Rechner kleiner werden. Damit ist offensichtlich, daß eine weitere Verkürzung der Taktzeit auf absehbare Zeit nur begrenzt möglich ist, mit Sicherheit jedoch nicht um mehrere Größenordnungen; seit mehreren Jahren werden auf der technologischen Seite auch mit erheblichem Aufwand keine Beschleunigungen im erwünschten Maße, d.h. um Größenordnungen erzielt.

Die stagnierende Entwicklung auf der technologischen Seite läßt der Rechnerarchitektur eine wachsende Bedeutung zukommen und hat bereits in der Vergangenheit einer Klasse von Rechnern den Weg bereitet, die Gegenstand der aktuellen Forschung und Entwicklung ist, der Klasse von Multiprozessorrechnern mit MIMD-Architektur (MIMD = *Multiple Instruction stream - Multiple Data stream* [Fly66]). Den bis vor etwa zehn Jahren im Supercomputing führenden Einprozessorrechnern haben Multiprozessorrechner zum Beispiel der Firmen Cray oder NEC den Rang abgelaufen. Diese Multiprozessorrechner nutzen heute bis zu 4, 8 oder 16 Prozessoren, die auf einem gemeinsamen Speicher arbeiten; wegen der kleinen Prozessorzahl werden diese Rechner als moderat-parallel bezeichnet.

Eine Entwicklung in Richtung wachsende Prozessorzahl zeichnet sich seit einigen Jahren ab: Das Erreichen einer wiederum technologisch bedingten Grenze hat bei Rechnern mit vergleichsweise sehr viel mehr Prozessoren (massiv-parallele Rechner) zur Realisierung einer abermals neuen Architektur geführt: Multiprozessorrechner mit verteiltem Speicher. Damit ergeben sich drei Klassen von Rechnerarchitekturen: Die Einprozessorrechner, die moderat-parallelen Rechner mit gemeinsamem Speicher und die massiv-parallelen Rechner mit verteiltem Speicher.

Bei der Entwicklung neuer Rechnerarchitekturen hat die Entwicklung der zugehörigen System-Software nicht schritthalten können. Die neuartigen Architekturkonzepte bedingen ebenfalls neue Konzepte für die Software-getriebene Steuerung der Rechner; ohne entsprechende Betriebssystemunterstützung ist die ihrer Leistungsfähigkeit angemessene Nutzung nicht gewährleistet. Bei Einprozessorrechnern hat die System-Software aufgrund der langjährigen Weiterentwicklungen des von-Neumann-Konzeptes einen soliden Status erreicht und kann als weitgehend ausgereift bezeichnet werden. Bei Multiprozessorrechnern mit gemeinsamem Speicher sind trotz der erheblichen Erfolge der letzten Jahre entsprechende Implementierungen bisher noch lückenhaft; weiterführende Konzepte sind Gegenstand aktueller Entwicklungen: Die Integration von parallelen Programmen in eine Mehrbenutzerumgebung zum Beispiel kann in bestimmten Situationen nur durch neuartige Strategien zufriedenstellend gelöst werden [Nag93]. Die bei Rechnern mit verteiltem Speicher verwendeten Konzepte basieren zum Teil auf denen für Rechner mit gemeinsamem Speicher und sind damit ebenfalls verbesserungswürdig. Darüber hinaus stellt die Architektur des verteilten Speichers neue Anforderungen an die System-Software: Die Mehrzahl heutiger Rechner mit verteiltem Speicher realisiert das *message-passing*-Programmiermodell. Der wesentliche Nachteil des *message passing* - die für viele Anwendungen aufwendige Programmierung durch die explizite Spezifikation der Datenkommunikation - hat bisher zu einer eingeschränkten Benutzerakzeptanz geführt. Verschiedene Forschungsprojekte beschäftigen sich heute mit der Programmierbarkeit von Rechnern mit verteiltem Speicher. Hinter den Schlagworten datenparallele Programmierung und virtuell gemeinsamer Speicher verbergen sich Programmiermodelle, die eine im Vergleich zum *message passing* einfachere Programmierung erlauben sollen [HPF93, Li86]. Eine allen Programmiermodellen zur Architektur des verteilten Speichers gemeinsame Problemstellung liegt in der geeigneten Gruppierung und Verteilung der Daten über den Speicher; die Programmausführung kann nur dann effizient sein, wenn die benötigten Daten im jeweiligen Speicher lokal verfügbar sind (Lokalität der Datenzugriffe). Bei der Definition von Programmiersprachen zu den Programmiermodellen werden Direktiven zur Beeinflussung der Datenverteilung und damit zur Erhöhung der Lokalität integriert [BGNP93, BKP93, HPF93].

Eine entscheidende Rolle für die Verbesserung der Lokalität kann in Rechnern mit verteiltem Speicher der Ablaufplanung zukommen. Die Ablaufplanung bei der Ausführung paralleler Programme umfaßt drei Problemstellungen: Die Identifikation von Parallelität innerhalb eines Programms, die Partitionierung der parallelen Programmteile in Teilaufgaben und das Scheduling der Teilaufgaben auf die Prozessoren. Die Identifikation von Parallelität bezieht sich auf den Algorithmus beziehungsweise auf dessen Darstel-

lung mithilfe der Programmiersprache; sie kann durch den Benutzer oder durch den Compiler erfolgen und soll in dieser Arbeit nicht weiter betrachtet werden. Die Partitionierung und das Scheduling sind jedoch von der zugrundeliegenden Rechnerarchitektur und dem Programmiermodell abhängig. In dieser Arbeit werden die Partitionierung und das Scheduling bei Rechnern mit virtuell gemeinsamem Speicher untersucht. Da es sich dabei um Parallelrechner mit verteiltem Speicher handelt, kommt insbesondere der Forderung nach Datenlokalität und den damit verbundenen zusätzlichen Anforderungen an Partitionierung und Scheduling große Bedeutung zu. Bisher wurden zu diesem Thema nur wenige Arbeiten veröffentlicht: Eine Arbeit aus jüngster Zeit behandelt die Partitionierung bei Rechnern mit virtuell gemeinsamem Speicher [GrWi93]. Für die Ablaufplanung bei Rechnern mit virtuell gemeinsamem Speicher ergibt sich eine zusätzliche Aufgabenstellung: Das Speicherkonzept unterstützt die automatische Partitionierung und Verteilung der Daten; um eine effiziente Ausführung zu erzielen, müssen Partitionierung und Verteilung auf die Datenzugriffsmuster von Anwendungsprogrammen abgestimmt werden können. Durch geeignete Strategien für diese Automatismen kann die Effizienz der Ausführung deutlich verbessert werden. Diesbezüglich sind in der Literatur bisher keine Arbeiten zu finden.

Das Modell des virtuell gemeinsamen Speichers überträgt das komfortable Progammiermodell des gemeinsamen Speichers auf Rechner mit verteiltem Speicher [Li86]. Daneben gibt es heute verwandte Modelle, die ähnliche Ziele verfolgen. Kapitel 2 versucht eine Abgrenzung der Modelle gegeneinander sowie deren Einordnung in bestehende Taxonomien.

Die in der Literatur beschriebenen Ansätze zum Modell des virtuell gemeinsamen Speichers basieren auf unterschiedlichen Teilkonzepten. Kapitel 3 stellt die verschiedenen Konzepte und bei der Implementierung verwendeten Mechanismen gegenüber; dabei stehen sowohl das Speichermanagement als auch die Prozeßverwaltung im Mittelpunkt.

In Multiprozessorrechnern bestimmt das Scheduling einen wesentlichen Teil der Ablaufplanung; im folgenden wird der Begriff Scheduling auch als Oberbegriff für die Kombination von Partitionierung und Scheduling verwendet. Die in der Vergangenheit entwickelten Ansätze basieren auf unterschiedlichen Rechnerarchitekturen und beschreiben eine Vielfalt von Algorithmen für das Scheduling auf verschiedenen Hierarchieebenen. Im Kapitel 4 werden Betrachtungen zum Scheduling angestellt, die sich in die Ebenen des Job-Scheduling und des *thread*-Scheduling gliedern. Für die Vergleichbarkeit werden zunächst allgemeine Beschreibungskriterien und jeweils zugrundeliegende Basismodelle erarbeitet.

Diese Arbeit stellt einen der ersten Schritte auf dem Gebiet des Scheduling in Rechnern mit virtuell gemeinsamem Speicher dar. Um eine Grundlage für weiterführende Arbeiten zu schaffen, werden die Betrachtungen auf das *thread*-Scheduling konzentriert. Kapitel 5 betrachtet das *thread*-Scheduling bei Rechnern mit virtuell gemeinsamem Speicher. Es werden Algorithmen und Strategien vorgestellt, die zu deutlichen Verbesserungen führen.

Die vorgestellten Algorithmen und Strategien werden mithilfe von Simulationsrechnungen untersucht. Kapitel 6 beschreibt ein Simulationssystem, das im Rahmen dieser

Arbeit implementiert worden ist. Das Simulationssystem basiert auf unabhängigen Modellen für das Multiprozessorsystem, für den virtuell gemeinsamen Speicher und für die auszuführenden Arbeitslasten.

In Kapitel 7 erfolgt die Untersuchung der vorgestellten Algorithmen. Zu diesem Zweck stehen zum einen statistische Daten aus den Simulationsläufen zur Verfügung, die eine vergleichende Bewertung erlauben. Darüber hinaus generiert das Simulationssystem Informationen zu diskreten Zustandswechseln von Prozessen und Speicher, die Einsichten in diese Abläufe geben und zu neuen Strategien für die Ablaufplanung führen.

Den Simulationsuntersuchungen liegen reale Arbeitslasten zugrunde; die Arbeitslasten und die entsprechenden Untersuchungsparameter wurden so gewählt, daß für die Ablaufplanung relevante Problemstellungen jeweils umfassend behandelt werden konnten. Kapitel 8 verdeutlicht die Bedeutung der Untersuchungen im Gesamtkontext der Ablaufplanung, faßt die gewonnenen Ergebnisse zusammen und stellt Folgerungen für die Ablaufplanung in Systemen mit virtuell gemeinsamem Speicher auf.

2. Globaler Adreßraum für Rechner mit verteiltem Speicher

Das Erreichen einer technologischen Grenze für die Geschwindigkeit einzelner Prozessoren und die Möglichkeit der Kombination von preiswerten Hochleistungs-Mikroprozessoren hat zur Entwicklung von Rechnern mit sehr großer Prozessorzahl geführt (massiv-parallele Rechner). Die mit wachsender Prozessorzahl entstehenden Anforderungen (eine ebenfalls wachsende Speicherbandbreite bei gleichbleibender Zugriffsverzögerung) bedingen eine Änderung des in Parallelrechnern mit nur wenigen Prozessoren bewährten Konzeptes eines globalen Speichers: Im Unterschied zu Rechnern, bei denen alle Prozessoren auf einen globalen Speicher einheitlich zugreifen können, wird in massiv-parallelen Systemen der Speicher über die Prozessoren verteilt, d.h. jedem Prozessor wird jeweils ein Speichermodul lokal zugeordnet. Aufgrund der Verteilung des Speichers ergibt sich eine Vielzahl neuer Problemstellungen für den Betrieb dieser Rechner. Die Programmierbarkeit massiv-paralleler Rechner stellt ein wesentliches Problem dar: In existierenden Systemen ermöglicht das *message-passing*-Programmiermodell eine komfortable Programmierung nur für wenige, besonders geeignete Anwendungen; für den universellen Einsatz massiv-paralleler Rechner stellt das *message-passing*-Programmiermodell ein entscheidendes Akzeptanzhemmnis dar. Für die meisten Anwendungen im wissenschaftlich-technischen Bereich ist die *shared-memory*-Programmierung einfacher als die *message-passing*-Programmierung, da bei der *shared-memory*-Programmierung die Datenallokation für den Programmierer transparent ist; das *shared-memory*-Programmiermodell erlaubt, im Gegensatz zu explizit lokalen und nicht-lokalen Referenzen beim *message-passing*-Programmiermodell, die Verwendung von globalen Referenzen. Die wachsende Bedeutung von Rechnern mit verteiltem Speicher in Verbindung mit der Überlegenheit des *shared-memory*-Programmiermodells über das *message-passing*-Programmiermodell hat zur Entwicklung von Konzepten geführt, die die *shared-memory*-Programmierung auch auf Rechnern mit verteiltem Speicher ermöglichen. Die unterschiedlichen Konzepte implementieren verschiedene *shared-memory*-Rechnermodelle, die sich durch die Art der Abbildung des *shared memory* auf den verteilten Speicher unterscheiden; die Abbildung wird durch die Hardware, durch das Betriebssystem, durch den Compiler oder durch eine Kombination dieser Komponenten unterstützt.

Ein wesentliches Rechnermodell dieser Kategorie verfolgt das Konzept des virtuell gemeinsamen Speichers; dieses Modell findet sich in der Literatur bereits seit 1986 [Li86]. Mit der Vielzahl der seitdem veröffentlichten Arbeiten zu diesem Konzept gibt es auch eine Vielfalt von Namen für dieses Rechnermodell, die von *shared virtual memory*, *virtual shared memory*, *distributed virtually shared memory* und *distributed shared memory* bis zu *ALLCACHE memory* reichen [Li86, GHSS91, BoIs91, BCZ90a, KSR92]. Diese Ansätze basieren auf einem Rechnermodell mit virtuell gemeinsamem Speicher, der jedoch nur logisch existiert und in der Realität auf einen verteilten Speicher abgebildet wird. Aus Gründen der Einheitlichkeit wird in diesem Zusammenhang im weiteren Verlauf dieser Arbeit von *virtual shared memory* (VSM) gesprochen.

Die Verschiedenartigkeit der Namen von VSM-Ansätzen geht einher mit unterschiedlichen Hardware-Plattformen und unterschiedlichen Implementierungen. Das Spektrum

der Hardware macht eine Einordnung des VSM-Modells in die Welt der Supercomputer-Parallelrechner problematisch. Abschnitt 2.1 setzt das Rechnermodell des *virtual shared memory* in Relation zu anderen Modellen aus dem Bereich der Supercomputer-Parallelrechner. In diesem Zusammenhang wird auch versucht, eine Abgrenzung des VSM-Modells gegenüber verwandten Modellen im Bereich der *shared-memory*-Rechnermodelle für Rechner mit verteiltem Speicher vorzunehmen; dabei werden Vor- und Nachteile von Modellen und Konzepten diskutiert. Eine wesentliche Motivation des VSM-Modells liegt in der Programmierbarkeit von massiv-parallelen Rechnern; in Abschnitt 2.2 wird die Betrachtung auf massiv-parallele Systeme eingeschränkt, und es werden drei Konzepte für die massiv-parallele Programmierung beschrieben.

2.1. Virtual Shared Memory - Versuche der Einordnung

In existierenden Taxonomien, die eine Einordnung von *virtual-shared-memory*-Systemen erlauben, sind Definitionen und Nomenklatur nicht einheitlich, es werden zum Teil die gleichen Begriffe mit verschiedenen Definitionen belegt. In einer informellen Taxonomie für MIMD-Systeme[1] definiert Karp zwei Systeme [Kar87]: Die Taxonomie klassifiziert Systeme mit einem global zugreifbaren, gemeinsamen Speicher als *shared-memory*-Systeme. In *message-passing*-Systemen verfügt laut Karp jeder Prozessor über lokal zugreifbaren Speicher, ein global zugreifbarer Speicher existiert jedoch nicht. Nach Karp gehören VSM-Systeme, die tatsächlich nur über lokalen Speicher verfügen, deren Betriebssystem jedoch die Illusion eines globalen Speichers erzeugt, zu einer Klasse von hybriden Systemen. Eine detailliertere Einordnung in den Taxonomien von Howe und Moxon [HoMo87] beziehungsweise von Johnson [Joh88] erfordert eine differenzierte Sichtweise. Beide Taxonomien definieren jeweils zwei Architekturmodelle und, orthogonal dazu, jeweils zwei Kommunikations- beziehungsweise Programmiermodelle. Ein Unterschied zwischen den beiden Taxonomien besteht in den Definitionen der Architekturmodelle: Howe und Moxon unterscheiden zwischen *shared-memory*-Architekturen mit von allen Prozessoren direkt zugreifbarem Speicher und *private-memory*-Architekturen, bei denen die Speichermodule anderer Prozessoren nicht direkt zugreifbar sind [HoMo87]. Johnson dagegen definiert *global-memory*-Architekturen mit einheitlicher Speicherzugriffszeit für alle Prozessoren und *distributed-memory*-Architekturen, die durch unterschiedliche Zugriffszeit für lokale beziehungsweise nicht-lokale Zugriffe charakterisiert sind. Aus Gründen der Eindeutigkeit bei der Begriffsbildung wird im weiteren Verlauf dieser Arbeit die Taxonomie von Johnson zugrunde gelegt, die im folgenden detailliert beschrieben und bezüglich der Einordnung von *virtual-shared-memory*-Systemen erweitert wird [Joh88]:

Die Einordnung des *virtual-shared-memory*-Modells mit real verteiltem, virtuell aber gemeinsamem Speicher macht eine differenzierte, zweidimensionale Sichtweise des Speichers notwendig. Es wird unterschieden zwischen der logischen Sicht des Speichers vom Standpunkt des Programmierers und der physikalischen Sicht des Speichers. Die logische Sicht entspricht der Sicht des Speichers aufgrund des Programmiermodells, die physikalische Sicht des Speichers entspricht der physikalischen Prozessor-Speicher-Anordnung.

[1] MIMD = *Multiple Instruction stream - Multiple Data stream* [Fly66]

Die Einordnung von Rechnermodellen (z.B. *virtual-shared-memory*-Rechnermodell) wird aufgrund von Programmiermodell und Prozessor-Speicher-Anordnung vorgenommen. Die logische Sichtweise unterscheidet zwischen zwei Programmiermodellen, dem *shared-memory*-Programmiermodell, und dem *message-passing*-Programmiermodell.

- Beim *shared-memory*-Programmiermodell können alle Prozesse auf den gesamten Speicher direkt zugreifen, alle Daten sind global bekannt. Die Kommunikation und die Synchronisation erfolgen mittels gemeinsamer Daten *(shared variables)*. Das *shared-memory*-Paradigma befreit den Programmierer von der Datenpartitionierung, die Transparenz der Datenallokation vereinfacht eine dynamische Lastbalancierung.
- Beim *message-passing*-Programmiermodell verfügt jeder Prozeß über einen Speicherbereich, auf den er nur selbst zugreifen kann *(private memory)*; alle Daten sind jeweils nur lokal bekannt. Die Kommunikation zwischen Prozessen und die Synchronisation der Prozesse können nur mithilfe von erfolgen. Die *message-passing*-Programmierung erfordert eine explizite Datenpartitionierung sowie explizit lokale und nicht-lokale Referenzen, was in vielen Anwendungen mit erheblichem Aufwand verbunden ist.

Aus physikalischer Sicht werden zwei Prozessor-Speicher-Anordnungen unterschieden:

- Ein globaler Speicher (*global memory*) ist für alle Prozessoren gleichwertig zugreifbar; die zugrundeliegende Hardware entspricht einer UMA- *(Uniform Memory Access)* Architektur [ThCr88, You87].
- Die zweite Prozessor-Speicher-Anordnung ist der verteilte Speicher (*distributed memory*), bei der jedem Prozessor ein Speichermodul lokal zugeordnet ist. Die *distributed-memory*-Architektur kann auf zwei verschiedene Arten realisiert werden, als NORMA- *(NO Remote Memory Access)* oder als NUMA- *(Non-Uniform Memory Access)* Architektur [ThCr88, You87]. Auf einer NORMA-Architektur erfolgt die Kommunikation *Message*-basiert, ein direkter Zugriff auf einen Speichermodul eines anderen Prozessors ist nicht möglich. Auf einer NUMA-Architektur ist ein solcher Zugriff möglich, jedoch ist die Zugriffszeit größer als für einen Zugriff auf den eigenen Speichermodul.

Das Konzept des *global memory* führt zu Skalierungsproblemen, d.h. bestimmte leistungsrelevante Eigenschaften können mit wachsender Prozessorzahl nicht erhalten werden, sie skalieren nicht. Eine wesentliche Eigenschaft ist z.B. die Speicherunterstützung: Eine wachsende Anzahl von Prozessoren erfordert ein entsprechendes Wachsen der Datenmenge, die der Speicher pro Zeiteinheit verfügbar machen kann (Speicherbandbreite). Die Forderung nach gleichbleibend geringer Verzögerung bei der Auflösung einer Speicherreferenz (Speicherverzögerung) läßt sich bei wachsender Speicherbandbreite nicht befriedigen, d.h. bei konstanter Speicherverzögerung kann die Speicherbandbreite nicht mit der Prozessorzahl skalieren. Ein weiteres wesentliches Skalierungsproblem stellt die schnelle Koordinierung einer wachsenden Anzahl von *tasks* dar. In existierenden Systemen wird die Koordinierung von *tasks* mittels gemeinsamer Variablen *(shared variables)* geregelt. Da der Zugriff auf die gemeinsamen Register sequentiell erfolgt, steigt die gesamte Verzögerung aufgrund der Koordinierung einer Serie von *tasks* linear mit der Anzahl der *tasks*; dieser Effekt wird als *convoy*-Problem bezeichnet [SHH90].

Das Konzept des *distributed memory* erlaubt eine Lösung der Skalierungsprobleme. Durch die Kopplung von lokalen Speichern mit Prozessoren kann, bei konstanter Speicherbandbreite je Prozessor, die Speicherbandbreite des Gesamtsystems mit der Prozessorzahl skalieren; die Speicherverzögerung bleibt dabei auf Prozessorebene, d.h. für lokale Datenzugriffe konstant. Aus dieser Strategie ergibt sich die Forderung nach Datenlokalität, ohne die eine effiziente Programmausführung nicht gewährleistet ist. Die Forderung nach Datenlokalität stellt einen wesentlichen Nachteil aller auf *distributed-memory*-Rechnern implementierten Rechnermodelle dar. Die Lösung des *convoy*-Problems für *distributed-memory*-Rechner bedarf einer verteilten Koordinierung von *tasks*, was eine essentielle Aufgabe beim Design von Schedulern für verteilte Betriebssysteme darstellt.
Global-memory-Rechner benötigen eine aufwendigere, kompliziertere Hardware als *distributed-memory*-Rechner; dies bedeutet für *global-memory*-Rechner einen zeitlich größeren Entwicklungsaufwand. Wenn das Preisleistungsverhältnis (MFLOPS/$) wichtiger ist als die absolute Leistung, dann ergibt sich aus dem Hardware-Aufwand ein weiterer Vorteil für *distributed-memory*-Rechner. Das Preisleistungsverhältnis fällt für *global-memory*-Rechner geringer aus, d.h. bei gegebenem Preis hat ein *global-memory*-Rechner eine geringere Spitzenleistung. Dabei ist jedoch zu beachten, daß die Spitzenleistung insbesondere beim Vergleich von *global-memory*-Rechnern mit *distributed-memory*-Rechnern eine weitgehend aussagelose Größe ist; auch mit besonders geeigneten Anwendungsprogrammen wird auf *distributed-memory*-Rechnern im allgemeinen ein weitaus geringerer Prozentsatz der Spitzenleistung erreicht als auf *global-memory*-Rechnern.
Tab. 1 beschreibt die zweidimensionale Kombination der logischen Sicht des Programmiermodells und der physikalischen Sicht der Prozessor-Speicher-Anordnung. Hybride Ansätze aus logischer beziehungsweise physikalischer Sicht lassen sich durch Erweiterung der Tabelle einbeziehen, sollen jedoch in diesem Zusammenhang nicht weiter betrachtet werden. Die Einordnung der Rechnermodelle *global shared memory*, *distributed memory message passing* und *virtual port memory* ist direkt und eindeutig, da innerhalb der jeweiligen Zuordnungsfelder der Tabelle keine weiteren Modelle existieren.

- **Global Shared Memory (GSM):**
  Die Kombination physikalisch *global* / logisch *shared* beschreibt das *global-shared-memory*-Modell; auf einer UMA-Hardware wird das *shared-memory*-

physikalisch **logisch**	**global memory**	**distributed memory**
shared memory	global shared memory (GSM)	distributed shared memory (DSM) virtual shared memory (VSM) compiler-bridged shared memory (CSM)
message passing	virtual port memory	distributed memory message passing (DMP)

Tab. 1. Eine Einordnung von Rechnermodellen bezüglich Programmiermodell (logisch) und Prozessor-Speicher-Anordnung (physikalisch) nach Johnson [Joh88]

Programmiermodell unterstützt. Dieses klassische Parallelrechnermodell ist z.B. in den Rechnern Cray X-MP, Cray Y-MP und IBM 3090 realisiert[2] [Com89].

- **Distributed memory Message Passing (DMP):**
  Die Kombination physikalisch *distributed* / logisch *message passing* ist in NORMA-Rechnern realisiert, die nach dem *message-passing*-Programmiermodell arbeiten. Das *distributed-memory-message-passing*-Modell stellt die einfachste Implementierung eines Programmiermodells auf einem Rechner mit physikalisch verteiltem Speicher dar und ist auf den meisten Rechnern dieses Typs implementiert (z.B. Intel Paragon [Int91] und nCUBE [Pal88]).

- **Virtual Port Memory:**
  Das Modell des *virtual port memory* [Joh87, Ols85] realisiert ein *message-passing*-Programmiermodell auf Rechnern mit physikalisch globalem Speicher. Das *virtual-port-memory*-Modell wird hier nicht weiter betrachtet.

Für die Kombination physikalisch *distributed* / logisch *shared* existiert eine Reihe von verwandten Ansätzen, die das *shared-memory*-Programmiermodell auf Rechnern mit verteiltem Speicher implementieren. Da sich die Ansätze zum Teil durch die Abbildung des *shared memory* auf den verteilten Speicher unterscheiden, läßt sich diesbezüglich eine weitere Aufteilung der Rechnermodelle vornehmen. Bei der Formulierung der Definitionen für die Rechnermodelle *distributed shared memory* (DSM), *virtual shared memory* (VSM) und *compiler-bridged shared memory* (CSM) wurde versucht, alle heute existierenden Ansätze einzubeziehen.

- **Distributed Shared Memory (DSM):**
  Bei diesem Modell hat jeder Prozessor direkten Zugriff auf alle Speichermodule, die Zugriffszeiten unterscheiden sich jedoch für den Zugriff auf verschiedene Module zum Teil erheblich; die zugrundeliegende Hardware entspricht der NUMA-Architektur, die z.B. mit dem Rechner BBN TC2000 kommerziell verfügbar ist [BBN90]. Beim *distributed-shared-memory*-Modell kann auf jede Referenz einzeln zugegriffen werden (feingranularer Zugriff).

- **Virtual Shared Memory (VSM):**
  Das *virtual-shared-memory*-Modell kann auf NUMA- oder auf NORMA-Architekturen implementiert werden. Bei NUMA-Architekturen wird mittels des *virtual shared memory* der Zugriff auf nicht-lokale Daten im Unterschied zum *distributed shared memory* Seiten-orientiert (grobgranular) durchgeführt. Bei NORMA-Architekturen können die Prozessoren nur mithilfe von *Messages* kommunizieren; ein direkter Zugriff auf Speichermodule anderer Prozessoren ist nicht möglich. Nicht-lokale Zugriffe werden mithilfe von *Messages* realisiert und erfolgen ebenfalls Seiten-orientiert (grobgranular). Ein entscheidender Unterschied zwischen DSM und VSM besteht in der Datenhaltung. Im Gegensatz zum DSM, wo jedes Datum genau einmal existiert, können beim VSM zeitweise mehrere gültige Kopien

[2] Die lokalen Speicher, d.h. die Register der Cray und die *Cache*-Speicher der IBM, werden als temporäre Zwischenspeicher benutzt und sind für diese Betrachtung irrelevant.

einer Seite existieren; in diesem Zusammenhang ist beim VSM-Modell die Kohärenz des Adreßraums ein wesentliches Problem (siehe Kapitel 3).

- **Compiler-bridged Shared Memory (CSM):**
  Das Programmiermodell des *compiler-bridged shared memory* basiert auf einem *shared memory*, das tatsächlich jedoch nur auf der Compiler-Ebene emuliert wird; Die Verbindung von physikalisch realem *distributed memory* und emuliertem *shared memory* wird vom Compiler erzeugt *(bridged)*. Das CSM-Modell kann auf NORMA-Architekturen implementiert werden, bei denen die Prozessoren über *Messages* kommunizieren; der Compiler generiert *message-passing*-Sprachprimitive, die die Bereitstellung von nicht-lokalen Daten gewährleisten (z.B. *High Performance FORTRAN* [HPF93]).

Weder von den Konzepten *global memory* und *distributed memory* noch bei den Programmiermodellen *shared memory* und *message passing*, noch bei den beschriebenen Rechnermodellen gibt es ein Konzept beziehungsweise Modell, das in jeder Hinsicht zu favorisieren ist. Leistung und Programmierbarkeit, Preisleistungsverhältnis und Skalierbarkeit gehören zu den entscheidenden Bewertungskriterien, wobei die Rechnermodelle jeweils Kompromisse bezüglich zumindest einem dieser Kriterien eingehen. Darüber hinaus hängt die Favorisierung eines Modells aufgrund eines einzelnen Bewertungskriteriums stark von den jeweiligen Anwendungsprogrammen ab; so können z.B. die Leistung und Programmierbarkeit verschiedener Anwendungsprogramme jeweils verschiedene Modelle favorisieren.

2.2. Konzepte der massiv-parallelen Programmierung

Heute existierende massiv-parallele Systeme haben einen physikalisch verteilten Speicher. Bei der Mehrzahl der Implementierungen ist das *message-passing*-Programmiermodell realisiert. Ein wesentlicher Nachteil des *message passing* liegt in der für viele Anwendungen sehr aufwendigen Programmierung. Die explizite Spezifikation der Kommunikation ist oft komplex und damit fehleranfällig. Ein weiteres Problem stellt die Untersuchung von verschiedenen Algorithmen dar. Da Zugriffe auf verteilte Datenstrukturen durch komplex verknüpfte Sequenzen von *message-passing*-Primitiven formuliert werden, ist die jeweilige Datenzerlegung eng mit der erzeugten Programmstruktur verbunden *(hard wired algorithm)*. Die notwendigen Kommunikationsstrukturen sind für verschiedene Datenzerlegungen im allgemeinen völlig verschieden, was einen entscheidenden Mangel an Flexibilität für das Testen von alternativen Datenzerlegungen darstellt. Die Programmierung auf der niedrigen Ebene des *message passing* führt, durch die Inflexibilität beim Testen von Alternativen, paradoxerweise zu geringer Leistung sowie zu unnötiger Arbeit für den Programmierer. Ein weiterer wesentlicher Nachteil ist in vielen Anwendungen die Unbalanciertheit zwischen Kommunikations- und Rechenaufwand; dies ist der Grund für die oft nur geringe Effizienz, d.h. das Verhältnis von erzielter Leistung zur Spitzenleistung. Neben der aufwendigen, unkomfortablen Programmierung sind insbesondere die Leistungsprobleme dafür verantwortlich, daß dem *message-passing*-Modell und damit

den massiv-parallelen Rechnern bisher der Durchbruch gegenüber den *global-shared-memory*-Rechnern verwehrt ist.
Modelle zur Verbesserung der massiv-parallelen Programmierung orientieren sich an der Programmierbarkeit. Alle neueren Modelle basieren auf dem *shared-memory*-Programmiermodell. Neben der besseren Programmierbarkeit wird für alle Modelle die Leistung im Vergleich zu *message-passing*-Programmen entscheidend sein. Da bei diesen Modellen ein *shared memory* nur als *shared memory* mit erhöhten Zugriffszeiten für nicht-lokale Zugriffe und nicht als *global shared memory* existiert, sind für Programme mit geringer Datenlokalität erhebliche Leistungsverluste zu erwarten. Durch eine geeignete, z.B. explizit Benutzer-gesteuerte Kontrolle der Datenverteilung und der Lastbalancierung kann die Lokalität erhöht werden.

2.2.1. Das DSM-Modell: Shared Memory auf einer NUMA-Architektur

Das *distributed-shared-memory*-Modell basiert auf einer direkten Abbildung des *shared-memory*-Programmiermodells auf eine NUMA-Architektur: Der physikalisch verteilte Speicher - jeder Speichermodul ist einem Prozessor zugeordnet - ist global adressierbar, d.h. jeder Prozessor kann auf jede Speicherstelle direkt zugreifen. Im Unterschied zu Zugriffen auf den lokalen Speichermodul - der dem Prozessor zugeordnete Speichermodul - ist die Zugriffszeit für Zugriffe auf einen *remote*-Speichermodul - ein Speichermodul, der einem anderen Prozessor zugeordnet ist - zum Teil erheblich höher. Der Grund für die nicht einheitliche *(non-uniform)* Zugriffszeit ist in der Architektur zu finden: In einem BBN TC2000 Rechner zum Beispiel sind die Speichermodule über ein mehrstufiges Butterfly-Verbindungsnetzwerk mit den Prozessoren verbunden [BBN89]. Der Zugriff auf einen *remote*-Speichermodul erfolgt über das Netzwerk und erfordert eine erheblich höhere Zugriffszeit als ein lokaler Zugriff, der ohne das Netzwerk direkt ausgeführt wird. Um die Verzögerung aufgrund von *remote*-Datenzugriffen zu verringern, wird in NUMA-Rechnern wie dem BBN TC2000 oder der IBM ACE Multiprozessor-Workstation [GFF89] in jedem Prozessor ein eigener *Cache*-Speicher genutzt. In diesem Zusammenhang werden spezielle NUMA-Speichermanagement-Techniken verwendet [BFS89, DMK92].

Massiv-parallele DSM-Rechner sind mit dem BBN Butterfly I seit 1979 kommerziell verfügbar. Mit Nachfolgemodellen bis hin zum BBN TC2000 wurde das DSM-Modell in immer leistungsfähigeren Rechnern realisiert.

2.2.2. Das CSM-Modell: Transformation von Shared-Memory-Programmen

Beim *compiler-bridged-shared-memory*-Modell erzeugt der Programmierer ein *shared-memory*-Programm. Die Grundidee besteht darin, daß der Compiler aufgrund von zusätzlichen Spezifikationen das *shared-memory*-Programm transformiert, so daß es auf einem Rechner mit verteiltem Speicher ausgeführt werden kann. Die zusätzlichen Spezifikationen bestehen im wesentlichen aus der Datenpartitionierung, d.h. aus der Zerlegung von Datenmengen und der Verteilung der Daten auf eine ebenfalls zu spezifizierende virtuelle oder physikalische Netzwerktopologie. Das *shared-memory*-Programm kann z.B. in ein *message-passing*-Programm transformiert werden: Dabei transformiert der

Compiler die globalen Datenzugriffe in lokale und nicht-lokale Zugriffe[3]; für nicht-lokale Zugriffe werden *message-passing*-Sprachprimitive generiert. Die Programmausführung erfolgt im *message-passing*-Betrieb.
Die existierenden Ansätze unterscheiden sich bezüglich der Datenzerlegung beziehungsweise -verteilung, die automatisch oder Benutzer-gesteuert sein kann. Das CSM-Modell ist seit 1987 in einer Vielzahl von Ansätzen untersucht und implementiert worden. In SUPERB [Ger89, ZBG88] ist die halbautomatische Parallelisierung von FORTRAN-Programmen für den SUPRENUM-Rechner [Gil88] implementiert worden: Aufgrund von Programmfluß- und Datenabhängigkeitsanalysen wird mit SUPERB eine interaktive Datenzerlegung durchgeführt, Daten und Schleifeniterationen werden auf die Knoten verteilt, und es wird die zur Auflösung von nicht-lokalen Referenzen notwendige Kommunikation eingefügt. Schwachpunkte dieses ersten Ansatzes eines CSM-Modells (keine automatische Datenpartitionierung, nur statische Datenverteilung) sind in nachfolgenden Ansätzen verbessert. In [KeZi89] wird ein dynamisches Schema für die Datenpartitionierung vorgeschlagen. Wie SUPERB sieht auch dieser Ansatz zunächst die Transformation von sequentiellen Programmen vor, d.h. mithilfe von Datenabhängigkeitsanalysen und automatischer Parallelisierung[4] werden sequentielle DO-Schleifen in *Single-Program-Multiple-Data*-Codesequenzen überführt [Kar87]. Die explizite Programmierung von parallelen Schleifen stellt diesbezüglich eine Erweiterung dar, die auch dann eine parallele Programmausführung ermöglichen soll, wenn für die Abhängigkeitsanalyse zur Übersetzungszeit keine hinreichenden Informationen zur Verfügung stehen (Fortran D [Fox90], Kali [MeRo91]). Explizit parallele Schleifen können jedoch zu semantischen Problemen führen, die in den bestehenden Ansätzen noch ungelöst sind [Ger92c]. Eine mögliche Alternative zu explizit parallelen Schleifen besteht in einer Abhängigkeitsanalyse zur Laufzeit mit automatischer Parallelisierung.
In Fortran D und Vienna FORTRAN [CMZ92] werden Datenzerlegung und -verteilung durch die Möglichkeit einer relativen Datenanordnung *(data alignment)*, d.h. der Anordnung von Datenfeldern relativ zur Verteilung eines anderen Datenfeldes, ergänzt. In FORTRAN D werden die Datenzerlegung und die Datenanordnung von der Datenverteilung getrennt. Dadurch kann das Problem-Mapping (Datenzerlegung und -anordnung) maschinenunabhängig durchgeführt werden. Maschinenabhängigkeiten können im zweiten Schritt beim Maschinen-Mapping (Datenverteilung auf die Prozessoren) getrennt behandelt werden. Sowohl die Datenverteilung als auch die Datenanordnung sind dynamisch.
Bei einigen Ansätzen verteilt der Compiler die Iterationen von parallelen Schleifen (FORTRAN D, Kali, ARF [KMSB90]); dabei werden *message-passing*-Sprachprimitive so eingefügt, daß auch nicht-lokale Daten vor der Ausführung der entsprechenden Iterationen bereitstehen. Es werden Kommunikationstupel (in(p,q)- beziehungsweise out(p,q)-Tupel) verwendet, die angeben, welche Daten dem Prozessor p gehören und von Prozessor q benötigt werden. Aufgrund der Kommunikationstupel sendet ein Prozessor alle ihm zugeordneten *Messages*, führt die Iterationen mit ausschließlich lokalen Daten-

[3] Dies gilt nicht für alle Realisationen.

[4] Vergleichbare Programmierumgebungen zur automatischen Parallelisierung wurden auch für *global-shared-memory*-Rechner entwickelt, z.B. im Rahmen von ParaScope [KMT91].

zugriffen aus, empfängt seine *Messages* und führt schließlich die übrigen Iterationen aus. Die Kommunikationstupel können zur Übersetzungszeit oder zur Laufzeit ermittelt werden. Zur Übersetzungszeit kann die Analyse nur dann durchgeführt werden, wenn hinreichend detaillierte Informationen bereits vorliegen (z.B. über die Verteilung der benötigten Datenstrukturen). In einigen Fällen können die Kommunikationstupel erst durch eine Laufzeit-Analyse bestimmt werden; wenn z.B. die Indizes der benötigten Feldelemente von Variablen abhängen, die erst zur Laufzeit bestimmt werden, ist die Analyse zur Übersetzungszeit nicht möglich. In solchen Fällen kann mithilfe des *inspector-/executor*-Konzepts [MSMB90] eine Bestimmung der Kommunikationstupel zur Laufzeit erfolgen. Der *inspector* führt eine modifizierte Version der parallelen Schleife aus, d.h. er erzeugt eine Liste nicht-lokaler Referenzen, aus denen die Tupel ermittelt werden. In einer anschließenden Kommunikationsphase werden die Tupel an die jeweiligen Prozessoren geschickt. Der *executor* führt nach der Kommunikationsphase die originale parallele Schleife aus.
In bestimmten Fällen können die Kommunikationstupel zur Übersetzungszeit bereits symbolisch bestimmt werden, zur Laufzeit müssen dann lediglich geschlossene Ausdrücke evaluiert werden; häufig muß jedoch aufgrund von nicht vorliegenden Informationen die gesamte Bestimmung der Tupel zur Laufzeit erfolgen.
Für die Laufzeit-Analyse ist die Effizienz bezüglich der Programmbeschleunigung aufgrund des zusätzlichen Overheads nicht a priori sichergestellt. In vielen Fällen ändern sich die Variablen bei der Ermittlung der Kommunikationstupel nur selten und können bei wiederholter Schleifenausführung wiederverwendet werden; auf diese Weise kann die Effizienz der Laufzeit-Analyse gesteigert werden [KMR90]. In der funktionalen Sprache Id Nouveau [ANP86, RoPi89] wird keine Wiederverwendung der Kommunikationstupel betrachtet, und die Laufzeit-Analyse wird als ineffizient bezeichnet.
Eine wesentliche Neuerung stellt die automatische Datenzerlegung dar (Crystal [LiCh90a], ASPAR [IFKF90]). In ASPAR generiert der Compiler automatisch die Datenzerlegung und Kommunikation für FORTRAN-Programme mit Block-Verteilung; eine automatische Parallelisierung für irreguläre Verteilungen ist jedoch nicht vorgesehen. Mit Erfahrungen aus den bisherigen Ansätzen hat sich das *High Performance FORTRAN Forum* zusammengefunden, um aufbauend auf dem FORTRAN-90-Standard Spracherweiterungen zu definieren, die einen Standard für eine datenparallele Programmierung darstellen sollen [HPF93]. Die Version 1.0 des *High Performance FORTRAN* spezifiziert viele der oben beschriebenen Techniken; darüber hinaus sind weitere Techniken Gegenstand aktueller Entwicklungen.

Im Bereich *compiler-bridged shared memory* gibt es zur Zeit eine Vielzahl an Aktivitäten. Die Entwicklung neuer Compiler-Techniken zur verbesserten Abhängigkeitsanalyse, die verbesserte Behandlung von Laufzeitabhängigkeiten in Schleifen, z.B. durch Laufzeit-Präprozessing, die Weiterentwicklung der automatischen Datenzerlegung sowie die Einbindung in interaktive Programmierumgebungen mit Leistungsvorhersagen für die Datenpartitionierung gehören zu den aktuellen Entwicklungen im Bereich *compiler-bridged shared memory*.

2.2.3. Das VSM-Modell: Globaler Adreßraum durch Demand Paging

In einem *virtual-shared-memory*-System wird ein globaler Adreßraum auf die lokalen Speichermodule eines *distributed-memory*-Rechners abgebildet. Der lokale Speicher eines Prozessors wird als *Cache*-Speicher aufgefaßt, so daß sich der gesamte Speicher ausschließlich aus *Cache*-Speichern zusammensetzt. Die Daten werden in Gruppen oder Seiten *(pages)* zusammengefaßt, die auf Anfrage zwischen den lokalen Speichern der Prozessorelemente transferiert werden können *(demand paging)*. Ähnlich wie in *multicache*-Systemen [Len90] können entsprechend dem Prinzip der Datenreplikation zeitweise mehrere gültige Kopien einer Seite existieren. Ein wesentliches Problem besteht darin, den *virtual-shared-memory*-Adreßraum kohärent zu halten: Der aus einer Leseoperation resultierende Wert muß immer mit dem jeweils zuletzt auf diese Adresse geschriebenen Wert übereinstimmen. Mögliche Lösungen zum Kohärenzproblem sowie eine ausführliche Beschreibung des VSM-Modells finden sich in Kapitel 3.

Das Modell des *virtual shared memory* ist in verschiedenartigen Ansätzen mit unterschiedlicher Motivation implementiert worden. Das System IVY realisiert ein *virtual shared memory* für ein Netzwerk von Workstations [Li86, Li88]. In Shiva wird mit den für IVY entwickelten Algorithmen ein VSM-System auf einem Intel iPSC/2 Hypercube implementiert [LiSc89a, LiSc89b]. Beide Ansätze sind auf Benutzerebene außerhalb des Betriebssystems implementiert und untersuchen alternative Algorithmen sowie die Effektivität des *virtual shared memory*. Mit KOAN wird der in Shiva gemachte Ansatz auf der Betriebssystemebene des iPSC/2-Systems implementiert [LaPr91]. Leistungsmessungen aufgrund von Anwendungsprogrammen zeigen die Effizienz des *virtual shared memory*; ein Vergleich mit Leistungsmessungen für entsprechende Anwendungsprogramme im *message-passing*-Betrieb auf dem gleichen Rechner führt zu ähnlichen Speedup-Werten [PrLa92] beziehungsweise zu einer Erhöhung der Speedup-Werte bei KOAN [LaPr91][5]. Wie in allen Rechnern mit verteiltem Speicher ist auch in VSM-Rechnern die Lokalität der Daten von entscheidender Bedeutung für die Effizienz. Geeignete Algorithmen und Mechanismen zur Verbesserung der Datenlokalität hängen von der konkreten Struktur der Anwendungen ab. Eine Anpassung an die Struktur einer Anwendung wird in Munin und Mirage versucht [BCZ90a, FlPo89]. In Munin werden verschiedene Kohärenzmechanismen angewandt, die eine adaptive, typspezifische Kohärenz implementieren. In Mirage wird ein Tuning-Parameter verwendet, der dynamisch in Abhängigkeit vom Programmverhalten erhebliche Auswirkungen auf spezifische Anwendungen als auch auf den Systemdurchsatz haben kann.
Eine wesentliche Erweiterung bezüglich der zugrundeliegenden Hardware stellt PLATINUM dar [CoFo89]; während allen bisher genannten Ansätzen eine NORMA-Architektur zugrunde liegt, ist PLATINUM auf einer NUMA-Architektur implementiert. Die Ausnutzung des bei einer NUMA-Architektur möglichen Zugriffs auf einen nichtlokalen Speichermodul erweist sich in einigen Situationen als sinnvolle Ergänzung zum *paging*.
Mit dem Design der Data Diffusion Machine wird eine spezielle *virtual-shared-memory*-

[5] Eine Erhöhung der Speedup-Werte von O(log n) auf O(n) wurde für einen Ray-Tracing-Algorithmus gemessen, für den die gleichzeitige Gewährleistung von Lastbalancierung und geeigneter Datenverteilung bei der Datenorientierten *message-passing*-Parallelisierung nicht gelöst werden konnte.

Hardware auf der Basis eines hierarchischen Bussystems vorgestellt [HaHa89].
Während die bisher betrachteten Ansätze als Forschungsprojekte entstanden sind, sollen Mach und Chorus als Grundlage für verteilte Netzwerk-Betriebssysteme dienen, zu denen auch kommerzielle Produkte zählen [You87, Roz88]. Mach ist als Kern eines verteilten Netzwerk-Betriebssystems implementiert, das den Betrieb von Ein- und Mehrprozessorrechnern in einem Netz mit einheitlichem Zugriff auf alle Daten erlaubt. Eine auf dem Mach-Kern aufbauende *virtual-shared-memory*-Implementierung ist z.B. PLATINUM (siehe oben). Auch MaX-SVM basiert auf Mach und ist auf der Benutzerebene implementiert [Moh93].
Die bisher einzig kommerziell verfügbaren *virtual-shared-memory*-Rechner wurden von den Firmen Myrias, Kendall Square und Cray entwickelt [BBZ88, Dun92, RSRM93, KeSc93, PMM93]. Der SPS-2-Rechner der Firma Myrias ist ein Mikroprozessor-basiertes System, das physikalisch in einer Baumstruktur realisiert ist. Der KSR 1 von Kendall Square nutzt RISC-Prozessoren, die über ein hierarchisches 2-Ebenen-Netzwerk von Ringen verbunden sind. Als erster massiv-paralleler Rechner der Firma Cray stellt der Cray T3D die derzeit aktuellste Realisierung in diesem Bereich dar. Die Hardware wird durch den Alpha-Prozessor der Firma DEC und eine dreidimensionale Torusstruktur für das Verbindungsnetzwerk charakterisiert, und das *virtual-shared-memory*-Modell realisiert vergleichsweise kleine Seiten und das Konzept der schwachen Kohärenz.

3. Virtual Shared Memory

Das Modell des *virtual shared memory* (VSM) implementiert auf Rechnern mit verteiltem Speicher das komfortable Programmiermodell eines *shared-memory*-Rechners. Ein virtueller Adreßraum wird auf den lokalen Speicher aller Prozessoren verteilt. Die Daten werden in Gruppen oder Seiten *(pages)* zusammengefaßt und z.B. nach dem *demand-paging*-Verfahren verwaltet. In konventionellen *demand-paging*-Systemen werden, zur Realisation des virtuellen Speicherkonzepts, auf Anfrage Seiten zwischen Massenspeicher und Hauptspeicher transferiert. In dem hier beschriebenen VSM-Modell wird ein *demand paging* zwischen den lokalen Speichern der Prozessorelemente durchgeführt. Der Programmierer sieht dadurch nur einen globalen Adreßraum; so entsteht für ihn die Illusion, einen *global-shared-memory*-Rechner zu programmieren.
Geeignete Algorithmen zur Ablaufplanung in VSM-Rechnern hängen stark vom Speichermanagement in solchen Systemen ab. Abschnitt 3.1 gibt einen Überblick über das Speichermanagement existierender VSM-Ansätze, d.h. über Speicherzugriffsmodelle, Kohärenzstrategien und deren Implementierung, über verschiedene Strategien zum Speicher-Mapping und zur Speicher-Allokation sowie über die Seitenersetzung. Ausgewählte Mechanismen zur Prozeßverwaltung in VSM-Systemen werden in Abschnitt 3.2 vorgestellt.

3.1. Speichermanagement in VSM-Systemen

In einem *virtual-shared-memory*-Adreßraum sind alle Daten von allen Prozessoren zugreifbar. Tatsächlich lokal verfügbar ist jedoch üblicherweise nur ein Teil der Daten. Das Auftreten einer Referenz auf eine lokal nicht vorhandene Adresse (inkohärente Referenz) erfordert die Auflösung dieser Referenz. Die Auflösung einer inkohärenten Referenz, die einen wesentlichen Teil der Kohärenzstrategie darstellt, unterscheidet sich bei VSM-Systemen im Vergleich zu klassischen *multicache-shared-memory*-Systemen. Ein klassisches *multicache*-System besteht meist aus einer relativ kleinen Anzahl von Prozessoren, die einen physikalisch globalen Speicher über ihren privaten *Cache* gemeinsam nutzen. In VSM-Systemen existiert kein physikalisch globaler Speicher; die als *Cache* genutzten lokalen Speicher sind relativ groß und erzeugen den gesamten VSM-Speicher.

3.1.1. Speicherzugriffsmodelle

Das Speicherzugriffsmodell eines VSM-Ansatzes wird bestimmt durch die Abbildung des virtuellen Adreßraums auf den physikalischen Adreßraum (Speicher-Mapping) sowie durch den Zugriffsmechanismus. Die Einheit für das Mapping stellt dabei eine charakteristische Größe dar. Analog zum klassischen Konzept des virtuellen Speichers wird in vielen Ansätzen die Seite *(page, segment, cache-line)* als Einheit für das Mapping benutzt (Seiten-orientiertes Mapping [RSRM93, LaPr91, Li86]). Die Seite wird mit einem Status wie *read-only* oder *read-write* versehen, der das Zugriffsrecht spezifiziert (Seiten-orientierter Zugriff). In einigen Ansätzen wird das Mapping auf der Basis von Objekten durchgeführt (Objekt-orientiertes Mapping [BCZ90a, CoFo89, RaKh91]); der Begriff des Objekts bezeichnet logisch zusammenhängende Daten. Die Objekte werden

in Seiten unterteilt, und das Zugriffsrecht wird wiederum auf Seitenebene spezifiziert (Seiten-orientierter Zugriff).
Als Beispiel für ein komplexes Speicherzugriffsmodell kann PLATINUM genannt werden [CoFo89]. Der PLATINUM-Ansatz ist auf einer NUMA-Architektur implementiert (BBN Butterfly Plus). Das traditionelle Konzept des virtuellen Speichers umfaßt drei Ebenen in der Speicherhierarchie: Den *Cache*, den einheitlich zugreifbaren Primärspeicher und den wesentlich langsameren Sekundärspeicher. Ein Charakteristikum des PLATINUM-Ansatzes besteht darin, daß aufgrund der NUMA-Architektur mit dem *remote*-Primärspeicher eine zusätzliche Ebene in der Speicherhierarchie existiert. Um die Anforderungen dieser zusätzlichen Ebene separat von denen des traditionellen Konzepts des virtuellen Speichers zu behandeln, wurde in PLATINUM ein 4-stufiges Speicherzugriffsmodell implementiert; das Modell besteht aus den virtuellen Adreßräumen, den Objekten, dem kohärenten Speicher und dem physikalischen Speicher. Alle Objekte innerhalb eines virtuellen Adreßraums sind *shared*-Objekte für alle *threads* in diesem Adreßraum (Objekt-orientiertes Mapping). Ein Objekt kann in mehrere virtuelle Adreßräume als zugreifbar abgebildet werden. Mit der Aufteilung der Objekte in Seiten wird ein Seiten-orientierter Zugriff realisiert. Der kohärente Speicher gewährleistet die Kohärenz des Mapping.

3.1.2. Kohärenzstrategien

Ähnlich wie bei der Nutzung von *Cache*-Speichern in den Prozessoren von *global-shared-memory*-Rechnern (*multicaches*) wird auch beim VSM-Modell das Prinzip der mehrfachen Datenhaltung angewandt, es existieren zeitweise mehrere gültige Kopien einer Seite (Datenreplikation). Ein wesentliches Problem besteht darin, den VSM-Adreßraum zu jeder Zeit kohärent zu halten, d.h. der aus einer Leseoperation resultierende Wert muß immer mit dem jeweils zuletzt auf diese Adresse geschriebenen Wert übereinstimmen (strikte Kohärenz). Eine temporäre Einschränkung dieser Kohärenzforderung (schwache Kohärenz) erlaubt in bestimmten Situationen eine Erhöhung der Parallelität.

Die Menge der möglichen Zugriffsrechte auf eine Seite charakterisiert die Kohärenzstrategie. Im einfachsten Fall wird das Zugriffsrecht auf eine Seite mit *read-only* (Nur-Lese-Zugriff) oder *read-write* (Schreib-Lese-Zugriff) bestimmt. In komplexeren Modellen werden weitere Zugriffsrechte spezifiziert, um die Seitenverwaltung zu beschleunigen. Es wird z.B. unterschieden zwischen *read-only*-Seiten mit mehreren Kopien und *read-only*-Seiten mit nur einer Kopie [CoFo89]. Eine wesentliche Erweiterung der Zugriffsrechte wird in [RaKh91] vorgenommen: Mit *weak-read* kann eine Seite im nicht-exklusiven Zugriff gehalten werden, jedoch ohne Garantie, ob sich die Seite ändert. Auf diese Weise kann aufgrund der möglichen Zugriffsrechte die Kohärenzstrategie bestimmt werden, die strikt kohärent oder schwach kohärent sein kann.

Die Strategie der strikten Kohärenz kann auf der Basis von Invalidationstechniken, durch Zurückschreiben (*writeback*), durch *write-update* und durch *remote*-Mapping realisiert werden (siehe Abb. 1). Beim Invalidationsansatz haben entweder mehrere Knoten *read-only*-Zugriff auf eine Seite, oder ein Knoten hat *read-write*-Zugriff. Will ein Knoten auf eine Seite zugreifen, für die er nicht das entsprechende Zugriffsrecht hat, so wird ein *read*

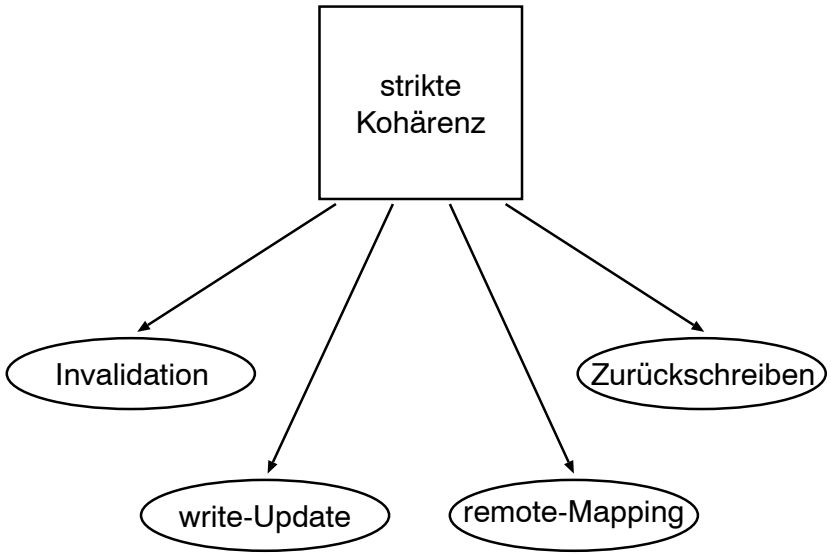


Abb. 1. Mechanismen der strikten Kohärenz

page fault oder ein *write page fault* ausgelöst. Bei einem *write page fault* werden alle Kopien der betroffenen Seite in Knoten mit *read-only*-Zugriff invalidiert (siehe Abb. 2). Nur der anfragende Knoten erhält eine gültige Kopie und *read-write*-Zugriff. Wird ein *read page fault* ausgelöst, dann wechselt das Zugriffsrecht des aktuell schreibberechtigten Knotens (falls existent) in *read-only*, und der anfragende Knoten erhält eine Kopie der Seite (Abb. 2: (5) und (6)). Die Invalidation beziehungsweise die Änderung des Zugriffsrechts für eine Seite kann mit drei unterschiedlichen Techniken erfolgen; es kann entweder ein individueller Prozessor, oder mittels *multicast* eine bestimmte Gruppe von Prozessoren angesprochen werden, oder es können mittels *broadcast* alle Prozessoren angesprochen werden. Die *broadcast*-Technik hat den Vorteil, daß keine Informationen über vorhandene Kopien der jeweiligen Seite (*copy set*) benötigt werden; für Systeme mit großer Prozessorzahl ist diese Lösung jedoch mit zuviel Overhead aufgrund der Vielzahl von *Messages* verbunden. Für große Systeme ist die Verwendung von *copy sets* in Verbindung mit der *multicast*-Technik effizienter.
Beim Zurückschreiben existieren nur Knoten mit *read-only*-Zugriff. Tritt ein *write page fault* auf, dann wird auf alle Kopien der Seite geschrieben. Da die beim Zurückschreiben notwendigen Aktualisierungen mit hohen Kommunikationskosten verbunden sind, erscheint dieser Ansatz ungeeignet für ein *distributed-memory*-System.
Mit der *write-update*-Technik (Abb. 3) wird bei einem *write page fault* der Zugriff auf die *read-only*-Kopien einer Seite nur vorübergehend gesperrt [GHSS91]. Der schreibende Prozeß gibt das *read-write*-Zugriffsrecht nach Beendigung aller Schreibzugriffe explizit ab (Abb. 3: (5)); danach werden die gesperrten *read-only*-Kopien aktualisiert und wieder für den Zugriff freigegeben. In Verbindung mit geeigneten Synchronisationsstrategien für mehrfache *page faults* kann mit dieser Technik der Kopier-Overhead auf Kosten von zusätzlichem *message-passing*-Overhead reduziert werden. Aufgrund von Untersuchungen in [GHSS91] hat sich die Invalidation in den meisten Fällen als effizienter erwiesen. Bei häufigem, feingranularem Schreibzugriff mehrerer Prozessoren auf unterschiedliche Daten auf derselben Seite treten beim Invalidationsansatz Effizienzprobleme auf; die

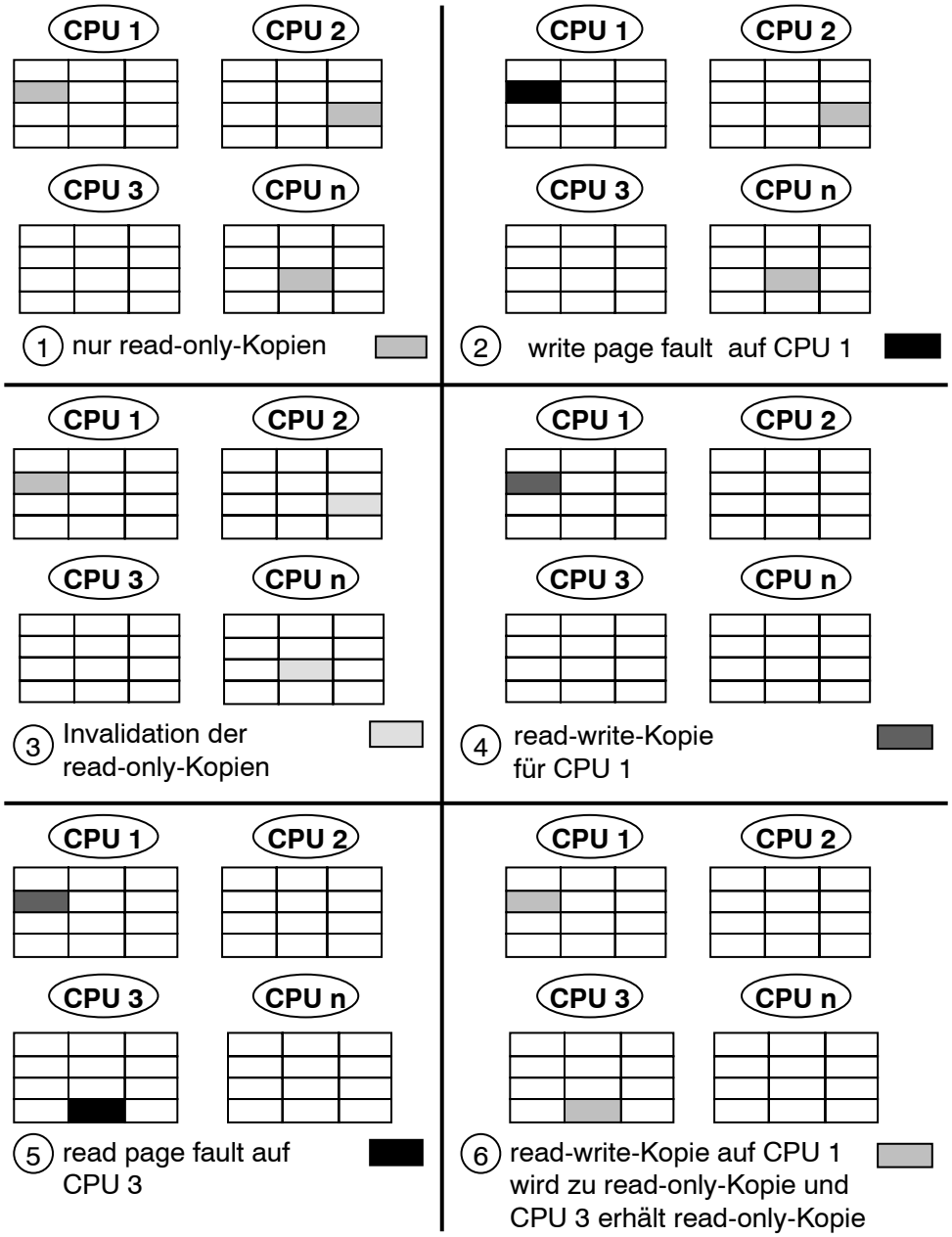


Abb. 2. Strikte Kohärenz durch Invalidation

entsprechende Seite muß ständig invalidiert beziehungsweise zwischen den Prozessoren transferiert werden (*page thrashing* [Li86]). Mögliche Lösungsansätze sind neben der Strategie der schwachen Kohärenz (siehe unten) die Datenreorganisation und das *remote*-Mapping. Bei einer Reorganisation der Daten werden die von verschiedenen *threads* benutzten Daten auf verschiedenen Seiten angelegt; die geeignete Anordnung der Daten wird vom Benutzer oder vom Compiler spezifiziert. Das *page thrashing* stellt ein wesentliches Problem des VSM-Modells dar; diesbezüglich ist die geeignete, Seiten-orientierte Anordnung von Daten eine wesentliche Anforderung an Compiler für VSM-Systeme. Ein Nachteil dieser Vorgehensweise liegt in dem zusätzlich benötigten Speicherplatz, da bei einer Verteilung von kontinuierlichen Datenfeldern auf verschiedene Seiten im allgemeinen jeweils Teile der Seiten ungenutzt bleiben (Fragmentierung). Beim

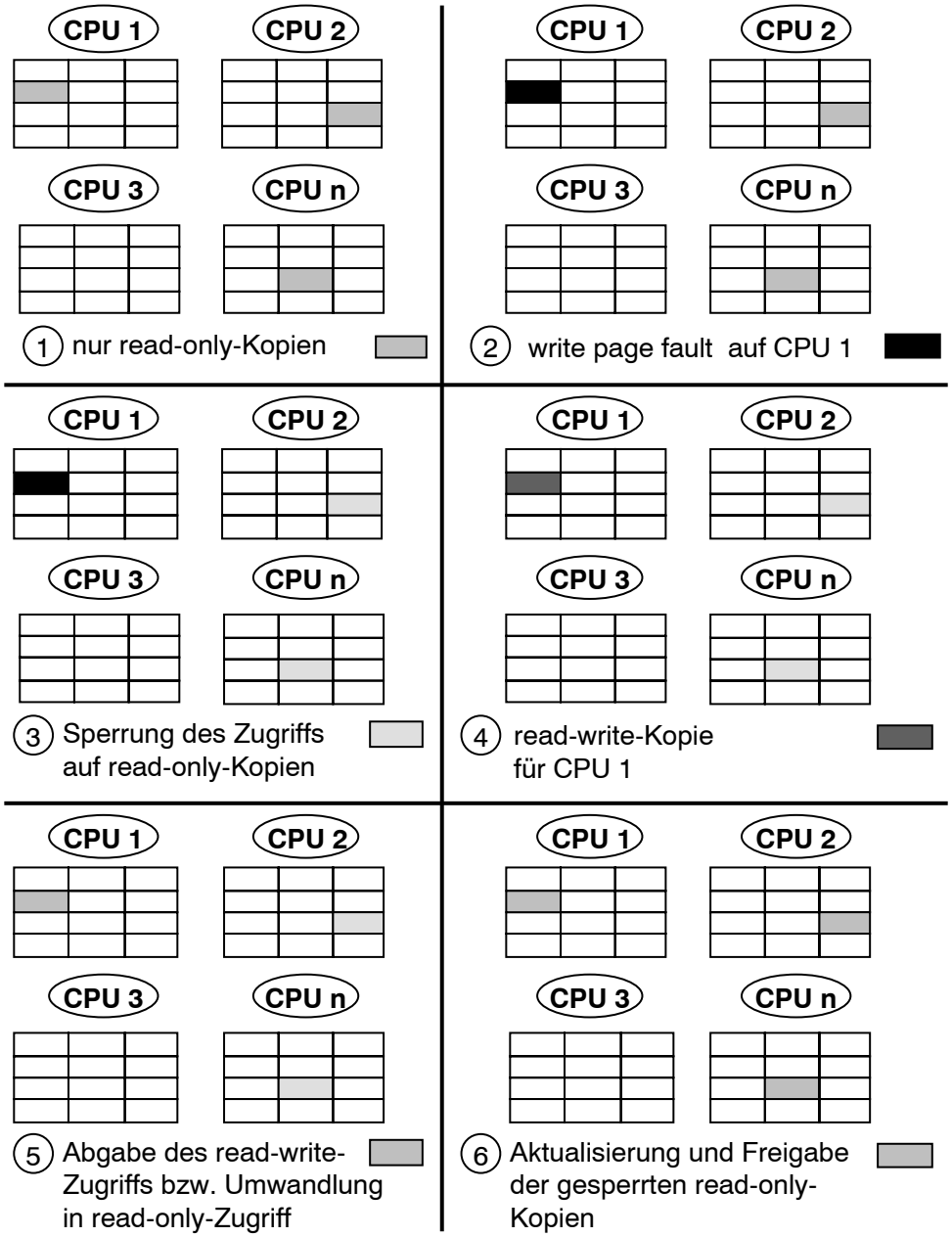


Abb. 3. Strikte Kohärenz durch die write-update-Technik

remote-Mapping (Abb. 4) wird anstatt der Replikation oder Migration einer Seite eine auf einem anderen (*remote*) Knoten existierende Kopie der Seite als zugreifbar abgebildet [CoFo89]; eine solche Seite entspricht einer *non-cachable*-Seite in *multicache-shared-memory*-Systemen. Anstatt die Seite in den eigenen lokalen Speichermodul (bzw. *Cache* im *multicache*-System) zu laden und die gewünschten Daten zuzugreifen, erfolgt der Zugriff jeweils auf ein einzelnes Datum direkt im *remote*-Speichermodul (bzw. *shared memory* im *multicache*-System). Diese Technik ist besonders geeignet, wenn viele Prozessoren häufig feingranular schreibend auf die gleichen Daten zugreifen.
Grundsätzlich sind Kombinationen der beschriebenen Techniken zur Realisierung der Strategie der strikten Kohärenz möglich. Bei einer Kombination von Invalidation und Zurückschreiben kann in Abhängigkeit von der Anzahl existierender Kopien jeweils eine

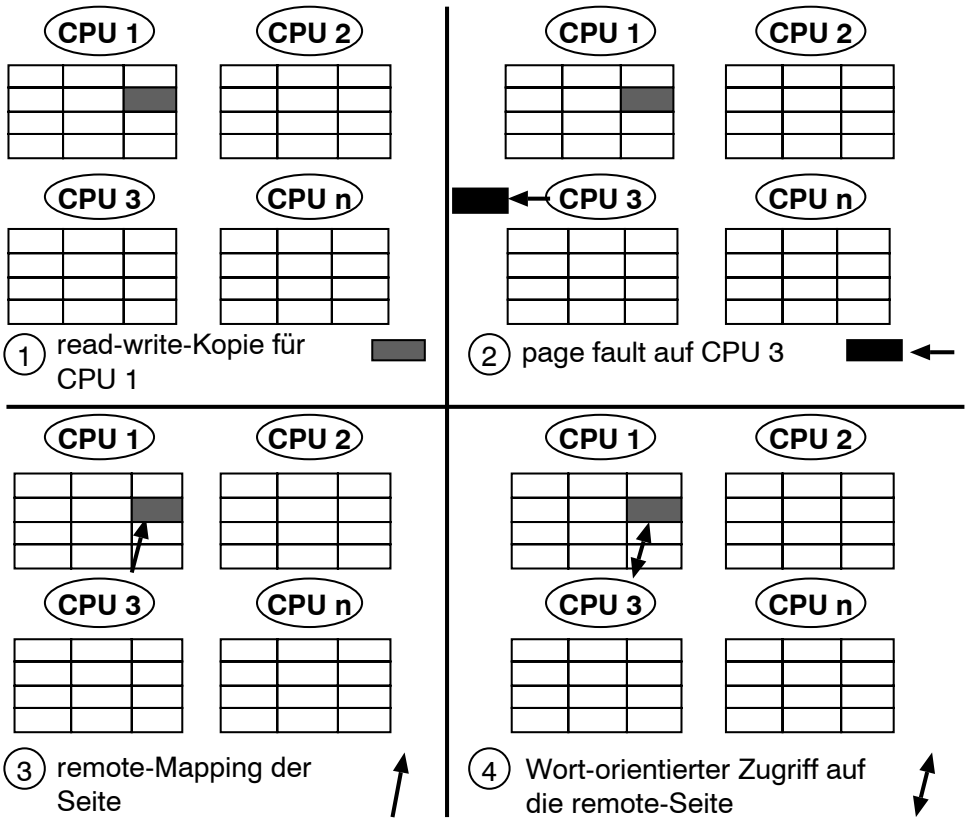


Abb. 4. Strikte Kohärenz durch remote-Mapping

der beiden Strategien angewandt werden. Bei der sogenannten selektiven Invalidation [CoFo89] ist eine Kombination von Invalidation und *remote*-Mapping implementiert. Dabei wird selektiv und dynamisch entschieden, welche der beiden Techniken genutzt wird. Die Invalidation und das Zurückschreiben können prinzipiell jeweils als eigenständige Implementierung eines VSM genutzt werden; die beiden Techniken *write-update* und *remote*-Mapping können nur in einer Kombination mit einer der beiden vorgenannten Techniken ein sinnvolles VSM implementieren.

Die Idee der schwachen Kohärenz basiert auf einer temporären Einschränkung der Kohärenzforderung mit dem Ziel, die Parallelität zu erhöhen [BNR89, BoIs91, GHSS91, KVW87]. Nach der Strategie der schwachen Kohärenz arbeitet jeder Prozeß auf einer eigenen Kopie einer entsprechenden Seite; dabei kann jeder Prozeß auf Daten auf derselben Seite schreibend zugreifen, d.h. es existieren gleichzeitig mehrere *read-write*-Kopien einer Seite (Abb. 5). Die Erzeugung einer aktuellen Kopie einer solchen Seite erfolgt an Synchronisationspunkten, wo alle Kopien der Seite vereinigt werden *(merge)*.
Eine Erweiterung der beiden Strategien (schwache Kohärenz und strikte Kohärenz) kann durch eine Kombination von strikt und schwach kohärenten Protokollen realisiert werden (Abb. 6, [BCZ90a, GHSS91, RaKh91]). In [GHSS91] wird dabei zunächst nach dem strikt kohärenten Protokoll verfahren; für Programmteile kann jedoch explizit das schwach kohärente Protokoll genutzt werden. Das Ende eines solchen Programmteils entspricht einem expliziten Synchronisationspunkt und veranlaßt die Vereinigung aller Kopien einer Seite und den Wiedereintritt in das strikt kohärente Protokoll.

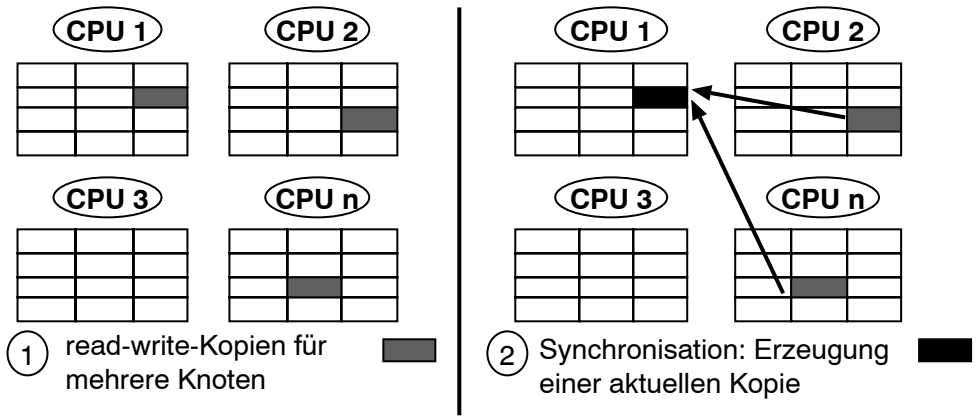


Abb. 5. Die schwache Kohärenz

Ein komplexerer Ansatz für eine Kombination von strikt und schwach kohärenten Protokollen wird in Munin gemacht [BCZ90a]. Die implementierte typspezifische Kohärenzstrategie unterstützt verschiedene Kohärenzmechanismen (strikte und schwache Kohärenz) für verschiedene Objekttypen. Die implementierten Mechanismen sind Replikation, *remote*-Mapping *(remote load store)*, Migration, verzögerte Aktualisierung (*delayed update*) und vorzeitige Bereitstellung von Objekten *(eager object moving)*.
Bei der Replikation oder Vervielfältigung von Objekten (*read-only* und *read-write*) werden mehrere Kopien der Objekte in verschiedenen Knoten erzeugt. Bei der Änderung einer Kopie werden alle übrigen Kopien aktualisiert; die Aktualisierung kann mittels Invalidation oder Zurückschreiben erfolgen. Die Replikation verursacht geringe Verzögerungen für Lesezugriffe, da diese lokal erfolgen, Schreibzugriffe bedeuten jedoch große Verzögerungen aufgrund der Aktualisierung aller *remote*-Kopien. Beim *remote*-Mapping existiert nur eine *remote*-Kopie, so daß Schreibzugriffe nur eine Aktualisierung und damit eine relativ geringe Verzögerung verursachen. Lesezugriffe sind beim *remote*-Mapping dagegen relativ teuer, da jeder Lesezugriff ein *remote*-Zugriff ist. Für *read-write*-Daten

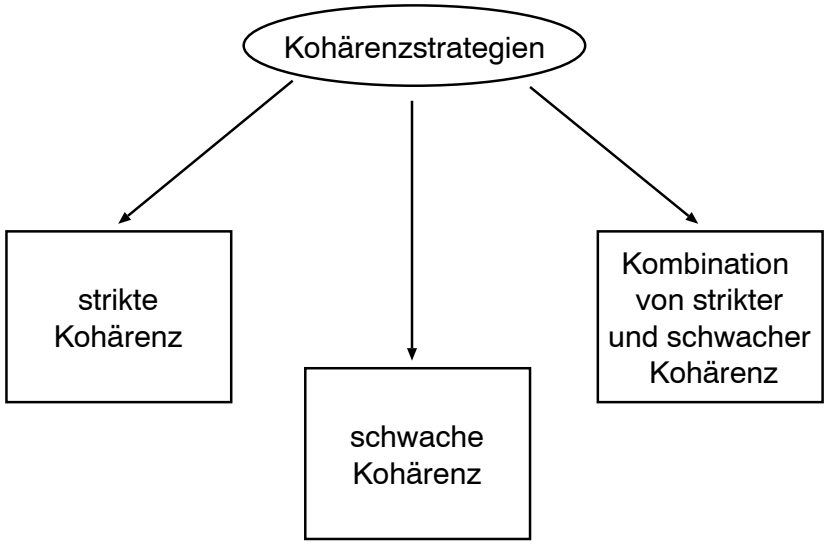


Abb. 6. Überblick über verschiedene Kohärenzstrategien

kann in Abhängigkeit von der relativen Häufigkeit der Schreib- beziehungsweise Lesezugriffe jeweils einer der beiden Mechanismen überlegen sein. Bei der Migration wird jeweils die einzige existierende Kopie einer Seite migriert. Die verzögerte Aktualisierung ähnelt der in [GHSS91] beschriebenen Form der schwachen Kohärenz; die Aktualisierung von *remote*-Kopien modifizierter Datenobjekte kann verzögert werden, bis die *remote-threads* auf die entsprechenden Daten zugreifen wollen. Die verzögerte Aktualisierung kann z.B. bei gleichzeitigem Schreiben von zwei *threads* auf verschiedene Objekte auf der gleichen Seite angewendet werden. An einem geeigneten Synchronisationspunkt erfolgt die Erzeugung einer aktuellen Kopie der Seite *(merge)*. Die vorzeitige Bereitstellung von Objekten entspricht einer Migration oder Replikation von Objekten ohne explizite Anforderung; im Idealfall stehen die Daten bereits zur Verfügung, bevor sie tatsächlich benötigt werden.
Zur Realisierung der typspezifischen Kohärenz sind 9 verschiedene Objekttypen definiert. Aufgrund von Benutzer-spezifizierten Informationen zum Objekttyp einer Datenstruktur wählt das Laufzeitsystem geeignete Kohärenzmechanismen für jedes Objekt. Geplant ist, daß diese Auswahl zur Laufzeit dynamisch erfolgt. In Tab. 2 sind die 9 Objekttypen mit dem jeweils geeigneten Kohärenzmechanismus angegeben.

Mit Clouds ist ein weiterer Ansatz einer Kombination von strikter und schwacher Kohärenz implementiert [RaKh91]. Im Unterschied zu Munin wird hier der Kohärenzmechanismus nicht auf das Objekt, sondern auf die Seite bezogen. Die beiden Zugriffsrechte *read-only* und *read-write* können in dieser Implementierung nur dann geändert werden,

Objekttyp	**Beschreibung des Objekttyps**	**Kohärenz-mechanismus**
write-once	werden einmal geschrieben (Initialisierung) und danach nur noch gelesen	Replikation
private	werden nur von einem *thread* zugegriffen	keine Kohärenz erforderlich
write-many	werden zwischen Synchronisationspunkten häufig von mehreren *threads* modifiziert	Verzögerte Aktualisierung
result	sammeln Ergebnisse, die einmal geschrieben und nur von einem *thread* gelesen werden	Verzögerte Aktualisierung
synchronization	wechselnder exklusiver Zugriff; Objekte wie z.B. *lock* und *monitor*, die zur Synchronisation benötigt werden	verteilte Synchronisation mittels verteilter *locks*
migratory	werden phasenweise jeweils nur von einem *thread* zugegriffen	Migration
producer-consumer	werden von einem *thread* geschrieben und von einer festen Gruppe von *threads* gelesen	Vorzeitige Bereitstellung
read-mostly	werden signifikant öfter gelesen als geschrieben	Replikation und *remote*-Mapping
general read-write	alle anderen Objekte; *read-write*- und *read-only*-Zugriffe von mehreren *threads*	*Berkeley-Ownership-Cache-Consistency-Protocol* [Kat85]

Tab. 2. Typspezifische Kohärenz in Munin: Objekttypen und geeignete Kohärenzmechanismen

wenn der jeweils berechtigte Knoten sein Zugriffsrecht auf die Seite explizit aufgibt. Als weitere Zugriffsrechte sind *weak-read* (nicht exklusiver *read-only*-Zugriff ohne Garantie, daß sich die Seite nicht ändert) und *none* (exklusiver *read-write*-Zugriff ohne Garantie, daß die Seite nicht abgegeben werden muß) implementiert. In Abb. 7 sind die beschriebenen Möglichkeiten der Kombination von strikter und schwacher Kohärenz dargestellt.

3.1.3. Implementierung von Kohärenzstrategien

Neben den konzeptionellen Unterschieden für die Kohärenzstrategien kann die Implementierung einer Kohärenzstrategie auf verschiedene Arten erfolgen und damit ein weiteres wesentliches Charakteristikum verschiedener VSM-Ansätze darstellen. Eine Kohärenzstrategie kann Software-basiert mithilfe von Handler- und Server-Techniken [BCZ90a, LaPr91, Li86, LiSc89a] oder aber Hardware-basiert implementiert werden [KSR92]. Während eine Hardware-basierte Implementierung möglicherweise physikalische Geschwindigkeitsvorteile einbringt, kann mithilfe einer Software-basierten Implementierung ein höheres Maß an Flexibilität erreicht werden. Wie in Munin können so z.B. Objekte entspechend ihrer Zugriffscharakteristik mit geeigneten Kohärenzmechanismen behandelt werden. Es kann sowohl mit variabler Seitengröße als auch mit dynamischen Entscheidungen für geeignete Kohärenzmechanismen gearbeitet werden.
Ein wesentliches Charakteristikum der Implementierung zeigt sich auch in der zugrundeliegenden Hardware-Architektur. So hat z.B. das *remote*-Mapping auf einer NUMA-Architektur (PLATINUM [CoFo89]) entscheidende Geschwindigkeitsvorteile gegenüber dem *remote*-Mapping auf einer NORMA-Architektur (Munin [BCZ90a]). Im folgenden werden Software- und Hardware-basierte Implementierungen näher untersucht.

3.1.3.1 Software-Mechanismen: Page-Fault-Handler und -Server

In allen beschriebenen Speicherzugriffsmodellen wird mit Seiten-orientiertem Zugriff gearbeitet, d.h. bei der Auflösung einer inkohärenten Referenz wird die gesamte Seite, in der die Referenz liegt, verfügbar gemacht. Es wird ein *page fault* ausgelöst, was

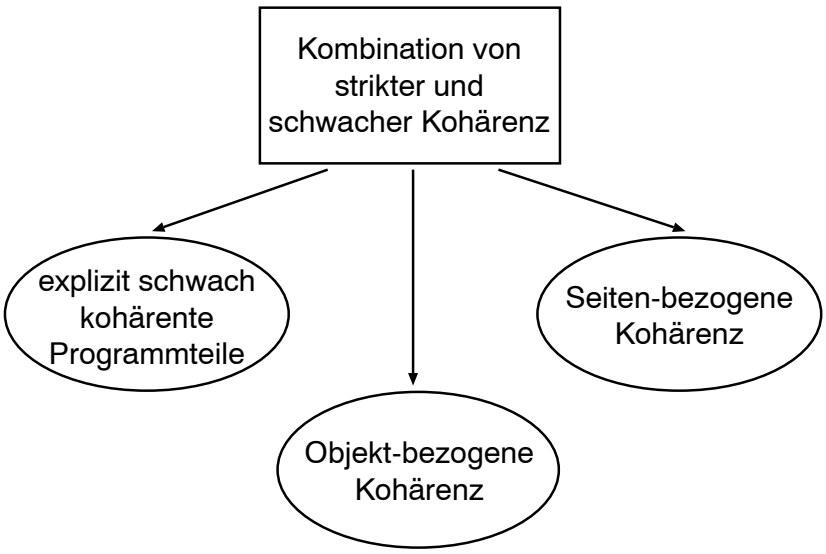


Abb. 7. Die Kombination von strikter und schwacher Kohärenz

eine Unterbrechung des auslösenden Prozesses und eine Aktivierung geeigneter *page-fault*-Handler und -Server bewirkt. In IVY [Li86] erzeugt der Handler im Prozessor mit dem *page fault* eine Anfrage für einen entsprechenden Zugriff auf die Seite (*read-only* oder *read-write*), der Server im Prozessor mit der gesuchten Seite bearbeitet die Anfrage und macht die Seite verfügbar. Die Handler und Server implementieren die Kohärenzstrategie. Verschiedene Algorithmen sind in [Li86, LiHu89] untersucht worden und in zum Teil abgewandelter oder erweiterter Form auch in anderen Ansätzen implementiert [GHSS91, LaPr91, LiSc89b]. Im folgenden werden drei Algorithmen betrachtet, der zentrale-Manager-Algorithmus *(centralized manager algorithm)*, der statisch-verteilte-Manager-Algorithmus *(fixed distributed manager algorithm)* und der dynamisch-verteilte-Manager-Algorithmus *(dynamic distributed manager algorithm)*. Bei allen drei Algorithmen gibt es jeweils einen Besitzer für jede Seite, der jedoch dynamisch wechseln kann *(dynamic page ownership)*; der Besitzer einer Seite verfügt über eine aktuelle Kopie der Seite. Eine wesentliche Aufgabe der Manager besteht in der Verwaltung der Informationen über die aktuellen Besitzer der Seiten *(ownership information)*. Für jede Seite werden außer dem Besitzer das Zugriffsrecht, das *copy set* (spezifiziert alle Prozessoren mit einer gültigen *read-only*-Kopie der Seite) und Synchronisationsvariablen (*lock*-Variablen) gespeichert.
Beim zentralen-Manager-Algorithmus verwaltet ein zentraler Manager auf einem einzelnen Prozessor alle Informationen über die aktuellen Besitzer aller Seiten. Bei einem *page fault* fragt der jeweilige Prozessor den Manager nach einer Kopie der entsprechenden Seite; der Manager gibt die Anfrage an den jeweiligen Besitzer der Seite weiter.
Beim statisch-verteilten-Manager-Algorithmus verwaltet jeder Prozessor eine vorbestimmte Menge von Seiten, d.h. es gibt einen Manager auf jedem Prozessor; die jeweilige Menge von Seiten wird durch eine Mapping-Funktion *H* bestimmt. Bei einem *page fault* fragt der jeweilige Prozessor den Prozessor *H(p)* (Manager) nach dem Besitzer der Seite, der Manager leitet die Anfrage weiter.
Beim dynamisch-verteilten-Manager-Algorithmus ist der Manager jeder Seite dynamisch verteilt: Neben den oben genannten Einträgen speichert jeder Prozessor Informationen über die Besitzer aller Seiten in seiner lokalen Seitentabelle; die Variable *probOwner* spezifiziert entweder den tatsächlichen Besitzer oder den wahrscheinlichen Besitzer einer Seite. Enthält der Eintrag nicht die Adresse des tatsächlichen Besitzers, so wird mithilfe des wahrscheinlichen Besitzers der Anfang einer Sequenz von Prozessoren gefunden, die schließlich zum tatsächlichen Besitzer führt. Bei der Initialisierung wird die Variable *probOwner* auf einen Default-Wert gesetzt, der den initialen Besitzer aller Seiten anzeigt. Zur Laufzeit wird die Variable *probOwner* einer Seite aktualisiert, wenn der Prozessor eine Invalidationsanfrage erhält, wenn der Prozessor den Besitz der Seite abgibt oder wenn der Prozessor eine *page-fault*-Anfrage weiterleitet.
In [LiHu89] wurden die drei genannten Algorithmen bezüglich ihrer Leistung gegenübergestellt. Da der zentrale Manager beim Auftreten vieler *page faults* einen Engpaß *(bottleneck)* darstellt, sind die verteilten Manager für massiv-parallele Systeme besser geeignet. Der statisch-verteilte-Manager-Algorithmus benötigt zwei *Messages* für die Mitteilung einer Seitenanfrage an den Besitzer (Abb. 8: (2) *page request* und (5) *request for page or copy to requester*). Beim dynamisch-verteilten-Manager-Algorithmus entfällt

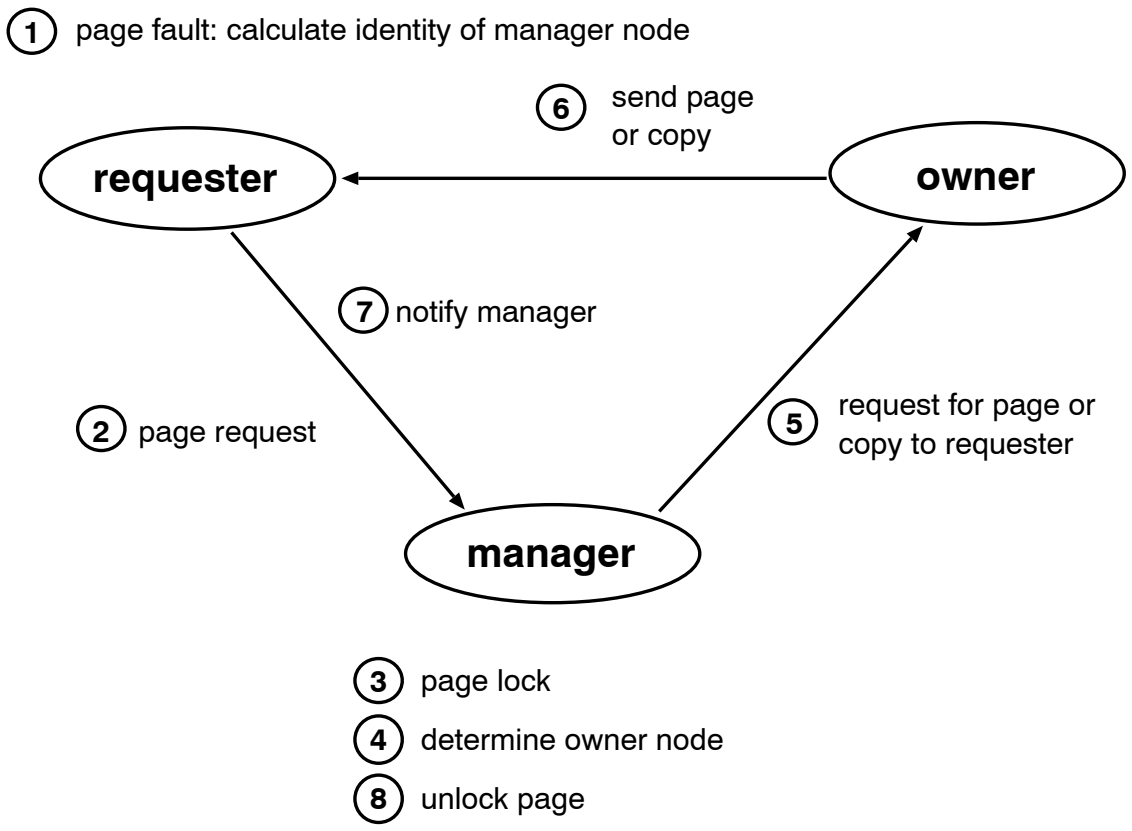


Abb. 8. Der statisch-verteilte-Manager-Algorithmus

im Vergleich dazu die *Message* an den Manager. Im Idealfall, d.h. wenn der Inhalt des Feldes *probOwner* den tatsächlichen Besitzer anzeigt (diese Voraussetzung wird in den Untersuchungen in [LiHu89] erfüllt), wird anstelle von zwei *Messages* nur eine *Message* für die Mitteilung der Seitenanfrage an den Besitzer benötigt. In den beschriebenen Untersuchungen ist der dynamisch-verteilte-Manager-Algorithmus dem statisch-verteilten-Manager-Algorithmus deutlich überlegen.

Weitere Software-basierte VSM-Ansätze sind ebenfalls durch Handler und Server implementiert. In Munin prüft der Server den Objekttyp und aktiviert einen dem Objekttyp entsprechenden Handler [BCZ90a]. Auf diese Weise wird mittels unterschiedlicher Handler eine adaptive Kohärenzstrategie implementiert. Das PLATINUM-System [CoFo89] implementiert den zusätzlichen Mechanismus des *remote*-Mapping (siehe Abschnitt 3.1.2): Für die *page-fault*-Handler steht eine Historie von Invalidationen für jede Seite zur Verfügung; aufgrund der Historie entscheidet der Handler z.B. bei einem *read page fault*, ob eine Replikation oder ein *remote*-Mapping vorgenommen wird.

Beim Auftreten mehrfacher *page faults* auf dieselbe Seite wird eine Synchronisation notwendig. In [Li86] wird die Synchronisation mittels der in den Seitentabellen enthaltenen *lock*-Variablen realisiert: Beim Auftreten mehrfacher *page faults* auf dieselbe Seite von verschiedenen Prozessen auf demselben Prozessor gewährleistet eine Seiten-bezogene *lock*-Variable, daß für jede Seite nur eine Anfrage abgeschickt wird. Neben dieser in jedem Prozessor für jede Seite verfügbaren *lock*-Variablen existiert für jede Seite eine weitere *lock*-Variable in dem jeweiligen Managerknoten. Beim Auftreten mehrfacher

page-fault-Anfragen für dieselbe Seite gewährleistet diese Variable den exklusiven Zugriff auf den Tabelleneintrag der Seite und so die sequentielle Bearbeitung der Anfragen durch den Manager (Abb. 8: (3) *page lock*). Die *lock*-Variable wird aufgrund einer Bestätigung (Abb. 8: (7) *notify manager*) nach der Abarbeitung der Anfrage zurückgesetzt (Abb. 8: (8) *unlock page*). In ähnlicher Weise wird in [LaPr91] ein Schiedsrichter *(arbiter)* verwendet, der mittels einer Synchronisationsvariable entscheidet, ob eine Anfrage zum entsprechenden Server weitergeleitet wird, ob die Anfrage in einer Warteschlange gespeichert wird oder ob die Anfrage zurückgewiesen wird.
In [RaKh91] werden Seitenzugriffe mithilfe von Primitiven explizit angefordert *(get)* und freigegeben *(discard)*. Bei *get*-Anfragen für exklusiven Zugriff auf eine Seite, die bereits von einem Prozeß in exklusivem Zugriff gehalten wird, warten alle anfragenden Prozesse in einer *First-come-first-serve*-Warteschlange auf eine explizite *discard*-Operation.

3.1.3.2 Hardware-Mechanismen

Die bisher einzigen Hardware-basierten Implementierungen einer Kohärenzstrategie werden mit dem KSR1 der Firma Kendall Square und dem Cray T3D der Firma Cray Research geliefert; im folgenden soll die Implementierung für den KSR1-Rechner betrachtet werden [Dun92, KSR92]. Ähnlich wie in [Li86] unterscheidet die verwendete Strategie *read-only*- und *read-write*-Seiten mittels strikter Kohärenz auf der Basis von Invalidationen. Der wesentliche Unterschied zu [Li86] liegt in der Implementierung der Kohärenzstrategie: Ein *page fault* bewirkt hier die Aktivierung der *Hardware Search Engine* (HSE). Die HSE ersetzt die Funktionen der Handler und Server, d.h. sie sucht die benötigten Seiten in den lokalen Speichern, macht die Seiten verfügbar und gewährleistet die Kohärenz. Ähnlich wie in [RaKh91] kann zur Synchronisation von Zugriffen mehrerer Prozessoren auf eine Seite optional mit expliziten *Get*- und *Release*-Instruktionen gearbeitet werden. Mit *Get* wird für eine Seite ein *lock* gesetzt, womit die Seite für Zugriffe anderer Prozessoren gesperrt wird. Im Unterschied zu [RaKh91] wird eine *Get*-Anfrage auf eine bereits gesperrte Seite nicht in eine Warteschlange eingereiht, sondern mit einem entsprechenden Vermerk zurückgeschickt. Mit *Release* wird ein *lock* explizit zurückgesetzt. Um die Skalierbarkeit der HSE zu gewährleisten, wurde eine hierarchische Konstruktion gewählt; die HSE arbeitet auf einer 2-Ebenen-Hierarchie von unidirektionalen Ringen, was der Anordnung der Knoten des Rechners entspricht. Die kleinen Ovale in Abb. 9 entsprechen den Knoten des Rechners, die großen Ovale entsprechen den Ringen. Jeder Knoten verfügt über ein *directory* mit einem Eintrag für jede Seite im lokalen Speicher des Knotens. Die HSE beginnt die Suche nach einer Seite in einem Knoten auf der unteren Hierarchieebene. Sequentiell wird die Suche im Ring fortgesetzt bis zum Verbindungsknoten zur oberen Hierarchieebene, der über ein *directory* mit Seiteneinträgen für den gesamten Ring der unteren Ebene verfügt. Bei erfolgloser Suche im Ring der unteren Ebene wird vom Verbindungsknoten zur oberen Ebene verzweigt und die Suche wird entsprechend fortgesetzt. Bei erfolgreicher Suche auf der oberen Ringebene, d.h. bei Lokalisierung eines Rings der unteren Ebene, der die gewünschte Seite enthält, wird in den entsprechenden Ring der unteren Ebene verzweigt, wo die Suche schließlich mit der Lokalisierung der gesuchten Seite endet. Die HSE arbeitet in allen Knoten simultan.

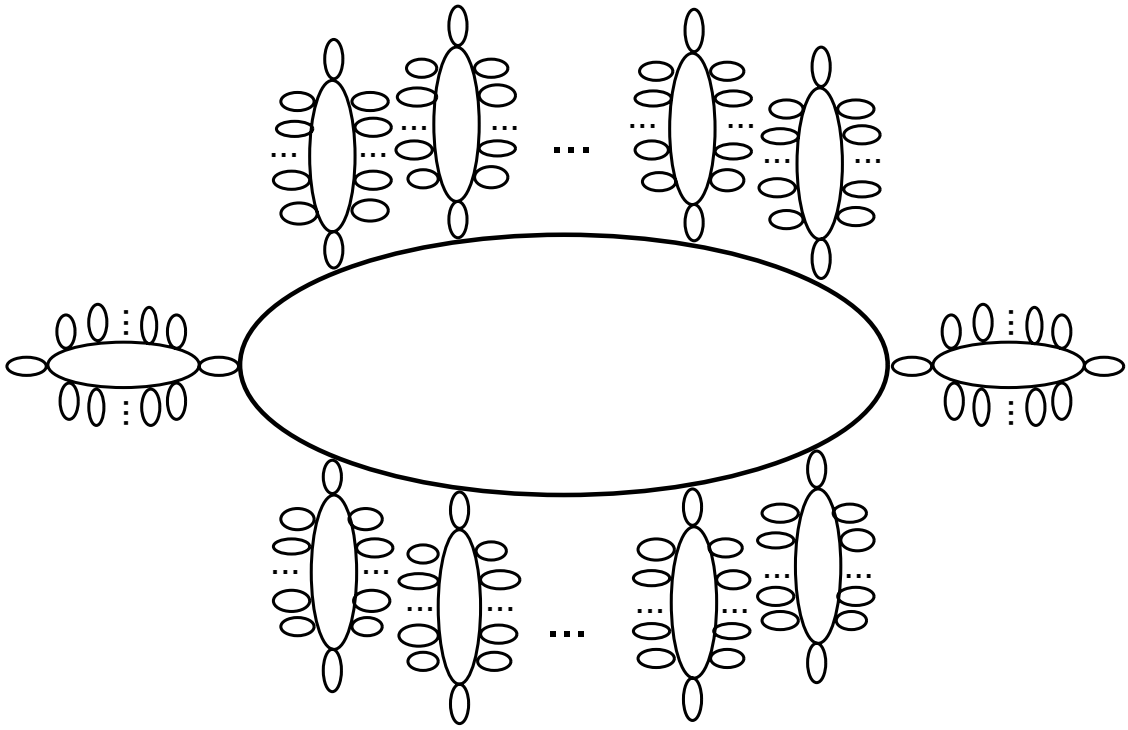


Abb. 9. Die 2-Ebenen-Hierarchie des KSR 1 der Firma Kendall Square

3.1.4. Speicher-Allokation

In *virtual-shared-memory*-Systemen beschreibt die Allokation die Anordnung der Daten im virtuellen Adreßraum; Prozesse, die Daten im virtuellen Adreßraum plazieren wollen, müssen jeweils Teile des virtuellen Adreßraums explizit allokieren. Kooperierende Prozesse, die auf einen logischen Adreßraum gemeinsam zugreifen, werden als *lightweight* Prozesse oder *threads* bezeichnet [SPG91]. In IVY und KOAN werden gemeinsame Adreßbereiche (*shared regions*) von einem Prozeß allokiert; allen *lightweight* Prozessen, die auf diesen Bereich zugreifen wollen, wird ein Zeiger übergeben [Li86, LaPr91]. Um dem Problem des *page thrashing* (siehe Abschnitt 3.1.2) entgegenzuwirken, beginnt jede Allokationsoperation auf einer neuen Seite. Ebenso kann für Daten mit unterschiedlichem Zugriffsmuster jeweils eine neue Seite verwendet werden; die dazu notwendigen Informationen muß der Compiler oder der Benutzer liefern [CoFo89]. Ähnlich zur Allokation werden Speicherbereiche explizit deallokiert, sobald sie nicht mehr benötigt werden.
Die dynamische Speicherallokation kann z.B. nach dem *boundary-tag*-Algorithmus implementiert werden, der mittels einer Liste freier Speicherbereiche *(free list)* eine *first-fit*- oder eine *best-fit*-Strategie realisieren kann [Knu73]. Die Informationen über den Status eines Speicherblocks (frei oder belegt, Größe) sind jeweils am Ende des Blocks angehängt; da die Speicherblöcke über die lokalen Speicher im VSM-System verteilt sind, sind auch die Statusinformationen verteilt, wodurch sich im ungünstigsten Fall für eine Allokation ebensoviele *page faults* ergeben können wie Speicherblöcke verfügbar sind. In [Li86] werden zwei Strategien zur dynamischen Speicherallokation in *virtual-shared-memory*-Systemen beschrieben, bei denen die Statusinformationen zentral verwaltet werden: Das einstufig-zentrale und das zweistufig-zentrale Speichermanagement.
Beim einstufig-zentralen Speichermanagement fungiert ein Prozessor, der zentral über

die Statusinformationen aller Speicherblöcke verfügt, als zentraler Allokator beziehungsweise Deallokator; alle anderen Prozessoren führen *remote*-Operationen aus, d.h. in einer *Message* wird eine entsprechende Anfrage an den zentralen Allokator geschickt. Diese Strategie erzeugt keine *page faults*, ein Effizienzproblem ergibt sich jedoch aus den *remote*-Operationen.
Beim zweistufig-zentralen Speichermanagement werden ebenfalls vom zentralen Allokator Speicherblöcke zugewiesen; es werden jedoch jeweils große Blöcke bereitgestellt, die von einem lokalen Allokator verwaltet werden, der wiederum mit *boundary tags* arbeitet. Durch diese Strategie wird die Anzahl der *remote*-Operationen reduziert, da die meisten Allokationen lokal erfolgen. Ein Nachteil gegenüber dem einstufig zentralen Speichermanagement besteht in der schlechteren Speichereffizienz.

Die Allokation hat keinen direkten Einfluß auf die Eignung von Scheduling-Algorithmen. Die Allokationsstrategie beeinflußt lediglich den Aufwand und damit die Geschwindigkeit von Allokation und Deallokation, was bei der Kreation beziehungsweise Terminierung von Prozessen zu entsprechenden Verzögerungen führt.

3.1.5. Speicher-Mapping

Das Speicher-Mapping beschreibt die Abbildung des virtuellen Adreßraums auf den physikalischen Adreßraum. Das Mapping kann sowohl Seiten-orientiert als auch Objekt-orientiert sein (siehe Abschnitt 3.1.1). Der Zugriff auf die Daten erfolgt unabhängig vom Mapping und ist prinzipiell sowohl Seiten-orientiert als auch Objekt-orientiert möglich. Die Größe einer Zugriffseinheit (Seite oder Objekt) kann einen entscheidenden Einfluß auf die Systemleistung haben: In *distributed-memory*-Rechnern ist das Senden von großen Datenpaketen (ca. 1000 Bytes) nicht wesentlich teurer als das Senden von kleinen Paketen (ca. 10 Bytes). Dadurch sind in VSM-Systemen grundsätzlich große Zugriffseinheiten möglich. Mit wachsender Größe von Seiten oder Objekten steigt jedoch auch die Wahrscheinlichkeit für Speicherzugriffskonflikte *(memory contention)*; Speicherzugriffskonflikte treten dann auf, wenn z.B. zwei Prozessoren gleichzeitig auf dieselbe Dateneinheit schreiben wollen. Um solche Konflikte zu vermeiden, sollte die Größe einer Zugriffseinheit relativ klein gewählt werden. Zu kleine Zugriffseinheiten erfordern jedoch große Tabellen zur Verwaltung der Zugriffsrechte, was mit großem Overhead bei der Behandlung von *page faults* verbunden ist. Die ideale Größe einer Zugriffseinheit wird also von mehreren Faktoren bestimmt. Ein Objekt bezeichnet eine logisch zusammenhängende Datenstruktur von unbestimmter Größe und ist daher als Zugriffseinheit ungeeignet. Aus diesem Grund erfolgt der Zugriff in allen VSM-Ansätzen Seiten-orientiert.
Die ideale Seitengröße hängt außerdem von den Datenstrukturen der jeweiligen Anwendung ab. In realen VSM-Systemen kann die Seitengröße durch die Hardware-Unterstützung einer Speichermanagementeinheit (MMU) physikalisch vorgegeben sein. Eine MMU ist für die Unterstützung von klassischen Virtualspeichersystemen konzipiert und implementiert die Verwaltung einer Seitentabelle. In einigen *virtual-shared-memory*-Systemen beträgt die Seitengröße typischerweise 1 oder 4 KByte. In anderen VSM-Systemen werden größere oder kleinere Seiten benutzt, z.B. 8 KByte in dem Netzwerk-VSM-System Clouds [RaKh91], 128 Byte im KSR 1 von Kendall Square [Dun92] oder 32 Byte im Cray T3D [KeSc93].

3.1.6. Seitenersetzung

Beim Auftreten eines *page fault* muß der anfragende Knoten für den Empfang einer Kopie der Seite den dazu notwendigen Speicherplatz in Form eines freien Seitenrahmens zur Verfügung stellen. Wenn alle Rahmen bereits belegt sind, wird mittels eines Seitenersetzungsmechanismus eine Seite ausgewählt, die an anderer Stelle zwischengespeichert oder gelöscht wird, um so den notwendigen Rahmen für die neue Seite zur Verfügung zu stellen. Da keine verläßlichen Vorhersagen über den zukünftigen Zugriff auf die Seiten getroffen werden können, existiert für die Seitenersetzung kein idealer Algorithmus. In VSM-Systemen werden ähnliche Algorithmen wie beim traditionellen Konzept des virtuellen Speichers verwendet. Ein Algorithmus ist in [LiSc89a] beschrieben: Für verschiedene Seitentypen (*read-write*, *owned read-only*, *read-only*) werden Prioritäten vergeben; innerhalb einer Prioritätsklasse wird nach der *Least-Recently-Used*-Strategie (LRU) verfahren. Die höchste Priorität bei der Ersetzung haben *read-only*-Seiten, da für die Ersetzung nur eine *Message* zur Benachrichtigung des Besitzers notwendig ist. *Owned-read-only*-Seiten haben höhere Priorität als *read-write*-Seiten, da die Ersetzung einer *owned-read-only*-Seite nur den Transfer des Besitztums erfordert, während bei der Ersetzung einer *read-write*-Seite neben dem Besitztum auch der Inhalt der Seite transferiert werden muß. Eine Technik zur Reduzierung der bei *page faults* entstehenden Wartezeiten aufgrund von Seitenersetzungen arbeitet mit Pegelmarken für die Anzahl unbenutzter Rahmen in einem Knoten. Wenn die CPU *idle* ist und die Anzahl unbenutzter Rahmen unter einer unteren Pegelmarke liegt, werden Seitenersetzungen vorgenommen, bis eine obere Pegelmarke erreicht wird.

3.2. Prozeßverwaltung

In Multiprozessor-Systemen hat die Prozeßverwaltung einen entscheidenden Einfluß auf die Ausführungszeit einzelner Programme sowie auf den Systemdurchsatz. Der Status von Prozessen wird typischerweise durch einen Wechsel zwischen der Ausführung und dem Warten auf Systemressourcen charakterisiert. Eine effiziente Nutzung der für einen Prozessor entstehenden Wartezeit für die Ausführung eines anderen Prozesses erfordert die gleichzeitige Existenz von mehreren Prozessen auf jedem Rechnerknoten.
Eine effiziente Prozeßverwaltung wird in *virtual-shared-memory*-Systemen im wesentlichen von zwei Effekten beeinflußt, dem Kontextwechsel *(context switch)* und der Prozeßmigration: Die Unterbrechung der Prozeßausführung durch die Behandlung von *page faults* ermöglicht durch eine Überlappung von Prozeßausführung und *page-fault*-Behandlung eine Verbesserung der Prozessorauslastung. Eine Voraussetzung für diese Überlappung ist die Möglichkeit schnelle Kontextwechsel zwischen Prozessen durchzuführen. Die zum effizienten Betrieb von VSM-Systemen notwendige Datenlokalität stellt spezielle Anforderungen an die Plazierung von Prozessen, die auf gemeinsame Daten zugreifen. Neben geeigneten Algorithmen zur Plazierung von Prozessen zum Zeitpunkt der Prozeßerzeugung ist zur dynamischen Plazierung die Migration von Prozessen eine wesentliche Problemstellung. Die Prozeßverwaltung wird durch drei Teile beschrieben, die Prozeßkontrolle, die Prozeßsynchronisation und das Prozeß-Scheduling.

3.2.1. Prozeßkontrolle

Die Prozeßkontrolle umfaßt die Erzeugung und die Terminierung von Prozessen. In IVY [Li86] wird mithilfe von Primitiven das *fork*/*join*-Paradigma unterstützt [DeHo66]. Eine Technik zur schnellen Erzeugung von Prozessen ist dabei die Initialisierung von freien Prozeß-Kontroll-Blöcken (PCBs) bei Programmstart (z.B. Initialisierung des *Stack*). Bei der Erzeugung eines Prozesses werden dann nur noch die notwendigen Werte (z.B. *program counter, stack pointer*) im PCB gesetzt, und der PCB wird in die *ready*-Warteschlange eingefügt. Für die Erzeugung eines Prozesses wird eine Zeitdauer angegeben, die der von einigen wenigen Prozeduraufrufen (*procedure calls*) entspricht [Li86]. Während die Prozeßerzeugung nur mit geringem Aufwand verbunden ist, erfordert die Migration von Prozessen erheblich mehr Aktivitäten (Transfer von Register- und *Stack*-Inhalten). Wenn ein Prozeß auf einem anderen Prozessor gestartet werden soll, kann durch die Erzeugung des Prozesses auf einem anderen Prozessor *(remote process creation)* der bei der Migration entstehende Overhead vermieden werden.

3.2.2. Prozeßsynchronisation

Bei der Koordinierung von parallelen Prozessen ergeben sich zwei unterschiedliche Aspekte der Prozeßsynchronisation, die Zugriffskontrolle und die Ablaufkontrolle [AlGo89]. Die Zugriffskontrolle soll den exklusiven Zugriff von Prozessen auf gemeinsame Daten gewährleisten (*mutual exclusion* [AnSc83]). Die Ablaufkontrolle bezieht sich auf die Reihenfolge von Ereignissen; die Ausführung von Prozessen kann solange unterbrochen werden, bis ein bestimmtes Ereignis eingetreten ist. Verbreitete Synchronisationsmechanismen sind die Semaphor-Synchronisation [Dij68], die Monitor-Synchronisation [Hoa74], die *signal*/*wait*-Synchronisation [AnSc83], die *eventcount*-Synchronisation [ReKa79] sowie die *event*- und die *barrier*-Synchronisation [Cra0222]. Voraussetzung für die korrekte Durchführung einer Synchronisation ist der exklusive Zugriff auf gemeinsame Variablen. Zu diesem Zweck werden unteilbare beziehungsweise nicht-unterbrechbare Operationen *(atomic operations)* verwendet, z.B. *Test-and-Set, Compare-and-Swap* oder *Fetch-and-Add* [And91, Her90, Got84]. Die *Test-and-Set*-Operation ist als blockierende oder *spin-waiting*-Operation von den beiden anderen als nicht-blockierende Operationen zu unterscheiden: Die *Test-and-Set*-Operation wird solange ausgeführt, bis auf die entsprechende Synchronisationsvariable zugegriffen werden kann. Dieses Verhalten wird mit *spin waiting*, *spinning* oder *busy waiting* bezeichnet. Die beiden nicht-blockierenden Operationen werden nur einmal ausgeführt, und bei gesperrtem Zugriff auf die Synchronisationsvariable kann der Prozeß seinen Prozessor zugunsten anderer Prozesse abgeben [Nag90e].
In *virtual-shared-memory*-Systemen können Mechanismen zur Prozeßsynchronisation sowohl auf der Basis des *shared memory* als auch *message-passing*-basiert implementiert werden. Bei der *shared-memory*-basierten Implementierung wird die zur Synchronisation notwendige Datenstruktur im *virtual-shared-memory*-Adreßraum gespeichert. Für den Zugriff auf eine lokal nicht vorhandene Datenstruktur muß dann mindestens eine Seite transferiert werden. Bei der Synchronisation mithilfe von *Messages* wird die Synchronisations-Datenstruktur im privaten Speicherbereich eines Knotens abgelegt. In einigen *virtual-shared-memory*-Ansätzen ist der Benutzeradreßraum zweigeteilt, in den

VSM-Bereich und in den privaten Bereich. In privaten Speicherbereichen werden zum Beispiel Programm-Code und Prozeß-Kontroll-Blöcke gespeichert. Die Ausführung von Synchronisationsoperationen mit nicht-lokalen Synchronisations-Datenstrukturen erfolgt in Form von *remote procedure calls* (RPCs). Eine Datenstruktur kann entweder statisch jeweils einem Knoten zugeteilt oder auf Anfrage migriert werden. Die dynamische Verteilung mittels Migration ist dabei aufgrund der größeren Lokalität bei Synchronisationen der statischen Verteilung überlegen. Die Effizienz von *shared-memory*- beziehungsweise *message-passing*-Implementierungen der Synchronisation hängt maßgeblich von der Hardware sowie von der jeweiligen VSM-Implementierung ab: In IVY hat sich die *shared-memory*-Synchronisation als effizienter erwiesen [Li88]. Für das auf einem iPSC/2-Hypercube implementierte VSM-System Shiva ist dagegen die Verwendung von *Messages* zur Synchronisation vorteilhaft [LiSc89b]. Die Ursache für die bessere Effizienz der *message-passing*-Synchronisation in Shiva liegt in der geringen Verzögerung für den Transfer von kurzen Nachrichten in Verbindung mit der Seitengröße von 4 KByte.

3.2.3. Prozeß-Scheduling

Im Bereich der *distributed-memory*-Rechner ergeben sich für *virtual-shared-memory*-Systeme besondere Möglichkeiten für das Prozeß-Scheduling. Nach dem *message-passing*-Programmiermodell arbeiten die Prozesse in verschiedenen Adreßräumen. Die Verwaltung der verschiedenen Adreßräume bedingt hohe Kosten für die Prozeßmigration und stellt so ein wesentliches Problem für das Prozeß-Scheduling dar. In VSM-Systemen bezieht sich die Migration auf Prozesse innerhalb eines Adreßraums; dadurch ergeben sich geringere Kosten für die Migration. Der Kontextwechsel beim Auftreten eines *page fault* stellt eine weitere Besonderheit für das Scheduling in *virtual-shared-memory*-Systemen dar.

3.2.3.1 Prozeßmigration

In VSM-Systemen stellt die Datenlokalität besondere Anforderungen an die Plazierung von Prozessen. Wenn zwei oder mehr Prozesse auf verschiedenen Prozessoren häufig auf die gleiche Seite schreiben müssen (*page thrashing*), dann kann die Ausführung der Prozesse auf einem Prozessor schneller sein als die Ausführung auf verschiedenen Prozessoren, da die Zeit für den häufigen Transfer einer Seite die Ausführungszeit unter Umständen deutlich dominieren kann. Die Prozeßsynchronisation kann einen ähnlichen Einfluß auf die Ausführungszeit haben: Wenn z.B. ein Prozeß auf die Terminierung eines anderen Prozesses wartet, dann ergeben sich für die Benachrichtigung eines wartenden Prozesses auf demselben Prozessor geringere Kosten als für eine Nachricht an einen Prozeß auf einem anderen Prozessor. Neben der Gruppierung von Prozessen auf einem Prozessor kann auch die Gruppierung auf mehreren benachbarten Prozessoren von Vorteil sein *(clustering)*. Sowohl das *page thrashing* als auch die *remote*-Operationen z.B. aufgrund der Synchronisation können durch geeignete Strategien für die Prozeßmigration vermieden oder zumindest begrenzt werden. Dabei ist zu beachten, daß die Reduzierung von *page thrashing* und *remote*-Operationen durch die Migration von Prozessen einer optimalen Lastbalancierung entgegenwirken kann.
In [Li86] sind Scheduling-Strategien für ein zentrales und für ein verteiltes Scheduling

beschrieben. Das zentrale Scheduling kann mittels einer zentralen *ready*-Warteschlange oder mittels einer Hierarchie einer zentralen *ready*-Warteschlange in Kombination mit lokalen *ready*-Warteschlangen auf jedem Knoten implementiert werden. Für skalierbare Systeme stellt die zentrale Warteschlange einen Engpaß dar. Beim verteilten Scheduling existieren nur lokale Warteschlangen: Auf jedem Prozessor existiert ein lokaler Scheduler mit zugehöriger lokaler *ready*-Warteschlange, das Scheduling innerhalb einer lokalen *ready*-Warteschlange erfolgt nach der *Last-In-First-Out*-Strategie. Die lokalen Scheduler kommunizieren miteinander, um geeignete Prozeßmigrationen durchführen zu können. Die ausgetauschten Informationen sind dabei z.B. die Anzahl der Prozesse und die Anzahl der *ready*-Prozesse. Der Informationsaustausch erfolgt durch das Anhängen von Extra-Bits an *Messages*, die aufgrund der Seitenverwaltung ausgetauscht werden, und erfordert somit nur geringen Overhead.
Die beschriebenen Scheduling-Strategien sind die aktive und die passive Lastbalancierungsstrategie sowie die *page-demand*-Lastbalancierungsstrategie. Aktiv bezieht sich dabei auf die aktive Anfrage von Prozessoren mit vielen Prozessen in ihrer *ready*-Warteschlange (überlastete Prozessoren), die bei einem anderen Prozessor um dessen Mitarbeit anfragen. Wenn der andere Prozessor zur Mitarbeit bereit ist, werden Prozesse migriert, andernfalls wird die Anfrage zurückgewiesen. Bei der passiven Lastbalancierungsstrategie warten überlastete Prozessoren auf eine Migrationsanfrage eines Prozessors mit wenigen Prozessen. Eine solche Anfrage wird von überlasteten Prozessoren akzeptiert, von den übrigen Prozessoren wird die Anfrage zurückgewiesen. Die *page-demand*-Lastbalancierungsstrategie ist speziell für die Prozeßmigration beim Auftreten von *page thrashing* konzipiert: Die Migration eines Prozesses wird insbesondere dann vorgenommen, wenn *page thrashing* auftritt. Eine Prozeßmigration zur Vermeidung von *page thrashing* und damit zur Vermeidung von *page faults* kann die Effizienz steigern, da die Kosten für die Prozeßmigration etwa denen für die Seitenmigration bei der Auflösung eines *page fault* entsprechen. Zur Erkennung des *page thrashing* wird bei jedem *page fault* eine Prozedur aufgerufen, die für jede Seite die Anzahl und die Zeitpunkte der *page faults* ermittelt; *page thrashing* wird definiert für das Auftreten von *k page faults* auf eine Seite innerhalb eines Zeitintervalls. Beim Erkennen von *page thrashing* während eines *page fault* kann der auslösende Prozeß zu dem Prozessor migriert werden, der die entsprechende Seite besitzt. Auf diese Weise kann ein *page thrashing* für weitere Zugriffe des migrierten Prozesses auf die Seite vermieden werden. Neben der Reduzierung des *page thrashing* kann diese Strategie jedoch auch ein neues *page thrashing* verursachen, wenn der migrierte Prozeß auf Seiten schreiben muß, die von anderen Prozessen auf seinem ursprünglichen Prozessor benötigt werden. Für eine optimale Lösung sind Informationen über zukünftige Datenzugriffe aller Prozesse auf dem ursprünglichen Prozessor notwendig. Da diese Informationen nicht verfügbar sind, wird eine Entscheidung über die Migration aufgrund von Regeln getroffen, die diese Informationen nicht benötigen. Die Regeln beruhen auf einem Vergleich der Anzahl der Prozesse beziehungsweise der Anzahl der *ready*-Prozesse auf den beiden betroffenen Prozessoren. Dabei wird versucht, sowohl eine maximale Lastbalancierung zu erzielen als auch das *page thrashing* zu vermeiden.

3.2.3.2 Kontextwechsel

Eine wesentliche Voraussetzung für die Durchführung von Kontextwechseln ist, daß die Prozeßausführung unterbrechbar *(preemptive)* ist. Eine Unterbrechung eines Prozesses wird vorgenommen, wenn der Prozeß auf Ressourcen wartet (z.B. bei der Ein-/Ausgabe, bei der Prozeßsynchronisation oder bei einem *page fault*) oder bei einem Timeout aufgrund einer abgelaufenen Zeitscheibe. Eine Besonderheit für das Scheduling in VSM-Systemen ist dabei der Kontextwechsel beim Auftreten eines *page fault*. Wenn die Kosten für die Auflösung eines *page fault* höher sind als für einen Kontextwechsel, dann kann auf diese Weise der Systemdurchsatz gesteigert werden. Für den einzelnen Prozeß kann sich ein Kontextwechsel jedoch sehr nachteilig auswirken, wenn eine Seite, die aufgrund eines *page fault* für einen Prozeß in den lokalen Speicher kopiert worden ist, beim Wiederaufsetzen des Prozesses bereits nicht mehr lokal verfügbar ist. Eine Vermeidung dieses Effekts kann z.B. durch ein schnelles Wiederaufsetzen des Prozesses nach Bereitstellung der Seite in Verbindung mit dem Festhalten der Seite bis zu diesem Zeitpunkt erreicht werden. Kontextwechsel können grundsätzlich sowohl innerhalb eines Jobs als auch zwischen verschiedenen Jobs sinnvoll sein (Intra- bzw. Inter-Job-Kontextwechsel). Beim Intra-Job-Kontextwechsel ist jedoch zum Beispiel für parallele Schleifen anzunehmen, daß der Datenzugriff für verschiedene Prozesse von ähnlicher Struktur ist; bei einem Kontextwechsel aufgrund eines *page fault* würde dann mit hoher Wahrscheinlichkeit auch der nächste Prozeß ein *page fault* auslösen. Eine auf diese Weise ausgelöste Lawine von *page faults* kann in vielen Fällen zu ungünstigen Effekten führen. Für den Inter-Job-Kontextwechsel sind diese negativen Effekte generell nicht zu erwarten.

4. Relevante Algorithmen zur Ablaufplanung

Mit der Entwicklung und Anwendung neuartiger Supercomputer-Parallelrechner treten vielfach auch neuartige Probleme auf, die den effizienten Betrieb dieser Rechner beeinträchtigen. Ein Teil dieser Probleme wird durch die wachsende Komplexität bei der Organisation von Systemen mit immer mehr Prozessoren (massiv-parallele Systeme) erzeugt. Ein wesentliches Problem betrifft die Ablaufplanung der im Rechner zu bewältigenden Aufgaben mit dem Ziel einer möglichst effizienten Nutzung der verfügbaren Betriebsmittel; für die Zuordnung von Aufgaben zu Betriebsmitteln, zum Beispiel von Prozessen zu Prozessoren, wird der Begriff Scheduling verwendet. Die Komplexität des Scheduling hängt sowohl von den Eigenschaften des Rechners als auch von den Eigenschaften der Aufgaben ab, die einen Rahmen für die zu betrachtenden Scheduling-Algorithmen vorgeben. Die folgenden Eigenschaften führen aus Effizienzgründen zu Forderungen an das Scheduling, die die Komplexität des Scheduling erhöhen:

- massiv-paralleler Rechner
- *virtual-shared-memory*-Rechnermodell
- kommerzielle Nutzung / Multiprogramming-Betrieb

Für massiv-parallele Rechner kann ein zentraler Scheduler mit wachsender Prozessorzahl zum Engpaß werden [FeRu90]. Ein verteiltes Scheduling kann diesen Engpaß aufweiten, was jedoch eine Erhöhung der Scheduling-Komplexität bedeutet. Eine weitere Erhöhung der Komplexität aufgrund von massiver Parallelität ergibt sich aus der hierarchischen Architektur solcher Systeme: In massiv-parallelen Systemen sind im allgemeinen die Abstände zwischen Prozessor und Speicherelementen verschieden, d.h. die Systeme weisen zumindest für lokale beziehungsweise nicht-lokale Speicherzugriffe unterschiedliche Zugriffszeiten auf; das Prinzip der räumlichen Lokalität fordert die Abbildung von häufig kommunizierenden Prozessen auf räumlich lokale Bereiche, da die Kommunikation über größere Distanzen mit höheren Kosten verbunden ist [FeRu90, ZhBr91].
Für das *virtual-shared-memory*-Rechnermodell ergeben sich Abhängigkeiten zwischen den zu bewältigenden Aufgaben und damit zwischen den Prozessen, an die die Aufgaben verteilt werden: Ein Abhängigkeitsverhältnis basiert auf der Zugehörigkeit verschiedener Prozesse zu einem Job. Eine weitere Abhängigkeit ergibt sich aus der Zuordnung von Prozessen zu Schleifen: Bei der Ausführung paralleler Schleifen können mehrere Prozesse in einer Schleife arbeiten; diese Prozesse müssen im allgemeinen miteinander kommunizieren, und sie können der gemeinsamen Schleife zugeordnet werden. Beim Scheduling unabhängiger Prozesse wird davon ausgegangen, daß zwischen den Prozessen keine Interaktion stattfindet. Diese Annahme vereinfacht die Scheduling-Algorithmen; da die Annahme jedoch für das Scheduling von parallelen Programmen im allgemeinen nicht zutrifft, ermöglicht die Berücksichtigung von Abhängigkeiten in Form einer koordinierten Zuordnung ein effizienteres Scheduling [GTU91]. Das VSM-Modell stellt darüber hinaus Anforderungen an die Datenlokalität: Insbesondere der Reduzierung von *page faults* und der Vermeidung beziehungsweise Reduzierung des *page thrashing* durch geeignete Scheduling-Algorithmen kommt diesbezüglich eine besondere Bedeutung zu [Li86].

Die Untersuchung von Scheduling-Algorithmen für die Ausführung eines einzelnen Programms kann einen ersten Schritt für die Lösung des Scheduling-Problems auf VSM-Rechnern darstellen. Mit dem kommerziellen Einsatz von derartigen, massiv-parallelen Rechnern ergibt sich die Notwendigkeit des Multiprogramming-Betriebs, was demgegenüber eine Erhöhung der Scheduling-Komplexität bedeutet. Eine mögliche Lösung dieses Scheduling-Problems ist die räumlich statische Aufteilung eines Rechners, so daß das Scheduling für jeden Job unabhängig erfolgen kann; beim Eintreffen neuer Jobs erfordert das statische Scheduling eine vollständige Reorganisation aller im System befindlichen Jobs. Ein effizientes statisches Scheduling benötigt a priori Kenntnisse über das Eintreffen neuer Jobs, die in einer Multiprogramming-Umgebung jedoch im allgemeinen nicht zur Verfügung stehen [Prz92]. Ein dynamisches Scheduling erlaubt eine flexible Integration von neu in das System eintretenden Jobs. Das dynamische Scheduling verwendet bei der Integration von Jobs zum Beispiel Informationen über die Auslastung des Systems. Kommerzielle Hersteller wenden in massiv-parallelen Systemen bisher nahezu ausschließlich das statische Scheduling an, zum Beispiel im Intel Paragon [Int91] und im Cray T3D [KeSc93].

Die Rahmenbedingungen eines massiv-parallelen *virtual-shared-memory*-Rechners im Multiprogramming-Betrieb favorisieren ein verteiltes, dynamisches Scheduling abhängiger Prozesse unter Berücksichtigung der Forderungen nach räumlicher Lokalität und nach Datenlokalität. Im folgenden werden existierende Scheduling-Algorithmen kurz erläutert; dabei erfolgt keine vollständige Beschreibung aller Details, es soll vielmehr die Idee des jeweiligen Algorithmus verdeutlicht werden. Als relevante Algorithmen werden auch solche Algorithmen verstanden, die nur einen Teil der getroffenen Annahmen befriedigen. Da in den Untersuchungen in Kapitel 7 zu Vergleichszwecken auch einfache Scheduling-Algorithmen betrachtet werden, sind diese in die folgenden Beschreibungen einbezogen.

4.1. Grundlagen für das Scheduling

Für die Parallelisierung von Programmen existieren zwei Konzepte, die sich durch die Granularität der parallel auszuführenden Einheiten unterscheiden. Bezüglich des Parallelisierungskonzeptes ergeben sich unterschiedliche Ebenen für das Scheduling. Als Grundlage für die Beschreibung von Scheduling-Algorithmen dienen einige charakteristische Kriterien, die sowohl eine Einordnung als auch die Bewertung der Algorithmen ermöglichen. Für die Analyse von Scheduling-Algorithmen lassen sich verschiedene Basismodelle definieren, auf denen die Algorithmen aufbauen.

4.1.1. Parallelisierungskonzepte

Die Eigenschaften der von Schedulern zuzuordnenden Einheiten hängen von dem jeweiligen Parallelisierungskonzept ab. Der Begriff Prozeß bezeichnet einen Teil eines Programms, der sich in der Ausführung befindet [And91, SPG91]. Kooperierende Prozesse bearbeiten gemeinsame Daten und kommunizieren miteinander. Die Kommunikation erfolgt sowohl beim Datenaustausch (*message passing*) als auch bei der Synchronisation zur Sicherung der Datenintegrität und der Abarbeitungsreihenfolge von Programmteilen.

Bei den existierenden Multiprozessorsystemen lassen sich bezüglich der Granularität der zuzuordnenden Einheiten zwei Parallelisierungskonzepte unterscheiden; die grobgranulare und die feingranulare Parallelisierung.
Bei der grobgranularen Parallelisierung setzt sich ein Programm aus Teilaufgaben *(tasks)* zusammen, die zum Teil parallel zueinander ausgeführt werden können. Ein *task*-Scheduler bildet die *tasks* auf Prozesse ab. Die Prozesse werden durch den Job-Scheduler den Prozessoren zugeordnet. Die Aufspaltung des Scheduling in *task*- und Job-Scheduling entspricht einer Aufteilung des Zuordnungsvorgangs auf verschiedene Ebenen: Während der Job-Scheduler auf Betriebssystemebene eine koordinierte Zuordnung von Prozessen zu Prozessoren vornimmt, kann der *task*-Scheduler auf einer darunterliegenden Ebene fungieren (zum Beispiel als *library*-Scheduler auf der Job-Ebene [Cra0222, IBM23]). Die Zuordnungsvorgänge auf der Ebene des *task*-Schedulers können im Vergleich zur Betriebssystemebene wesentlich effizienter erfolgen. Ungeachtet der möglichen Vorteile durch die Aufteilung des Zuordnungsvorgangs auf verschiedene Ebenen behandeln viele Algorithmen nur die Ebene des Job-Scheduling. Im einfachsten Fall wird das *task*-Scheduling durch eine eins-zu-eins-Abbildung von *tasks* auf Prozesse ersetzt; in diesem Fall erfolgen alle Zuordnungsvorgänge auf der Betriebssystemebene.
Auch bei der feingranularen Parallelisierung ergibt sich eine Aufteilung des Scheduling auf verschiedene Ebenen. Auf Betriebssystemebene erfolgt wiederum die Zuordnung von Prozessen zu Prozessoren durch den Job-Scheduler. Ein Programm besteht aus Blöcken, die nacheinander abgearbeitet werden. Eine *task* stellt den privaten Kontext (*Stack*, Programmzähler, usw.) für die Ausführung eines Blocks bereit. Jede *task* ist einem Prozeß zugeordnet und beschreibt dessen Zustand. Einem Prozeß können auch mehrere *tasks* zugeordnet werden. Die Zuordnung von *tasks* zu Prozessen erfolgt durch den *task*-Scheduler. Ein sequentieller Block wird von einer *task* ausgeführt. Ein paralleler Block kann von einer oder von mehreren *tasks* ausgeführt werden. Im letzteren Fall werden *sub-tasks* erzeugt, die zusammen mit der ursprünglichen *(master-)task* mehrere private Kontexte zur Verfügung stellen und so die Ausführung eines parallelen Blocks durch mehrere Prozesse ermöglichen. Ein *thread* entspricht einem Programmstück und repräsentiert nur den Programm-Code, d.h. eine Sequenz von Instruktionen, nicht jedoch den Kontext. Innerhalb einer parallelen Schleife zum Beispiel entspricht der Programm-Code einer Iteration einem *thread*. Eine Gruppe von *threads* wird als *chunk* bezeichnet. Der *thread*-Scheduler ordnet die *threads* eines parallelen Blocks den entsprechenden *tasks* unteilbar zu.
Auch bei der feingranularen Paralleliserung kann das *task*-Scheduling im einfachsten Fall durch eine eins-zu-eins-Abbildung realisiert werden (zum Beispiel bei der Cray Autotasking Implementierung [Cra3074]). Ein wesentlicher Unterschied zwischen dem *thread*-Scheduling und dem *task*-Scheduling ergibt sich für den Scheduling-Overhead: Auf der einen Seite erfordert die Verwaltung von *threads* einen wesentlich geringeren Aufwand als die Verwaltung von *tasks*: Da im allgemeinen mehrere *threads* innerhalb eines Kontextes ausgeführt werden, ist das Scheduling von *threads* nur mit geringem Overhead verbunden. Auf der anderen Seite besitzen *threads* im allgemeinen eine feinere Granularität als *tasks*, so daß der *thread*-Scheduler wesentlich häufiger aufgerufen wird.

4.1.2. Beschreibungskriterien

Die in dieser Arbeit betrachteten Scheduling-Algorithmen beschreiben eine dynamische Zuordnung von Prozessen zu Prozessoren. Zu den wesentlichen Kriterien für die Beschreibung dieser Algorithmen gehören die Strategien für die Zuteilung und die Abgabe von Prozessoren. Einfache Strategien für die Zuteilung von Prozessoren an Prozesse in einer Warteschlange sind zum Beispiel *First-In-First-Out* (FIFO), *Last-In-First-Out* (LIFO) und *Random* (zufällige Auswahl) [SPG91]. Bei der Abgabe von Prozessoren wird zwischen unterbrechenden *(preemptive)* und nicht-unterbrechenden *(nonpreemptive)* Strategien unterschieden. Bei einer nicht-unterbrechenden Strategie behält ein Prozeß den Prozessor bis zu seiner Fertigstellung. Bei einer unterbrechenden Strategie kann ein Prozeß vom Scheduler unterbrochen werden; der Prozessor kann dann einem anderen Prozeß zugeteilt werden, und der unterbrochene Prozeß wird zum Beispiel in eine Warteschlange eingereiht.
Für die Bewertung von Scheduling-Algorithmen sind die Effizienz für die Ausführung einer Arbeitslast und die Fairness bei der Behandlung von Prozessen beziehungsweise Jobs entscheidend. Als Maß für die Effizienz werden häufig die mittlere Antwortzeit der Jobs und der Systemdurchsatz verwendet. Zusätzlich kann die Häufigkeit der Kontextwechsel als Maß für den Scheduling-Overhead verwendet werden. Bezüglich der Fairness kann sowohl eine Gleichbehandlung als auch eine Bevorzugung erwünscht sein.

4.1.3. Basismodelle

Eine Analyse verschiedener Scheduling-Algorithmen kann aufgrund der bisher beschriebenen Grundlagen für das Scheduling erfolgen. Ein bisher nicht betrachteter Aspekt betrifft die Infrastruktur der Kontrollelemente und -pfade, auf der die Algorithmen basieren und die bei verschiedenen Algorithmen zum Teil identisch ist. In der Literatur existieren vereinzelt Definitionen für Modelle solcher Infrastrukturen, eine vollständige Kategorisierung verschiedener Modelle existiert jedoch nicht. Um die Analyse von Scheduling-Algorithmen weiter zu strukturieren, werden im folgenden verschiedene Basismodelle definiert, die eine Infrastruktur für das Design von Scheduling-Algorithmen beschreiben. Die vorliegende Kategorisierung basiert auf [FeRu90].

1. Das *Self-service-* oder *Load-Sharing*-Modell implementiert eine zentrale *ready*-Warteschlange. Die Verteilung der Prozesse erfolgt durch aktive Anfragen von *idle*-Prozessoren, denen jeweils ein Prozeß aus der Warteschlange zugeteilt wird. Ein Vorteil dieses Modells liegt in der gleichmäßigen Verteilung der Arbeitslast über die Prozessoren, so daß bei der Existenz von *ready*-Prozessen keine Prozessoren *idle* sein können. Zu den Nachteilen gehört die zentrale *ready*-Warteschlange, die in massiv-parallelen Systemen zum Engpaß werden kann. Ein weiterer Nachteil betrifft die Lokalität der Programmausführung. Da ein unterbrochener Prozeß nur mit geringer Wahrscheinlichkeit wieder auf demselben Prozessor gestartet wird, können lokal vorhandene Daten wie zum Beispiel der *Cache*-Inhalt nicht wiederverwendet werden. Ein wesentlicher Nachteil ist die fehlende Koordinierung bei der Zuteilung von Prozessoren, die keine Berücksichtigung von *task*-Abhängigkeiten zuläßt.

2. Beim *Central-Pool*-Modell für ein zentrales Scheduling wird die Vergabe von Prozessoren über einen zentralen Prozessor-Pool gesteuert. Die Prozessoren werden den Jobs von einem *Master* zentral zugeteilt. Durch die zentrale Kontrolle kann die Zuteilung von Prozessoren koordiniert erfolgen, es können zum Beispiel *task*-Abhängigkeiten berücksichtigt werden. Einen wesentlichen Nachteil in Form eines Engpasses stellt die zentrale Kontrolle jedoch für massiv-parallele Systeme dar. In Realisierungen dieses Modells sind die Scheduler nicht-unterbrechend; diesbezüglich wird dieses Modell auch mit *Nonpreemptive Mapping* bezeichnet.
3. Das *Local-Queue*-Modell ist die einfachste Realisierung für ein verteiltes Scheduling und wird durch die Verwendung von lokalen *ready*-Warteschlangen charakterisiert: Auf jedem Knoten existiert ein lokaler Scheduler, der unabhängig eine eigene lokale *ready*-Warteschlange verwaltet (siehe Abb. 10). Eine effiziente Lastbalancierung ist für dieses Modell im allgemeinen nur durch einen zusätzlichen Laufzeitmechanismus zu gewährleisten, der Prozesse von überlasteten Prozessoren zu *idle*-Prozessoren migriert. Ein Beispiel für dieses Modell ist der in Abschnitt 3.2.3 beschriebene verteilte Scheduler des IVY-Systems [Li86].
4. Beim *Multiple-Pool*-Modell wird ein verteiltes Scheduling durch die Partitionierung eines Multiprozessorsystems in mehrere Prozessor-Pools realisiert; jeder Prozessor wird genau einem Prozessor-Pool zugeordnet. Jeder Pool verfügt über eine *ready*-Warteschlange, die von einem lokalen Scheduler unabhängig verwaltet wird. Ein

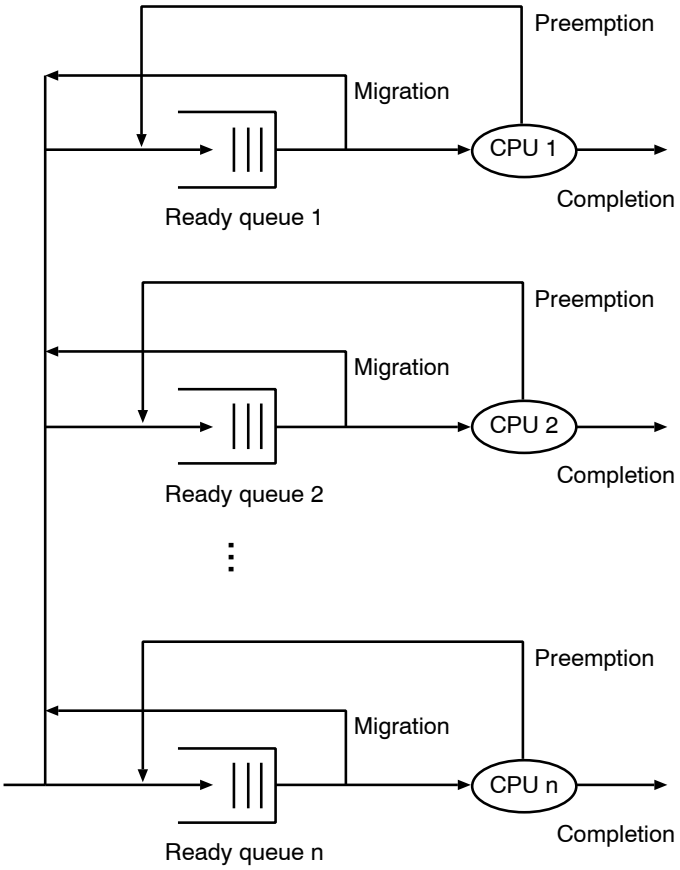


Abb. 10. Ein verteiltes Scheduling nach dem Local-Queue-Modell [Li86]

Vorteil dieses Modells kommt durch die Zuordnung von abhängigen *tasks* zu einem Pool zum Tragen; auf diese Weise kann die erforderliche Kommunikation auf räumlich lokale Bereiche begrenzt werden. Untersuchungen zu diesem Modell sind in [ZhBr91] beschrieben.

5. Das *Hierarchical-Control*-Modell für ein verteiltes Scheduling wird durch die Verwendung von hierarchischen Kontrollstrukturen charakterisiert. Die Zuteilung von Prozessoren erfolgt ähnlich wie beim *Central-Pool*-Modell durch einen *Master*, der im *Hierarchical-Control*-Modell jedoch selbst verteilt ist. Eine Baum-förmige Kontrollstruktur dient der Verwaltung der Prozessoren (siehe Abb. 11). Die Blätter in der unteren Ebene des Baumes repräsentieren die Prozessoren. Die Knoten der nächsthöheren Ebene verwalten jeweils eine Gruppe der Prozessoren, zum Beispiel zwei Prozessoren für einen Binärbaum. In den höherliegenden Ebenen werden entsprechend der Baumstruktur die Prozessorgruppen der darunterliegenden Ebenen zu größeren Gruppen zusammengefaßt. Die Wurzel des Baumes verwaltet schließlich alle Prozessoren als eine Gruppe. Mit diesem Modell wird der Vorteil einer koordinierten Kontrolle genutzt, d.h. es können *task*-Abhängigkeiten berücksichtigt werden. Im Gegensatz zum *Central-Pool*-Modell stellt die koordinierte Kontrolle jedoch keinen Engpaß dar, da die hierarchische Kontrollstruktur durch eine Abbildung auf mehrere Prozessoren verteilt verwaltet wird. Beispiele für dieses Modell sind das *Wave Scheduling* und ein verteiltes *Gangscheduling* mit hierarchischer Kontrolle (siehe Abschnitt 4.3, [TiWi84, FeRu90]).

Die beschriebenen Basismodelle stellen nur eine Auswahl dar. Dem Entwurf neuer Scheduling-Algorithmen liegen oft auch neue Basismodelle zugrunde. Für ein Basismodell können im allgemeinen verschiedene Scheduling-Algorithmen implementiert werden; zum Teil sind die Scheduling-Algorithmen jedoch so eng mit dem Basismodell verknüpft, daß sie durch das Basismodell bereits nahezu vollständig bestimmt sind (siehe *Pool-based Scheduling*, Abschnitt 4.3). Diesbezüglich ist die Beschreibung der Basismodelle nicht in jedem Fall orthogonal zu der folgenden Beschreibung der Scheduling-Algorithmen; in

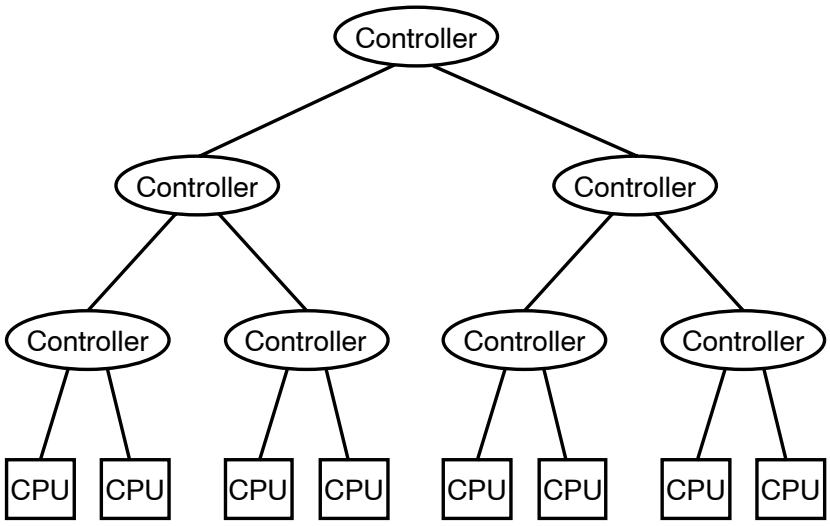


Abb. 11. Die Baumstruktur der Kontrollhierarchie [FeRu90]

den meisten Fällen erlaubt die Unterscheidung zwischen Basismodell und Scheduling-Algorithmus jedoch eine differenziertere Betrachtungsweise.

4.2. Scheduling-Algorithmen für unabhängige Tasks

Einfache Scheduling-Algorithmen für das Scheduling unabhängiger *tasks* sind zum Beispiel *Run-To-Completion*, *Round-Robin* und *Priority-based Scheduling*. Der *Prozessorabgabe*-Algorithmus stellt nur eine Detaillösung dar und kann mit anderen Scheduling-Algorithmen kombiniert werden. Die genannten Algorithmen behandeln das Job-Scheduling und spezifizieren die Prozessorzuteilung beziehungsweise -abgabe für die Verwaltung von *ready*-Warteschlangen unabhängig von einem Basismodell; sie können zum Beispiel für die Modelle *Self-service*, *Central Pool*, *Local Queue* und *Multiple Pool* realisiert werden. Im Sinn eines verteilten Scheduling können diese Algorithmen für lokale Scheduler eingesetzt werden, die zum Beispiel nach dem *Local-Queue*-Modell arbeiten. Eine Interaktion zwischen den lokalen Schedulern tritt dann nur bei der Migration von Prozessen auf.

- **Run-To-Completion (RTC):**
  Der *Run-To-Completion*-Algorithmus implementiert ein nicht-unterbrechendes Scheduling, d.h. ein einmal gestarteter Prozeß wird ohne Unterbrechung vollständig bearbeitet [ZaMc90]. Der RTC-Algorithmus spezifiziert nur die Abgabe von Prozessoren, die Prozessorzuteilung kann wahlweise zum Beispiel nach der FIFO-, LIFO- oder *Random*-Strategie erfolgen. Für eine Prozessorzuteilung nach der FIFO-Strategie wird dieser Algorithmus auch als *First-Come-First-Serve*-Algorithmus bezeichnet [MEB88].

- **Round-Robin (RR):**
  Der *Round-Robin*-Algorithmus realisiert ein unterbrechendes Scheduling, d.h. ein Prozessor wird jeweils nur für ein festes Zeitinterval (Zeitscheibe) zugeteilt [MEB88]. Nach Ablauf der Zeitscheibe wird der jeweilige Prozeß unterbrochen, und der Prozessor muß abgegeben werden. Ist ein Prozeß nach dem Ablauf der Zeitscheibe noch nicht vollständig bearbeitet, so wird er erneut in die *ready*-Warteschlange eingereiht. Nach dem RR-Algorithmus wird die verfügbare Prozessorzeit gleichmäßig über die Prozesse verteilt, d.h. der Algorithmus ist fair bezüglich der Prozesse; da die Anzahl der Prozesse pro Job verschieden sein kann, ergibt sich hier jedoch keine Fairneß bezüglich der Jobs[6]. Auch der RR-Algorithmus spezifiziert nur die Abgabe von Prozessoren, die Prozessorzuteilung kann wiederum zum Beispiel nach der FIFO-, LIFO- oder *Random*-Strategie erfolgen.

- **Priority-based Scheduling:**
  Nach dem *Priority-based-Scheduling*-Algorithmus werden den Prozessen in der *ready*-Warteschlange Prioritäten für die Zuteilung der Prozessoren zugewiesen. Die Berechnung der Prioritäten kann auf verschiedene Arten erfolgen: Die Priorität kann

[6] Dies gilt zum Beispiel bei einer Prozessorzuteilung nach der FIFO-Strategie.

zum Beispiel umgekehrt proportional zur bisherigen Prozessornutzung eines Prozesses sein; um kurze Prozesse dabei nicht zu stark zu bevorzugen, kann die bisherige Prozessornutzung mit der Zeit zum Beispiel exponentiell abgewertet werden [GTU91]. Bei einer Kombination des *Priority-based Scheduling* mit einem unterbrechenden Algorithmus für die Prozessorabgabe (*Round-Robin*) kann eine Unterbrechung entweder jeweils nach dem Ende einer Zeitscheibe oder, bei der Existenz eines Prozesses mit höherer Priorität, bereits vor dem Ende der Zeitscheibe erfolgen.

- **Prozessorabgabe:**
  Eine wesentliche Strategie für die Behandlung von Prozessen paralleler Jobs kann mit *Prozessorabgabe* bezeichnet werden. Die Strategie behandelt die Abgabe von Prozessoren durch Prozesse mit dem Ziel, durch eine günstigere Zuordnung den Systemdurchsatz zu verbessern. Die *Prozessorabgabe* beschreibt keinen vollständigen Scheduling-Algorithmus, sondern nur die Abgabe von Prozessoren in bestimmten Situationen. Die Strategie der *Prozessorabgabe* kann mit vielen Scheduling-Algorithmen sinnvoll kombiniert werden.
  Ein Beispiel für die Implementierung dieser Strategie ist das *Cray Autotasking Scheduling* [Cra3074, BKLR90]: Nach diesem Algorithmus geben die Prozesse eines parallelen Jobs ihren Prozessor ab, sobald sie für einen festen Zeitraum nicht an der Ausführung beteiligt waren. Beim Ablauf einer Zeitscheibe implementiert dieser Algorithmus eine Bevorzugung für Prozesse von parallelen Jobs: In diesem Fall wird dem Prozeß der Prozessor nicht sofort entzogen, sondern dem Prozeß wird die Möglichkeit gegeben, innerhalb eines festgelegten Zeitraums weitere Arbeit zu verrichten und den Prozessor zu einem günstigen Zeitpunkt freiwillig abzugeben. Eine ähnliche Strategie wird im folgenden Abschnitt durch den *Dynamic-Allocation*-Algorithmus beschrieben, der eine *Prozessorabgabe* unter Berücksichtigung von *task*-Abhängigkeiten realisiert.

4.3. Scheduling-Algorithmen für abhängige Tasks

Das Scheduling von abhängigen *tasks* bezieht sich bei existierenden Algorithmen im allgemeinen auf die Berücksichtigung der Jobzugehörigkeit von Prozessen. Für diesen Fall kann das Scheduling auf zwei Ebenen verteilt werden und mehr oder weniger unabhängig erfolgen: Auf einer oberen Ebene werden die Prozessoren den Jobs zugewiesen (Job-Scheduling); auf der darunterliegenden Ebene verwalten die Jobs ihre Prozessoren unabhängig und weisen sie den *tasks* beziehungsweise *threads* zu (*task*- beziehungsweise *thread*-Scheduling). Die im folgenden beschriebenen, für *virtual-shared-memory*-Systeme relevanten Scheduling-Algorithmen, sind auf massiv-parallele oder auf *shared-memory*-Systeme als Zielrechner zugeschnitten.

4.3.1. Scheduling in massiv-parallelen Systemen

Für ein dynamisches Scheduling abhängiger *tasks* in massiv-parallelen Systemen existieren bisher nur wenige Ansätze. Die Grundidee dieser Ansätze basiert auf dem *Coscheduling*-Algorithmus [Ous82]. In diesem Algorithmus werden zusammengehörige

(abhängige) *tasks* gruppiert, und die Gruppen *(task forces)* werden jeweils vollständig zugeordnet. Ein wesentlicher Vorteil des Scheduling von *task*-Gruppen besteht in der Reduzierung des *spin waiting* bei der Synchronisation (siehe Abschnitt 3.2.2): Wenn ein Prozeß seinen Prozessor abgeben muß, während er eine Synchronisationsvariable im Zugriff hat und diese somit für andere Prozesse sperrt, dann kann für andere Prozesse der Zugriff auf diese Variable mit erheblichem *spin waiting* verbunden sein. Mit dem gleichzeitigen Scheduling aller Prozesse, die auf gemeinsame Synchronisationsvariablen zugreifen, kann dieser Effekt vermieden werden. Eine zweite Form des *spin waiting* tritt auf, wenn ein in der Ausführung befindlicher Prozeß eine Synchronisationsvariable im Zugriff hat und andere Prozesse auf diese Variable zugreifen wollen. Dieser Effekt wird durch das Scheduling von *task*-Gruppen verstärkt. In Simulationen in [GTU91] führt das Scheduling von *task*-Gruppen in der Summe beider Effekte insgesamt zu einer Reduzierung des *spin waiting*. Ein weiterer Vorteil für das Scheduling von *task*-Gruppen ergibt sich zum Beispiel in *message-passing*-Systemen für die Kommunikation von Prozessen. Für Prozesse, die häufig miteinander kommunizieren, können bei nicht-gleichzeitigem Scheduling erhebliche Wartezeiten auftreten, wenn zum Beispiel ein Prozeß auf die *Message* eines nicht in der Ausführung befindlichen Prozesses wartet. Auch dieser Effekt kann durch das Scheduling von *task*-Gruppen vermindert werden. Neben diesen Vorteilen kann sich das Scheduling von *task*-Gruppen auch nachteilig auswirken: Das gleichzeitige Scheduling aller *tasks* unabhängig davon, ob die *tasks* gerade aktiv sind, kann *idle*-Phasen für die Prozessoren erzeugen und damit den Durchsatz verringern. Auf dem *Coscheduling* basierende Ansätze sind zum Beispiel das *Wave Scheduling* und das *Distributed-Hierarchical-Control-Gangscheduling*.

- **Coscheduling:**
  Für das *Coscheduling* werden in [Ous82] drei verschiedene Algorithmen untersucht, die auf dem *Central-Pool*-Modell basieren. Der *Matrix*-Algorithmus verwendet eine Matrix, deren Spaltenzahl der Anzahl der Prozessoren des Rechners entspricht. Die *tasks* einer *task*-Gruppe werden vollständig einer Zeile der Matrix zugeordnet. Nach dem Zeitscheibenverfahren wird jeweils allen *tasks* einer Zeile gleichzeitig ein Prozessor zugeteilt. Die beiden anderen Algorithmen, der *Continuous*- und der *Undivided*-Algorithmus unterscheiden sich vom *Matrix*-Algorithmus durch eine lineare Zuordnung der *task*-Gruppen zu den Prozessoren; die *task*-Gruppen werden in einer linearen Liste plaziert. Für die Zuteilung der Prozessoren an die *tasks* wird ein Fenster über die lineare Liste gelegt, das nach dem Ablauf einer Zeitscheibe verschoben wird. Das *Coscheduling* ist im Rahmen des Cm*-Projektes implementiert und untersucht worden [GSS87].
  Neben den für das Scheduling von *task*-Gruppen allgemein geltenden Vor- und Nachteilen ergeben sich für das *Coscheduling* noch weitere Nachteile. Ein Nachteil liegt in der zentralen Verwaltung der Prozessoren (*Central-Pool*-Modell), die in massiv-parallelen Systemen einen Engpaß darstellen kann. Als zweiter Nachteil ist die Fragmentierung der Prozessoren zu nennen, die auftreten kann, wenn eine *task*-Gruppe nicht alle Prozessoren benötigt, jedoch nicht genügend Prozessoren für alle *tasks* einer anderen *task*-Gruppe übrig sind.

- **Wave Scheduling:**
  Aufbauend auf den *Coscheduling*-Algorithmen wurde das *Wave Scheduling* insbesondere zur Verbesserung des Zuordnungsvorgangs bezüglich der Vermeidung der Fragmentierung entwickelt [ReFu87, TiWi81, TiWi84]. Die Idee des *Wave Scheduling* basiert auf dem *Hierarchical-Control*-Modell; die hierarchische Kontrollstruktur wird auf die Prozessoren abgebildet, es wird zum Beispiel jeder Knoten eines Baumes auf einen Knoten des Rechners abgebildet. Jeder Knoten verwaltet Informationen über die aktuelle Belegung der ihm zugeordneten Prozessoren; ein Austausch von Informationen findet nur zwischen Knoten aus benachbarten Hierarchieebenen statt. Eine Anfrage für die Zuteilung von Prozessoren an eine *task*-Gruppe wird an einen Knoten der entsprechenden Hierarchieebene gestellt. Dieser Knoten überprüft die Anzahl freier Prozessoren in dem unter ihm liegenden Teilbaum und ordnet der *task*-Gruppe gegebenenfalls die angeforderten Prozessoren dediziert zu. Aus Gründen der Lastbalancierung kann der zuordnende Knoten eine Unterbrechung der *task*-Gruppe vornehmen, ein Zeitscheibenverfahren ist jedoch nicht vorgesehen.
  Im Gegensatz zum *Coscheduling*, wo die zentrale Kontrolle aufgrund des *Central-Pool*-Modells einen Engpaß darstellen kann, realisiert das beim *Wave Scheduling* verwendete *Hierarchical-Control*-Modell eine verteilte, koordinierte Kontrolle und kann somit nicht zum Engpaß werden. Ein Nachteil des *Wave Scheduling* ergibt sich bezüglich der Fairness: Da die Zuordnung von Prozessen zu Prozessoren dediziert erfolgt und, im Fall einer balancierten Systemlast, keine Unterbrechung der Prozesse vorgenommen wird, müssen neu in das System eintretende Jobs gegebenenfalls bis zur Beendigung eines Jobs warten.

- **Distributed Hierarchical Control Gangscheduling (DHC-Gangscheduling):**
  Wie das *Wave Scheduling* basiert auch das *DHC-Gangscheduling* auf dem *Hierarchical-Control*-Modell [FeRu90]. Eine entscheidende Verbesserung bezüglich der Fairness wird durch eine Kombination mit dem *Round-Robin*-Algorithmus erzielt: Während beim *Wave Scheduling* ein Kontrollknoten die Kontrolle über ein Teilnetzwerk für die gesamte Dauer eines Jobs behält, wechselt die Kontrolle beim *DHC-Gangscheduling* im Zeitscheibenverfahren zwischen den Kontrollknoten; dabei kann ein Kontrollknoten auch mehrere *task*-Gruppen verwalten. Im einfachsten Fall wird die Kontrolle für die Dauer einer Zeitscheibe jeweils einer Ebene der Kontrollstruktur zugeteilt, so daß die Kontrolle in Form einer *Scheduling-Front* über die Kontrollstruktur läuft. Im Unterschied zum *Wave Scheduling*, wo die Kontrollstruktur über die Prozessoren des Rechnersystems verteilt wird, wird in [FeRu90] die Verwendung von Kontrollprozessoren vorgeschlagen, die in einer Baumstruktur mit Querverbindungen auf jeder Ebene *(X-tree)* orthogonal zum Rechnernetzwerk mit den Prozessoren verbunden sind (siehe Abb. 12). Die Querverbindungen in der Baumstruktur dienen der Kommunikation innerhalb einer Hierarchieebene und ermöglichen eine Lastbalancierung beim Mapping der *task*-Gruppen. Um den lokalen Kontext von Prozessen wiederzuverwenden und die Effizienz der *Cache*-Nutzung zu verbessern, werden einmal gestartete Prozesse nicht mehr migriert. Der *DHC-Gangscheduling*-Algorithmus realisiert bisher als einziger ein unterbrechendes

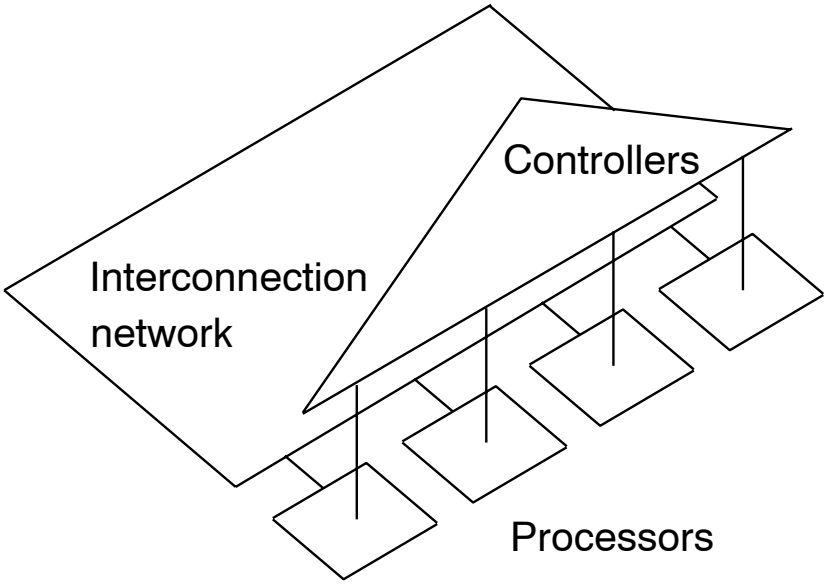


Abb. 12. Distributed Hierarchical Control: Die Kontrollhierarchie ist orthogonal zum Rechnernetzwerk [FeRu90].

Scheduling von *task*-Gruppen mit verteilter, koordinierter Kontrolle.
Ein Scheduling-Mechanismus, der wie das Scheduling von *task*-Gruppen eine Reduzierung des durch gesperrte Synchronisationsvariablen verursachten *spin waiting* zum Ziel hat, wird mit *Handoff Scheduling* bezeichnet.

- **Handoff Scheduling:**
  Das *Handoff Scheduling* beschreibt keine vollständige Scheduling-Strategie, sondern einen Mechanismus zur Reduzierung des *spin waiting*, der auf dem *Round-Robin Scheduling* basiert [Bla90, GTU91]. Die Idee des *Handoff Scheduling* ist, daß ein blockierter Prozeß den Rest seiner Zeitscheibe an einen *ready*-Prozeß des gleichen Jobs weitergibt, damit dieser Prozeß die Blockierung aufheben kann: Wenn ein unterbrochener Prozeß eine Synchronisationsvariable im Zugriff hat, dann geben Prozesse, die aufgrund dieser Variable blockiert werden, ihren Prozessor an den unterbrochenen Prozeß weiter.

4.3.2. Scheduling in Shared-Memory-Systemen

Mit der wachsenden Bedeutung von *shared-memory*-Systemen für die Ausführung von parallelen Programmen im Multiprogramming-Betrieb sind in den letzten Jahren vermehrt Untersuchungen über das Scheduling in derartigen Umgebungen durchgeführt worden. Mithilfe von Simulationen wurden verschiedene Scheduling-Algorithmen gegenübergestellt und neue Scheduling-Algorithmen evaluiert. Die Simulationen beziehen sich auf jeweils andere Multiprozessormodelle und Arbeitslasten und favorisieren jeweils andere Algorithmen.

4.3.2.1 Job-bezogenes Scheduling

Das Job-bezogene Scheduling behandelt die Zuordnung von Prozessoren zu Prozessen in Abhängigkeit von der Jobzugehörigkeit der Prozesse. Im folgenden werden zunächst zwei

Shortest-Job-First-Algorithmen und der *Round-Robin-job*-Algorithmus beschrieben. Der *Dynamic-Allocation*-Algorithmus erzielt gegenüber dem *Round-Robin-job*-Algorithmus eine Verkürzung der Prozessor-*idle*-Phasen. Alle vier Algorithmen behandeln das Job-Scheduling und basieren auf dem *Central-Pool*-Modell. Der *Auto-Scheduling*-Algorithmus vereint die Vorteile der Algorithmen *Round-Robin-job* und *Dynamic Allocation* durch ein geeignetes *task*-Scheduling. Während all diese Algorithmen zum Job-Scheduling unabhängig von den darunterliegenden Scheduler-Komponenten (*task*- und *thread*-Scheduler) arbeiten, realisiert das *Cooperative-Scheduling* eine Kommunikation zwischen den Komponenten.

- **Shortest Job First (SJF):**
  Mit *Shortest-Job-First* werden zwei verschiedene Algorithmen bezeichnet, der *Shortest-Number-of-Processes-First*-Algorithmus und der *Shortest-Cumulative-Demand-First*-Algorithmus. Bei beiden Algorithmen erfolgt das Scheduling zwar unter Berücksichtigung der Jobzugehörigkeit der Prozesse, eine gleichzeitige Ausführung aller Prozesse eines Jobs steht dabei jedoch nicht im Vordergrund.

  – **Smallest Number of Processes First (SNPF):**
    Der SNPF-Algorithmus verwendet die Anzahl der Prozesse eines Jobs als Maß für die Prozessorzuteilung [MEB88]. Ein freier Prozessor wird einem Prozeß des Jobs zugewiesen, der aktuell die kleinste Anzahl an Prozessen hat. Nach diesem Algorithmus werden *task*-Abhängigkeiten nur begrenzt berücksichtigt, da nur mit geringer Wahrscheinlichkeit alle *tasks* eines Jobs gleichzeitig ausgeführt werden. Der SNPF-Algorithmus spezifiziert nur die Zuteilung von Prozessoren und kann sowohl unterbrechend als auch nicht-unterbrechend sein.

  – **Smallest Cumulative Demand First (SCDF):**
    Der SCDF-Algorithmus orientiert sich am Prozessorzeitbedarf eines Jobs [MEB88]. Ein freier Prozessor wird einem Prozeß des Jobs zugewiesen, der aktuell den kleinsten verbleibenden Bedarf an Prozessorzeit hat. Wie schon beim SNPF-Algorithmus werden *task*-Abhängigkeiten nur begrenzt berücksichtigt, und die Zuteilung von Prozessoren kann wiederum unterbrechend oder nicht-unterbrechend sein.

- **Round-Robin-job (RRjob):**
  Der RRjob-Algorithmus arbeitet analog zum regulären *Round-Robin*-Algorithmus nach dem Zeitscheibenverfahren [LeVe90]. Anstelle von Prozessen befinden sich jedoch Jobs in der *ready*-Warteschlange (Job-Warteschlange); jedem Eintrag in der Job-Warteschlange entspricht eine eigene Prozeß-Warteschlange. Bei jeder Zuordnung werden einem Job p Einheiten an Prozessorzeit zugeteilt; p entspricht der Prozessorzahl des Systems. Für Jobs mit weniger als p Prozessen werden größere Einheiten zugeteilt, so daß das Produkt aus der Anzahl der Prozesse und der Größe der Einheit konstant ist. Für Jobs mit p oder mehr Prozessen in ihrer Prozeß-Warteschlange werden p Prozessoren zugewiesen, und es werden jeweils nur p Prozesse bedient. Ein Nachteil des RRjob-Algorithmus ist die statische Festlegung der Prozessorzahl für einen Job beim Programmstart; da die Jobs im allgemeinen nicht immer alle Pro-

zessoren nutzen können, verursacht diese Inflexibilität Phasen, in denen Prozessoren *idle* sind[7].

- **Dynamic Allocation:**
  Der *Dynamic-Allocation*-Algorithmus erzielt im Vergleich zum RRjob-Algorithmus eine Verbesserung bezüglich der Prozessor-*idle*-Phasen durch eine dynamische Prozessorzuordnung: Die Anzahl der einem Job zugeordneten Prozessoren wird der Systemlast dynamisch angepaßt. Der Algorithmus versucht, jedem Job immer wenigstens einen Prozessor zuzuordnen [ZaMc90]. Erst wenn diese Bedingung erfüllt ist, werden noch freie Prozessoren nach der FIFO-Strategie verteilt. Wenn beim Eintreffen neuer Jobs keine freien Prozessoren verfügbar sind, werden von Jobs mit mehr als einem Prozessor Prozessoren abgegeben, so daß jedem neu eintreffenden Job ein Prozessor zugeteilt werden kann. Auf der Job-Ebene verwaltet jeder Job die ihm zugeteilten Prozessoren selbst, zum Beispiel mittels einer Jobbezogenen *ready*-Warteschlange. Die Kommunikation der Jobs untereinander gewährleistet, daß nicht benötigte Prozessoren bei Bedarf an andere Jobs abgegeben werden können. Wenn es keinen Bedarf durch andere Jobs gibt, dann werden nicht benötigte Prozessoren in einer Warteposition gehalten und können neu eintreffende Prozesse ohne Kontextwechsel ausführen. Die Untersuchungen in [ZaMc90] favorisieren die dynamische Prozessorzuordnung gegenüber der statischen Zuordnung (RRjob), sofern der Kontextwechsel-Overhead nicht extrem groß ist.
  Die beiden Algorithmen *Round-Robin-job* und *Dynamic Allocation* beschreiben einen Konflikt: Der RRjob-Algorithmus bietet den Vorteil der gleichzeitigen Ausführung aller Prozesse eines Jobs, wobei jedoch Prozessor-*idle*-Phasen erzeugt werden. Der *Dynamic-Allocation*-Algorithmus vermeidet diese *idle*-Phasen, führt aber im allgemeinen nicht alle Prozesse eines Jobs gleichzeitig aus, was wiederum mit Nachteilen bezüglich der Prozeßsynchronisation beziehungsweise -kommunikation verbunden ist. Eine Lösung dieses Konflikts wird durch den *Auto-Scheduling*-Algorithmus beschrieben:

- **Auto-Scheduling:**
  Der *Auto-Scheduling*-Algorithmus implementiert zusätzlich zu einer dynamischen Prozessorzahl pro Job eine ebenfalls dynamische Anzahl von Prozessen pro Job [Pol91]. Zu diesem Zweck wird ein vom Compiler generierter Prozeßgraph *(Hierarchical Task Graph)* verwendet, der aus elementaren Knoten und aus zusammengesetzten Knoten besteht; die zusammengesetzten Knoten bestehen wiederum aus elementaren Knoten und aus zusammengesetzten Knoten. Die Knoten repräsentieren Prozesse, die in Abhängigkeit von der Anzahl der verfügbaren Prozessoren als zusammengesetzte Knoten oder als elementare Knoten ausgeführt werden: Für einen zusammengesetzten Knoten werden nur dann mehrere Prozesse erzeugt, wenn genügend Prozessoren verfügbar sind; andernfalls wird der zusammengesetzte Knoten nur von einem Prozeß ausgeführt. Nach dem *Auto-Scheduling*-Algorithmus werden

[7] Eine Kombination des RRjob-Algorithmus mit der Prozessorabgabe-Strategie (freiwillige Abgabe von Prozessoren zum Beispiel in *idle*-Phasen) kann die *idle*-Phasen deutlich vermindern. Wenn der Job die Prozessoren jedoch nur zeitweise nicht benötigt, dann entfällt der Vorteil der gleichzeitigen Ausführung aller Prozesse.

alle Prozesse eines Jobs gleichzeitig ausgeführt, und die beim RRjob-Algorithmus möglichen *idle*-Phasen treten nicht auf. Die Zuteilung von Prozessoren an Jobs ist in [Pol91] nicht spezifiziert; sie kann zum Beispiel nach dem *Central-Pool*-Modell erfolgen. Eine wesentliche Voraussetzung für diesen Algorithmus ist die Verwendung des Compiler-generierten Prozeßgraphen. Damit ist der *Auto-Scheduling*-Algorithmus nur dann anwendbar, wenn bereits auf der Compiler-Ebene der beschriebene Prozeßgraph verwendet wird. Die Erzeugung des Prozeßgraphen auf der Compiler-Ebene birgt den Nachteil, daß für die Parallelisierung wesentliche Laufzeitabhängigkeiten nicht berücksichtigt werden können. Für den *Auto-Scheduling*-Algorithmus existiert eine Implementierung im Rahmen der *Parafrase-2*-Umgebung [Pol89b].

- **Cooperative-Scheduling:**
  Alle bisher in diesem Abschnitt beschriebenen Algorithmen beschäftigen sich im wesentlichen mit der Ebene des Job-Scheduling. Das Job-Scheduling hat einen erheblichen Einfluß auf die darunterliegenden Scheduler-Komponenten (*task*- und *thread*-Scheduler) und bildet eine Grundlage für diese. Im einfachsten Fall arbeiten die Scheduler-Komponenten unabhängig voneinander. In allen kommerziellen Realisationen findet dieses Konzept Anwendung. Dadurch kann eine Scheduling-Entscheidung eine unmittelbar zuvor getroffene Entscheidung einer anderen Komponente unter Umständen aufheben oder sogar umkehren. Beim *Cooperative-Scheduling* kommunizieren die Scheduler-Komponenten der verschiedenen Ebenen über gemeinsame Datenbereiche miteinander, so daß eine Abstimmung von Scheduling-Entscheidungen stattfinden kann [Nag93]: Damit soll zum Beispiel verhindert werden, daß der Job-Scheduler einem Prozeß einer parallelen Anwendung ohne Zustimmung der *thread*-Scheduler-Komponente einen Prozessor entzieht; dadurch kann ein erhöhtes *spin waiting* bei der Synchronisation vermieden werden. Eine ganz wesentliche Abstimmung erfolgt durch eine Anforderung des *thread*-Scheduler an den Job-Scheduler: Der *thread*-Scheduler stellt seine Information über die aktuell benötigte Prozessorzahl zur Verfügung; der Job-Scheduler teilt dann entweder genau diese Anzahl von Prozessoren zu, oder die entsprechende Anwendung erhält genau einen Prozessor und wird zunächst sequentiell ausgeführt, oder die Anwendung wird zunächst nicht bedient und aus der *ready*-Warteschlange in eine *blocked*-Warteschlange transferiert. Auf diese Weise können ungünstige Kombinationen von Anzahl der Schleifeniterationen und Anzahl der Prozessoren vermieden werden, was möglicherweise eine schlechte Lastbalance zur Folge hätte. Die Strategie des *Cooperative-Scheduling* kann auf herkömmliche Scheduling-Algorithmen angewendet werden. Dabei werden diese Algorithmen bezüglich Informationsaustausch und geeigneter Nutzung der Informationen adaptiert. Das *Cooperative-Scheduling* ermöglicht dann vergleichsweise effizientere Entscheidungen, und die Untersuchungen in [Nag93] zeigen deutliche Vorteile gegenüber den herkömmlichen Realisationen.

4.3.2.2 Locality-Scheduling

Der Einfluß des Scheduling auf die Lokalität ist bisher nur in einigen wenigen Ansätzen untersucht worden: Zielsysteme für die untersuchten Algorithmen sind *shared-memory*-

Systeme mit *Multicache*- oder NUMA-Architektur, für die ein effizientes Scheduling entscheidend von der Berücksichtigung der Lokalität abhängt. Die Algorithmen *Affinity-Scheduling* und *Process Control* berücksichtigen den Einsatz von *Cache*-Speichern, das *Pool-based Scheduling* ist für den Einsatz in NUMA-Architekturen konzipiert.

- **Affinity Scheduling:**
  In *Multicache*-Systemen ist eine effiziente Programmausführung eng mit der effizienten Nutzung der lokalen *Cache*-Speicher verbunden. Wenn ein Prozeß nach einer Unterbrechung auf einem anderen Prozessor fortgeführt wird, dann muß im allgemeinen zunächst eine große Datenmenge in den *Cache* geladen werden (*working set* [Den68]), bevor eine effiziente Programmausführung möglich ist. Aufgrund des *Cache*-Inhaltes existiert eine enge Beziehung (Affinität) zwischen Prozeß und Prozessor. Beim *Affinity-Scheduling* wird diese Affinität als Kriterium für das Scheduling genutzt [SqNe91]: Ein unterbrochener Prozeß wird auf dem Prozessor fortgeführt, auf dem er unterbrochen wurde. Dadurch kann der sonst entstehende Overhead für das Laden des *working set* zum Teil vermieden werden. Da eine strikte Einhaltung dieser Strategie keine Lastbalancierung erlaubt, werden Prozeßmigrationen dann durchgeführt, wenn Prozessoren *idle* sind. Die Untersuchungen in [SqNe91] zum *Affinity-Scheduling* beschäftigen sich insbesondere mit einer adaptiven Scheduling-Strategie, die in Abhängigkeit von der Systemlast geeignete Grenzwerte für die Migration von Prozessen bestimmt.

- **Process Control:**
  Wenn die Anzahl der Prozesse die Anzahl der Prozessoren übersteigt, dann werden von unterbrechenden Schedulern aus Gründen der Fairness Kontextwechsel durchgeführt. Neben dem *spin waiting* bei der Synchronisation und dem mit Kontextwechseln verbundenen Overhead verursacht die Existenz von mehr Prozessen als Prozessoren in *Multicache*-Systemen ein weiteren negativen Effekt: Selbst wenn ein unterbrochener Prozeß zum Beispiel nach dem *Affinity-Scheduling* wieder auf demselben Prozessor gestartet wird, dann sind die von ihm benötigten Daten im allgemeinen teilweise oder vollständig verloren und müssen erneut geladen werden (*cache corruption*). Für die Durchführung von Kontextwechseln ergibt sich damit in *Multicache*-Systemen ein wesentlich höherer Overhead als in anderen *shared-memory*-Systemen.
  Der *Process-Control*-Algorithmus vermeidet diesen Overhead, indem die Anzahl der *ready*-Prozesse auf die Anzahl der Prozessoren beschränkt wird [GTU91, TuGu89]; auf diese Weise verfügt jeder *ready*-Prozeß über einen Prozessor, und Kontextwechsel können vermieden werden. Die Zuteilung von Prozessoren an Jobs erfolgt nach dem *Central-Pool*-Modell: Ein zentraler *Server* verteilt die Prozessoren gleichmäßig an die Jobs; ein Job erhält jedoch maximal soviele Prozessoren wie er nutzen kann. Die Anzahl der zugewiesenen Prozessoren bestimmt die Anzahl der *ready*-Prozesse eines Jobs: Wenn ein Job weniger Prozessoren erhält als er *ready*-Prozesse hat, dann muß er Prozesse suspendieren um Kontextwechsel zu vermeiden. Die Zuteilung der Prozessoren durch den zentralen *Server* erfolgt dynamisch, d.h. wenn ein Job gestartet oder beendet wird, erfolgt eine Neuverteilung aller Prozessoren.

Der *Process-Control*-Algorithmus ähnelt dem *Auto-Scheduling*; beide Algorithmen vermeiden Kontextwechsel durch das gleichzeitige Scheduling aller existierenden Prozesse: Der *Auto-Scheduling*-Algorithmus realisiert durch die Verwendung eines Compiler-generierten *task*-Graphen eine dynamische Granularität auf *task*-Ebene; auf diese Weise können auch mit einer wechselnden Anzahl von Prozessen immer alle *ready-tasks* gleichzeitig ausgeführt werden. Der *Process-Control*-Algorithmus basiert auf der Zuordnung von *threads* zu *Server*-Prozessen (feingranulare Parallelisierung). Bei statischer Granularität auf der *thread*-Ebene wird durch die Reduzierung der *Server*-Prozesse ein gleichzeitiges Scheduling aller existierenden Prozesse realisiert. Im Unterschied zu anderen Scheduling-Algorithmen, die ebenfalls nach dem *Central-Pool*-Modell arbeiten, erfolgt die zentrale Kontrolle beim *Process-Control*-Algorithmus seltener: Die Zuteilung von Prozessoren an Jobs erfolgt zentral; die Verwaltung der Prozesse eines Jobs wird auf einer darunterliegenden Ebene durchgeführt (*task*- beziehungsweise *thread*-Scheduling) und erfordert keine Aktivitäten der zentralen Kontrollinstanz, die somit seltener aktiviert wird. Diesbezüglich ist der durch die zentrale Kontrolle entstehende Engpaß für massiv-parallele Systeme beim *Process-Control*-Algorithmus weniger signifikant.

- **Pool-based Scheduling:**
  Das *Pool-based Scheduling* ist für NUMA-Architekturen konzipiert, die nach dem *distributed-shared-memory*-Rechnermodell arbeiten; die Anforderungen an die Lokalität sind jedoch ähnlich wie in *Multicache*-Systemen [ZhBr91]. Der *Pool-based-Scheduling*-Algorithmus basiert auf dem *Multiple-Pool*-Modell, das ein Multiprozessorsystem in mehrere logische Prozessor-Pools partitioniert und jeden Prozessor einem Pool zuordnet. Die Prozesse eines Jobs werden auf einen oder mehrere Pools verteilt: In Abhängigkeit von der Systemlast wird die Anzahl der Prozesse eines Jobs begrenzt; auf diese Weise wird auch die Anzahl der Pools für die Prozesse eines Jobs der aktuellen Last angepaßt. In realen Systemen sind Architektur-bedingte Prozessor-Cluster für die Bildung von logischen Pools prädestiniert. Neben einer statischen Pool-Einteilung ist auch eine dynamische Zerlegung und Vereinigung der Pools möglich. Ein Konflikt ergibt sich für das *Pool-based Scheduling* zwischen der Berücksichtigung der räumlichen Lokalität und der Lastbalancierung: Einerseits ermöglicht das *Pool-based Scheduling* durch die Berücksichtigung der Lokalität eine effizientere Programmausführung. Andererseits behindert die Berücksichtung der Lokalität eine systemweite Lastbalancierung.

4.4. Loop-Scheduling

Die bisher betrachteten Algorithmen sind für ein Scheduling ohne Berücksichtigung einer bestimmten Programmstruktur konzipiert. In *shared-memory*-Systemen ergibt sich jedoch eine wesentliche Form von Parallelismus, für die spezielle Scheduling-Algorithmen entwickelt worden sind: Voneinander unabhängige Operationen lassen sich häufig in Form von parallelen Schleifen und geschachtelten parallelen Schleifen darstellen. Das Loop-Scheduling bezeichnet eine Form des *thread*-Scheduling für den Fall von parallelen Schleifen als Programmstruktur, d.h. die Verteilung der Iterationen einer parallelen

Schleife oder von geschachtelten parallelen Schleifen auf die verfügbaren Prozesse [Pol89b, PoKu87]. Bei einer statischen Verteilung kann der Benutzer oder der Compiler a priori bestimmen, welche Iterationen von welchem Prozeß ausgeführt werden.

- **Block-Scheduling:**
  Beim *Block-Scheduling* wird für p Prozesse und N Iterationen jedem Prozeß ein Block mit N/p Iterationen zugewiesen. Der Modulo-Rest der Division wird zusätzlich über die Prozesse verteilt. Das *Block-Scheduling* ist der einfachste statische Scheduling-Algorithmus und wird in vielen Systemen verwendet.

- **Cyclic-Scheduling:**
  Das *Cyclic-Scheduling* resultiert in der gleichen Anzahl von Iterationen für jeden Prozeß wie das *Block-Scheduling*. Die Gruppierung der Iterationen erfolgt jedoch nicht blockweise, sondern zyklisch. Das *Cyclic-Scheduling* ist insbesondere für *virtual-shared-memory*-Systeme relevant [LaPr91].

Da die Ausführungszeit verschiedener Iterationen zum Teil einer großen Streuung unterworfen ist und im allgemeinen nicht vorhergesagt werden kann, ist eine bezüglich der Lastbalancierung optimale Zuordnung nur mittels dynamischer Verfahren möglich; dabei fordern die Prozesse jeweils neue Blöcke von Iterationen (*chunks*) an. Die dynamischen Algorithmen für das Loop-Scheduling werden mit *Self-Scheduling*, *Chunk-Scheduling*, *Guided Self-Scheduling* und *Factoring* bezeichnet; diese Algorithmen unterscheiden sich bezüglich der Größe der jeweils zugeordneten *chunks*: Beim *Self-Scheduling* und beim *Chunk-Scheduling* werden *chunks* konstanter Größe zugeordnet, beim *Guided Self-Scheduling* und beim *Factoring* variiert die *chunk*-Größe. Alle vier Algorithmen basieren auf dem *Self-service*-Modell [FeRu90]:

- **Self-Scheduling:**
  Beim *Self-Scheduling* wird einem Prozeß jeweils eine Iteration zugewiesen [TaYe86]. Für eine Schleife mit N Iterationen ergeben sich daraus N Zuweisungen (*dispatch*-Operationen). Da nach dem *Self-Scheduling* jeweils nur eine Iteration zugewiesen wird, ist dies bezüglich der Lastbalancierung der beste dynamische Algorithmus. Der optimalen Lastbalancierung steht jedoch ein hoher Overhead mit N *dispatch*-Operationen gegenüber. Für feingranularen Parallelismus kann der Overhead der wesentliche Faktor für die Bewertung des *Self-Scheduling* sein.

- **Chunk-Scheduling:**
  Beim *Chunk-Scheduling* wird einem Prozeß jeweils ein *chunk* mit einer festen Anzahl von Iterationen zugewiesen [KrWe85, Pol89b]. Auf diese Weise kann der Overhead durch die Reduzierung der *dispatch*-Operationen verringert werden, wobei die Lastbalancierung jedoch verschlechtert wird. Die Größe der *chunks* beschreibt einen Konflikt zwischen Lastbalancierung und Overhead: In einem Extremfall sind die *chunks* so groß, daß jeder Prozeß nur eine *dispatch*-Operation ausführt. Im anderen Extrem ist die *chunk*-Größe mit nur einer Iteration minimal (*Self-Scheduling*). Für dazwischenliegende *chunk*-Größen kann die Effizienz des Scheduling im Vergleich zu den Extremfällen sowohl besser als auch schlechter sein. Der Nachteil des

Chunk-Scheduling besteht in der Abhängigkeit einer geeigneten *chunk*-Größe von der Granularität der Schleife sowie von der Anzahl und der zeitlichen Varianz der Schleifeniterationen, die auch zur Laufzeit nicht in jedem Fall vorhergesagt werden können.

- **Guided Self-Scheduling:**
  Das *Guided Self-Scheduling* stellt einen geeigneten Kompromiß zwischen Lastbalancierung und Overhead dar [PoKu87]. Für *p* Prozesse wird einem Prozeß von den noch zu bearbeitenden *N* Iterationen ein *chunk* von $\lceil N/p \rceil$ Iterationen zugewiesen. Auf diese Weise variiert die Größe der *chunks* zwischen den beiden Extremen: Die Zuweisung von großen *chunks* zu Beginn der Schleifenausführung erzeugt eine geringe Anzahl von *dispatch*-Operationen und damit einen geringen Overhead. Durch die Zuweisung von kleinen *chunks* am Ende der Schleifenausführung kann in vielen Fällen gleichzeitig eine optimale Lastbalancierung erzielt werden.

- **Factoring:**
  Beim *Guided Self-Scheduling* kann es, bei großer Varianz für die Ausführungszeit der Iterationen einer Schleife, zu einer unbalancierten Ausführung kommen: In diesem Fall werden zu früh bereits zuviele Iterationen zugewiesen, so daß nicht genügend Iterationen übrig sind, um die Ausführungszeit der großen *chunks* auszugleichen. Beim *Factoring* werden zu Beginn einer Schleifenausführung im Vergleich zum *Guided Self-Scheduling* kleinere *chunks* verteilt [HSF91]; es werden jeweils *p chunks* der gleichen Größe zugewiesen. Die Berechnung der *chunk*-Größe basiert dabei auf einer Schätzung der Ausführungszeit für diese Zuweisung. Untersuchungen in [HSF91, Nag93] favorisieren das *Factoring* gegenüber dem *Guided Self-Scheduling*, insbesondere bei großer Varianz für die Ausführungszeit oder bei ungünstiger Prozessorzuteilung.

Die dynamischen Scheduling-Algorithmen ermöglichen auch für Schleifen mit großer Varianz in der Ausführungszeit der Iterationen eine mehr oder weniger lastbalancierte Ausführung; diesbezüglich werden diese Algorithmen im Gegensatz zu den statischen als lastadaptiv bezeichnet.
Die bisher betrachteten Algorithmen sind auf *shared-memory*-Architekturen zugeschnitten, die keine Anforderungen an die Lokalität der Ausführung stellen. Im Gegensatz dazu erfordern *multicache*-Architekturen und insbesondere auch *virtual-shared-memory*-Systeme die Einbeziehung der Lokalitätsbedingungen in die Scheduling-Entscheidungen. Das *Affinity-Scheduling*[8] wird in [MaLe92] für einen *multicache*-Rechner untersucht. Es ist teilweise statisch und teilweise dynamisch und ist damit als bedingt lastadaptiv einzuordnen. Das *Align-Scheduling* wurde im Rahmen des Cray MPP FORTRAN Programmiermodells implementiert und gehört zu den statischen, nicht-lastadaptiven Algorithmen [PMM93]. Das *Loop-Blocking* ist eine speziell auf *virtual-shared-memory*-Rechner zugeschnittene Scheduling-Strategie für ein dynamisches, lastadaptives Scheduling [GrWi93].

[8] Im Gegensatz zu dem in Abschnitt 4.3.2.2 betrachteten *Affinity-Scheduling* der Job-Scheduling Ebene bezieht sich das *Affinity-Scheduling* hier auf die Ebene des *thread*-Scheduling.

- **Affinity-Scheduling:**
  Das *Affinity-Scheduling* [MaLe92] ist auf eine charakteristische Schachtelung von Schleifen zugeschnitten, bei der sich eine parallele Schleife innerhalb einer sequentiellen Schleife befindet (siehe Abb. 13). Jeder *thread* der parallelen Schleife benötigt in aufeinanderfolgenden Iterationen der sequentiellen Schleife jeweils dieselben Daten. Während der ersten Iteration der sequentiellen Schleife werden für jeden *thread* der parallelen Schleife die entsprechenden Daten lokal verfügbar gemacht. Bei der Verwendung herkömmlicher Algorithmen werden die *threads* der parallelen Schleife in den folgenden Iterationen der sequentiellen Schleife im allgemeinen zufällig jeweils anderen Prozessoren zugewiesen; dieser Zufall beruht auf der nicht vorhersagbaren Reihenfolge für die Vergabe des *thread*-Scheduler an die Prozesse. Mit diesem "wandern" der *threads* von einem Prozeß zum anderen müssen auch die zugehörigen Daten jeweils transferiert werden. Im Gegensatz zu herkömmlichen Algorithmen gewährleistet das *Affinity-Scheduling*, daß ein *thread* J_k für alle Iterationen der sequentiellen Schleife demselben Prozeß zugewiesen wird; dadurch kann die Datenlokalität gegenüber herkömmlichen Algorithmen deutlich verbessert werden. Für die erste Iteration der sequentiellen Schleife kann ein dynamisches und damit lastadaptives Scheduling genutzt werden. Da das Scheduling für alle weiteren Iterationen der sequentiellen Schleife durch die erste Iteration festgelegt ist, handelt es sich danach um ein statisches Scheduling. Inwiefern ein lastadaptives Scheduling in der ersten Iteration eine lastbalancierte Ausführung auch in den übrigen Iterationen gewährleistet, hängt von der jeweiligen Schleife ab.

- **Align-Scheduling:**
  Beim *Align-Scheduling* [PMM93] erfolgt die Zuordnung von Iterationen zu Prozessen aufgrund der aktuellen Verteilung der Daten über die Prozessoren *(iteration-data-alignment)*. Das *alignment* ist jeweils auf genau ein Datenfeld - zum Beispiel *A(I)* - bezogen, welches über den Schleifenindex *I* der parallelen Schleife indiziert wird. Eine Iteration I_k wird dem Prozeß zugewiesen, der auf dem Prozessor ausgeführt wird, auf dem sich das Datum $A(I_k)$ befindet. Für das Cray MPP FORTRAN Programmiermodell muß der Index dabei von der Form *(aI+b)* sein. Das *Align-Scheduling* ist durch die Verteilung der Daten vollständig festgelegt und damit nicht-lastadaptiv. Eine andere Form des *Align-Scheduling* läßt sich mithilfe von *templates* realisieren: Ähnlich wie in *High Performance FORTRAN* kann ein *template* als abstrakter Indexraum definiert werden [HPF93]; ein *template T(I)* zum Beispiel ordnet jedem Wert von *I* einen Wert *T(I)* zu. Auf diese Weise kann für das *Align-Scheduling*

Abb. 13. Affinität in geschachtelten Schleifen: Parallele Schleifen innerhalb sequentieller Schleifen

anstelle der Verteilung eines Datenfeldes ein *template* und damit eine beliebige Verteilung als Grundlage für die Verteilung der Iterationen auf die Prozessorelemente dienen.

- **Loop-Blocking:**
  Das *Loop-Blocking* ist eine speziell auf *virtual-shared-memory*-Systeme zugeschnittene Scheduling-Strategie, die prinzipiell auf alle herkömmlichen Loop-Scheduling-Algorithmen angewendet werden kann [GrWi93]. Insbesondere die herkömmlichen lastadaptiven Algorithmen führen zu negativen Effekten, da häufig zwei oder mehr Vektoren eines Datenfeldes auf jeweils einer Seite liegen und verschiedene Iterationen einer parallelen Schleife gleichzeitig jeweils einen anderen dieser Vektoren benötigen. Dabei benötigen die Iterationen verschiedener Prozesse verschiedene Daten, die jedoch auf derselben Seite liegen; dieser Effekt wird mit *false sharing* [BoIs91] bezeichnet. In Abb. 14 liegen jeweils 10 Vektoren eines Datenfeldes auf einer Seite; das in diesem Beispiel verwendete *Factoring* erzeugt eine Partitionierung, die insbesondere bei den letzten *chunks* zu *false sharing* führt. Beim *Loop-Blocking* wird zur Vermeidung des *false sharing* eine Gruppierung von Iterationen entsprechend der Gruppierung der Vektoren durch deren Abbildung auf die Seiten vorgenommen. Für das Scheduling bildet dann nicht die einzelne Iteration, sondern eine Gruppe von Iterationen eine zuzuordnende Einheit. Die Güte der Lastbalancierung wird dadurch um den Faktor der Größe der Gruppen reduziert. In Abb. 15 ist das *Loop-Blocking* am Beispiel des *Factoring* dargestellt.

4.5. Analyse der Algorithmen

Entsprechend Abschnitt 3.2 ist das Scheduling in *virtual-shared-memory*-Systemen eng mit den übrigen Belangen der Prozeßverwaltung verknüpft, insbesondere mit der Prozeßsynchronisation. Neben den Algorithmen für die Prozessorzuteilung und -abgabe gehören die Migration und die Kontextwechsel zu den entscheidenden Problemen. Im folgenden werden die in Abschnitt 3.2 beschriebenen Scheduling-Algorithmen und Strategien existierender VSM-Systeme und die in Abschnitt 4.3 beschriebenen Algorithmen für massiv-parallele Systeme und *shared-memory*-Systeme bezüglich der zu Beginn dieses Kapitels festgelegten Rahmenbedingungen für das Scheduling analysiert. Die Rahmenbedingungen spezifizieren ein verteiltes, dynamisches[9] Scheduling abhängiger *tasks* unter Berücksichtigung der Forderungen nach räumlicher Lokalität und nach Datenlokalität. Alle betrachteten Algorithmen realisieren ein dynamisches[9] Scheduling; auf diesen Punkt wird daher im weiteren Verlauf dieser Arbeit nicht näher eingegangen.
Die Untersuchungen in dieser Arbeit beziehen sich auf die Parallelität innerhalb von Schleifen. Diesbezüglich wird im folgenden das in Abschnitt 4.1 beschriebene feingranulare Parallelisierungskonzept im Vordergrund stehen; die Analysen beziehen sich auf das Job-Scheduling und auf das *thread*-Scheduling.

[9] Dynamisch bezieht sich hier auf das Scheduling auf der Job-Ebene und nicht, wie in Abschnitt 4.4, auf das *thread*-Scheduling.

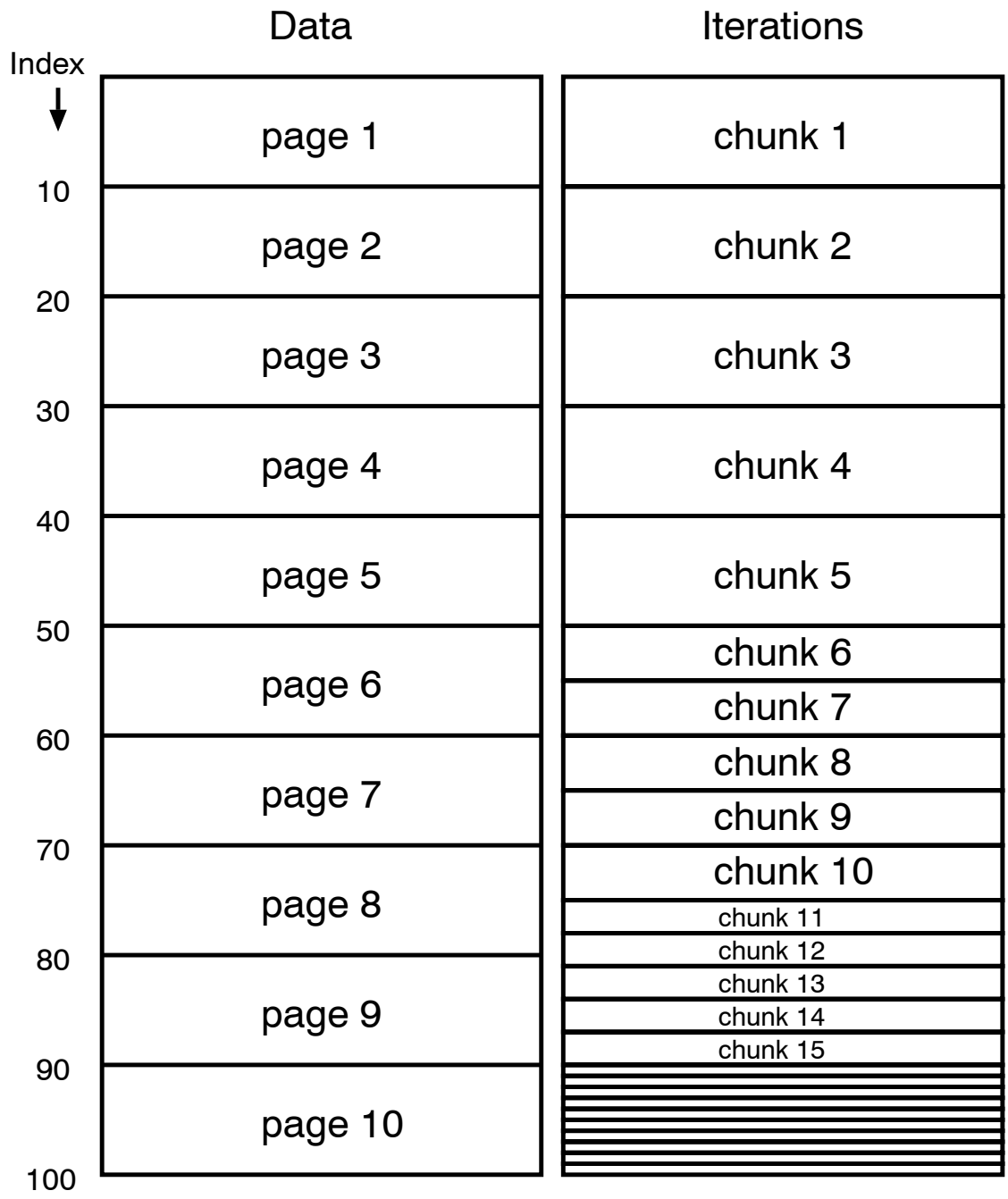


Abb. 14. Loop-Scheduling für Multi-Vektor-Seiten: Factoring [HSF91]

4.5.1. Das Job-Scheduling

Für das Job-Scheduling sind alle zu Beginn dieses Kapitels aufgestellten Forderungen relevant. Die Aufteilung des Scheduling auf verschiedene Hierarchiestufen realisiert bereits ein in begrenztem Umfang verteiltes Scheduling. Da die Prozesse aller Jobs den Job-Scheduler nur exklusiv nutzen können, bleibt für massiv-parallele Systeme ein Engpaß und damit die Forderung nach einer Verteilung auch auf der Ebene des Job-Scheduler.

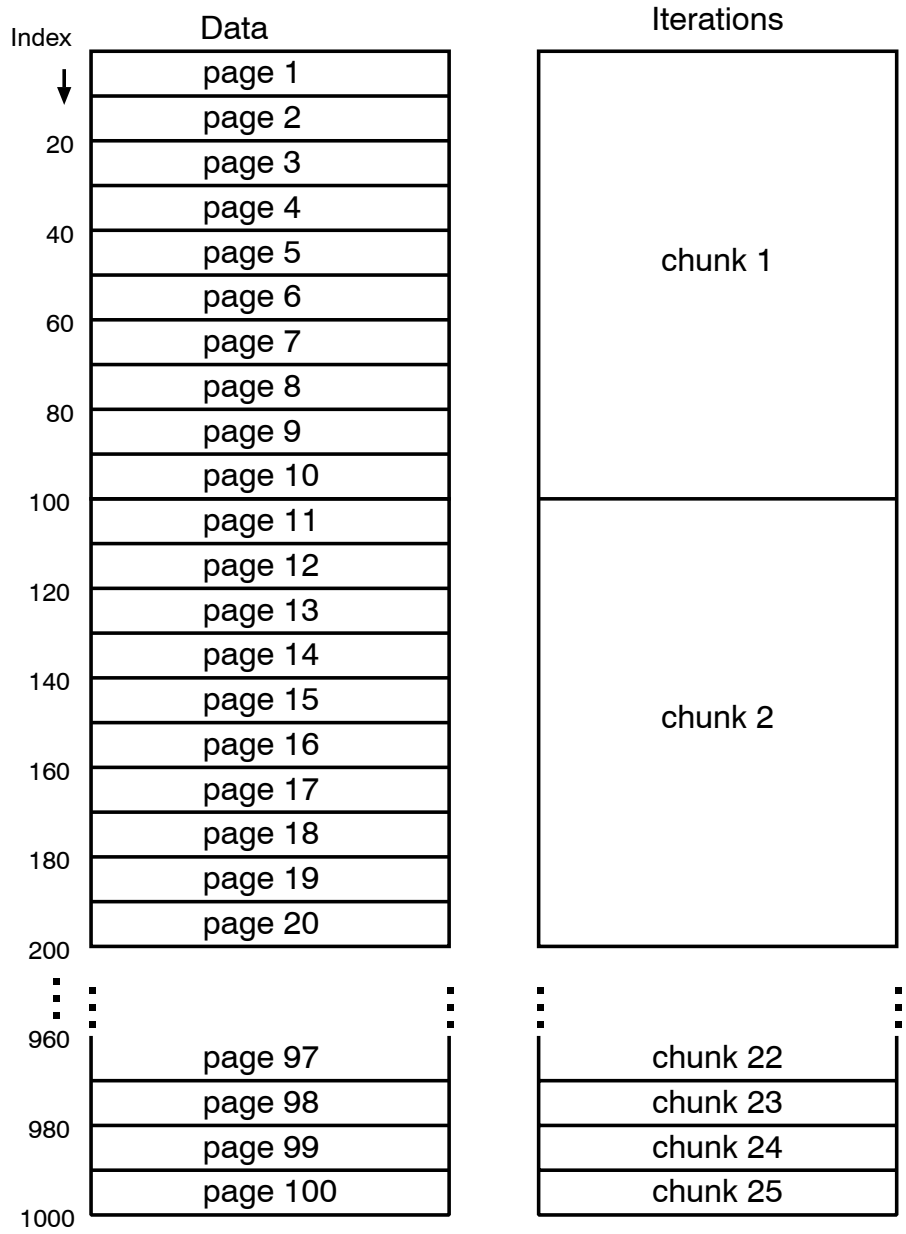


Abb. 15. Loop-Scheduling für Multi-Vektor-Seiten: Factoring mit Loop-Blocking [GrWi93]

4.5.1.1 Task-Abhängigkeiten

Bezüglich der Effizienz von Algorithmen für das Job-Scheduling spielt die Einbeziehung von *task*-Abhängigkeiten eine entscheidende Rolle. Bei allen vorgestellten Scheduling-Algorithmen für massiv-parallele Systeme, dem *Coscheduling*, dem *Wave Scheduling* und dem *DHC-Gangscheduling*, ist diesbezüglich das Scheduling von *task*-Gruppen realisiert. Die Überlegenheit des Scheduling von *task*-Gruppen wächst mit der Abhängigkeit der *tasks* untereinander. Häufig kommunizierende oder synchronisierende *tasks* können die Leistung stark reduzieren, wenn sie nicht gleichzeitig ausgeführt werden. Das heute auf massiv-parallelen Rechnern verbreitete *message-passing*-Programmiermodell bedingt durch den Austausch von *Messages* zwischen *tasks* eine große Abhängigkeit der *tasks* untereinander. In massiv-parallelen Systemen ist bezüglich des *message-passing-*

Programmiermodells das Scheduling von *task*-Gruppen zu favorisieren.
In *shared-memory*-Systemen ist die Abhängigkeit von *tasks* im wesentlichen durch die Synchronisation bestimmt; auch hier spielt das Scheduling von *task*-Gruppen eine entscheidende Rolle: Nur wenige, im wesentlichen einfache Algorithmen wie zum Beispiel die *Shortest-Job-First*-Algorithmen und der *Priority-based-Scheduling*-Algorithmus realisieren kein Scheduling von *task*-Gruppen. Diese Algorithmen orientieren sich zwar zum Teil an der Job-Zugehörigkeit von Prozessen, eine Vermeidung des *spin waiting* bei der Synchronisation steht dabei jedoch nicht im Vordergrund. Eine Reduzierung des *spin-waiting*-Overhead kann für diese Algorithmen durch die Verwendung von nichtblockierenden Synchronisationsoperationen und den Einsatz des *Handoff-Scheduling*-Mechanismus realisiert werden [GTU91]. Aus den Simulationen in [GTU91] geht jedoch hervor, daß das Scheduling von *task*-Gruppen am Beispiel des *Coscheduling*- und des *Process-Control*-Algorithmus den beiden vorgenannten Verfahren überlegen ist. Auch die Analyse der drei Algorithmen *Round-Robin-job*, *Dynamic Allocation* und *Auto-Scheduling* favorisiert das Scheduling von *task*-Gruppen: Der *Round-Robin-job*-Algorithmus stellt eine einfache Realisierung des Scheduling von *task*-Gruppen dar. Beim *Dynamic-Allocation*-Algorithmus werden die Nachteile des *Round-Robin-job*-Algorithmus vermieden, ein Scheduling von *task*-Gruppen ist dabei jedoch nicht realisiert. Ein direkter Vergleich der beiden Algorithmen *Round-Robin-job* und *Dynamic Allocation* kann in Abhängigkeit von der untersuchten Arbeitslast sowohl den einen als auch den anderen Algorithmus favorisieren. Der *Auto-Scheduling*-Algorithmus verbindet die Vorteile des *Dynamic-Allocation*-Algorithmus mit dem Scheduling von *task*-Gruppen und steht damit als weiteres Argument für das Scheduling von *task*-Gruppen. Das *Cooperative-Scheduling* implementiert ebenfalls ein Scheduling von *task*-Gruppen; durch die Kommunikation der Scheduler-Komponenten untereinander verfügt der Job-Scheduler im Vergleich zu den übrigen Scheduling-Algorithmen über zusätzliche Informationen und kann damit bestehende Task-Abhängigkeiten in seinen Entscheidungen besser berücksichtigen.
Die Analyse der vorgestellten Algorithmen bezüglich der Berücksichtigung von *task*-Abhängigkeiten favorisiert demnach das Scheduling von *task*-Gruppen sowohl für massiv-parallele Systeme mit *message-passing*-Programmiermodell als auch für *shared-memory*-Systeme. In *virtual-shared-memory*-Systemen ergeben sich im Vergleich zu *message-passing*-Systemen für den Austausch von *Messages* vollkommen andere Verhältnisse. Zur Auflösung von *page faults* werden zwar ebenfalls *Messages* ausgetauscht, die Kommunikation findet jedoch nicht zwischen Prozessen (Benutzer-Prozessen) sondern zwischen Knoten (System-Prozessen) statt. Diesbezüglich sind durch das Scheduling von *task*-Gruppen keine Vorteile zu erwarten. Für eine Favorisierung des Scheduling von *task*-Gruppen in VSM-Systemen spricht dagegen die Synchronisation: In VSM-Systemen treten bezüglich der Synchronisation die gleichen Effekte auf wie in anderen *shared-memory*-Systemen; das Scheduling von *task*-Gruppen läßt somit in VSM-Systemen ähnliche Vorteile erwarten. Auf der Ebene des Job-Scheduling ist deswegen das Scheduling von *task*-Gruppen zu favorisieren.

4.5.1.2 Verteiltes Scheduling

Der Forderung nach einem verteilten Scheduling genügen die in Abschnitt 4.1.3 vorgestellten Basismodelle *Local Queue*, *Multiple Pool* und *Hierarchical Control* [FeRu90, ZhBr91]. Das *Local-Queue*-Modell ermöglicht auf der Ebene des Job-Scheduling keine koordinierte Kontrolle und ist somit für ein Scheduling von *task*-Gruppen unbrauchbar. Das *Multiple-Pool*-Modell ist für ein Scheduling von *task*-Gruppen nur mit Einschränkungen zu verwenden: Ein Scheduling von *task*-Gruppen ist bei diesem Modell nur dann möglich, wenn jeder Job jeweils vollständig einem Pool zugeordnet werden kann. Zum einen stellt diese Forderung eine weitere Einschränkung der ohnehin nur bedingt möglichen Lastbalancierung dar. Zum anderen muß für diese Forderung eine dynamische Pool-Einteilung realisiert werden, was mit erheblichem Aufwand bezüglich Implementierung und Overhead verbunden sein kann. Das einzige Basismodell, das für ein verteiltes Scheduling von *task*-Gruppen sinnvoll erscheint, ist das *Hierarchical-Control*-Modell.

4.5.1.3 Forderung nach Lokalität

Die beschriebenen Algorithmen zum *Locality Scheduling* berücksichtigen die Anforderungen an die Lokalität für *multicache-shared-memory*-Systeme und *distributed-shared-memory*-Rechner mit NUMA-Architektur. In VSM-Systemen sind die Anforderungen an die Lokalität durch das Auftreten von *page faults* und den Effekt des *page thrashing* bestimmt. Die Grundidee der beschriebenen Algorithmen wie zum Beispiel die Beachtung von Affinitäten oder die Realisierung des Scheduling von *task*-Gruppen durch eine Begrenzung der Prozeßanzahl beim *Process-Control*-Algorithmus können im Prinzip auch für VSM-Systeme genutzt werden. Unmittelbar auf VSM-Systeme zugeschnitten sind jedoch die beiden in Abschnitt 3.2 vorgeschlagenen Strategien, die Prozeßmigration zur Behandlung von *page thrashing* und der Kontextwechsel beim Auftreten von *page faults*. Diese Strategien stellen jedoch keine vollständigen Scheduling-Algorithmen dar.

4.5.2. Das Thread-Scheduling

Für das *thread*-Scheduling sind die Forderung nach einem verteilten Scheduling und die Forderung nach Lokalität relevant. Ein verteiltes *thread*-Scheduling ist in den beschriebenen Ansätzen bisher nicht implementiert; innerhalb einer parallelen Schleife wird der *thread*-Scheduler nur exklusiv vergeben. Die einzige verteilte Funktion ist ein Mechanismus zur Migration von *threads* mit dem Ziel der Lastbalancierung; ein solcher Mechanismus ist für das IVY-System und im Rahmen des *Affinity-Scheduling* implementiert worden [Li86, MaLe92]. Bei den Algorithmen *Block-Scheduling*, *Cyclic-Scheduling*, *Self-Scheduling*, *Chunk-Scheduling*, *Guided Self-Scheduling* und *Factoring* sowie beim *Loop-Blocking* kann das Scheduling nicht ohne weiteres verteilt durchgeführt werden und muß zentral erfolgen. Durch eine Aufteilung einer parallelen Schleife *(tiling* [Wol92]) in zwei oder mehr Schleifen kann eine einfache parallele Schleife in geschachtelte parallele Schleifen umgewandelt werden. Das *thread*-Scheduling erfolgt dann durch für jede Schleife separate und damit unabhängige Scheduler, die entsprechend der Schleifenschachtelung eine Hierarchie bilden. Eine solche Verteilung des *thread*-Scheduling ist jedoch bisher nicht untersucht worden. Für das *Affinity-Scheduling* und

das *Align-Scheduling* ist im Prinzip keine zentrale Kontrolle notwendig; da jeder Prozeß unabhängig erkennen kann, welcher Anteil der Schleifeniterationen ihm selbst zugeordnet wird, muß der *thread*-Scheduler nicht notwendigerweise exklusiv vergeben werden, was einer Verteilung entspricht. Dabei ist jedoch die Vollständigkeit der Ausführung sicherzustellen, so daß keine Iterationen verlorengehen[10]. Auch diese Möglichkeit für ein verteiltes *thread*-Scheduling ist bislang nicht untersucht worden.

Eine Beachtung der Datenlokalität erfolgt nur bei den Algorithmen *Affinity-Scheduling*, *Align-Scheduling* und *Loop-Blocking*. Für *virtual-shared-memory*-Systeme sind diese Algorithmen bisher jedoch nicht alternativ untersucht worden. Es ist zu erwarten, daß die Effizienz dieser Algorithmen für verschiedene Anwendungen unterschiedlich groß ist.

4.6. Bewertung der Algorithmen

Wie die Analyse der in diesem Kapitel beschriebenen Algorithmen zeigt, existieren bisher keine beziehungsweise nur wenige Algorithmen, die auf die besonderen Anforderungen beim Scheduling in *virtual-shared-memory*-Systemen zugeschnitten sind. Damit können von den zu Beginn dieses Kapitels gestellten Forderungen an das Scheduling insbesondere die nicht befriedigt werden, die sich auf das VSM-Rechnermodell beziehen. Die Untersuchung diesbezüglich geeigneter Algorithmen soll den weiteren Rahmen dieser Arbeit abstecken.
Für das Job-Scheduling ergeben sich keine vollständigen Algorithmen, die die speziell für VSM-Systeme relevanten Forderungen befriedigen. Im Bereich der Algorithmen für das *thread*-Scheduling sind bisher nur einige wenige Untersuchungen durchgeführt worden. Die Untersuchungen zum Scheduling in *virtual-shared-memory*-Systemen in dieser Arbeit können sich damit nur auf wenige Erfahrungen in diesem Bereich beziehen und stellen einen ersten Schritt in diese Richtung dar. Diesbezüglich sind bestimmte Abstraktionen und Idealisierungen für den Rahmen der Untersuchungen sinnvoll, so daß zunächst Teilaspekte isoliert betrachtet werden können. Dabei ergibt sich eine Analogie zur Untersuchung von parallelen Progammen: Für die Nutzung von parallelen Programmen haben sich die Untersuchungen in der Vergangenheit zunächst nur auf dedizierte Maschinen bezogen [HwBr84, KnNa86, Nag88]; die Betrachtungen wurden erst in einem zweiten Schritt auf reale, nicht idealisierte Systeme und damit auf den Multiprogramming-Betrieb ausgeweitet [Bie88, DiIy90, NaLi91, NaLi93]. Auch für die Untersuchung von Scheduling-Algorithmen erscheint es sinnvoll, die Betrachtungen zunächst auf die dedizierte Ausführung eines Jobs zu beschränken; auf diese Weise kann die Ebene des *thread*-Scheduling zunächst isoliert betrachtet werden. Alle weiteren Betrachtungen in dieser Arbeit sollen damit auf das *thread*-Scheduling beschränkt werden.

[10] Für das *Align-Scheduling* ist für diesen Fall z.B. denkbar, daß ein Teil der den Iterationen zugeordneten Daten aktuell für keinen Prozeß lokal verfügbar ist und sich möglicherweise auf einem Hintergrundspeicher befindet; wenn in diesem Fall keine globale Kontrollinstanz die Vollständigkeit der Ausführung kontrolliert, dann gehen die entsprechenden Iterationen verloren. Diese Tatsache stellt eine Einschränkung für ein verteiltes *thread*-Scheduling dar.

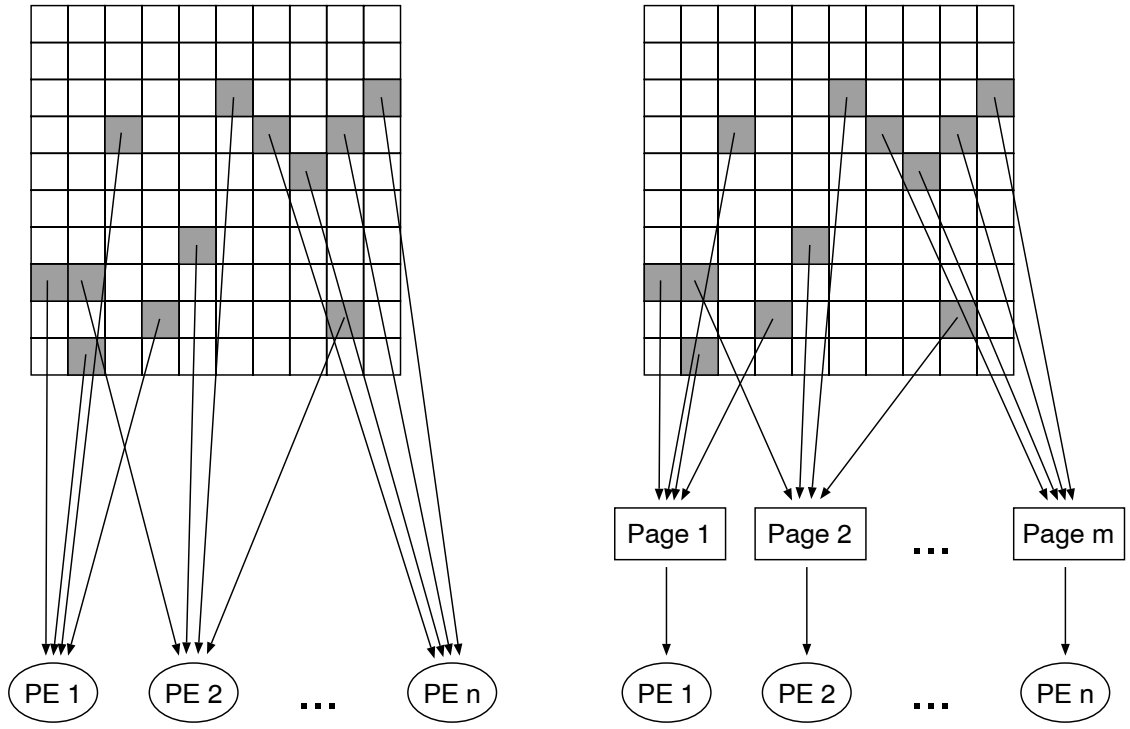
5. Die Ablaufplanung in VSM-Systemen

Bei der Parallelverarbeitung auf Schleifenebene werden die dabei verwendeten Algorithmen für die Schleifenpartitionierung ständig weiterentwickelt oder erneuert. Die Entwicklung von der statischen zur dynamischen Partitionierung mit schließlich immer effizienteren, lastadaptiven Algorithmen bis hin zur Einbeziehung von kooperativen Entscheidungen kennzeichnet den Werdegang von heute auf *shared-memory*-Rechnern etablierten Algorithmen und dauert aktuell an [KrWe85, TaYe86, PoKu87, HSF91, Nag93]. Die Anforderungen an das *thread*-Scheduling bei *shared-memory*-Systemen beziehen sich dabei in den meisten Fällen auf eine maximale Lastbalancierung bei minimalem Overhead. Im Gegensatz dazu sind die Anforderungen für das *thread*-Scheduling in *virtual-shared-memory*-Systemen um eine ganz wesentliche Komponente erweitert: In VSM-Systemen wird ein effizientes *thread*-Scheduling durch eine Abhängigkeit von der aktuellen Verteilung der Daten gekennzeichnet sein. Da die Daten in Form von Seiten gruppiert sind, kann die Vorgabe für diese Gruppierung, d.h. die Abbildung von Daten auf Seiten, von entscheidender Bedeutung sein. Die Abhängigkeit der Algorithmen zum *thread*-Scheduling von der Abbildung der Daten auf Seiten wird in Abschnitt 5.1 betrachtet. Neben der Abbildung der Daten wird natürlich die Anwendung selbst einen charakteristischen Einfluß auf die aktuelle Datenverteilung haben. Abschnitt 5.2 untersucht Algorithmen für das *thread*-Scheduling auf deren Eignung für charakteristische Schleifen.
Die möglichen Vor- und Nachteile alternativer VSM-Modelle und einzelner Strategien sind in der Literatur bereits mehrfach untersucht und in Kapitel 3 dokumentiert. Das zugrundeliegende VSM-Modell hat auf die Problemstellungen des *thread*-Scheduling einen wesentlichen Einfluß. Die Vielfalt der existierenden Modelle und Strategien und der komplexe Einfluß des jeweiligen Modells auf die Ablaufplanung erfordern eine weitere Einschränkung des Themengebietes dieser Arbeit. Die Untersuchungen im Rahmen dieser Arbeit sollen sich auf ein ausgewähltes VSM-Modell beziehen; alternative Modelle beziehungsweise Strategien sollen nur insoweit betrachtet werden, wie konkrete Problemstellungen Rückschlüsse auf die besondere Eignung dieser Modelle zulassen. Das von K. Li entwickelte VSM-Modell ist in existierenden Projekten bereits mehrfach implementiert, untersucht und auch erweitert worden (IVY, Shiva, KOAN, Rosi, MaX SVM [Li86, LiSc89b, LaPr91, Ber92, Moh93]). Die Eignung dieses Modells für wissenschaftliche Untersuchungen, die auf vielfältigen Projekten basierende breite Erfahrungsgrundlage und der Detaillierungsgrad der verfügbaren Informationen zu diesem Modell sprechen für die Zugrundelegung dieses Modells. Für alle weiteren Untersuchungen bildet deshalb das VSM-Modell von K. Li mit statisch verteiltem Manager die Grundlage; als Alternativen beziehungsweise Erweiterungen dieses Modells werden die explizite Abbildung von Daten auf Seiten, die Strategie der schwachen Kohärenz und die Option, Seiten bereits vor der tatsächlichen Zugriffsanforderung anzufordern (*prefetch*-Operation), vorgestellt und diskutiert.

5.1. Die Abbildung von Daten auf Seiten

Bei den vielfältigen Untersuchungen bezüglich der effizienten Programmierung von Rechnern mit verteiltem Speicher wird die dabei allen gemeinsame Aufgabenstellung im we-

sentlichen durch zwei Teilaspekte beschrieben: Die Verteilung von Daten und die Verteilung von Arbeit. In Untersuchungen zum *compiler-bridged-shared-memory*-Modell (z.B. Vienna FORTRAN, FORTRAN D, High Performance FORTRAN [CMZ92, Fox90, HPF93]) und beim Cray MPP FORTRAN Programmiermodell [PMM93] kann die Verteilung von Daten auf Prozessorelemente durch den Anwender gesteuert werden (siehe Abb. 16.a). Die möglichen Verteilungsmuster erlauben eine an die jeweilige Anwendung angepaßte Verteilung der Daten auf Wortebene. Für *virtual-shared-memory*-Rechner nach dem Modell von K. Li erfolgt eine Abbildung von Daten auf Seiten und eine Verteilung der Seiten auf Prozessorelemente (Abb. 16.b). Zur Laufzeit wird die Verteilung der Daten in Form von Seiten automatisch vorgenommen. Dieser Automatismus macht die Lokalität der Daten für den Anwender vollständig unsichtbar und erfüllt so das *shared-memory*-Paradigma. Die im Vergleich zum einzelnen Datum grobe Granularität von Seiten kann zu einer unzulänglichen Datenverteilung führen. Eine Verbesserung der Datenverteilung durch eine der jeweiligen Anwendung angepaßte Abbildung der Datenobjekte auf Seiten ist bisher nicht untersucht worden.
In existierenden VSM-Realisationen erfolgt die Abbildung von Daten auf Seiten in Anlehnung an die FORTRAN-übliche spaltenorientierte Abbildung von Datenfeldern auf den Adreßraum; der Adreßraum wird dann blockweise auf die Seiten abgebildet, so daß die Abbildung von Daten auf Seiten in der Summe spaltenweise und ohne Beachtung der Struktur der Daten erfolgt (siehe Abb. 17.a). Dabei liegen im allgemeinen Teile von Spalten auf verschiedenen Seiten.



a) Abbildung von Daten auf PEs

b) Abbildung von Daten auf Seiten und Verteilung der Seiten auf PEs

Abb. 16. Verteilung von Daten

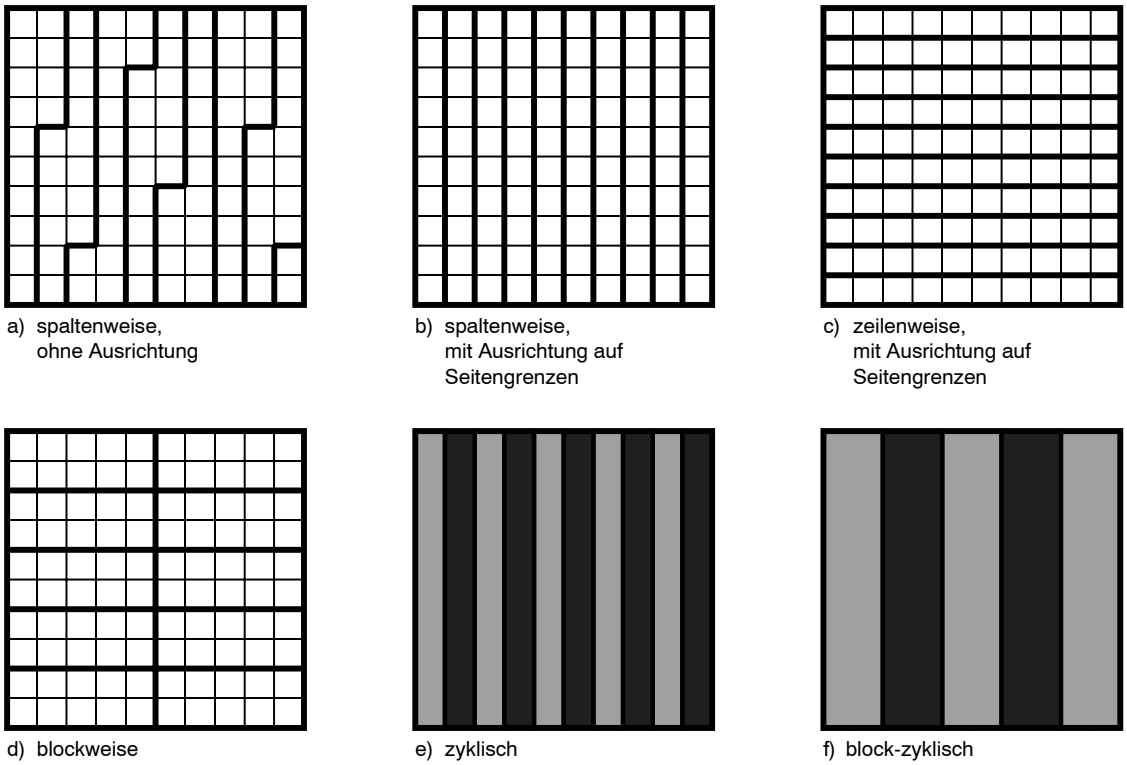


Abb. 17. Abbildung von Daten auf Seiten

Der Datenzugriff innerhalb von Schleifen erfolgt für einzelne Iterationen in den meisten Fällen in Form von Vektoren, also zum Beispiel spalten- oder bzw. und zeilenweise. Bei zeilenweisem Datenzugriff durch Iterationen einer parallelen Schleife auf ein spaltenweise abgebildetes Datenfeld benötigt jede Iteration jede Seite des Datenfeldes. Für Lesezugriffe ist dabei ein unnötig hohes Kommunikationsaufkommen zu erwarten; da jeder Prozeß jede Seite benötigt, die Seiten aber nicht gleichzeitig an alle Prozesse gesendet werden können, ergeben sich außerdem erhöhte Wartezeiten bei *page faults* aufgrund von Kommunikationsengpässen. Insbesondere für Schreibzugriffe führt diese Situation zu erheblichem *page thrashing*; dabei ist es möglich, daß alle Prozesse auf alle Seiten parallel zugreifen wollen, was die Rechenleistung um Größenordnungen verschlechtern kann. Eine Anpassung von Datenabbildung und Datenzugriff erscheint unerläßlich, d.h. ein zum Beispiel zeilenorientierter Datenzugriff erfordert eine ebenfalls zeilenorientierte Datenabbildung. Wenn innerhalb eines Programms ein Datenobjekt sowohl spalten- als auch zeilenweise zugegriffen wird, dann kann möglicherweise auch ein erheblicher Aufwand für eine Neuabbildung der Daten auf Seiten vorteilhaft sein. Ob eine solche dynamische Abbildung zu besserer Rechenleistung führt, kann nicht einheitlich beurteilt werden und hängt von Anwendung und Problemgröße ab.

Auch bei einer Anpassung von Datenabbildung und Datenzugriff kann es zu unnötigen Zugriffskonflikten kommen: Bei spaltenweiser Datenabbildung ohne Ausrichtung auf Seitengrenzen gemäß Abb. 17.a und spaltenweisem Datenzugriff benötigen benachbarte Iterationen zum Teil dieselben Seiten. Diese Situation läßt die oben beschriebenen negativen Effekte zumindest in abgeschwächter Form erwarten; das Ausmaß der Effekte kann dabei vom Verhältnis der Seitengröße zur Vektorgröße abhängen. Um die Datenzugriffe

der Iterationen von parallelen Schleifen unabhängig auf verschiedenen Seiten durchzuführen, ergibt sich für das beschriebene Beispiel eine spaltenweise Abbildung von Daten auf Seiten mit Ausrichtung von Spalten auf Seitengrenzen gemäß Abb. 17.b.
Aufgrund der bisherigen Ausführungen sind für die Abbildung von Daten auf Seiten entsprechend den Abb. 17.b und 17.c zum Teil deutliche Vorteile für die Effizienz von VSM-Systemen zu erwarten. In Abhängigkeit von der jeweiligen Anwendung kann auch eine blockweise, zyklische oder block-zyklische Abbildung vorteilhaft sein (siehe Abb. 17.d bis 17.f).
Da es sich bei den beschriebenen negativen Effekten um *false sharing* handelt, kann das *page thrashing* bei Schreibzugriffen natürlich durch die Strategie der schwachen Kohärenz verhindert werden. Bei einer Datenabbildung ohne Ausrichtung auf Seitengrenzen gemäß Abb. 17.a ergibt sich jedoch nach wie vor *false sharing*, und damit treten unnötige Datentransfers und Kommunikationsengpässe auf: Die zu erwartenden Effekte für Schreibzugriffe mit schwacher Kohärenz entsprechen dabei den oben beschriebenen Effekten für Lesezugriffe. Damit kann eine nicht angepaßte oder nicht ausgerichtete Datenabbildung auch für schwach kohärente Zugriffe zu erheblichen Leistungsverlusten führen.
Die Datenabbildungen mit Ausrichtung auf Seitengrenzen - dies gilt für alle Darstellungen in Abb. 17.b bis 17.f - verursachen eine verschlechterte Speichereffizienz: Durch die Ausrichtung ergeben sich im allgemeinen auf jeder Seite Bereiche, die ungenutzt bleiben. Dabei geht insbesondere auch die FORTRAN-Semantik beim Zugriff auf hintereinander liegende Datenfelder verloren. Ob diese Bereiche für die Abbildung anderer Daten günstig genutzt werden können, kann möglicherweise der Compiler entscheiden; dieser Aspekt soll hier jedoch nicht weiter betrachtet werden.

Die Realisierung der beschriebenen Datenabbildungen geht aus Abb. 18 hervor: Die spaltenweise Abbildung ohne Ausrichtung - für das Datenfeld *A(10,10)* - wird durch die Direktive *Column* spezifiziert; bei der Seitengröße von 14 Worten in diesem Beispiel werden die erste Spalte des Feldes *A* und die ersten vier Worte der zweiten Spalte auf die erste Seite abgebildet, die zweite Seite enthält den übrigen Teil der zweiten Spalte und einen Teil der dritten Spalte. Bei allen anderen Abbildungen werden die Dimensionen der Datenfelder unabhängig voneinander abgebildet; für jede Dimension wird unabhängig je eine Direktive angegeben:

- **Block:**
  Die Direktive *Block* spezifiziert die Abbildung einer Dimension in Blöcken und gibt außerdem die Größe der Blöcke in Worten an; mit *Block 10* wird zum Beispiel angegeben, daß diese Dimension in Blöcken von je 10 Worten auf jeweils andere Seiten abgebildet wird. Durch die spezielle Wahl der Blockgröße ergibt sich für die Felder *B(10,10)* beziehungsweise *C(10,10)* in Abb. 18 eine spaltenweise beziehungsweise eine zeilenweise Abbildung. Wie für das Feld *D(10,10)* lassen sich unter Beachtung der Seitengröße beliebige Blockaufteilungen angeben.

- **Cyclic:**
  Beim Feld *E(10,10)* wird die erste Dimension durch die Direktive *Block 10* abgebildet, so daß jede Spalte vollständig auf jeweils einer Seite liegt, und die zweite Di-

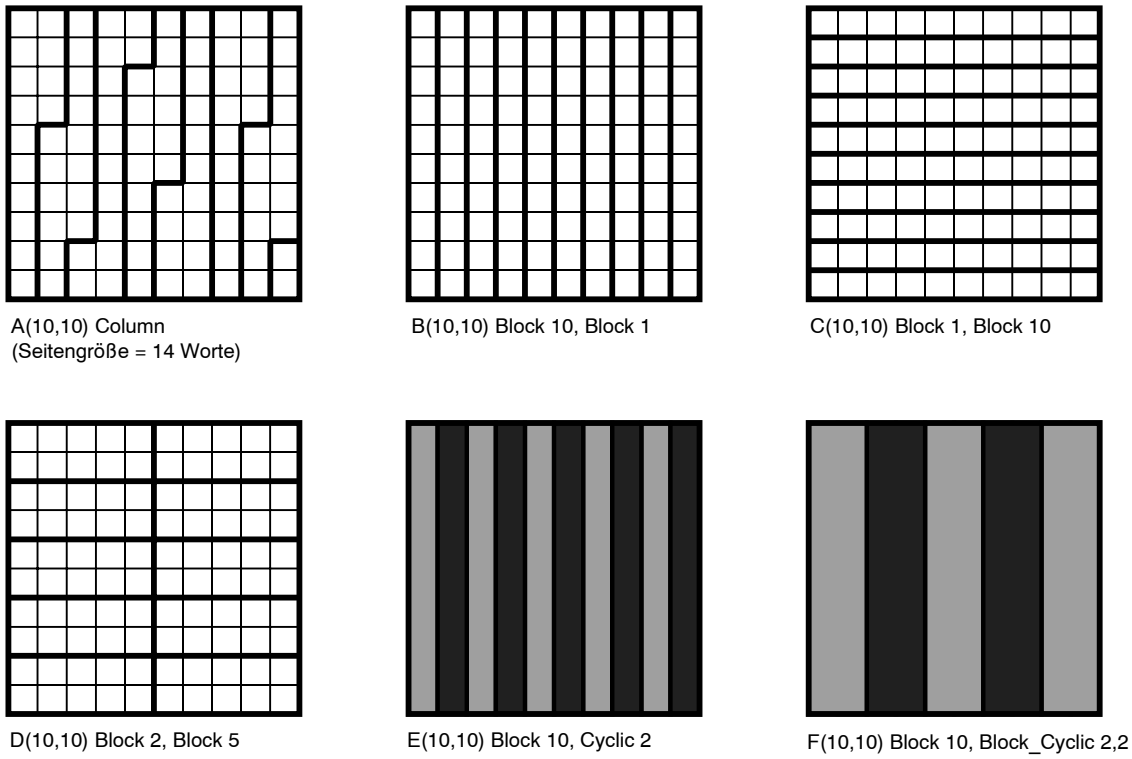


Abb. 18. Sprachdirektiven zur Abbildung von Daten auf Seiten

mension wird durch die Direktive *Cyclic 2* zyklisch abgebildet; der Parameterwert *'2'* spezifiziert die Anzahl der Seiten, auf die diese Dimension verteilt werden soll. In der Summe ergibt sich für das Feld *E* eine Abbildung, bei der die Spaltenvektoren zyklisch auf 2 Seiten verteilt werden. Eine zyklische Abbildung beider Dimensionen würde einer schachbrettartigen Abbildung entsprechen.

- **Block_Cyclic:**
  Für das Feld *F(10,10)* ergibt die Direktive *Block 10* für die erste Dimension wieder eine Abbildung von jeweils allen Datenelementen einer Spalte auf eine Seite; die Direktive *Block_Cyclic 2, 2* für die zweite Dimension spezifiziert eine Aufteilung dieser Dimension in Blöcke der Größe 2 und eine zyklische Verteilung der Blöcke auf insgesamt 2 Seiten.

Neben der herkömmlichen Datenabbildung durch die Direktive *Column* ergeben sich damit - bezogen auf eine Dimension - die drei für VSM-Systeme neuartigen Abbildungsvarianten *Block, Cyclic* und *Block_Cyclic*; für Datenfelder mit zwei oder mehr Dimensionen können die drei neuen Varianten untereinander beliebig kombiniert werden. Die dadurch entstehende Vielfalt für die Abbildungsmöglichkeiten von Daten auf Seiten muß - im Vergleich zur herkömmlichen Datenabbildung - mit einer komplexeren Adreßberechnung bezahlt werden: Bei allen Abbildungen werden die Dimensionen eines Datenfeldes auf die eine Dimension der fortlaufenden Seitennumerierung reduziert. Bei der Abbildung ohne Ausrichtung ist diese Reduktion, wie bei der FORTRAN-üblichen Abbildung von zwei- oder mehrdimensionalen Datenfeldern auf den Adreßraum, direkt und einfach. Die jedoch bei der Ausrichtung entstehenden Lücken - die jeweils ungenutzten Adreß-

bereiche - erhöhen die Komplexität für die Adreßberechnung. Die Adreßberechnung beinhaltet die Berechnung der Seitenidentifikation (*PAGE_ID*) und die Berechnung der Position des jeweiligen Datenelements innerhalb der Seite (*PAGE_OFFSET*); die Zuordnung von *PAGE_IDs* zu den Seiten erfolgt analog zur Speicherung von Datenfeldern in FORTRAN; die Seiten werden spaltenweise in aufsteigender Reihenfolge numeriert (siehe Abb. 19). Die Adreßberechnung für die Abbildungsdirektiven *Column, Block, Cyclic* und *Block_Cyclic* wird durch den in Abb. 20 angegebenen Algorithmus beschrieben: Die Prozedur *Calculate_Page_Id_Offset* durchläuft, nach der Initialisierung, für jede Dimension des Datenfeldes einmal die *FOR*-Schleife. In Abhängigkeit von der Abbildungsdirektive für die entsprechende Dimension *(DIM.DISTRIBUTION)* wird jeweils ein Alternativzweig der *CASE*-Anweisung ausgeführt:

- Für die Direktive *Column* wird zunächst der Beitrag jeder Dimension zur Position des entsprechenden Datenelements innerhalb des Feldes gemäß der FORTRAN-Semantik berechnet *(COLUMN_POSITION)*. Nach der vollständigen Berechnung dieses Wertes, d.h. nach Beendigung der *FOR*-Schleife, ergeben sich in Verbindung mit der Seitengröße *(PAGE_SIZE)* die Werte für *PAGE_ID* und *PAGE_OFFSET*.
- Bei den Direktiven *Block, Cyclic* und *Block_Cyclic* werden die Beiträge zu *PAGE_ID* und *PAGE_OFFSET* für jede Dimension unabhängig berechnet und aufsummiert. Für die Direktiven *Block* und *Block_Cyclic* ergibt sich eine komplexe Berechnungsvorschrift; die Direktive *Cyclic* führt diesbezüglich zu einem vergleichsweise günstigeren Verhalten. Für alle drei Direktiven wächst die Komplexität der Adreßberechnung mit der Anzahl der Dimensionen schneller als für die Direktive *Column*. Da sich die erhöhte Komplexität bei der Adreßberechnung für jeden Datenzugriff wiederfindet, können sich erhebliche Leistungsverluste ergeben. Bei einer Beschränkung der in den Berechnungsvorschriften relevanten Zahlenwerte auf Zweierpotenzen kann der Berechnungsaufwand deutlich gesenkt werden; in diesem Fall werden Multiplikationen und Divisionen zu *Shift*-Operationen, und die *Modulo*-Operation wird zu einer Maskierung.

Bezüglich der Komplexität entsprechen die in diesem Abschnitt betrachteten, neuartigen Abbildungsmuster denen für die Abbildung von Daten auf Prozessorelemente im Cray MPP FORTRAN Programmiermodell. Untersuchungen zu einer effizienten Adreßberechnung für die Abbildungen beim Cray MPP FORTRAN Programmiermodell sind in [MPM92] beschrieben.

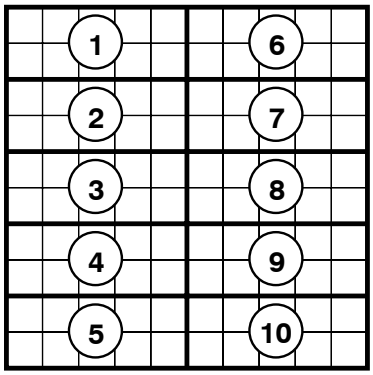


Abb. 19. Zuordnung von PAGE_IDs zu Seiten

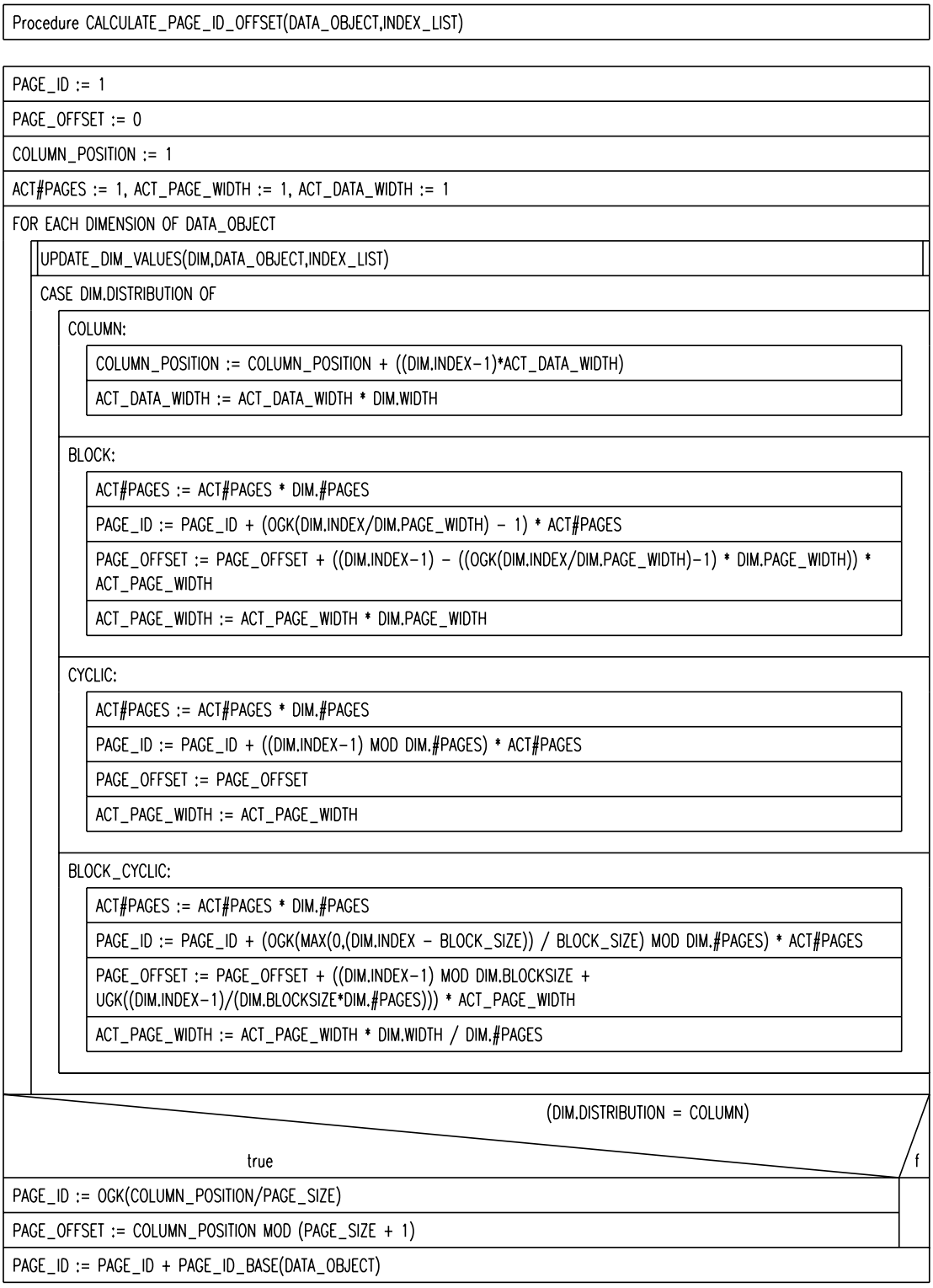


Abb. 20. Die Berechnung von PAGE_ID und PAGE_OFFSET für die beschriebenen Datenabbildungen

5.2. Thread-Scheduling für charakteristische Schleifen

In der Vergangenheit war die Entwicklung von Algorithmen zum *thread*-Scheduling für *shared-memory*-Rechner weitgehend universell, d.h. die Effizienz der Algorithmen war in den meisten Fällen nicht vom Charakter der jeweiligen Anwendung abhängig. Neue Algorithmen waren herkömmlichen Algorithmen bei wesentlichen Problemstellungen wie z.B. der Lastbalancierung überlegen, und mögliche Nachteile für bestimmte Anwendungen - z.B. ein erhöhter Scheduling-Overhead - waren im allgemeinen vernachlässigbar. Damit waren die Algorithmen für den universellen Einsatz geeignet, unabhängig von der jeweiligen Anwendung. Die Abhängigkeit des Scheduling von der Datenlokalität kann die Universalität von Algorithmen für VSM-Systeme einschränken: Das in Abschnitt 4.4 beschriebene *Align-Scheduling* implementiert einen universellen Algorithmus mit Beachtung der Datenlokalität. Für die Partitionierung nach dem *Align-Scheduling*-Algorithmus muß zur Laufzeit die aktuelle Verteilung der maßgeblichen Daten ermittelt werden; damit ist für das *Align-Scheduling* ein vergleichsweise hoher Scheduling-Overhead zu erwarten. Selbst für optimale Scheduling-Entscheidungen kann sich in der Summe ein ungünstiges Zeitverhalten ergeben. Unter diesen Umständen können solche Algorithmen vorteilhaft sein, die die aktuelle Verteilung der Daten nicht explizit ermitteln, aber dennoch beachten. Das Wissen um die Datenverteilung kann nur aus der Struktur und dem Charakter der jeweiligen Anwendung gewonnen werden. Für diese Algorithmen stellt die Abhängigkeit von der Datenlokalität eine Abhängigkeit von der konkreten Anwendung dar, d.h. sie können nicht universell sein. Im folgenden soll versucht werden, Scheduling-Algorithmen dieser Art zumindest auf eine Klasse von charakteristischen Schleifennestern zuzuschneiden.

5.2.1. Schleifennester mit Affinität

Einen ganz erheblichen Einfluß auf die Effizienz der Ausführung kann das *thread*-Scheduling in VSM-Systemen für Schleifennester mit Affinität haben. Abb. 21 beschreibt eine einfache Form von Schleifennestern dieser Art. Das für *multicache-shared-memory*-Systeme entwickelte *Affinity-Scheduling* (siehe Abschnitt 4.4) ist auf diese Klasse von Anwendungen zugeschnitten. Für VSM-Systeme ist die Effizienz des *Affinity-Scheduling* bislang nicht untersucht worden; der Algorithmus läßt jedoch ein ähnlich vorteilhaftes Verhalten erwarten wie in *multicache*-Systemen [MaLe92].
Eine charakteristische Form von Schleifennestern mit etwas andersartigen Affinitätsbeziehungen ist in Abb. 22 beschrieben. Die Schleifengrenze der inneren, parallelen Schleife hängt vom Index der äußeren, sequentiellen Schleife ab; damit erhält der die Anzahl der Iterationen und die damit verbundenen Datenzugriffe beschreibende Indexraum

Schleifennester mit Affinität

```
do I = 1, N
   doall J = 1, M
      A(J) = ...
   enddo
enddo
```

Abb. 21. Schleifennester mit Affinität: Parallele Schleifen innerhalb sequentieller Schleifen

Triangulare Schleifennester mit Affinität

```
do K = 1, N
   doall I = aK+b, N
      A(I,K) = ...
   enddo
enddo
```

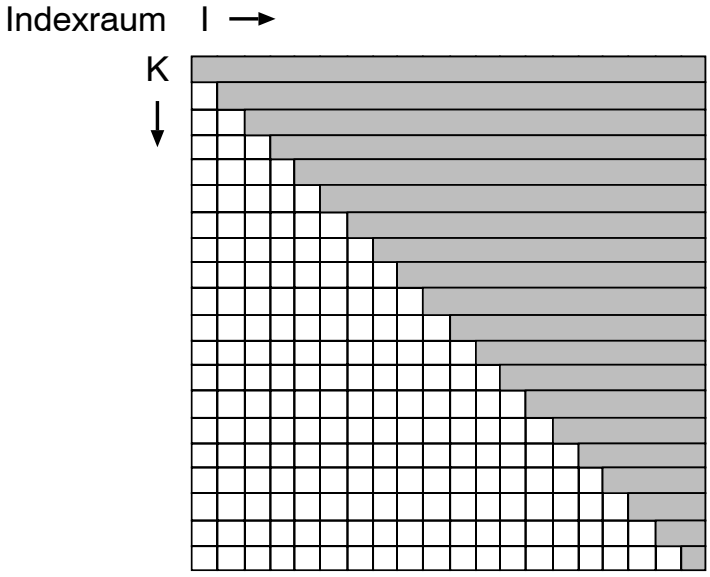
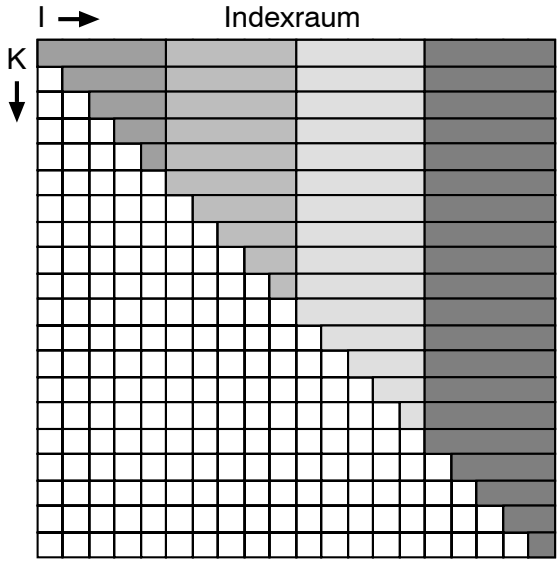


Abb. 22. Triangulare Schleifennester mit Affinität

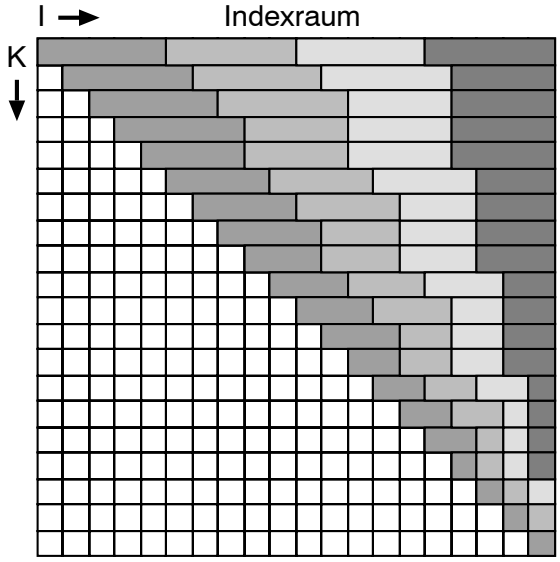
eine triangulare Form (siehe Abb. 22). Die mit wachsendem Index *K* der sequentiellen Schleife stetig kleiner werdende parallele Schleife stellt bezüglich der Beachtung von Affinitäten besondere Anforderungen an die Schleifenpartitionierung[11]: Bei einer Partitionierung nach dem *Affinity Block-Scheduling*-Algorithmus ergeben sich die in Abb. 23.a dargestellten Verhältnisse. Die strikte Beachtung der Affinitäten durch die feste Zuordnung von Iterationen zu Prozessen resultiert in einem festen Rahmen für die Partitionierung, unabhängig von der aktuellen Schleifengröße. Wie unmittelbar einzusehen ist, ergibt sich für diese Partitionierung mit wachsendem *K* ein Leerlaufen der Prozessoren und damit eine unbalancierte Ausführung. Eine Abschwächung der Affinitätsforderung durch eine geeignete Verschiebung des Rahmens für die Partitionierung ermöglicht eine bessere Lastbalancierung (siehe Abb. 23.b). Wie die Abbildung zeigt, ergeben sich für verschiedene Werte von *K* Überschneidungen der zugeordneten Blöcke; in diesen Fällen werden in aufeinanderfolgenden Iterationen der sequentiellen Schleife jeweils dieselben Daten von verschiedenen Prozessen benötigt, so daß Datentransfers durchgeführt werden müssen. Eine strikte Beachtung der Affinitäten bei bestmöglicher Lastbalancierung wird durch das *Cyclic-Scheduling* realisiert (siehe Abb. 23.c): Die zyklische Partitionierung der Schleife realisiert wie das *Affinity Block-Scheduling* mit strikter Beachtung der Affinitäten eine ebenfalls feste Zuordnung von Iterationen zu Prozessen. Bezüglich der

[11] Die folgenden Ausführungen gelten in gleicher Weise auch für solche Schleifennester, bei denen die parallele Schleife mit dem Schleifenindex der sequentiellen Schleife größer wird.

a) Affinity Block-Scheduling



b) Affinity Block-Scheduling mit abgeschwächter Affinitätsforderung



c) Affinity Cyclic-Scheduling

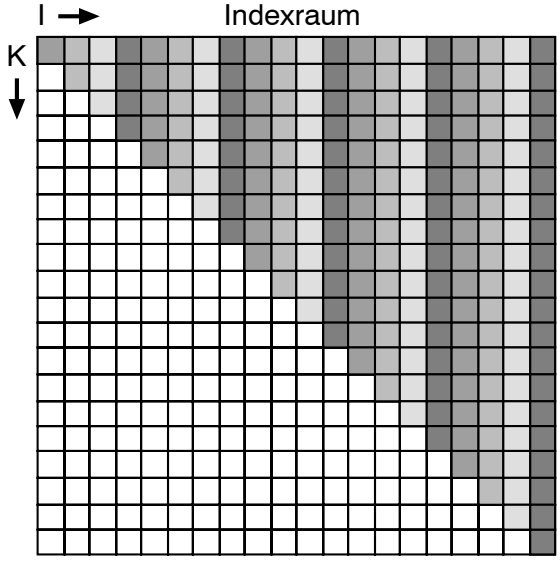


Abb. 23. Scheduling-Algorithmen für triangulare Schleifennester mit Affinität

Lastbalancierung ergeben sich die gleichen Verhältnisse wie in Abb. 23.b. Das *Cyclic-Scheduling* ist im Rahmen des Koan-Projektes für den Modifizierten Gram-Schmidt Algorithmus untersucht worden [LaPr91]. In den Untersuchungen wird jedoch das *Affinity Block-Scheduling* mit abgeschwächter Affinitätsforderung favorisiert. Die Ursache für diese zunächst nicht erwarteten Verhältnisse liegt in der Abbildung der Daten auf Seiten. In den durchgeführten Messungen wurden jeweils Blöcke von Vektoren auf eine Seite abgebildet; da beim *Cyclic-Scheduling* benachbarte Vektoren von unterschiedlichen Prozessen zugegriffen werden, ergeben sich beim *Cyclic-Scheduling* vergleichsweise mehr *page faults* als beim *Block-Scheduling*. Eine dem *Cyclic-Scheduling* angepaßte Abbildung der Daten muß ebenfalls zyklisch sein (siehe Abb. 17); auf diese Weise werden die Vektoren und die Iterationen nach dem gleichen Muster gruppiert, und die in [LaPr91] beobachteten *page faults* können eliminiert werden. In Verbindung mit der geeigneten Abbildung von Daten auf Seiten sind für triangulare Schleifennester die besten Ergebnisse für das *Affinity Cyclic-Scheduling* zu erwarten.

Die bisher beschriebenen Partitionierungen für das jeweils bestmögliche *Affinity-Scheduling* können natürlich prinzipiell durch ein geeignetes *Align-Scheduling* nachgebildet werden. Da das *Align-Scheduling* jedoch mit vergleichsweise deutlich größerem Overhead verbunden ist, ist das *Affinity-Scheduling* in diesen Fällen zu favorisieren.

5.2.2. Schleifennester ohne Affinität

Die Verbesserung der Datenlokalität durch die Algorithmen zum *Affinity-Scheduling* mit blockweiser oder zyklischer Partitionierung basiert auf der Existenz von Affinitäten beim Datenzugriff. Schleifennester ohne Affinität beim Datenzugriff stellen für VSM-Systeme eine wesentlich problematischere Klasse von Anwendungen dar. Das Fehlen von Affinitäten beim Datenzugriff ergibt sich für die in Abb. 24 beschriebenen Schleifennester mit variabler Indexadressierung: Der Zugriff auf ein Datenfeld innerhalb der parallelen Schleife erfolgt indiziert, und für die wiederholte Ausführung der parallelen Schleife ändert sich das Indexfeld. Im allgemeinen ergibt sich eine variable und außerdem regellose Zuordnung von Datenzugriffen und Iterationen. Für diese charakteristische Form von Schleifennestern kann nur das *Align-Scheduling* eine Beachtung der aktuellen Datenverteilung gewährleisten. Für die parallele Schleife in Abb. 24 ergibt sich ein *Align-Scheduling* bezüglich *A(INDEX(J))*; das entspricht einer Zuordnung der Iteration *J* zum Datum *A(INDEX(J))*. Bei der Anwendung des *Align-Scheduling* auf ein Schleifennest mit

Variable Indexadressierung

```
do I = 1, N
   doall J = 1, M
        ... A(Index(J)) ...
   enddo
   do J = 1, M
      Index(J) = ...
   enddo
enddo
```

Abb. 24. Schleifennester mit variabler Indexadressierung

variabler Indexadressierung und nur einem Datenzugriff pro Iteration auf ein gemeinsames Datenobjekt sind gegenüber anderen Partitionierungsstrategien deutliche Vorteile zu erwarten. Erfolgen pro Iteration jedoch zwei oder mehr Datenzugriffe auf gemeinsame Datenobjekte, dann können die benötigten Daten im allgemeinen auf verschiedenen Prozessorelementen liegen. Für diesen allgemeinen Fall sind durch das *Align-Scheduling* allenfalls begrenzte Vorteile zu erwarten.

5.2.3. Geschachtelte parallele Schleifen

Die Parallelisierung von zwei oder mehr ineinander geschachtelten Schleifen - geschachtelte parallele Schleifen (siehe Abb. 25) - ist bisher nur sehr wenig untersucht worden. Für *global-shared-memory*-Rechner ermöglicht in vielen Fällen bereits die Parallelisierung der äußeren Schleife den gewünschten Parallelitätsgrad; die Parallelisierung einer weiter innen liegenden Schleife führt aufgrund der vergleichsweise geringeren Granularität zu erhöhtem *thread*-Scheduler-Overhead und damit zu einer schlechteren Effizienz, außerdem kann eine innere Schleife zum Beispiel zusätzlich zur Parallelisierung der äußeren Schleife vektorisiert werden. Mögliche Vorteile durch geschachtelte, parallele Schleifen sind bei *global-shared-memory*-Rechnern nur dann zu erwarten, wenn keine einzelne der parallelisierbaren Schleifen einen hinreichend hohen Parallelitätsgrad ermöglicht, d.h. wenn die Zahl der Schleifeniterationen nur einer Schleife zu klein ist; in diesem Fall kann der Parallelitätsgrad gesteigert werden, indem zwei oder mehr ineinander geschachtelte Schleifen parallelisiert werden. Der resultierende, maximale Parallelitätsgrad entspricht dann dem Produkt der Schleifeniterationen aller parallelisierten Schleifen.
Bei *virtual-shared-memory*-Rechnern sind geschachtelte, parallele Schleifen bisher nicht untersucht worden. In [Wol92] wird auf Compiler-Ebene die Maximierung der Anzahl von parallelen Schleifen bei gleichzeitiger Maximierung der Datenlokalität mithilfe von Schleifentransformationen untersucht. Bezüglich der Datenlokalität sind die Untersuchungen auf *multicache-shared-memory*-Systeme zugeschnitten. Im Gegensatz dazu soll in dieser Arbeit keine Transformation von Schleifen betrachtet werden, sondern die Anzahl der parallelisierbaren Schleifen soll vorgegeben sein. Für zwei parallelisierbare Schleifen stellt dabei die alternative Parallelisierung von jeweils nur einer Schleife einen Untersuchungsparameter dar. Dabei wird die Maximierung der Datenlokalität ein entscheidendes Kriterium für die Effizienz der gewählten Parallelisierung sein.
Im folgenden sollen mögliche Vor- und Nachteile am Beispiel der Matrixmultiplikation untersucht werden; es wird der Speicherbedarf für die Parallelisierung der äußeren beziehungsweise der mittleren Schleife ermittelt. Die Abb. 26 gibt Aufschluß über den Speicherbedarf für eine Parallelisierung der äußeren Schleife: Es wird der lokale Speicherbedarf für eine vollständige Wiederverwendung der Daten ermittelt; dabei ist der

Abb. 25. Geschachtelte parallele Schleifen

Matrixmultiplikation

```
doall I = 1, N
    do J = 1, N
        do K = 1, N
            C(I,J) = C(I,J) +A(I,K)*B(K,J)
        enddo
    enddo
enddo
```

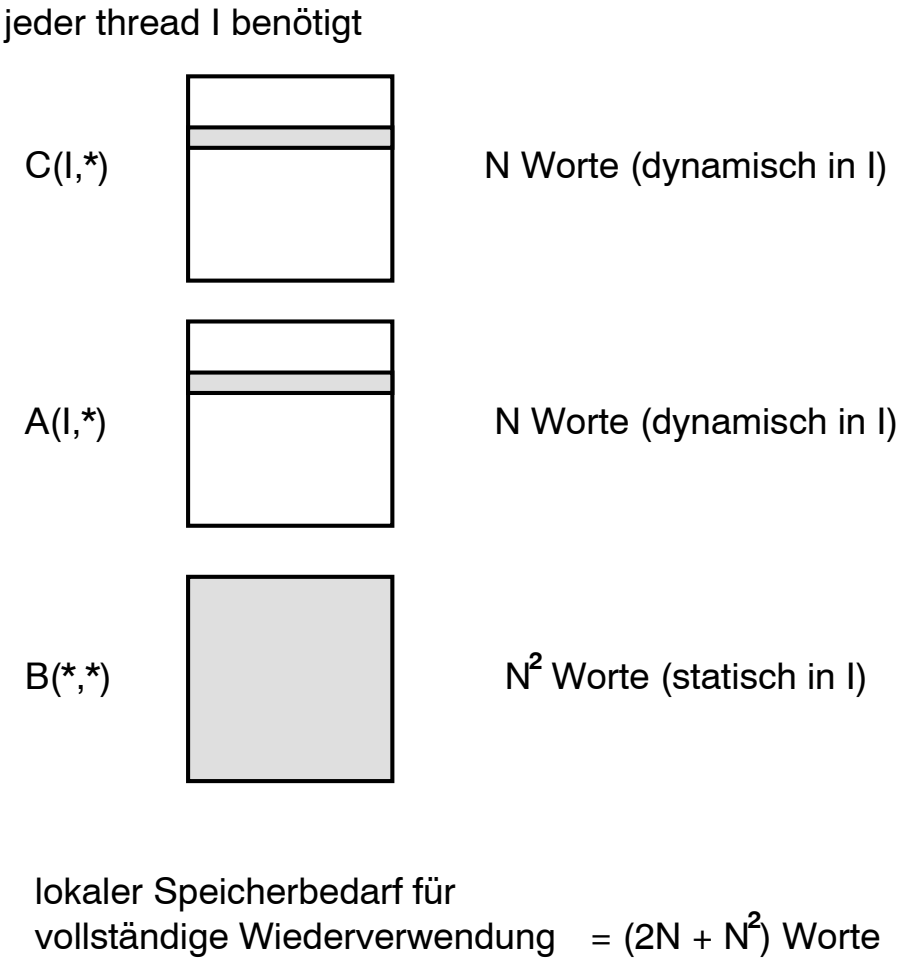


Abb. 26. Speicherbedarf bei einer Matrixmultiplikation für die Parallelisierung der äußeren Schleife

Datenbedarf für die Ausführung von einem *thread* der parallelen *I*-Schleife relevant. Bei der beschriebenen Matrixmultiplikation benötigt jeder *thread* der parallelen *I*-Schleife folgende Daten der drei Matrizen:

- Von der Matrix *C* benötigt jeder *thread I* die Datenelemente für einen Wert von *I* und für alle Werte von *J*, das entspricht genau einer Zeile der Matrix. Da jeder *thread* einem anderen Wert für *I* entspricht, benötigt jeder *thread* eine andere Zeile der Matrix; diesbezüglich sind die benötigten Daten "dynamisch in *I*". Mit der

Problemgröße *N* ergibt sich bezüglich der Matrix *C* ein Speicherbedarf von *N* Worten, der dynamisch in *I* ist.

- Für die von der Matrix *A* benötigten Daten liegen dieselben Verhältnisse vor: Jeder *thread I* benötigt eine jeweils andere Zeile der Matrix *A*; damit ergibt sich auch hier ein Speicherbedarf von *N* Worten, der dynamisch in *I* ist.
- Von der Matrix *B* benötigt jeder *thread I* die Datenelemente für alle Werte von *K* und für alle Werte von *J*, das entspricht der gesamten Matrix. Da die jeweils benötigten Daten in diesem Fall unabhängig von *I* sind, werden diese als "statisch in *I*" bezeichnet. Der Speicherbedarf bezüglich der Matrix *B* liegt damit bei N^2 Worten und ist statisch in *I*.

Der gesamte Speicherbedarf pro Prozessorelement ergibt sich als Summe der drei ermittelten Komponenten zu

$$Local\_Memory\_Needs \;=\; (2N \;+\; N^2)$$

Worten. Im Vergleich dazu beschreibt die Abb. 27 den Speicherbedarf bei Parallelisierung der mittleren Schleife; die Schleife soll auf *P* Prozessoren parallel ausgeführt werden. Für den lokalen Speicherbedarf zur vollständigen Wiederverwendung der Daten sind hier die für die Ausführung von *N*/*P threads* der parallelen *J*-Schleife benötigten Daten relevant. *N*/*P threads* der parallelen *J*-Schleife benötigen die folgenden Daten der drei Matrizen:

- Von der Matrix *C* benötigen *N*/*P threads* die Datenelemente für einen Wert von *I* und für *N*/*P* Werte von *J*, das entspricht *N*/*P* Worten aus einer Zeile der Matrix. Da sich für jeden Wert von *I* wieder neue *threads* ergeben und jeder *thread* einem anderen Wert für *J* entspricht, sind die benötigten Daten dynamisch in *I* und *J*. Der Speicherbedarf bezüglich der Matrix *C* liegt bei *N*/*P* Worten und ist dynamisch in *I* und *J*.
- Von der Matrix *A* benötigt jeder *thread J* die Datenelemente für einen Wert von *I* und für alle Werte von *K*, das entspricht einer Zeile der Matrix. Die benötigten Daten hängen vom Index *I* ab, sie sind dynamisch in *I*; vom Index *J* sind die Daten jedoch unabhängig und damit statisch in *J*. Damit ergibt sich bezüglich der Matrix *A* ein Speicherbedarf von *N* Worten, der dynamisch in *I* und statisch in *J* ist.
- Von der Matrix *B* benötigen *N*/*P threads* die Datenelemente für alle Werte von *K* und für *N*/*P* Werte von *J*, das entspricht *N*/*P* Spalten der Matrix. Der Speicherbedarf bezüglich der Matrix *B* liegt damit bei N^2/P Worten und ist statisch in *I* und dynamisch in *J*.

Der gesamte Speicherbedarf pro Prozessorelement ergibt sich in diesem Fall zu

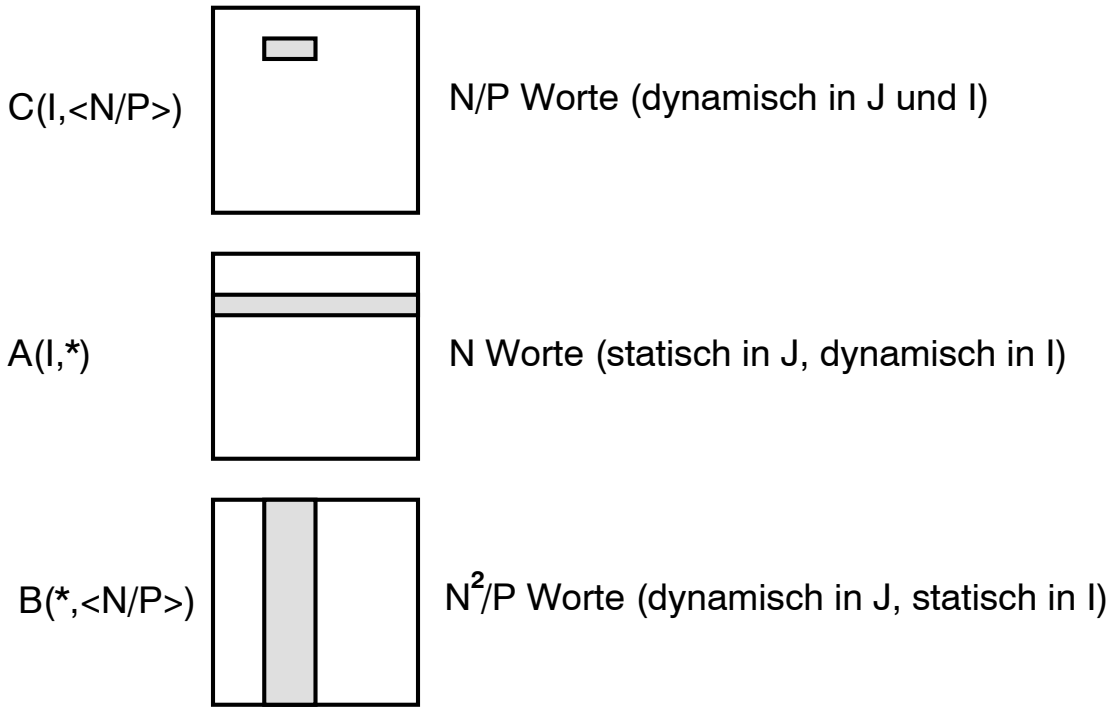
$$Local\_Memory\_Needs \;=\; (1 \;+\; \frac{1}{P})\,N \;+\; \frac{N^2}{P}$$

Worten. Für große Matrizen ergibt ein Vergleich der beiden betrachteten Varianten im wesentlichen eine Reduzierung des Speicherbedarfs von der Ordnung N^2 auf die Ordnung N^2/P. Die wesentlichen Anteile beziehen sich damit ausschließlich auf die Matrix *B*. Für die Parallelisierung der äußeren Schleife sind die benötigten Daten der Matrix *B* statisch in *I*, d.h. sie werden für alle *threads I* gleichermaßen benötigt und sollten aus

Matrixmultiplikation

```
do I = 1, N
    doall J = 1, N (ncpus = P)
        do K = 1, N
            C(I,J) = C(I,J) +A(I,K)*B(K,J)
        enddo
    enddo
enddo
```

N/P threads J benötigen



lokaler Speicherbedarf für
vollständige Wiederverwendung $= (1 + \frac{1}{P})N + \frac{N^2}{P}$ Worte

Abb. 27. Speicherbedarf bei einer Matrixmultiplikation für die Parallelisierung der mittleren Schleife

Effizienzgründen auf jedem Prozessorelement ständig komplett verfügbar sein. Für die Parallelisierung der mittleren Schleife sind die benötigten Daten der Matrix *B* dynamisch in *J*, d.h. ein Prozeß benötigt für jeden seiner *threads J* eine Spalte der Matrix. Da diese Daten außerdem statisch in *I* sind, kann im Prinzip eine Wiederwendung für aufeinanderfolgende Iterationen der *I*-Schleife erfolgen. Dazu muß gewährleistet sein, daß einem Prozeß für alle Iterationen der *I*-Schleife jeweils dieselben *threads* zugeteilt

werden; diese Forderung wird durch das *Affinity-Scheduling* erfüllt (siehe Abschnitt 5.2.1).

Aus diesen Überlegungen folgt, daß durch eine Parallelisierung der mittleren anstelle der äußeren Schleife bei der Matrixmultiplikation der Speicherbedarf reduziert werden kann. Daraus kann sich ein entscheidender Vorteil ergeben, wenn der Speicherplatz je Prozessorknoten in dem Maße zu klein ist, daß bei der Parallelisierung der äußeren Schleife die Matrix *B* nicht vollständig in den Speicher paßt und deswegen immer nur teilweise vorliegt; dabei muß während der Ausführung eines *thread* der parallelen Schleife jeweils ein Teil der Matrix gelöscht werden, um so einem anderen Teil Platz zu schaffen, so daß jeder Teil der Matrix für jeden neuen *thread* wieder neu geladen werden muß. Dieser Effekt ist analog zu dem für ein virtuelles Speichersystem, das aufgrund eines zu kleinen Realspeichers ständig Seiten laden und invalidieren muß. Durch die Parallelisierung der mittleren Schleife wird der Speicherbedarf im wesentlichen um den Faktor der Prozessorzahl reduziert, so daß ein vorher bestehender Speicherengpaß eliminiert werden kann, was die Effizienz der Ausführung deutlich verbessert.

Die Verlagerung der parallelen Ausführung in eine weiter innen liegende Schleife führt - wie oben beschrieben - zu einer Erhöhung des *thread*-Scheduler-Overhead. Um den möglicherweise notwendigen *trade-off* zwischen Speicherengpaß beziehungsweise *thread*-Scheduler-Overhead zu untersuchen, soll insbesondere auch der Fall betrachtet werden, daß beide Schleifen parallelisiert werden.
Die Verwaltung von geschachtelten, parallelen Schleifen erfordert im Vergleich zu einfach parallelen Schleifen erweiterte Konzepte: Bei der Ausführung einer parallelen Schleife werden die Iterationen der Schleife in Partitionen an die beteiligten Prozesse verteilt. Die Datenstruktur für die Speicherung der Informationen über eine zugeteilte Partition und die darin enthaltene Arbeit soll für die folgenden Betrachtungen mit *Thread-Queue* bezeichnet werden. Damit wird jedem aktiven Prozeß eine eigene *Thread-Queue* zugeordnet. Nach der vollständigen Ausführung einer *Thread-Queue* versucht der entsprechende Prozeß einen weiteren Teil der Schleifeniterationen - eine weitere Partition - für sich zu reservieren. Die Verwaltung der noch verbleibenden Iterationen wird durch die Datenstruktur *Loop-Descriptor* realisiert; damit wird der *Loop-Descriptor* von allen Prozessen einer parallelen Schleife gemeinsam genutzt. Bei der Verwaltung von geschachtelten, parallelen Schleifen können gleichzeitig mehrere *Loop-Descriptor*-Strukturen aktiv sein, die auf verschiedenen Schleifen oder auf derselben Schleife basieren können. Um eine eindeutige Zuordnung von Prozessen zu *Loop-Descriptor*-Strukturen zu garantieren, wird eine allen Prozessen eines Jobs gemeinsame, hierarchische Datenstruktur verwendet, der *Shared-Thread-Queue-Tree* (siehe Abb. 28). Der *Shared-Thread-Queue-Tree* realisiert eine Baumstruktur; jeder Knoten des Baumes enthält einen *Process-Pointer*, der auf einen aktiven Prozeß zeigen kann; anderenfalls enthält der *Process-Pointer* den Wert *NIL*. Bei der Ausführung eines sequentiellen Programmteils zeigt der *Process-Pointer* der Wurzel auf einen aktiven Prozeß, und es existieren keine weiteren Knoten im Baum. Beim Eintritt in eine parallele Schleife wird für jeden partizipierenden Prozeß ein Kindknoten erzeugt. Der bisher der Wurzel zugeordnete Prozeß wird einem der Kindknoten zugeordnet; den übrigen Kindknoten werden bisher nicht aktive Prozesse zugeordnet. Der

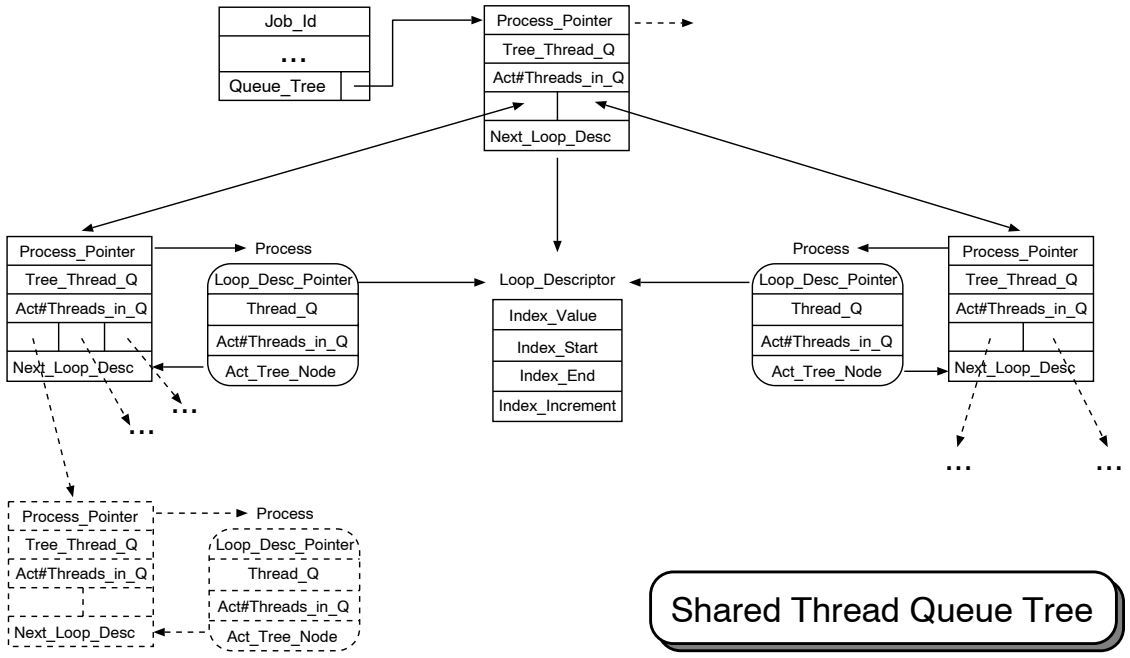


Abb. 28. Shared Thread Queue Tree zur hierarchischen Verwaltung von geschachteltem Parallelismus

Elternknoten und alle partizipierenden Prozesse verfügen über einen Zeiger auf den *Loop-Descriptor* der parallelen Schleife. Bei geschachtelten, parallelen Schleifen trifft jeder Prozeß auf der Kindebene wieder auf eine parallele Schleife. In diesem Fall wird die dem Prozeß zugeordnete *Thread-Queue*, die Informationen zu den verbleibenden Schleifeniterationen beinhaltet, im entsprechenden Kindknoten des Baumes als *Tree-Thread-Queue* gespeichert; für jeden an der neuen parallelen Schleife partizipierenden Prozeß wird ein Enkelknoten erzeugt, und alle Enkelknoten sowie der zugehörige Kindknoten zeigen auf einen gemeinsamen *Loop-Descriptor*. Wenn sich dieser Vorgang für zwei oder mehr Kindknoten wiederholt, dann existieren neben dem *Loop-Descriptor* auf der Kindebene mehrere *Loop-Descriptor*-Strukturen auf der Enkelebene; jeder *Loop-Descriptor* ist dabei eindeutig einem Knoten des Baumes und damit den jeweiligen Prozessen zugeordnet. Abb. 29 beschreibt eine abstrahierte Sicht des *Shared-Thread-Queue-Tree*; die Darstellung ist auf die *Thread-Queues* beschränkt und bezieht sich auf eine Schachtelung von zwei parallelen Schleifen. In diesem Beispiel wird die äußere Schleife auf drei Prozesse verteilt, die innere Schleife wird für jede Iteration der äußeren Schleife von jeweils zwei Prozessen ausgeführt; damit ergibt sich in der Summe ein Parallelitätsgrad von sechs.
Die Speicherung der *Thread-Queue* in einem Knoten des *Shared-Thread-Queue-Tree* erweist sich als ausgesprochen vorteilhaft; diese Loslösung der *Thread-Queue* vom Prozeß erfolgt nicht nur beim Wechsel eines Prozesses auf eine tiefere Ebene des Baumes, sondern auch bei einer Unterbrechung eines Prozesses, dem der Prozessor entzogen wird: Dadurch kann irgendein anderer Prozeß die Ausführung der *Thread-Queue* wiederaufnehmen. Durch die Speicherung der *Thread-Queue* im Benutzerbereich kann die Wiederaufnahme ohne Intervention durch das Betriebssystem erfolgen; diese Strategie ist im Cray Autotasking seit dem UNICOS Release 6.0 für einfach parallele Schleifen implementiert und hat sich als sehr vorteilhaft erwiesen [Oed91, Cra3074].

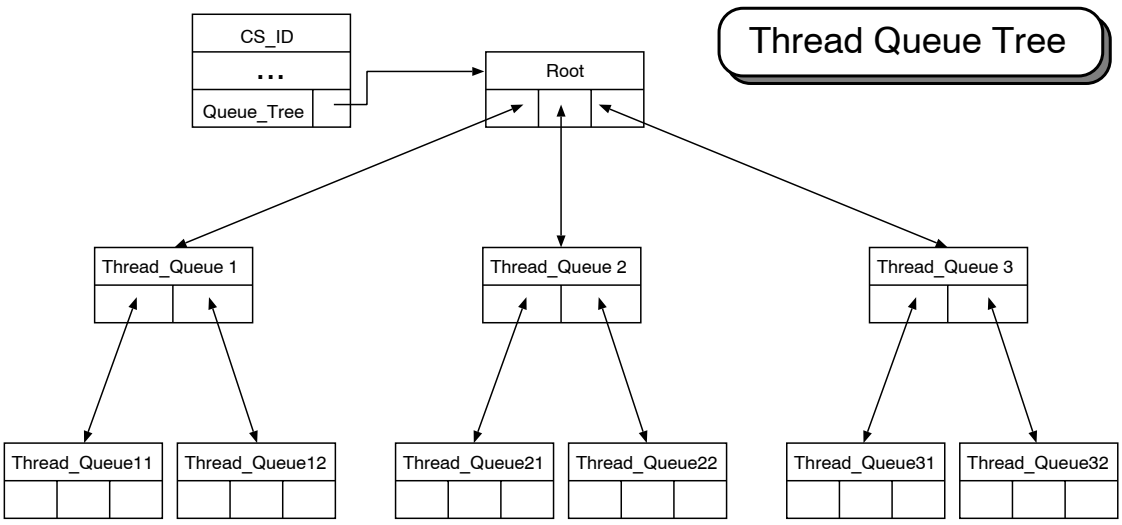


Abb. 29. Die Hierarchie der Thread Queues

Der *Shared-Thread-Queue-Tree* ermöglicht die hierarchische Verwaltung von beliebig geschachtelten, parallelen Schleifen. In Verbindung mit einer Verteilung der Knoten des Baumes auf unabhängige Datensegmente, zum Beispiel auf Seiten für VSM-Rechner, kann ein verteiltes *thread*-Scheduling für geschachtelten Parallelismus realisiert werden.

Bei der Ausführung von geschachtelten, parallelen Schleifen treten Affinitätsbeziehungen beim Datenzugriff auf: Die wiederholte Ausführung einer inneren parallelen Schleife durch deren Einbettung in eine äußere, ebenfalls parallele Schleife kann zu wiederholten Zugriffen auf jeweils dieselben Daten oder Seiten führen. Für die Schachtelung von zwei und mehr parallelen Schleifen ergibt sich gegenüber der herkömmlichen Version des *Affinity-Scheduling* eine gesteigerte Komplexität: Die Affinitätsbeziehungen existieren dann auf mehreren Ebenen und erfordern eine entsprechend angepaßte Verwaltung; da sich die Affinitäten an der Lokalität von Daten orientieren, die wiederum einem Prozessorelement zuzuordnen sind, verwaltet jeder Prozeß die Affinitätsinformationen für jede Schachtelung mit Bezug zu seinem aktuellen Prozessor. Wenn während der Ausführung ein Prozessor abgegeben wird, dann können die bestehenden Affinitäten nicht weiter genutzt werden; wird die Ausführung auf einem anderen Prozessor weitergeführt, so werden die entsprechenden Daten dorthin migriert, und es entstehen neue Affinitäten.

6. Ein Virtual-Shared-Memory-Simulator

Die Untersuchung von Scheduling-Algorithmen zur Ablaufplanung bei Parallelrechnern durch die Implementierung der Algorithmen auf realen Systemen ist im allgemeinen aufgrund des hohen Aufwandes und der Inflexibilität in bezug auf Parameterstudien nur in begrenztem Umfang möglich: Zum einen erlauben Untersuchungsparameter, die sich auf das Multiprozessormodell und damit auf die Hardware beziehen, in realen Systemen keine hinreichende Flexibilität. Zum anderen erfordert die alternative Untersuchung von Scheduling-Algorithmen flexible Änderungen auf der Betriebssystemebene, die mit erheblichem Aufwand bezüglich Implementierung und Betriebsunterbrechungen verbunden sind; für Produktionsumgebungen sind Betriebsunterbrechungen im erforderlichen Umfang nahezu ausgeschlossen. Eine weitere, ganz wesentliche Problematik stellt die Verfügbarkeit von Informationen über interne Abläufe dar: Die Komplexität der relevanten Abläufe fordert neben einer Bewertung aufgrund absoluter Leistungsvergleiche eine Analyse der Ursachen. Erst die Analyse mithilfe von Detailinformationen zum Beispiel über die Abläufe innerhalb der Auflösung eines *page fault* kann das Verständnis der Effekte fördern und so zu neuen, effizienten Algorithmen zur Ablaufplanung führen. In vielen Fällen sind Informationen dieser Art gar nicht verfügbar, oder aber die erforderlichen Messungen verfälschen den eigentlichen Ablauf. Eine Lösung all dieser Probleme kann durch die Simulation der maßgeblichen Abläufe realisiert werden. Das im Rahmen dieser Arbeit entwickelte Simulationssystem für *virtual-shared-memory*-Rechner implementiert einen Ereignis-gesteuerten Simulator, der auf drei unabhängigen Modellen basiert. Die Modularität der Implementierung gewährleistet die weitgehende Unabhängigkeit von VSM-Modell, Multiprozessormodell und Arbeitslastmodell; die Modelle können über Parameter modifiziert oder durch Alternativmodelle vollständig ersetzt werden.
Die Komplexität und der Umfang der zum Verständnis der Abläufe notwendigen Informationen lassen eine leistungsfähige Visualisierung zu einem wesentlichen Bestandteil der Untersuchungen werden. Unter dieser Vorgabe wurde am Zentralinstitut für Angewandte Mathematik im Forschungszentrum Jülich (KFA) das Visualisierungstool PARvis entwickelt [Arn93, NaAr93, Mul94]. Die Kombination von Simulation und Visualisierung resultiert in der Werkzeugumgebung PARtools. PARtools umfaßt insbesondere die Komponenten PARsim - einen Simulator für *global-shared-memory*-Rechner [Nag93] und PARvis für die Visualisierung der Simulationsergebnisse [Arn93]. Das im Rahmen dieser Arbeit entwickelte Simulationssystem stellt eine neue Funktionalität - die Simulation von *virtual-shared-memory*-Rechnern - zur Verfügung und erweitert damit die Komponente PARsim. Zur Visualisierung des bei VSM-Rechnern relevanten Speicherverhaltens wurde auch die Komponente PARvis deutlich erweitert [Mul94].

6.1. Das VSM-Modell

Wie bereits in Kapitel 5 motiviert worden ist, wird das von K. Li entwickelte VSM-Modell mit statisch verteiltem Manager zugrunde gelegt; um in bestimmten Situationen ein günstigeres Programmverhalten zu erzielen, werden in Erweiterung dieses Modells

die explizite Abbildung von Daten auf Seiten, die schwache Kohärenz und die *prefetch*-Operation implementiert. Der Simulator basiert auf einer verbesserten Version des statisch verteilten Managers: Die Verbesserung ist in [Li86] für den zentralen Manager beschrieben *(improved centralized manager)*; gegenüber der in Kapitel 3 beschriebenen Version des statisch verteilten Managers wird nach Erhalt der Seite die Bestätigung des anfragenden Knotens an den Manager eingespart (siehe Abb. 8 auf Seite 27: (7) *notify manager*). Abb. 30 beschreibt den verbesserten Algorithmus zum statisch verteilten Manager. Bei einem *page fault* wird eine entsprechende Anfrage an den Manager der Seite geschickt. Wenn ein *write fault* vorliegt und sich damit eine Änderung bezüglich des Besitztums der Seite ergibt, dann wird bereits zu diesem Zeitpunkt der Eintrag des Managers über den Besitzer aktualisiert. Dadurch wird eine später beim Manager eintreffende Anfrage für dieselbe Seite bereits zu dem neuen Besitzer weitergeleitet. Bei diesem Algorithmus besteht die Möglichkeit, daß eine Anfrage bei einem vermeintlichen Besitzer eintrifft, bevor dieser die entsprechende Seite tatsächlich erhalten hat; in diesem Fall wird die Anfrage beim zukünftigen Besitzer in einer Warteschlange gespeichert. Der Vorteil dieser Verbesserung wird für das Beispiel von zwei kurz hintereinander beim Manager eintreffenden *write-fault*-Anfragen von Knoten 1 und Knoten 2 deutlich: Für den herkömmlichen Algorithmus wird die zweite Anfrage (von Knoten 2) solange beim Manager zwischengespeichert, bis die Benachrichtigung von Knoten 1 über den Erhalt der Seite beim Manager eintrifft. Erst dann wird die Anfrage von Knoten 2 an den

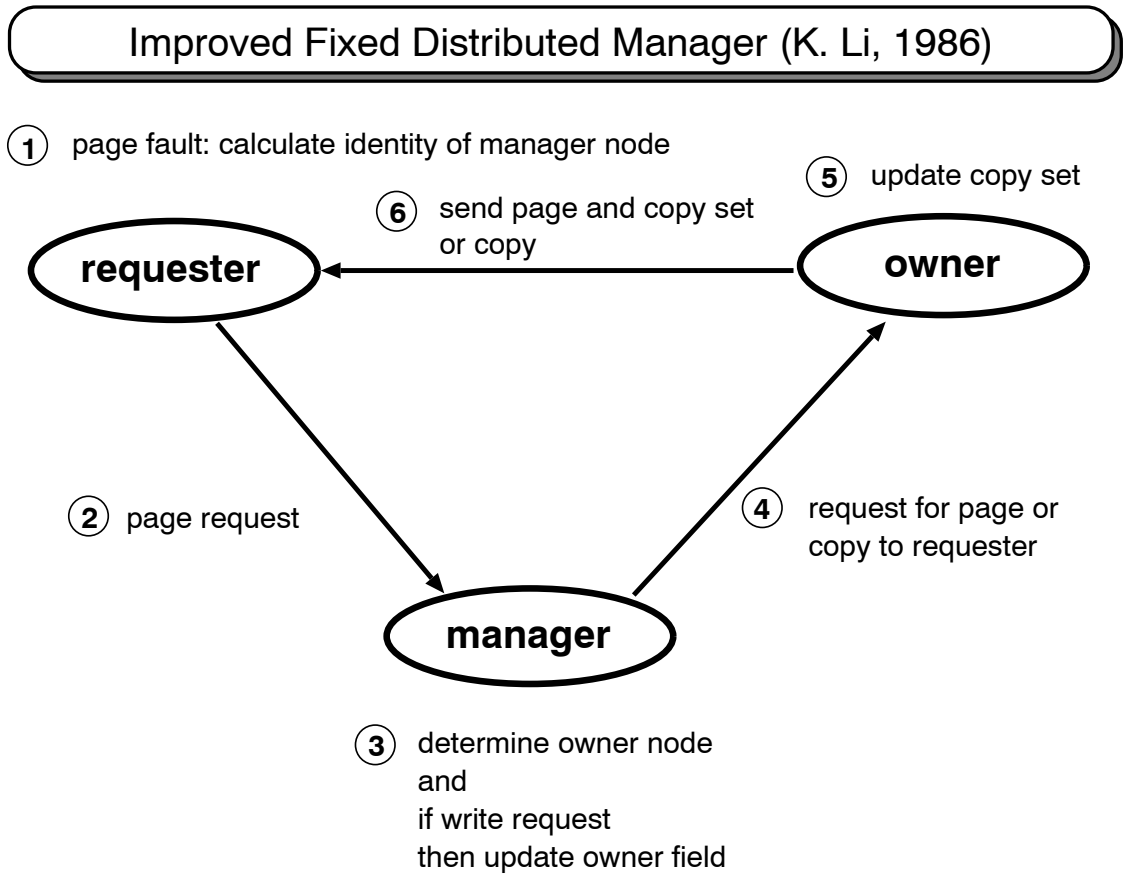


Abb. 30. Der verbesserte statisch verteilte Manager [Li86]

neuen Besitzer (Knoten 1) weitergeleitet. Nach dem verbesserten Algorithmus wird die Anfrage des Knotens 2 vom Manager sofort an den Knoten 1 weitergeleitet. Damit trifft die Anfrage bei Knoten 1 bereits früher ein und kann früher bearbeitet werden[12].
Die explizite Abbildung von gemeinsamen Daten auf Seiten soll den Effekt des *false sharing* begrenzen; mögliche Abbildungen für Datenfelder sind bereits in Abschnitt 5.1 beschrieben worden. Zur weiteren Vermeidung des *false sharing* erfolgt die Abbildung von Datenobjekten unabhängig, d.h. auf eine Seite werden jeweils nur Teile genau eines Datenobjekts abgebildet; für jedes Datenobjekt kann explizit eine Abbildung spezifiziert werden; dabei sind Datenfelder beziehungsweise Skalare gegebenenfalls gesondert zu behandeln:

- **Datenfelder:**
  In den Simulationsuntersuchungen handelt es sich bei den gemeinsamen Datenobjekten im wesentlichen um Datenfelder. Für diesen Fall kann der Speicherplatz pro Seite bei den meisten Abbildungen nahezu vollständig genutzt werden.

- **Skalare:**
  Die unabhängige Abbildung von gemeinsamen Skalaren kann dagegen insbesondere bei großen Seiten zu Effizienzproblemen führen; die Abbildung von jeweils einem Skalar auf eine Seite kann Speicherengpässe hervorrufen. Deswegen kann für die Abbildung von Skalaren eine Einschränkung der Unabhängigkeit durch eine geeignete Gruppierung der Skalare zum Beispiel durch den Compiler sinnvoll sein.

Die Erweiterungen des VSM-Modells bezüglich der schwachen Kohärenz und der *prefetch*-Operation sind wie folgt realisiert: Für jeden Schreibzugriff kann explizit spezifiziert werden, daß es sich um einen schwach kohärenten Schreibzugriff handelt; die entsprechende Seite wird dann mit dem Zugriffsattribut *weak-write* verfügbar gemacht. Von einer solchen Seite können mehrere Kopien gleichzeitig mit Seiten für den Nur-Lese-Zugriff existieren. Erst beim Auftreten eines normalen (strikt kohärenten) Schreibzugriffs auf die jeweilige Seite wird eine aktuelle Kopie durch Zusammenfügen der schwach kohärenten Kopien erzeugt. Eine *prefetch*-Operation bezieht sich auf jeweils ein Datum sowie auf die jeweils zugehörige Zugriffsart *(read, write, weak-write)* und kann explizit in der Programmspezifikation angegeben werden (z.B. *Prefetch_Write A(I,K))*. Die *prefetch*-Operation wird wie ein normaler Datenzugriff behandelt, nur daß nicht auf die Fertigstellung der Operation gewartet wird. Tritt nach einer *prefetch*-Operation ein Zugriff auf die entsprechende Seite auf, bevor die Seite tatsächlich verfügbar ist, dann wird kein weiterer *page fault* auf die Seite ausgelöst.

[12] Für den verbesserten Algorithmus kann sich der minimale Verfügbarkeitszeitraum einer Seite auf einem Knoten verkürzen. Daraus können sich auch Nachteile ergeben, wenn in diesem Zeitraum nicht alle erforderlichen Schreibzugriffe durchgeführt werden können. In diesem Zusammenhang kann das explizite Festhalten und Freigeben einer Seite *(freeze and thaw)* vorteilhaft sein.

6.2. Das Multiprozessormodell

Für das simulierte Multiprozessormodell kann über Parameter wahlweise ein physikalisch gemeinsamer oder verteilter Speicher vorgegeben werden. Da sich beide Modelle auf ein *shared-memory*-Programmiermodell beziehen sollen, werden die Bezeichnungen *global-shared-memory*- und *virtual-shared-memory*-Multiprozessormodell gewählt (siehe Abb. 31). Die Prozessorzahl und die Größe des Arbeitsspeichers (Lokalspeichergröße für das VSM-Modell) stellen zwei wesentliche Parameter für die Multiprozessormodelle dar. Für das VSM-Modell kann ein zu kleiner Lokalspeicher zu zusätzlichem *paging* führen und so die Effizienz der Ausführung erheblich beeinflussen. Zur Hardware-Unterstützung der Datenkommunikation ist für das VSM-Multiprozessormodell jeder Prozessorknoten mit einem Kommunikationsprozessor ausgestattet (siehe Abb. 32). Eine vergleichbare Hardware-Konfiguration ist zum Beispiel für den Intel Paragon realisiert [Int91]. Auf diese Weise muß die Prozeßausführung auf einem Prozessorknoten nur dann unterbrochen werden, wenn der lokale Prozeß ein *page fault* auslöst und selbst auf eine Seite wartet; die Behandlung von fremden, d.h. von Prozessen anderer Knoten ausgelösten *page faults* erfolgt ausschließlich durch den Kommunikationsprozessor und erfordert keine Unterbrechung des lokalen Prozesses. Der Kommunikationsprozessor kann die anstehenden Anfragen nur sequentiell bearbeiten: Treffen zwei oder mehr *page-fault*-Anfragen gleichzeitig ein, oder ist ein Sendevorgang beim Eintreffen einer neuen Anfrage noch nicht abgeschlossen, dann werden jeweils alle nicht unmittelbar ausführbaren Anfragen in einer FIFO-Warteschlange gespeichert, bis der Kommunikationsprozessor wieder frei ist.
Die zugrundegelegten Kosten für die Auflösung von *page faults* stellen weitere wesentliche Vorgaben für die Simulation dar. Um die Komplexität und damit den Simulationsaufwand zu begrenzen, geht eine Abhängigkeit für den Kommunikations-Overhead von einer bestimmten Netzwerktopologie oder von der aktuellen Netzwerkbelastung in

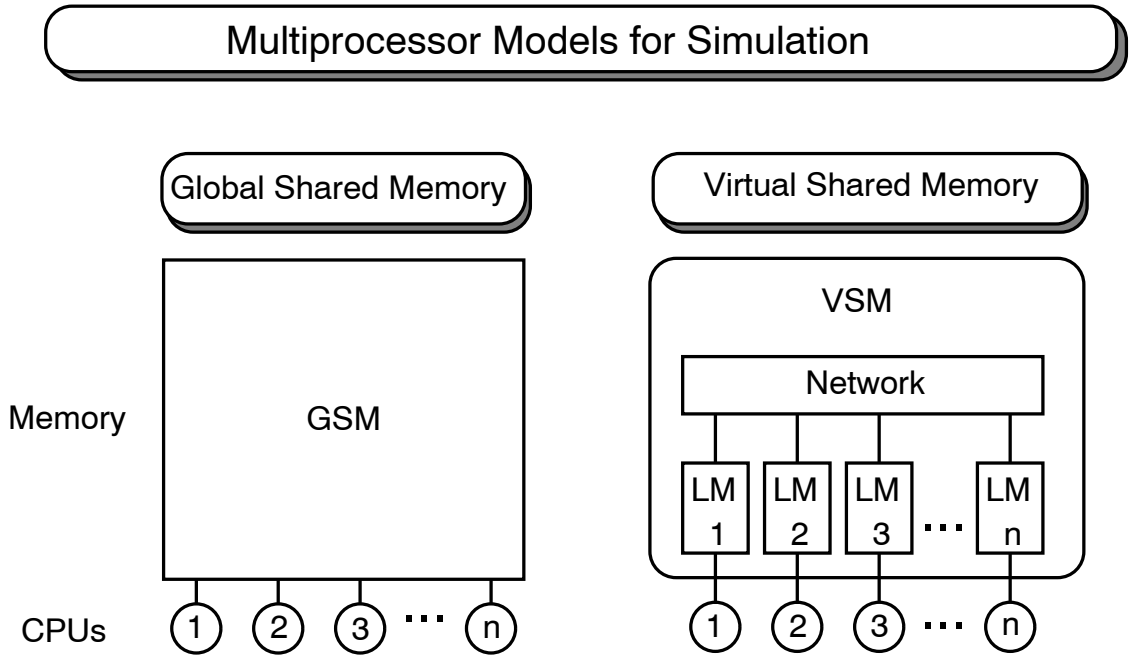


Abb. 31. Multiprozessormodelle für die Simulation

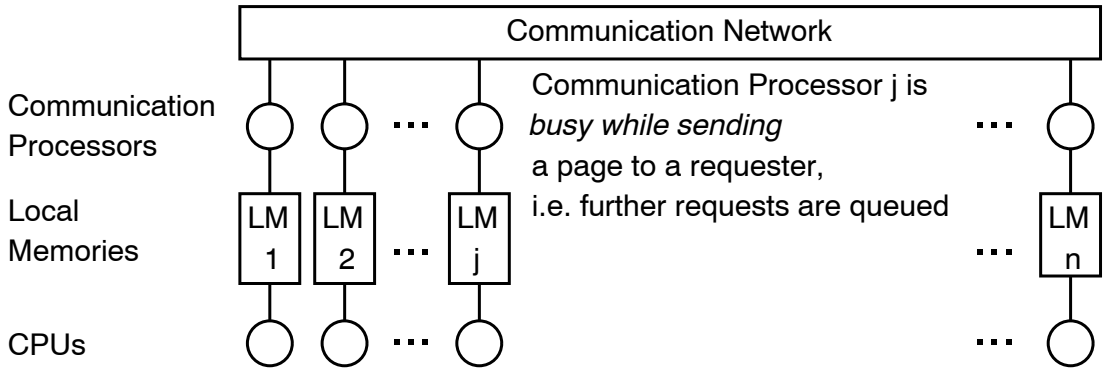


Abb. 32. Kommunikationsprozessoren für Datentransfers bei physikalisch verteiltem Speicher

die Simulationen nicht ein. Die Kosten ergeben sich gemäß Abb. 33: *T_Fault* entspricht dem Overhead für das Aufsetzen eines *page fault*; da dieser Vorgang durch den Kommunikationsprozessor von der Hardware unterstützt wird, werden die Kosten im wesentlichen vernachlässigt. Dieser Parameter eröffnet die Möglichkeit, auch Fallstudien ohne Kommunikationsprozessor durchzuführen; in diesem Fall muß die CPU die *page-fault*-Behandlung selbst durchführen, was zunächst einen Kontextwechsel erfordert und damit einen vergleichsweise höheren Overhead verursacht. *T_Message* entspricht dem Overhead für das Senden einer *page-fault*-Anfrage, z.B. vom Manager zum Besitzer; da keine Netzwerktopologie berücksichtigt wird, sind die Kosten unabhängig von der Lokalität des Sender- beziehungsweise des Empfängerknotens; außerdem geht auch keine Abhängigkeit der Übertragungskosten von der aktuellen Netzwerkbelastung ein, so daß dieser Overhead immer gleich ist. *T_Startup* und *T_Send* beschreiben die Kosten für die Übertragung einer Seite in Abhängigkeit von der Seitengröße; *T_Startup* entspricht dem Overhead für das Aufsetzen der Übertragung und *T_Send* gibt die darüber hinaus entstehenden Kosten für die Übertragung eines Wortes an. Die Kosten für die Invalidierung

Page_Fault_Time =	T_Fault	page fault startup time
	+ T_Message	optional message to manager
	+ T_Message	optional message to owner
	+ T_Startup + T_Send * Pagesize	optional for send page
	+ T_Startup + T_Send * RC(Page)/2	optional for send copy set
	+ T_Inval * RC(Page)	optional for invalidation
	+ T_Delay	optional when communication processor is busy
T_Fault	= 1 cycle	page fault startup time
T_Message	= 250 cycles	internode message time
T_Startup	= 250 cycles	startup time for send
T_Send	= 10 cycles	time to send one word
T_Inval	= 300 cycles	page invalidation time

Abb. 33. Overhead für *page faults*

Read_Data_Overhead	= 5 cycles	overhead for read operation
Write_Data_Overhead	= 5 cycles	overhead for write operation
Add_Overhead	= 6 cycles	overhead for add operation
Sub_Overhead	= 6 cycles	overhead for subtract operation
Mul_Overhead	= 7 cycles	overhead for multiply operation
Div_Overhead	= 7 cycles	overhead for divide operation

Abb. 34. Overhead für Speicherzugriffe und Rechenoperationen

einer Seite werden mit *T_Inval* bezeichnet; für jede Invalidation ergibt sich jeweils eine Nachricht vom Besitzer zu jedem Knoten mit einer Kopie der Seite. Die tatsächlichen Kosten für die Auflösung eines *page fault* ergeben sich gemäß Abb. 33 in Abhängigkeit vom Typ des *page fault*, von der Identität der beteiligten Knoten, von der Seitengröße und gegebenenfalls von der Anzahl der existierenden Kopien. Als weitere wesentliche Größe geht die Laufzeit-abhängige Belastung des Kommunikationsprozessors in die *page-fault*-Kosten ein *(T_Delay)*. Damit können die entstehenden Kosten für ein *page fault* z.B. bei einer Seitengröße von 128 Worten und einer Konfiguration mit 10 Prozessoren zwischen 5 und 5000 Maschinentakten zuzüglich der Größe *T_Delay* liegen; der günstigste Overhead von nur 5 Takten[13] ergibt sich für ein *write fault*, wenn der anfragende Knoten zugleich der Manager und der Besitzer der Seite ist und darüber hinaus keine Kopie der Seite auf anderen Knoten existiert, so daß tatsächlich kein Datentransfer zur Auflösung des *page fault* erforderlich ist. Ein Overhead von etwa 5000 Takten berechnet sich aus je einer *Message* vom anfragenden Knoten zum Manager und von diesem zum Besitzer, aus dem Transfer der Seite und des *copy set* vom Besitzer zum anfragenden Knoten[14] und aus der Invalidation der maximal 9 Kopien der Seite.
Neben den *page-fault*-Kosten ergibt sich ein Overhead für die Ausführung von Speicherzugriffen und von Rechenoperationen (siehe Abb. 34). Um den Simulationsaufwand zu begrenzen, werden für die in dieser Arbeit beschriebenen Simulationen alle Speicherzugriffe ohne *Cache* durchgeführt; ebenso wird keine Vektorverarbeitung und keine Überlappung von Datenzugriffen und Rechenoperationen realisiert.

6.3. Das Arbeitslastmodell

In *virtual-shared-memory*-Systemen hat die Datenverteilung einen erheblichen Einfluß auf die Lokalität der Prozesse. Da die Abhängigkeit zwischen der Datenverteilung und der Prozeßausführung wechselseitig ist, kann eine stochastische Beschreibung die Datenverteilung einer realen Anwendung nur unzureichend annähern. Um realistische Vorgaben für die Datenverteilung zu erhalten, wurde für den Simulator ein Arbeitslastmodell gewählt, das die Datenzugriffe auf der Basis von realen Anwendungen generiert und damit die Datenverteilung realistisch steuert.
Abb. 35 beschreibt die Abbildung eines realen Programmkerns auf das verwendete Arbeitslastmodell am Beispiel der Matrixmultiplikation: Datenzugriffe und Rechenoperationen werden auf Arbeitspakete abgebildet; zum Beispiel resultiert die Initialisierung der

[13] Der Wert von 5 Takten ergibt sich in diesem Fall aufgrund einer Vorgabe bezüglich der minimalen *page-fault*-Kosten.

[14] Für die Übertragung des *copy set* wird vorausgesetzt, daß jeweils zwei Knotenkennungen in einem Wort zusammengefaßt werden; mit dem *Read_Count* der Seite ergibt sich die Anzahl der zu übertragenden Worte zu *RC(Page)/2*.

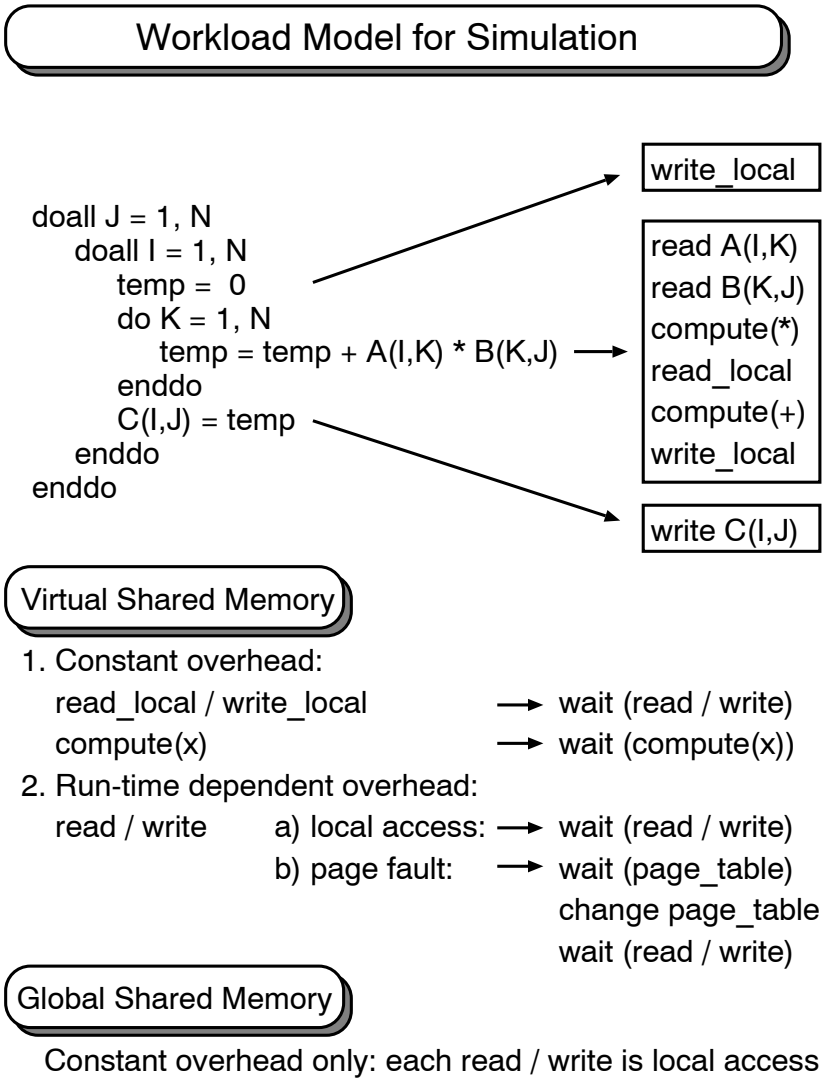


Abb. 35. Arbeitslastmodell für die Simulation

privaten Variable *temp* in einem Arbeitspaket mit der Bezeichnung *write-local* für einen lokalen Schreibzugriff auf den Speicher. Die Lese- beziehungsweise Schreibzugriffe auf die gemeinsamen Variablen *A(I,K), B(K,I)* und *C(I,J)* werden in Arbeitspakete mit der Bezeichnung *read* beziehungsweise *write* umgesetzt, die jeweils auch die zuzugreifende Variable enthalten. Die Addition und die Multiplikation resultieren in Arbeitspaketen für Rechenoperationen mit Angabe der jeweiligen Operation *(compute(+)* und *compute(*))*. Bei der Ausführung einer Arbeitslast auf einem VSM-Multiprozessor ergibt sich ein Unterschied zwischen Arbeitspaketen mit konstantem Overhead und Arbeitspaketen mit Laufzeit-abhängigem Overhead: Lokale Datenzugriffe *(read-local, write-local)* und alle Rechenoperationen *(compute)* verursachen immer den gleichen konstanten Overhead. Für *read-* und *write*-Arbeitspakete ergibt sich jedoch naturgemäß eine Abhängigkeit des Overhead von der Datenverteilung zur Laufzeit: Wenn das zuzugreifende Datum zur Laufzeit lokal verfügbar ist, dann ergeben sich lokale Datenzugriffe und damit die gleichen Kosten wie für *read-local-* bzw. *write-local*-Arbeitspakete. Wenn jedoch ein *page fault* ausgelöst wird, dann berechnet sich der Overhead aus der aktuellen Datenverteilung

anhand der Seitentabelle und kann gemäß Abschnitt 6.2 deutlich variieren. Die Simulation eines *global-shared-memory*-Multiprozessors ergibt sich in Erweiterung dazu durch die Vorgabe, daß jeder Datenzugriff lokal ist und sich damit ausschließlich konstante Kosten für die Ausführung der Arbeitspakete ergeben.
Die Beschreibung der DO-Schleifen im Rahmen des Arbeitslastmodells erfolgt durch eine Datenstruktur *Loop-Descriptor*; neben verwaltungstechnischen Daten zum Beispiel über die Schleifengrenzen enthält ein *Loop-Descriptor* eine Liste mit allen Arbeitspaketen der Schleife. Im Fall einer Schleifenschachtelung wird in die Liste ein Arbeitspaket mit der Bezeichnung *Loop* eingefügt; dieses Paket enthält einen Zeiger auf den *Loop-Descriptor* der inneren Schleife. Die resultierende Datenstruktur ist rekursiv und erlaubt so die Verwaltung beliebig geschachtelter Schleifen. Bei der Ausführung eines Arbeitspaketes *Loop* wird mithilfe des *Loop-Descriptor* eine *Thread-Queue* generiert, die in einer Liste für jede Iteration der Schleife einen Eintrag enthält. Jedem dieser Einträge wird dann eine Liste mit Arbeitspaketen zugeordnet, die den Schleifenkörper vollständig beschreibt (siehe Abb. 36). In dieser Darstellung liegen die Arbeitspakete gegenüber der ursprünglichen Form bereits modifiziert vor: Die Datenzugriffe auf gemeinsame Daten sind bezüglich der Indizierung aufgelöst und damit bereits auf den Zugriff einer Seite bezogen, und die *compute*-Pakete enthalten bereits den Wert für den Overhead.

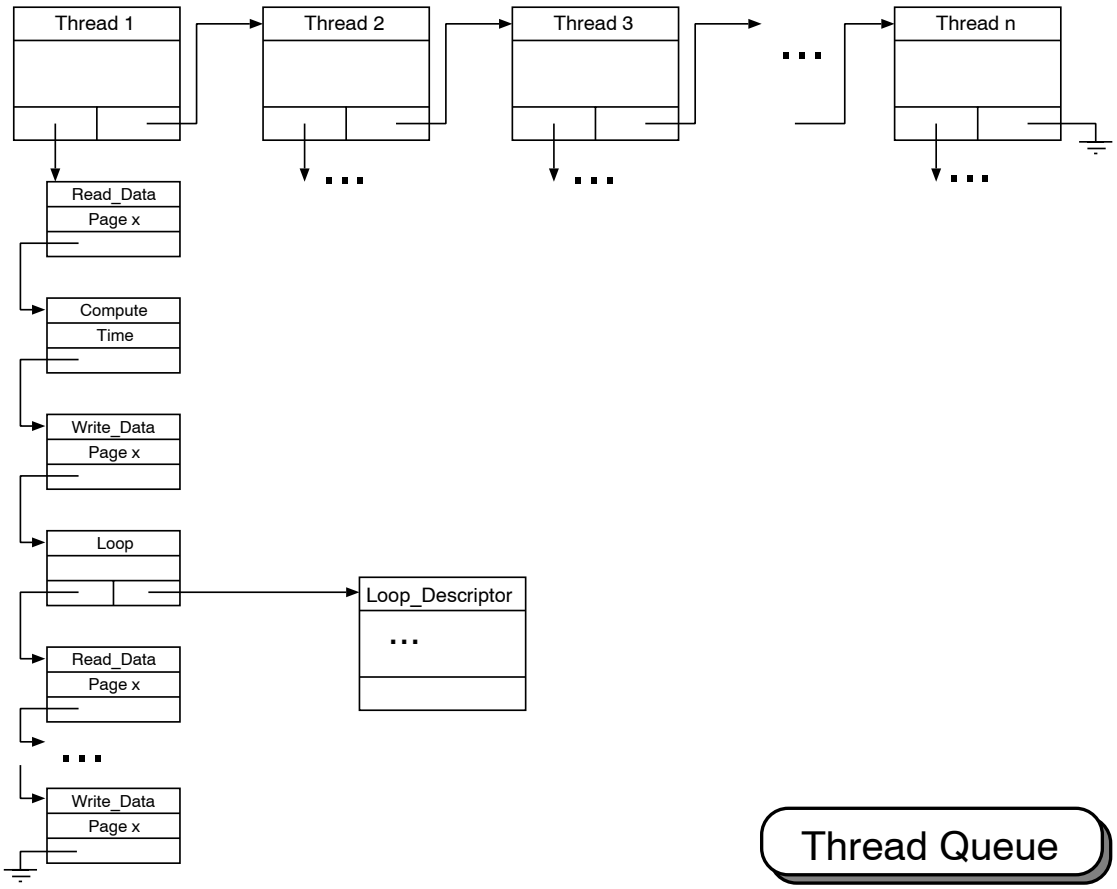


Abb. 36. Thread Queue

6.4. Simulationsparameter

Die Steuerung des Simulationssystems läßt sich in vier Bereiche aufteilen; die Konfiguration des Rechners, die Beschreibung der Arbeitslast, die Vorgaben für den Overhead von Rechenoperationen, Datenzugriffen, *page-fault*-Behandlung und Scheduling-Entscheidungen und die Steuerung der internen Simulator-Parameter. Da die internen Simulator-Parameter nur für die Steuerung des Simulationsvorgangs relevant sind, jedoch keinen Einfluß auf die Simulationsergebnisse haben, wird auf diesen Bereich hier nicht weiter eingegangen. Die Konfiguration des Rechners wird durch die in Abb. 37 beschriebenen Parameter gesteuert: Der Parameter *Shared-Memory-Type* bestimmt das Multiprozessormodell zu GSM (*global shared memory*) oder VSM (*virtual shared memory*). Mit dem Parameter *VSM-Type* kann zwischen verschiedenen Realisationen eines VSM-Modells gewählt werden: Die Modularität des Simulators gewährleistet eine einfache Schnittstelle bezüglich dem Wechsel zwischen verschiedenen VSM-Modellen; aufgrund des eingestellten Parameterwertes werden jeweils andere Routinen ausgewählt. Damit ist das Simulationssystem einfach um neue Modelle erweiterbar. Für die Untersuchungen in dieser Arbeit wird ausschließlich das in Abschnitt 6.1 beschriebene VSM-Modell mit der Bezeichnung *FDM* für *Fixed Distributed Manager* betrachtet. Weitere Konfigurationsparameter sind die Anzahl der CPUs, die Taktzeit, die Speichergröße und - für VSM-Rechner - die Seitengröße *(Number-of-CPUs, Machine-Cycle-Time, Local-Memory-Size* und *Pagesize).* Die Problemgröße der untersuchten Arbeitslast ist ein begrenzender Faktor für die maximale Anzahl der CPUs: Mit der Problemgröße wachsen Speicherbedarf und Rechenzeitbedarf für den Simulator; dadurch ist die Problemgröße begrenzt. Um eine hinreichend grobe Granularität der Ausführung zu gewährleisten, muß das Verhältnis der Problemgröße zur Anzahl der CPUs vernünftig gewählt werden. Die durchgeführten Simulationen haben gezeigt, daß zum Beispiel für eine Matrixmultiplikation der Problemgröße 200 bei einer Konfiguration mit 128 CPUs die Grenzen für Speicherbedarf, Rechenzeitbedarf und Granularität erreicht werden. Aus pragmatischen Gründen wird die Speichergröße eines Knotens als Anzahl von Seiten spezifiziert, so daß sich die tatsächliche Speichergröße in Worten oder Byte erst in Verbindung mit der Seitengröße ergibt. Das Simulationssystem wurde auch im Hinblick auf zukünftige Untersuchungen im Multiprogramming-Betrieb entworfen. Da in dieser Betriebsart der Job-Scheduler bei der Untersuchung nur eines Jobs unnötige Unterbrechungen verursacht, wird mit dem Parameter *Dedicated-Mode* eine dedizierte Maschine realisiert.
Die Beschreibung der Arbeitslast besteht gemäß Abb. 38 aus drei Teilen, nämlich aus den Steuerparametern, aus der Deklaration von gemeinsamen Daten und deren Abbildungsvorschrift auf Seiten einschließlich einer Vorschrift für die initiale Verteilung der Seiten

Shared_Memory_Type	VSM (Virtual Shared Memory), GSM (Global Shared Memory)
VSM_Type	SDM, FDM, KSR, MPP
Number_of_CPUs	1 to 128
Pagesize	1 to *
Local_Memory_Size	1 to *
Machine_Cycle_Time	*
Dedicated_Mode	true or false

Abb. 37. Steuerparameter für die Konfiguration des Rechners

Repetitions *Number_of_Processes* *Start_Time*
1 10 0

Data_Mapping

DID	*DW1*	*DW2*	*DD1*	*DD2*	*Initial_Page_Distribution*
A	100	100	Block 1	Block 100	Block Read
B	100	100	Block 100	Block 1	Each Read
C	100	100	Block 1	Block 100	Block Write

Loop-Nest

```
Loop (parallel) I = 1 to 100 step 1 by Chunk-Scheduling 10
   Loop (sequential) J = 1 to 100 step 1
      Write Local
      Loop (sequential) K = 1 to 100 step 1
         Read A(I,K)
         Read B(K,J)
         Compute (*,+)
         Write Local
      Endloop
      Write C(I,J)
   Endloop
Endloop
```

Abb. 38. Arbeitslastbeschreibung für eine Matrixmultiplikation: Datenverteilung und Programmkern

auf die Prozessorelemente und aus der Beschreibung des Programmkerns. Der Steuerparameter *Repetitions* spezifiziert, wie oft der beschriebene Programmkern ausgeführt werden soll. *Number-of-Processes* gibt die Anzahl der Prozesse an, die der Job nutzen soll. Mit dem Parameter *Start-Time* wird der Startzeitpunkt für den Job festgelegt; dieser Parameter ist nur für die Ausführung des Jobs in einer Multiprogramming-Umgebung relevant.

Unter *Data-Mapping* wird die Abbildung der Daten auf Seiten beschrieben: Jede Zeile spezifiziert ein Datenobjekt bezüglich Deklaration und Abbildungsvorschrift. Die Verteilung der Seiten auf Prozessorelemente hat einen wesentlichen Einfluß auf die Datenlokalität. Während der Ausführung wird diese Verteilung in Abhängigkeit von verschiedenen Einflußfaktoren bestimmt. Bei den Untersuchungen in Kapitel 7 werden rechenintensive Kerne von Programmen betrachtet; in der Realität wird die beim Eintritt in einen Programmkern vorherrschende Seitenverteilung von der Vorgeschichte abhängen; bei den Untersuchungen ist diese Vorgeschichte unbekannt, und die Seitenverteilung wird synthetisch generiert. Die *Initial-Page-Distribution* legt die initiale Verteilung der Seiten auf die Prozessorelemente fest; die Verteilung erfolgt für jedes Datenobjekt unabhängig. Mögliche Verteilungen sind *Block, Cyclic, Nondistributed* und *Each*. Bei der *Block*-Verteilung erhält jeder Knoten einen zusammenhängenden Block von Seiten. Die *Cyclic*-Verteilung bewirkt eine zyklische Verteilung von jeweils einer Seite auf einen Knoten je Zyklus. Bei *Nondistributed* erfolgt keine Verteilung der Seiten; alle Seiten werden einem Knoten zugewiesen. Für die drei Verteilungen *Block, Cyclic* und *Nondistributed* kann außerdem festgelegt werden, ob das Zugriffsattribut der Seiten *Read* oder *Write* sein soll. Bei der Verteilung *Each* erhält jeder Knoten eine Kopie aller Seiten; für diese Verteilung muß das Zugriffsattribut *Read* heißen. Die initiale Verteilung der Seiten muß auf die Lokalspeichergröße der Knoten abgestimmt sein.

Die Beschreibung der Arbeitslast erfolgt in Anlehnung an die Abbildung auf Arbeitspa-

Loop Type	sequential, parallel
Algorithm for Loop-Scheduling	Block-Scheduling n (n = number of processes) Cyclic-Scheduling l (l = cycle length) Self-Scheduling Chunk-Scheduling b (b = block width) Guided-Self-Scheduling n Factoring n Block-Factoring n b (Factoring with Loop-Blocking) Align-Scheduling data n (data = data array) Template-Scheduling t n (t = template)
Loop Index Variable	variable
Loop Index Start and End Value	integer constant or variable or term of variable, operator (+,-,*,/) and integer constant
Loop Index Increment	integer constant

Abb. 39. Parametervorgaben für eine Schleife

kete (siehe Abb. 38, *Loop-Nest*). Es wird jeweils ein Arbeitspaket pro Zeile spezifiziert. Für ein Arbeitspaket *Loop* werden der Schleifentyp, der Schleifenindex, die Schleifengrenzen und das Schleifeninkrement spezifiziert. Bei einer parallelen Schleife wird außerdem der Loop-Scheduling-Algorithmus angegeben. Abb. 39 beschreibt die jeweils möglichen Parametervorgaben. Der Schleifentyp kann sequentiell oder parallel sein. Der Loop-Scheduling-Algorithmus wird zum Teil zusammen mit entsprechenden Zahlenwerten angegeben. Schleifenindex, -grenzen und -inkrement werden als Konstante, Variable oder Term spezifiziert. Abb. 40 gibt einen Überblick über die definierten Arbeitspakete. Neben den bereits aus Abb. 38 bekannten Arbeitspaketen sind drei Varianten für die Abbildung von *prefetch*-Operationen definiert, die sich jeweils durch das Zugriffsattribut unterscheiden. Darüber hinaus kann mithilfe von *Index-Change* ein Indexfeld zufällig geändert werden. Das in Abb. 40 benutzte Datenelement *data* kann folgendermaßen spezifiziert werden: Ein Datenfeldindex kann eine Variable, ein Term oder ein Indexfeld sein. Ein Datenelement dieser Form wird für Datenzugriffs- und *prefetch*-Operationen sowie für die Spezifikation des *Align-Scheduling* benötigt.

Loop	loop start
Endloop	loop end
Read data	read access (data = data element, e.g. A(K+1,J)))
Write data	write access
Weak data	weak coherent access
Read Local	local read access
Write Local	local write access
Compute operator	compute operation (operator = '+,-,*,/')
Prefetch_Read data	prefetch operation for read access
Prefetch_Write data	prefetch operation for write access
Prefetch_Weak data	prefetch operation for weak coherent access
Index_Change r	change index address array r by random

Abb. 40. Die Spezifikation von Arbeitspaketen

6.5. Auswertung der Simulationen

Bei Vergleichsmessungen auf einer dedizierten Maschine stellt die Realzeit für die Ausführung eines Programms ein wesentliches Bewertungskriterium dar. Auf dieser Grundlage können zum Beispiel die jeweils am besten geeigneten Scheduling-Algorithmen identifiziert werden. Für das Erkennen von Effekten und Ursachen in bezug auf Ineffizienzen sind jedoch weitere, detailliertere Informationen nötig: Die Ausführung eines Programms läßt sich durch die zeitliche Abfolge der Zustände der an der Ausführung beteiligten Prozesse beschreiben. Zunächst befinden sich alle Prozesse im *idle*-Zustand. Bei Programmstart benötigen alle Prozesse den Job-Scheduler. Da der Job-Scheduler nur exklusiv zugeteilt wird, geht nur ein Prozeß in den Zustand *job-scheduler*; alle anderen Prozesse warten auf die Zuteilung des Job-Scheduler und gehen in den Zustand *system-waiting*, der das Warten auf Systemressourcen kennzeichnet. Vom Zustand *job-scheduler* wechselt der Prozeß zum Zustand *user*, der schließlich die tatsächliche Programmausführung kennzeichnet.
Bei der Programmausführung im Bereich paralleler Kontrollstrukturen kann die Beschreibung der Prozeßzustände ergänzt werden: Um an der Bearbeitung einer parallelen Schleife zu partizipieren, geht ein Prozeß zunächst in den Zustand *thread-scheduler* über. Wird dem Prozeß von dieser Instanz ein Teil der Schleifeniterationen zur Bearbeitung übergeben, dann kann wieder ein Wechsel zum Zustand *user* erfolgen. Wenn die Iterationen der Schleife bereits vollständig verteilt sind, dann kann jedoch keine weitere Zuweisung erfolgen. In dieser Situation bestimmt sich der neue Zustand des Prozesses wie folgt: Wenn andere Prozesse noch mit der Bearbeitung der Schleife beschäftigt sind, dann muß auf deren Fertigstellung gewartet werden; der betroffene Prozeß wechselt in den Zustand *busy-waiting*. Handelt es sich jedoch um den letzten Prozeß, der mit der Bearbeitung der Schleife fertig wird, dann ist der Synchronisationspunkt für alle beteiligten Prozesse erreicht: Der Prozeß setzt die Bearbeitung des Programms hinter der Schleife fort und wechselt in den Zustand *user*; beim erneuten Eintritt in eine parallele Kontrollstruktur werden die Prozesse im Zustand *busy-waiting* vom aktuellen Prozeß "geweckt" und nehmen mit der Zustandsfolge *thread-scheduler*, *user* die Bearbeitung wieder auf.
Für VSM-Systeme ergibt sich in diesem Zusammenhang ein weiterer, für die Beschreibung ganz wesentlicher Prozeßzustand: Beim Autreten eines *page fault* entsteht eine Wartezeit, in der der Prozeß auf die Bereitstellung der entsprechenden Seite warten muß; dieser Wartezustand wird mit *page-waiting* bezeichnet. In Abhängigkeit von der jeweiligen Zugriffsanforderung kann dabei zwischen *page-waiting-read* und *page-waiting-write* unterschieden werden. Um detaillierte Einblicke in das Verhalten einer Anwendung zu ermöglichen, realisiert das Simulationssystem eine solche Beschreibung eines Programmablaufes. Der zeitliche Anteil der Prozeßzustände - über die gesamte Ausführung betrachtet - kann zu einem ersten Anhaltspunkt für eine Bewertung führen; das Simulationssystem liefert die akkumulierten Werte dieser Größen zum Beispiel aufgeschlüsselt für jeden Prozessor.
Der zeitliche Anteil der Zustände *user*, *page-waiting* und *busy-waiting* an der gesamten Ausführungszeit erlaubt bereits eine grobe Bewertung bezüglich der Lokalität und der Parallelität einer Ausführung. Um zum Beispiel das Auftreten von hohen *page-waiting*-Anteilen nicht nur zu erkennen sondern im Detail zu verstehen, sind jedoch

weitere Informationen nötig: Erst die zeitliche Abfolge der Prozeßzustände mit Bezug auf den jeweiligen Prozessor ermöglicht die Zuordnung von konkreten Zustandswechseln zu den Operationen einer Anwendung und damit das Verständnis von Zusammenhängen. Das Simulationssystem erzeugt eine *Trace*-Datei mit chronologischen Einträgen über die Zustandswechsel und die jeweiligen Bezugszeitpunkte für alle Prozesse; die Auflösung beträgt einen Maschinentakt.
Die Verhältnisse in VSM-Systemen hängen in erheblichem Maß von der aktuellen Seitenverteilung ab. In vielen Situationen sind zum Verständnis der Prozeßzustandswechsel Informationen über die aktuelle Verteilung der Seiten unerläßlich. Um dieser Anforderung gerecht zu werden, erzeugt das Simulationssystem weiterhin Einträge zur Verteilung der Seiten über die Prozessorelemente. Auch diese Einträge erfolgen chronologisch und werden in derselben *Trace*-Datei abgelegt.
Die Vielseitigkeit der Einträge in der *Trace*-Datei über die Wechsel von Prozeß- und Seitenzuständen und die hohe zeitliche Auflösung der Simulation resultieren in einer erheblichen Menge von Daten; für die Ausführung einer Matrixmultiplikation können zum Beispiel für ein Intervall von 6 Millionen Maschinentakten etwa 200,000 Zeilen *Trace*-Daten erzeugt werden - das entspricht etwa 20 Mbyte. Das Auffinden von relevanten Einträgen innerhalb einer solchen Datei und der Vergleich von verschiedenen Messungen anhand mehrerer Dateien ist aufgrund der großen Datenmenge ohne weiteres nahezu unmöglich. Um die Auswertung der Simulationsdaten zu unterstützen, wird das Visualisierungstool PARvis eingesetzt. PARvis ermöglicht die graphische Darstellung der Zustandswechsel von Prozessen in Multiprozessorsystemen auf der Basis von *Trace*-Dateien und ist insbesondere auch auf die Visualisierung der Seitenverteilung in VSM-Systemen zugeschnitten. Die Mächtigkeit von PARvis durch eine Vielfalt von Sichtweisen und Filterfunktionen ermöglicht die effektive Verfikation und Analyse von Simulationsläufen. Eine Beschreibung von PARvis findet sich in [Arn93, NaAr93, Mul94] und erfolgt im Rahmen dieser Arbeit anhand der in Kapitel 7 genutzten Visualisierungen.

7. Simulationsergebnisse

Das entwickelte Simulationssystem stellt eine flexible und leistungsfähige Umgebung für Effizienzuntersuchungen in *virtual-shared-memory*-Rechnern zur Verfügung: Es können sowohl die zu bearbeitenden Arbeitslasten als auch die Konzepte und Algorithmen für die Ablaufplanung innerhalb des Laufzeitsystems auf deren Eignung für VSM-Rechner untersucht werden; dabei können aufgrund der bei den Simulationen erzeugten Daten negative Effekte identifiziert und gegebenenfalls durch geeignete Maßnahmen behoben werden. Die Modularität und die Parametersteuerung des Simulationssystems ermöglichen insbesondere die Untersuchung von alternativen Algorithmen zur Ablaufplanung. Die Abhängigkeit geeigneter Algorithmen vom Speicherverhalten der jeweils zugrundeliegenden Anwendung erfordert die beispielhafte Untersuchung von charakteristischen Anwendungen; damit können die Ergebnisse der Untersuchungen keine universelle Verifikation der propagierten Verfahren darstellen. Das Ziel dieses Kapitels ist es vielmehr, für charakteristische Beispielanwendungen die Anforderungen an die Ablaufplanung anhand von Effekten zu identifizieren und die Möglichkeit der Leistungssteigerung durch geeignete Algorithmen aufzuzeigen. Die folgende Beschreibung der durchgeführten Untersuchungen strukturiert sich daher anhand der jeweils betrachteten Programmkerne. Bei den Untersuchungen in diesem Kapitel wird die Ablaufplanung bezüglich der folgenden Bereiche beeinflußt:

- die Abbildung von Daten auf Seiten
- die initiale Verteilung der Seiten auf die Prozessorelemente
- das *thread*-Scheduling, d.h. die Partitionierung der Iterationen von parallelen Schleifen und die Verteilung der dabei entstehenden *chunks* auf die Prozesse

Für die drei Bereiche in dieser Liste ergibt sich eine Abhängigkeit in der Form, daß eine geeignete Ablaufplanung in einem Bereich eine wesentliche Grundlage für die Ablaufplanung des oder der in der Liste folgenden Bereiche darstellt. Für einzelne Beispielanwendungen können sich besondere Abhängigkeiten von der zugrundeliegenden Multiprozessorkonfiguration ergeben; diesbezüglich werden Parameterstudien mit Variation von Prozessorzahl, Lokalspeichergröße oder Seitengröße durchgeführt. Wenn keine anderen Angaben gemacht werden, dann wurde eine Multiprozessorkonfiguration mit 10 Prozessoren, mit unbegrenztem Lokalspeicher und mit einer Seitengröße von 128 Worten zugrunde gelegt. Die zu einem beliebigen Zeitpunkt während der Programmausführung vorherrschende Verteilung der Seiten auf Prozessorelemente wird in der Realität durch die Vorgeschichte bestimmt. Im folgenden werden Kerne von Programmen betrachtet, und die initiale Seitenverteilung zu Beginn der Ausführung eines Programmkerns wird synthetisch generiert: Wo keine besonderen Angaben gemacht werden, wurde versucht, eine realistische Seitenverteilung nachzuempfinden; um den Einfluß der Seitenverteilung zu untersuchen, werden in Abschnitt 7.6 Messungen mit günstiger Verteilung und Messungen mit ungünstiger Verteilung gegenübergestellt.

7.1. Der Einfluß der Abbildung von Daten auf Seiten

Eine geeignete Abbildung der Daten auf Seiten stellt eine wesentliche Grundlage für ein effizientes *thread*-Scheduling dar; deswegen soll dieser Aspekt zunächst isoliert betrachtet werden. Bei Vergleichsmessungen wird der heute üblichen, spaltenorientierten Datenabbildung ohne Ausrichtung von Vektoren auf Seitengrenzen eine jeweils geeignete, explizite Datenabbildung gegenübergestellt. Das Ausmaß der beobachteten Effekte soll ein Urteil darüber ermöglichen, ob eine explizite Datenabbildung für VSM-Rechner relevant ist. Die Abhängigkeit der Rechenleistung von verschiedenen Datenabbildungen wird für unterschiedliche Programmkerne untersucht; dabei wurde versucht, bezüglich des Speicherverhaltens typische Fälle zu betrachten.

7.1.1. Fallstudien mit verschiedenen Programmkernen

In einer ersten Fallstudie wird ein Algorithmus zur Matrixmultiplikation untersucht, der bezüglich der äußeren Schleife parallelisiert wird und der durch die Reihenfolge der Schleifen einen spaltenorientierten Datenzugriff realisiert. Auch die Abbildung der Daten auf Seiten erfolgt in allen betrachteten Fällen spaltenorientiert; es soll der Unterschied einer Datenabbildung ohne beziehungsweise mit Ausrichtung der zugegriffenen Vektoren auf Seitengrenzen untersucht werden.
Die Abb. 41 enthält zwei Darstellungen von Simulationsdaten, die mithilfe von PARvis visualisiert worden sind: Die Darstellungen geben den zeitlichen Verlauf der Prozeßzustände für alle Prozessoren wieder; die angegebene Zeitskala bezieht sich auf Maschinentakte. Die Abb. 41.a beschreibt die gesamte Ausführung einer Matrixmultiplikation der Dimension 100 und bezieht sich auf einen Zeitraum von etwa 4 Millionen Maschinentakten, so daß nur sehr wenige Details erkennbar sind. Dargestellt sind die Prozeßzustände für die Ausführung der Matrixmultiplikation mit einer Datenabbildung ohne Ausrichtung der Vektoren auf Seitengrenzen: Zu Beginn der Ausführung befinden sich die Prozesse die meiste Zeit im Zustand *page-waiting-read* (PAGE-WAIT-R); dieser Zustand ist durch das Warten auf Seiten für den Lesezugriff gekennzeichnet: Jeder Prozeß benötigt jeweils Teile der beiden zu multiplizierenden Matrizen. Die entsprechenden Seiten werden beim ersten Bedarf angefordert; da es keine Schreibzugriffe auf diese Matrizen gibt und auch keine Speicherengpässe auftreten, bleiben die Seiten für die gesamte weitere Ausführung lokal verfügbar. Nachdem alle *read page faults* aufgelöst sind - das ist nach etwa 500,000 Maschinentakten der Fall - wird die weitere Ausführung im wesentlichen von zwei Prozeßzuständen dominiert: Mit LOOP 2 wird die Ausführung der parallelen Schleife und damit der Zustand *user* gekennzeichnet; in diesem Zustand führt der entsprechende Prozeß die substantielle Arbeit aus, also zum Beispiel Rechenoperationen und lokale Datenzugriffe. Der zweite dominante Zustand ist durch die vielen kurzzeitigen Unterbrechungen des *user*-Zustandes in Form von schmalen dunklen Strichen zu erkennen; die Unterbrechungen beruhen auf *write page faults* (PAGE-WAIT-W). Die Ursache für die große Häufigkeit der *write page faults* wird in Abb. 41.b deutlich; die Darstellung gibt einen vergrößerten Detailausschnitt zu Abb. 41.a wieder. Die Zahlen in den dunklen Abschnitten für PAGE-WAIT-W geben die Identifikationsnummern der Seiten an. Die Zahlen in den hellen Abschnitten für LOOP 2 geben die Identifikationsnummern der Prozesse an. Das sich ständig wiederholende Auftreten von *write page faults* auf

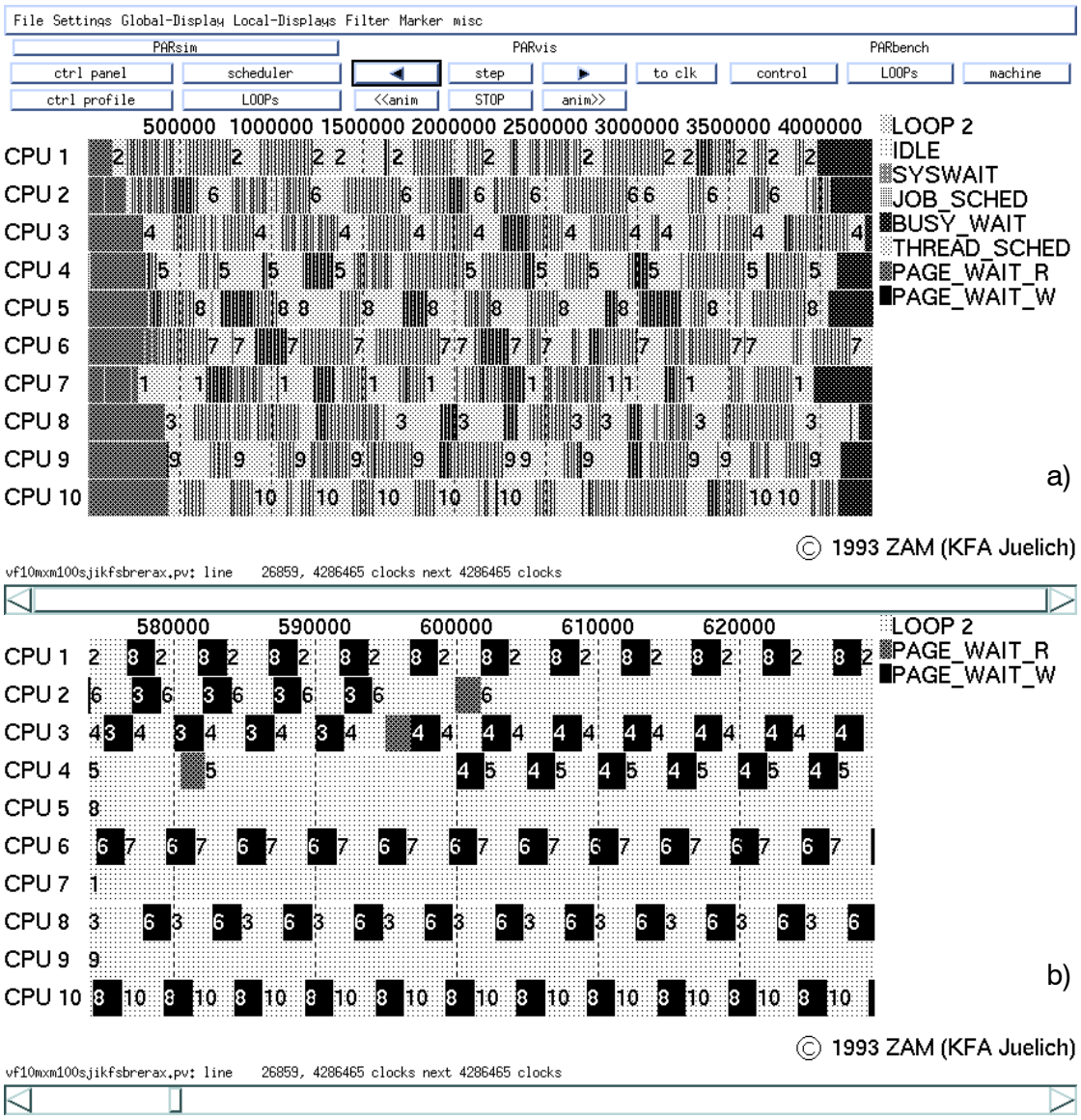


Abb. 41. Matrixmultiplikation ohne Ausrichtung der Vektoren auf Seitengrenzen: Zeitliniendarstellung der Prozeßzustände, Matrixmultiplikation 100*100, Seitengröße 128, 10 CPUs, spaltenweise Datenabbildung ohne Ausrichtung, Datenzugriff spaltenweise, a) Darstellung der gesamten Ausführung b) Detailausschnitt zu Abb. a: page thrashing für die Seiten 3, 4, 6 und 8

jeweils dieselbe Seite und auf jeweils zwei Prozessoren - in der Abb. 41.b für die Seiten 3, 4, 6 und 8 - kennzeichnet den Effekt des *page thrashing*: Für einen Schreibzugriff fordert ein Prozessor eine Seite an; parallel dazu ergibt sich ein Schreibzugriff auf dieselbe Seite auf einem anderen Prozessor, so daß die Seite dem ersten Prozessor wieder entzogen wird. Da beide Prozessoren jeweils mehrere Schreibzugriffe auf der Seite ausführen müssen, wird die Seite mehrfach zwischen den Prozessoren ausgetauscht. Bei der untersuchten Matrixmultiplikation wird jedes Feldelement der Produktmatrix genau einmal schreibend zugegriffen. Daraus folgt, daß die parallelen Datenzugriffe verschiedener Prozessoren auf dieselbe Seite durch den Effekt des *false sharing* hervorgerufen

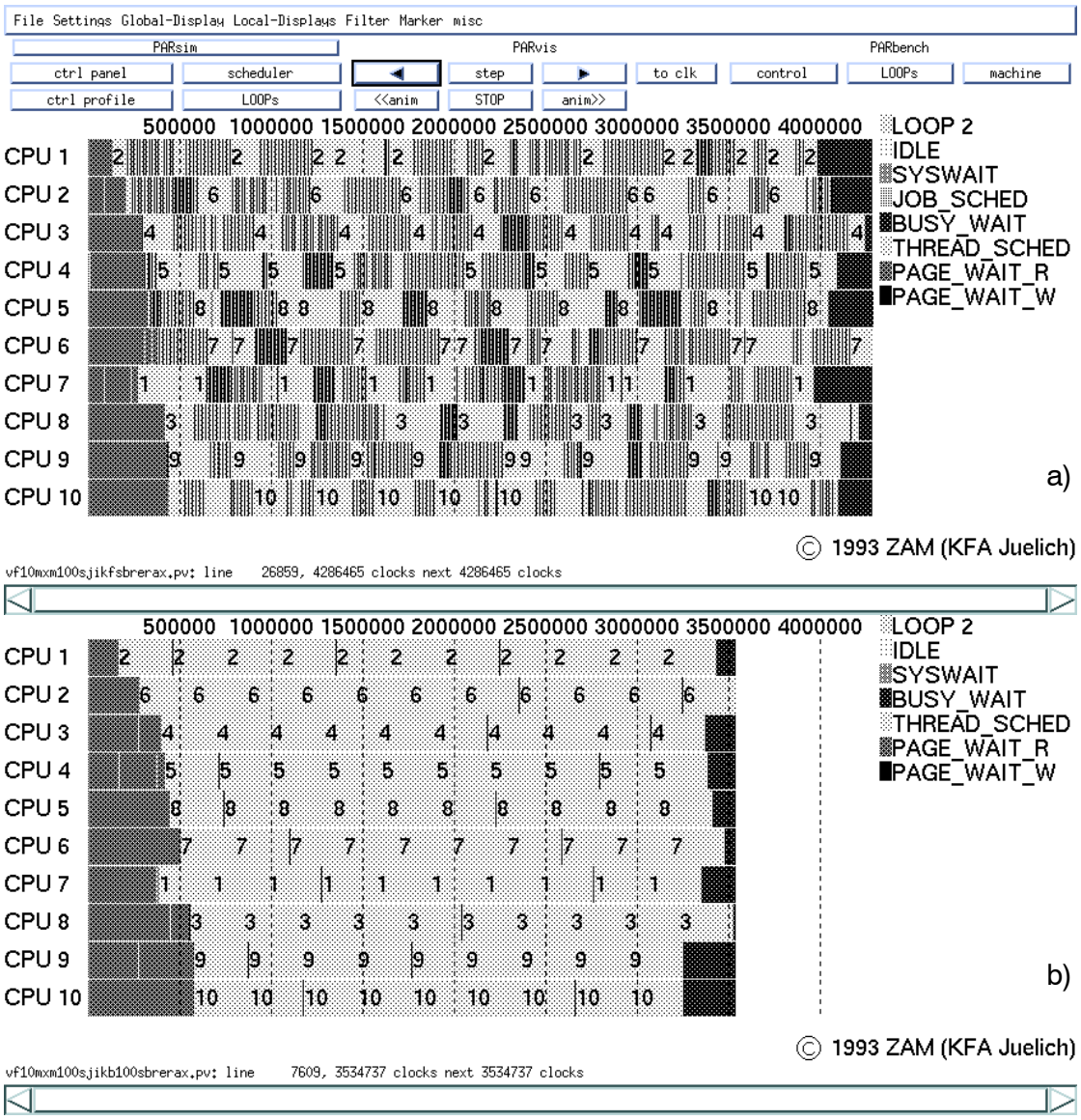


Abb. 42. Matrixmultiplikation ohne und mit Ausrichtung der Vektoren auf Seitengrenzen: Zeitliniendarstellung der Prozeßzustände, Matrixmultiplikation 100*100, Seitengröße 128, 10 CPUs, Datenzugriff spaltenweise, a) Datenabbildung spaltenweise ohne Ausrichtung b) Datenabbildung spaltenweise mit Ausrichtung auf Seitengrenzen

werden, d.h. es werden jeweils unterschiedliche Adressen zugegriffen, die jedoch auf derselben Seite liegen. Die Ursache für das *false sharing* liegt in der Datenabbildung: Die Datenabbildung ohne Ausrichtung der Spaltenvektoren auf Seitengrenzen bewirkt, daß auf eine Seite jeweils Teile verschiedener Spaltenvektoren abgebildet werden. Da verschiedene Spaltenvektoren durch die Schleifenpartitionierung im allgemeinen auf verschiedenen Prozessoren benötigt werden, stellen sich das *false sharing* und die damit verbundenen Zugriffskonflikte ein.

Eine Vermeidung des *false sharing* kann durch die Ausrichtung der Spaltenvektoren auf Seitengrenzen erreicht werden: Die Abb. 42 stellt die bereits untersuchte Messung ohne

Ausrichtung (Abb. 42.a) und eine Messung mit Ausrichtung (Abb. 42.b) gegenüber. In der Vergleichsmessung wird der Bereich vor 500,000 Maschinentakten wie zuvor durch *read page faults* dominiert; ein genauer Vergleich ergibt, daß der Bereich im Unterschied zu Abb. 42.a etwas ausgedehnt ist. Die Ursache hierfür liegt in der Datenabbildung: Durch die Ausrichtung der Spaltenvektoren mit der Vektorlänge 100 auf die Seitengrenzen ergibt sich bei einer Seitengröße von 128 Worten ein Speicherplatzverlust von 28 Worten pro Seite. Damit wird insgesamt Speicherplatz verschenkt, so daß die Anzahl der Seiten für die Speicherung einer Matrix vergleichsweise höher ist. Um die Matrix auf die Prozessoren zu verteilen, müssen damit insgesamt auch mehr Seiten verschickt werden. Der erhöhte Aufwand beruht also auf dem Transfer von nicht belegtem Speicherplatz; ob dieser Speicherplatz für andere Daten sinnvoll genutzt werden kann, kann möglicherweise der Compiler entscheiden. Dieser nachteilige Effekt wird durch die Vermeidung des *false sharing* und das damit verbundene Ausbleiben des *page thrashing* überkompensiert: Nach der Auflösung aller *read page faults* tritt fast ausschließlich der Zustand *user* (LOOP 2) auf. Es treten nur vereinzelt *write page faults* auf; dabei bleibt eine Seite jedoch nach der Auflösung eines *page fault* im Zugriff. Das Ende der Ausführung wird in beiden Fällen (Abb. 42.a und 42.b) durch den Zustand *busy-waiting* gekennzeichnet, der durch eine Synchronisation am Ende der parallelen Schleife verursacht wird. Ein Vergleich der Gesamtausführungszeit für die beiden betrachteten Fälle ergibt eine Effizienzsteigerung durch die Datenabbildung mit Ausrichtung der Vektoren auf Seitengrenzen. Abb. 43 stellt für die beiden betrachteten Messungen den relativen Overhead in Prozent dar: Der relative Overhead ergibt sich als die Summe der zeitlichen Anteile der Prozeßzustände *page-waiting-read* und *page-waiting-write* (zusammengefaßt in PAGE), *busy-waiting* (BUSY) und *thread-scheduler* (THREAD) bezogen auf die Summe der Anteile des Zustandes

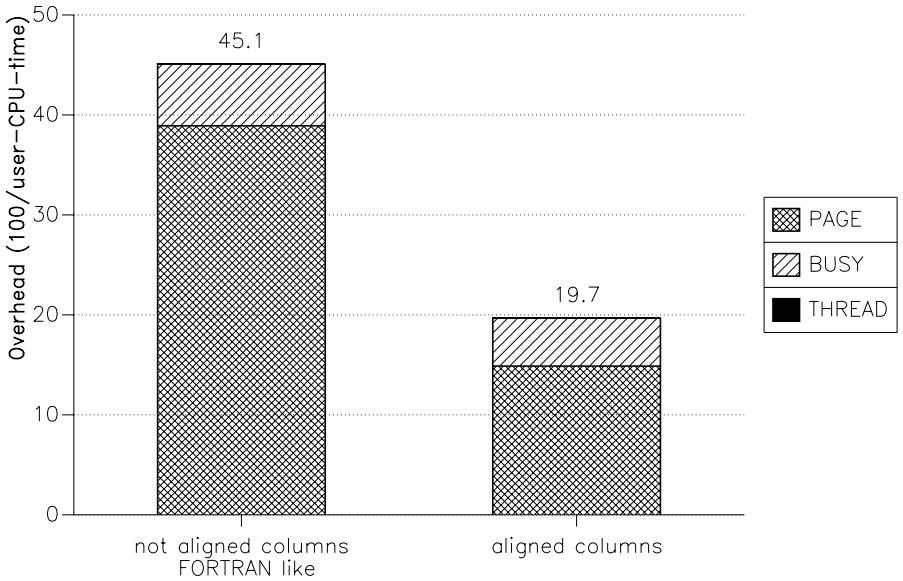


Abb. 43. Datenabbildung ohne und mit Ausrichtung der Vektoren auf Seiten: Overhead bezogen auf die user-CPU-time in Prozent, Matrixmultiplikation 100*100 (Datenzugriff spaltenweise), Seitengröße 128, 10 CPUs, links: spaltenorientierte Abbildung ohne Ausrichtung, rechts: spaltenorientierte Abbildung mit Ausrichtung

user; im Zustand *user* wird die substantielle Arbeit in Form von Rechenoperationen und Speicherzugriffen ausgeführt. Alle anderen Zustände können anteilmäßig vernachlässigt werden. Der Anteil des Zustandes *thread-scheduler* ist in der Abb. 43 so gering, daß er in Verbindung mit dem groben Maßstab nicht zu erkennen ist. Da die Werte für den Zustand *user* bei den beiden Messungen identisch sind, können die Werte für den relativen Overhead absolut verglichen werden. Der linke Balken steht für die Messung bei der Datenabbildung ohne Ausrichtung, der rechte Balken für die Messung bei der Datenabbildung mit Ausrichtung: Durch die Ausrichtung ergibt sich eine Verringerung des relativen Overhead von 45.1 auf 19.7 Prozent, die nahezu ausschließlich auf einer Verkürzung der *page-waiting*-Anteile beruht. Diese Effizienzsteigerung favorisiert die explizite Abbildung von Daten auf Seiten; im folgenden soll untersucht werden, wie sich der Einfluß der Datenabbildung bei anderen Arbeitslasten darstellt.

Bei der Gaußelimination gemäß Abb. 44.a erfolgt der Datenzugriff zeilenorientiert; jede Iteration der parallelen *I*-Schleife greift schreibend auf die gesamte *I*-te Zeile des Datenfeldes *A(N,N)* zu. Für diese Zugriffsorientierung werden die Zeitverhältnisse bei der heute üblichen spaltenorientierten Datenabbildung ohne Ausrichtung untersucht; zum Vergleich wird wiederum eine Messung mit einer geeigneten, in diesem Fall zeilenorientierten Datenabbildung mit Ausrichtung gegenübergestellt: Abb. 45 beschreibt den relativen Overhead für die beiden Messungen. Bei der herkömmlichen spaltenorientierten Datenabbildung ergibt sich im wesentlichen ausschließlich *page-waiting* und *busy-waiting*. Der durch diese beiden Anteile bestimmte relative Overhead beträgt mehr als 10,000 Prozent, so daß der zeitliche Anteil für den Zustand *user* an der gesamten Ausführungszeit weniger als 1 Prozent beträgt. Die katastrophalen Verhältnisse sind in diesem Ausmaß nicht unerwartet: Durch die Orthogonalität von Datenabbildung und Datenzugriff benötigt jeder Prozessor jede Seite; das hierdurch bedingte massive Kommunikationsaufkommen führt zu einem Engpaß, der die Wartezeit bei *page faults* deutlich verlängert. Für massivparallele Anwendungen ist deswegen eine deutliche Vergrößerung der negativen Effekte zu erwarten.

Gaussian elimination

a) accessing rows

```
do K = 1, N-1
   doall I = K+1, N
      A(I,K) = A(I,K) / A(K,K)
      do J = K+1, N
         A(I,J) = A(I,J) - A(I,K) * A(K,J)
      enddo
   enddo
enddo
```

b) accessing columns

```
do K = 1, N-1
   doall I = K+1, N
      A(K,I) = A(K,I) / A(K,K)
      do J = K+1, N
         A(J,I) = A(J,I) - A(K,I) * A(J,K)
      enddo
   enddo
enddo
```

Abb. 44. Die Gaußelimination:
a) mit zeilenorientiertem Datenzugriff
b) mit spaltenorientiertem Datenzugriff

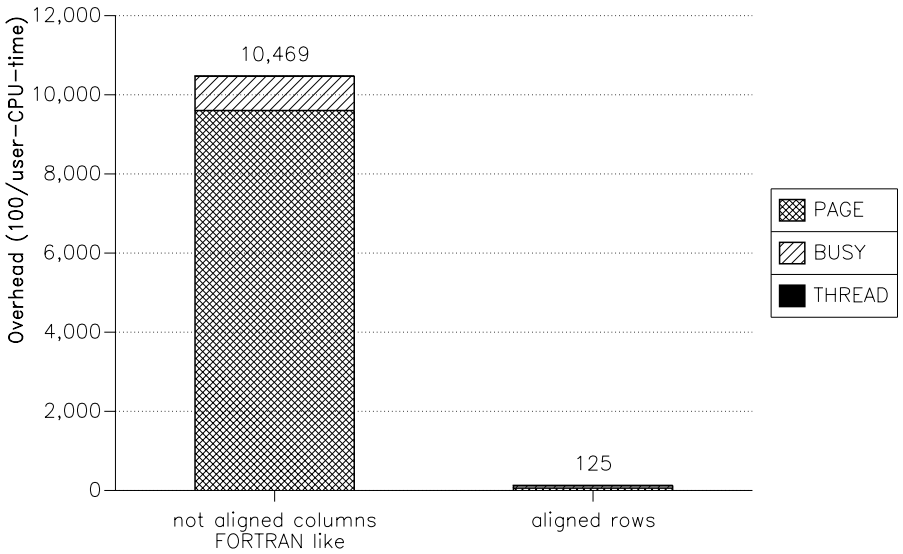


Abb. 45. Datenabbildung ohne und mit Ausrichtung der Vektoren auf Seiten: Overhead bezogen auf die user-CPU-time in Prozent, Gaußelimination 100*100 (Datenzugriff zeilenweise), Seitengröße 128, 10 CPUs, links: spaltenorientierte Abbildung ohne Ausrichtung, rechts: zeilenorientierte Abbildung mit Ausrichtung

Die Vergleichsmessung (Abb. 45 rechts) mit zeilenorientierter Datenabbildung ist durch den Maßstab der Darstellung nicht erkennbar. Tatsächlich ergibt sich hier ein um über 10,000 Prozent verringerter relativer Overhead. Mit 125 Prozent liegt der Overhead jetzt in derselben Größenordnung wie der *user*-Anteil, ein signifikant besseres, aber kein gutes Ergebnis; die Untersuchungen in den weiteren Abschnitten dieses Kapitels werden jedoch zeigen, daß die Effizienz bei der Ausführung dieses Algorithmus weiter gesteigert werden kann. Zunächst untermauert dieses Ergebnis die Notwendigkeit der Anpassung von Datenabbildung und Datenzugriff. Ob jedoch besser die Datenabbildung oder der Datenzugriff angepaßt wird, läßt sich aus dieser Untersuchung nicht ableiten. Prinzipiell kann nämlich für jede Schleifenschachtelung durch eine Vertauschung der Dimensionen ein zeilenorientierter Datenzugriff in einen spaltenorientierten Datenzugriff transformiert werden. Ob diese Möglichkeit in der Praxis für bestehende Programme handhabbar ist, oder ob sich durch eine solche Transformation innerhalb von anderen Programmteilen Nachteile ergeben können, soll hier nicht weiter diskutiert werden.
Für die Gaußelimiantion ergibt sich durch diese Transformation ein Schleifennest gemäß Abb. 44.b; jede Iteration der parallelen *I*-Schleife greift jetzt schreibend auf die gesamte *I*-te Spalte des Datenfeldes *A(N,N)* zu. Um diesen Programmkern bezüglich des Einflusses der Ausrichtung von Vektoren auf Seitengrenzen zu untersuchen, wird für zwei weitere Vergleichsmessungen wie bei der Matrixmultiplikation die spaltenorientierte Datenabbildung einmal ohne Ausrichtung und einmal mit Ausrichtung zugrunde gelegt. Abb. 46 stellt die beiden Messungen gegenüber (links ohne Ausrichtung, rechts mit Ausrichtung). Die auf der Datenabbildung ohne Ausrichtung beruhenden negativen Effekte sind für die Gaußelimination deutlich größer als für die Matrixmultiplikation (siehe Abb. 43). Die Verbesserung bezüglich dem relativen Overhead von 872 auf 125 Prozent ist weitaus signifikanter als für die Matrixmultiplikation. Dabei ist jedoch zu beachten, daß in beiden

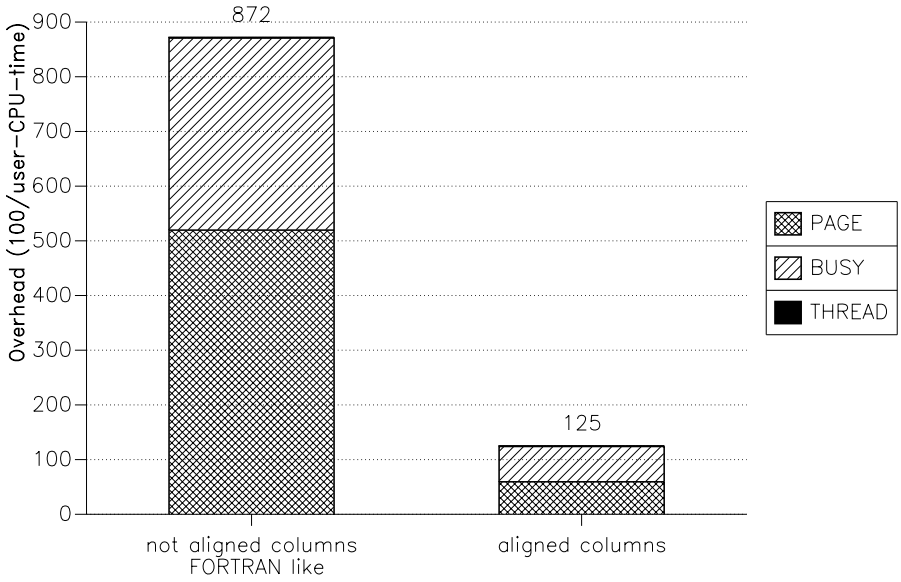


Abb. 46. Datenabbildung ohne und mit Ausrichtung der Vektoren auf Seiten: Overhead bezogen auf die user-CPU-time in Prozent, Gaußelimination 100*100 (Datenzugriff spaltenweise), Seitengröße 128, 10 CPUs, links: spaltenorientierte Abbildung ohne Ausrichtung, rechts: spaltenorientierte Abbildung mit Ausrichtung

Darstellungen der Abb. 46 der relative Overhead um etwa eine Größenordnung höher ist als bei der Matrixmultiplikation. Die Abb. 47 gibt Aufschluß über die Verhältnisse bei der Gaußelimination ohne Ausrichtung der Datenabbildung: Die Abbildung zeigt einen Ausschnitt der gesamten Ausführung; die im gezeigten Ausschnitt vorkommenden Prozeßzustände sind *user* (LOOP 2), *page-waiting-read* und *page-waiting-write*. Wie schon bei der Matrixmultiplikation ist der Effekt des *page thrashing* durch gleichzeitige Schreibanforderungen von jeweils zwei Prozessoren auf dieselbe Seite zu beobachten, zum Beispiel für CPU 3 und CPU 4 auf die Seite 4 und für CPU 6 und CPU 9 auf die Seite 8. Die dominierende Rolle bezüglich des Zeitverhaltens spielt bei der Gaußelimination jedoch das *page thrashing* aufgrund der Schreibanforderung eines Prozessors bei gleichzeitiger Leseanforderung aller übrigen Prozessoren. In der Abb. 47 ist dieser Effekt für die Seite 2 zu beobachten; CPU 1 versucht im dargestellten Zeitintervall ständig auf Seite 2 zu schreiben, während alle anderen CPUs häufig von dieser Seite lesen wollen. Die Ursache für diesen Effekt kann der Abb. 44.b entnommen werden: Innerhalb der parallelen Schleife führt ein Prozeß die Iteration *I=K+1* aus und schreibt dabei auf die Datenelemente *A(K,I)* und *A(J,I)* mit *J=K+1* bis *N*; diese Datenelemente befinden sich in der *K+1*-ten Spalte der Matrix *A*. Alle Prozesse innerhalb der parallelen Schleife lesen die Datenelemente *A(K,K)* und *A(J,K)* mit *J=K+1* bis *N*; diese Datenelemente befinden sich in der *K*-ten Spalte der Matrix *A*. Durch die Plazierung von benachbarten Spalten der Matrix auf einer Seite ergeben sich damit die in Abb. 47 beobachteten Konflikte. Insgesamt nimmt der Zustand *page-waiting* mit über 500 Prozent relativ zur *user*-CPU-Zeit den größten Anteil bezüglich der Ausführung ein. Die große Zahl von *page faults* verursacht eine sehr unbalancierte Ausführung, so daß an den Synchronisationspunkten, d.h. am Ende einer parallelen Schleifenausführung, ein erhebliches *busy-waiting* von ins-

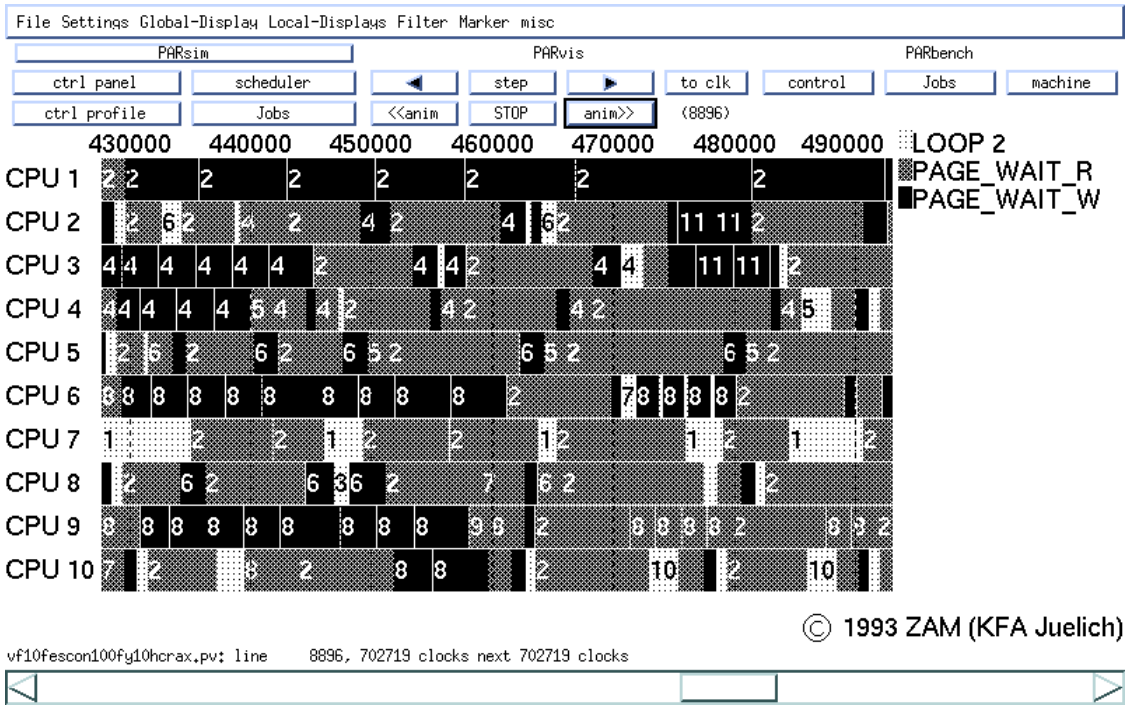


Abb. 47. Datenabbildung ohne Ausrichtung auf Seitengrenzen; page-thrashing: Zeitliniendarstellung der Prozeßzustände, Gaußelimination 100*100 (Datenzugriff spaltenweise), Seitengröße 128, 10 CPUs, spaltenorientierte Datenabbildung ohne Ausrichtung

gesamt über 300 Prozent relativ zur *user*-CPU-Zeit auftritt[15].
Die Abb. 48 beschreibt einen Zeitausschnitt für die Gaußelimination mit Ausrichtung der Vektoren auf Seitengrenzen; dargestellt sind die Prozeßzustände für drei Ausführungen der parallelen Schleife, d.h. für drei Iterationen der äußeren sequentiellen Schleife. Jede der Iterationen beginnt mit dem Zustand *page-waiting-read* für alle Prozessoren bezüglich derselben Seite, jedoch mit unterschiedlicher Dauer. Nach der Auflösung der *page faults* führen alle Prozessoren für ein jeweils gleichgroßes Zeitintervall im Zustand *user* (LOOP 2) einen Teil der parallelen Schleife aus. Die tatsächlich balancierte Verteilung der Schleifeniterationen führt durch die unterschiedlichen Wartezeiten für die Auflösung der *page faults* insgesamt zu einer unbalancierten Ausführung; dadurch ist die Synchronisation am Schleifenende mit *busy-waiting* vom Ausmaß der *page-fault*-Wartezeit verbunden.
Verursacht werden diese Effekte durch einen Kommunikationsengpaß: Innerhalb einer parallelen Schleifenausführung benötigen alle Prozesse jeweils dieselbe Seite für den Lesezugriff; gemäß Abb. 44.b handelt es sich um die Datenelemente *A(K,K)* und *A(J,K)* mit *J=K+1* bis *N*, also um die *K*-te Spalte der Matrix. In der Abb. 48 sind das die Seiten 5, 6 beziehungsweise 7 für jeweils eine der drei Schleifenausführungen. Da der aktuelle Besitzer der jeweiligen Seite die Anfragen für die Seite nur sequentiell bearbeiten kann, ergeben sich unterschiedliche Wartezeiten für die Auflösung der *page faults*. In diesem Zusammenhang kann eine *prefetch*-Operation auf die jeweilige Seite sinnvoll sein; dieser Gedanke wird an dieser Stelle jedoch nicht weiter verfolgt. Insgesamt er-

[15] Der in Abb. 47 dargestellte Detailausschnit enthält keine *busy-waiting*-Anteile, da diese nur in der Nähe von Synchronisationspunkten auftreten.

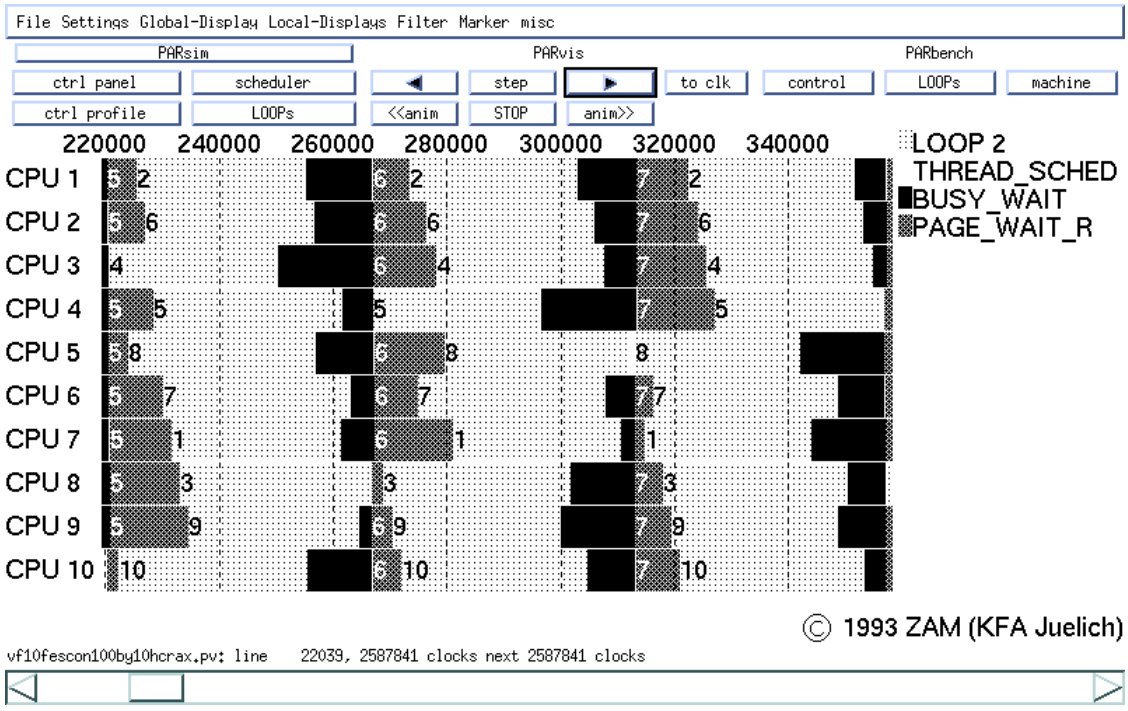


Abb. 48. Datenabbildung mit Ausrichtung auf Seitengrenzen; Kommunikationsengpaß: Zeitliniendarstellung der Prozeßzustände, Gaußelimination 100*100 (Datenzugriff spaltenweise), Seitengröße 128, 10 CPUs, spaltenorientierte Datenabbildung mit Ausrichtung

geben sich für die Ausführung der transformierten Gaußelimination mit Ausrichtung der Spaltenvektoren exakt die gleichen Verhältnisse wie für die Ausführung der originären Gaußelimination mit Ausrichtung der Zeilenvektoren.

Zusammenfassend können aufgrund der beschriebenen Untersuchungen folgende Aussagen für die Ablaufplanung in *virtual-shared-memory*-Rechnern gemacht werden: Eine Anpassung von Datenabbildung und Datenzugriff ist für die beschriebenen Untersuchungen aus Effizienzgründen unerläßlich. Aufgrund der gewonnenen Einsichten ist zu vermuten, daß diese Aussage auch für beliebige andere Anwendungen Gültigkeit behält. Das Ausmaß der Verringerung des relativen Overhead von über 10,000 Prozent auf 125 Prozent wird in vielen Fällen auch eine dynamische Anpassung der Datenabbildung rechtfertigen.
Die Ausrichtung von Vektoren auf Seitengrenzen hat in beiden betrachteten Anwendungen zu verbesserter Effizienz geführt: Bei der Matrixmultiplikation konnte der relative Overhead von 45.1 auf 19.7 Prozent reduziert werden. Durch eventuell erhöhte Kosten bei der Adreßberechnung aufgrund der expliziten Datenabbildung kann der tatsächliche Gewinn geringer ausfallen; damit ist ein Vorteil durch die Ausrichtung der Vektoren bei der betrachteten Matrixmultiplikation möglicherweise nur gering. Für die Gaußelimination ist die Verringerung des relativen Overhead von 872 auf 125 Prozent jedoch erheblich. Für diese Anwendung ist damit insgesamt eine deutliche Effizienzsteigerung zu erwarten. Der Nutzen der Ausrichtung von Vektoren auf Seitengrenzen hängt bei den beschriebenen Untersuchungen vom Speicherverhalten der jeweiligen Anwendung ab. Diese Abhängigkeit ist auch für beliebige andere Anwendungen zu erwarten. Ein

Speicherzugriffsmuster ähnlich dem der Matrixmultiplikation, bei dem nur jeweils zwei Prozesse auf benachbarte Vektoren zugreifen, läßt einen ebenfalls eher geringen Vorteil erwarten. Greifen jedoch wie bei der Gaußelimination zum Teil alle Prozesse auf einen Vektor zu, in dessen Nachbarschaft ebenfalls Datenzugriffe erfolgen, dann ist der Effizienzgewinn durch die Ausrichtung vermutlich sehr deutlich.

7.1.2. Fallstudien bezüglich der Seitengröße

Die bisher in diesem Kapitel durchgeführten Untersuchungen beziehen sich auf eine Seitengröße von 128 Worten und eine Vektorgröße von 100. Damit ergibt sich für die favorisierte Datenabbildung mit Ausrichtung der Vektoren jeweils ein Vektor pro Seite. Für größere Seiten mit jeweils mehr Vektoren pro Seite ergeben sich besondere Abhängigkeiten für das *thread*-Scheduling; dieser Fall wird in Abschnitt 7.2 betrachtet. Die negativen Effekte bei einer Datenabbildung ohne Ausrichtung der Vektoren gründen sich auf die Aufteilung von Seiten über verschiedene Vektoren. Wenn die Vektorgröße wie in den bisherigen Untersuchungen ungefähr gleich der Seitengröße ist, dann ergibt sich für nahezu alle Seiten eine Aufteilung. Ist das Verhältnis von Vektorgröße zu Seitengröße jedoch deutlich größer als eins, dann wird der Anteil der aufgeteilten Seiten kleiner sein. Um den Einfluß vom Verhältnis Vektorgröße zu Seitengröße auf die beobachteten Effekte zu studieren, wurde eine Meßreihe durchgeführt, bei der die Messungen aus dem vorigen Abschnitt für unterschiedliche Seitengrößen wiederholt wurden.

Die Abb. 49 zeigt die Messungen für die Matrixmultiplikation bei Seitengrößen von 8 bis 128 Worten; es sind jeweils die Messungen ohne Ausrichtung *(not aligned)* und mit Ausrichtung *(aligned)* dargestellt. Bereits bei einer Seitengröße von 64 Worten treten die vorher beobachteten Effekte nicht mehr auf. Tatsächlich ergeben sich sogar geringfügige Vorteile für die Datenabbildung ohne Ausrichtung. Der Vorteil gründet sich, wie schon in Abb. 42 beobachtet, auf den durch die Ausrichtung verschenkten Speicherplatz. Das *false sharing* tritt bei den Seitengrößen von 8, 16, 32 beziehungsweise 64 Worten nicht mehr auf: Es liegen zwar nach wie vor Teile benachbarter Vektoren auf einer Seite, so daß eine solche Seite von verschiedenen Prozessoren benötigt wird; da eine Seite jedoch kleiner ist als ein Vektor, ergibt sich durch die Abbildung keine Seite, die von zwei Vektoren die gleichen Indexbereiche enthält. Der Zugriff auf eine Seite durch verschiedene Prozessoren erfolgt damit nicht gleichzeitig, und es ergibt sich kein *page thrashing*. Mit abnehmender Seitengröße steigt der relative Overhead durch *page-waiting* für jeweils beide Messungen an. Da die Anzahl der Seiten mit abnehmender Seitengröße wächst, müssen die *startup*-Kosten pro *page fault* entsprechend häufiger bezahlt werden; deswegen steigt der *page-fault*-Overhead bei konstanter übertragener Datenmenge mit abnehmender Seitengröße.

Für die Gaußelimination sehen die Ergebnisse wiederum deutlich anders aus (siehe Abb. 50). Das Verschwinden der im vorigen Abschnitt beobachteten Effekte stellt sich erst bei sehr kleinen Seiten ein. Für eine Seitengröße von 64 Worten ist der Vorteil durch die Ausrichtung der Vektoren noch immer erheblich; gegenüber der Messung mit 128 Worten je Seite kann der relative Overhead nahezu um den Faktor 3 reduziert werden. Auch bei einer Seitengröße von 32 Worten ergibt sich eine Effizienzsteigerung

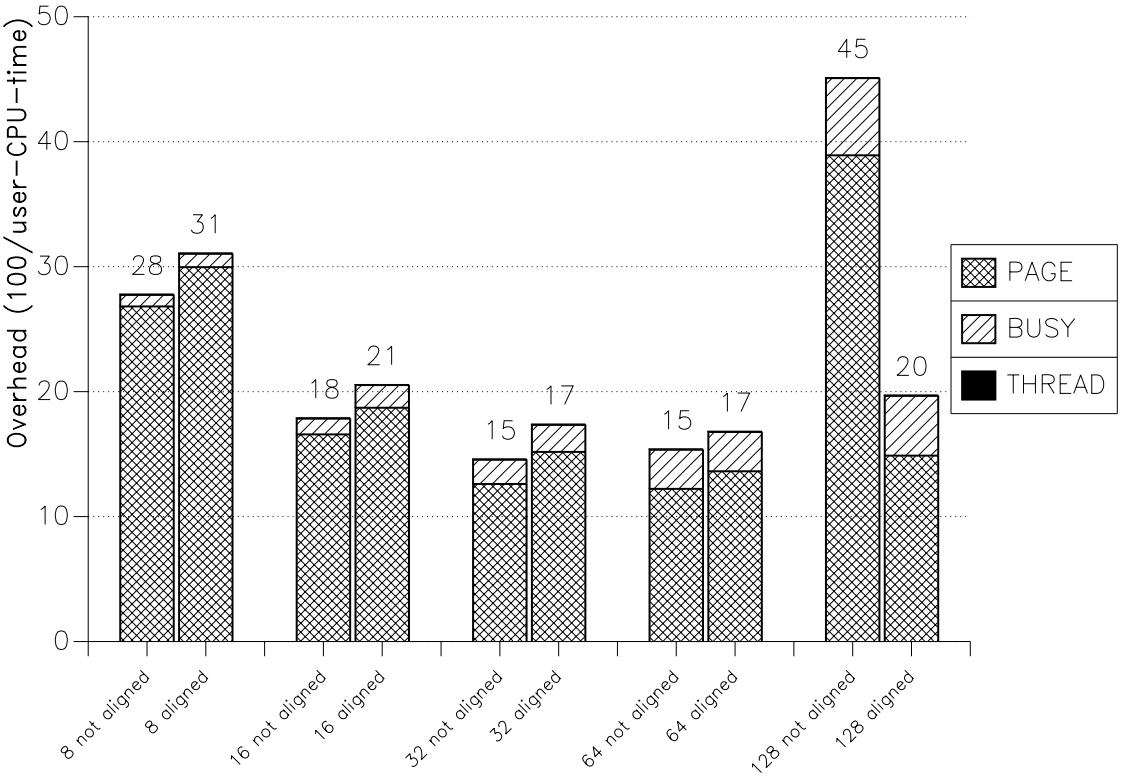


Abb. 49. Datenabbildung ohne und mit Ausrichtung auf Seitengrenzen bei variabler Seitengröße: Overhead bezogen auf die user-CPU-time in Prozent, Matrixmultiplikation 100*100 (Datenzugriff spaltenweise), Seitengröße 8 bis 128, 10 CPUs, *not aligned***: spaltenorientierte Abbildung ohne Ausrichtung,** *aligned***: spaltenorientierte Abbildung mit Ausrichtung**

durch die Ausrichtung. Erst für die Seitengrößen von 16 beziehungsweise 8 Worten verschwinden die Vorteile fast vollständig. Die Ursache für das deutlich andere Verhalten liegt wiederum im Speicherzugriffsmuster: Sowohl die lesenden Prozesse als auch der schreibende Prozeß durchlaufen innerhalb einer parallelen Schleifenausführung mehrmals den entsprechenden Spaltenvektor. Durch das Rundlaufen ergeben sich Überlappungen auch noch bei relativ kleinen Seiten. Bei diesem Effekt spielt möglicherweise nicht nur das Verhältnis von Vektorgröße zu Seitengröße eine Rolle, sondern auch die absolute Größe einer Seite.

Zusammenfassend lassen sich die folgenden Feststellungen machen: Die Relevanz der Ausrichtung von Vektoren bei der Datenabbildung hängt von dem Speicherzugriffsverhalten der jeweiligen Anwendung ab. Im ungünstigen Fall hat sich erst bei einem Verhältnis der Vektorgröße zur Seitengröße von etwa 10 das Zeitverhalten für die Ausführungen mit beziehungsweise ohne Ausrichtung der Vektoren angeglichen. Für Anwendungen mit relativ zur Seitengröße sehr langen Vektoren werden die ohne Ausrichtung beobachteten negativen Effekte vermutlich nicht auftreten. Damit wird die Ausrichtung von Vektoren auf Seitengrenzen für VSM-Rechner mit sehr kleinen Seiten von zum Beispiel 4 Worten (Cray T3D) ohne praktische Relevanz sein; dieses Faktum stellt somit ein Argument für die Realisierung von kleinen Seiten dar.
Bei kleinen Vektoren oder mittelgroßen Vektoren in Verbindung mit ähnlich großen Seiten

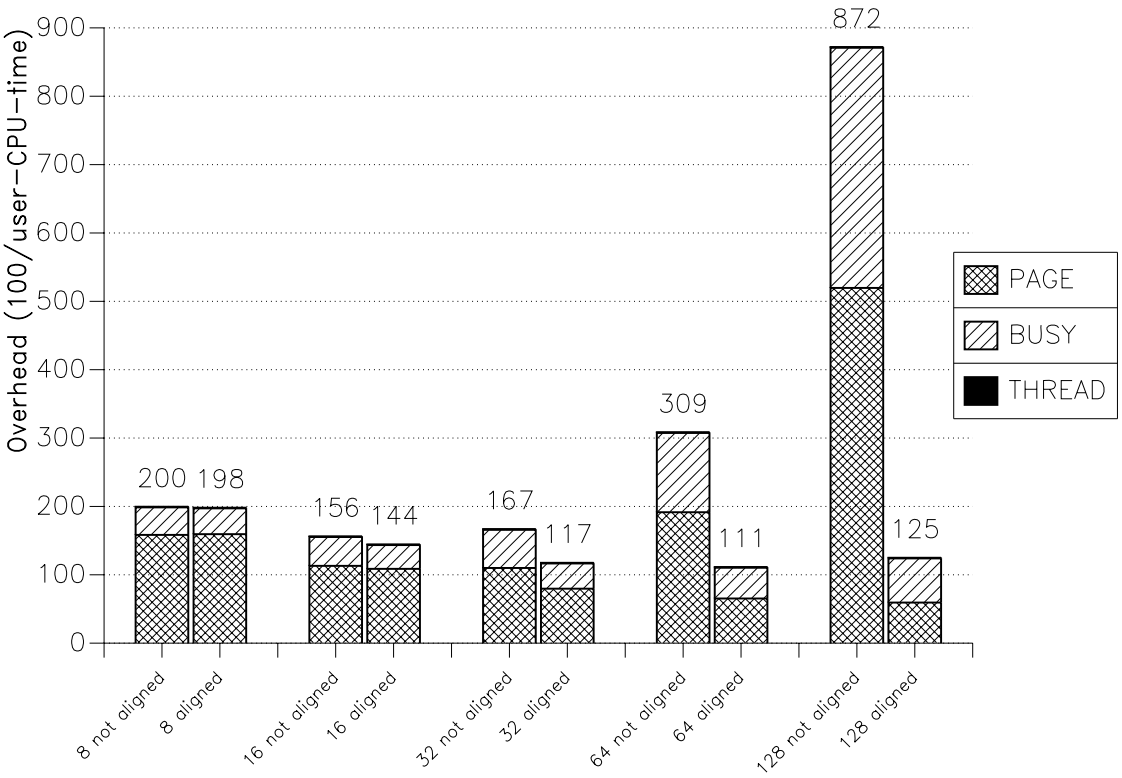


Abb. 50. Datenabbildung ohne und mit Ausrichtung auf Seitengrenzen bei variabler Seitengröße: Overhead bezogen auf die user-CPU-time in Prozent, Gaußelimination 100*100 (Datenzugriff spaltenweise), Seitengröße 8 bis 128, 10 CPUs, *not aligned***: spaltenorientierte Abbildung ohne Ausrichtung,** *aligned***: spaltenorientierte Abbildung mit Ausrichtung**

können jedoch für bestimmte Anwendungen erhebliche Vorteile durch die Ausrichtung der Vektoren erwartet werden. In der Praxis können Programme mit großen Vektorlängen auch Daten mit kurzen Vektoren beinhalten, und bei der Programmentwicklung werden üblicherweise auch Programmläufe für kleine Problemgrößen und damit für kurze Vektoren durchgeführt; um Rechner mit einer entsprechenden Seitengröße nicht auf Anwendungen mit sehr langen Vektoren zu beschränken, sollte hier die Ausrichtung der Vektoren auf Seitengrenzen implementiert sein.

7.2. Thread-Scheduling für Seiten mit mehreren Vektoren

Die Abhängigkeit der Ablaufplanung vom Verhältnis der Vektorgröße zur Seitengröße ist für die Fälle Vektorgröße ungefähr gleich Seitengröße und Vektorgröße größer als Seitengröße im vorigen Abschnitt untersucht worden. Für den Fall Vektorgröße kleiner als Seitengröße können sich Seiten mit mehreren Vektoren ergeben; eine solche Gruppierung von Vektoren stellt besondere Anforderungen an das *thread*-Scheduling und soll in diesem Abschnitt untersucht werden: Ein effizientes *thread*-Scheduling in *virtual-shared-memory*-Rechnern hängt in vielen Fällen ganz entscheidend von der jeweiligen Anwendung ab. Um diese Abhängigkeit für die folgenden Untersuchungen zu eliminieren, wird eine Anwendung gewählt, die keine besonderen Anforderungen an das *thread*-Scheduling stellt. Die Matrixmultiplikation erweist sich bei einer Parallelisierung

der äußeren Schleife als weitgehend indifferent bezüglich des *thread*-Scheduling, sofern jeweils nur ein Vektor auf eine Seite abgebildet wird. Mit der Wahl der Matrixmultiplikation kann eine von anderen Effekten weitgehend unabhängige Untersuchung für das *thread*-Scheduling mit Multi-Vektor-Seiten erfolgen.
Die Abb. 51 beschreibt den relativen Overhead für die Ausführung einer Matrixmultiplikation der Dimension 100 mit verschiedenen Scheduling-Algorithmen; die Seitengröße beträgt bei allen Messungen 1000 Worte, so daß jeweils genau 10 Vektoren auf einer Seite liegen. Um die auftretenden Effekte leichter einordnen zu können, liegen die beiden Ausgangsmatrizen auf allen Prozessorelementen vollständig vor; durch diese Vorgabe beziehen sich alle *page faults* ausschließlich auf die Produktmatrix. Es wurden sowohl lastadaptive als auch nicht-lastadaptive Scheduling-Algorithmen untersucht; der dargestellten Meßreihe liegt eine balancierte Arbeitslast zugrunde, so daß auch für die nicht-lastadaptiven Algorithmen eine balancierte Ausführung erzielt werden kann. Die untersuchten Scheduling-Algorithmen sind in der Abbildung gruppiert; links sind die Messungen für die nicht-lastadaptiven Algorithmen *Block-Scheduling*, *Cyclic-Scheduling* und *Align-Scheduling* dargestellt, rechts die Messungen für die lastadaptiven Algorithmen *Self-Scheduling*, *Chunk-Scheduling*, *Guided Self-Scheduling*, *Factoring* und *Factoring* mit *Loop-Blocking*. Für die nicht adaptiven Algorithmen ergeben sich nahezu optimale Verhältnisse: Der relative Overhead liegt in allen Fällen unter 0.5 Prozent; es ist je-

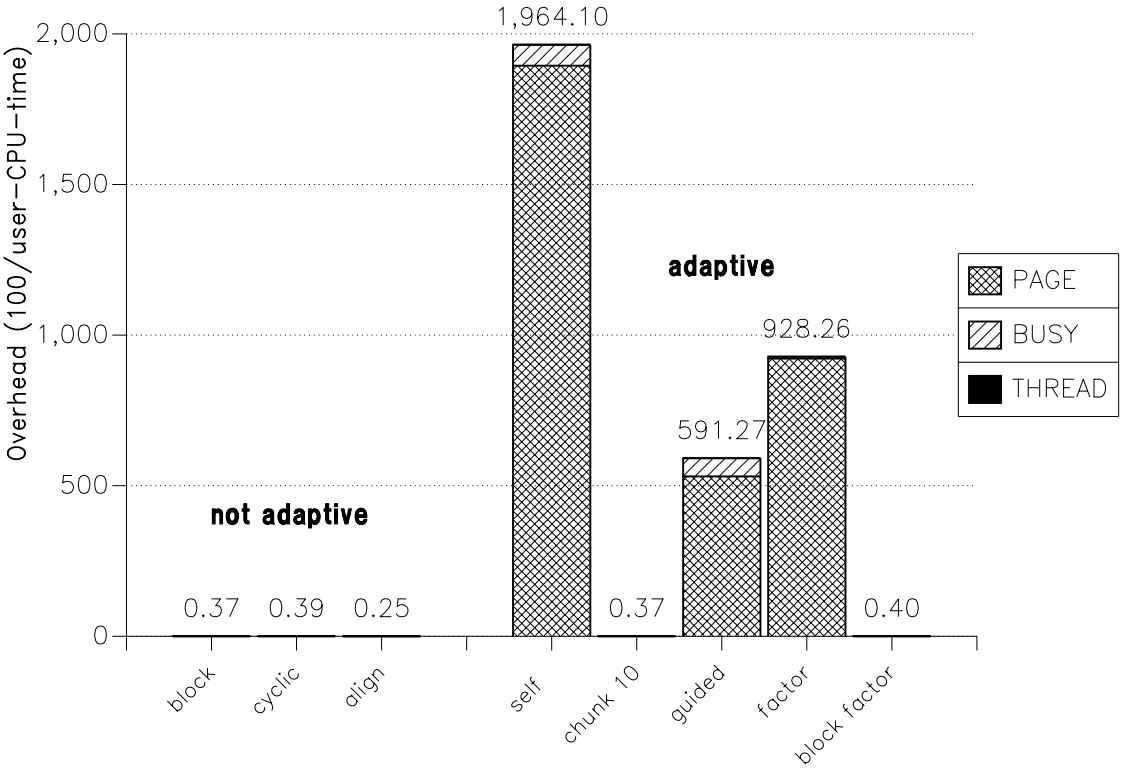


Abb. 51. Unterschiedliche Algorithmen zum thread-Scheduling bei balancierter Arbeitslast: Overhead bezogen auf die user-CPU-time in Prozent, Matrixmultiplikation 100*100, Seitengröße 1000 (10 Vektoren/Seite), 10 CPUs,
block, cyclic, align: **nicht-lastadaptiv,**
self, chunk, guided, factor, block factor: **lastadaptiv**

doch zu beachten, daß die Parametervorgaben für die Scheduling-Algorithmen optimal gewählt wurden:

- Für das *Block-Scheduling* ergibt sich aufgrund der Schleifenlänge von 100 und der Prozessorzahl von 10 eine Blockgröße von 10; dadurch erfolgt eine Gruppierung der Iterationen, die der Gruppierung der Vektoren durch die Seiten entspricht.
- Für das *Cyclic-Scheduling* wurde eine zyklische Datenabbildung gewählt; in Verbindung mit Schleifenlänge und Prozessorzahl ist die Gruppierung von Iterationen und Vektoren wiederum identisch.
- Das *Align-Scheduling* bezieht sich auf die Verteilung der Elemente der Produktmatrix; durch eine gleichmäßige Verteilung der Matrix wird eine lastbalancierte Ausführung erzielt.

Für die lastadaptiven Algorithmen ergibt sich zum Teil eine erheblich schlechtere Effizienz. Das *Self-Scheduling* schneidet mit Abstand am schlechtesten ab; mit fast 2,000 Prozent ist der relative Overhead in dieser Meßreihe maximal. Auch für das *Guided Self-Scheduling* und das *Factoring* ergeben sich mit etwa 600 und über 900 Prozent für den relativen Overhead erhebliche Effizienzverluste. Lediglich das *Chunk-Scheduling* und das *Factoring* mit *Loop-Blocking* zeigen die gleichen optimalen Ergebnisse wie die nicht adaptiven Algorithmen. Die Ursachen für diese Ergebnisse lassen sich wie folgt erklären:

- Beim *Self-Scheduling* werden die Schleifeniterationen unabhängig verteilt; jedem Prozeß wird immer nur eine Iteration ausgehändigt. Da benachbarte Iterationen, die somit meist auf verschiedenen Prozessoren ausgeführt werden, auf benachbarte Vektoren zugreifen, die jedoch meist auf derselben Seite liegen, ergeben sich ständig Zugriffskonflikte (*page thrashing*). Etwa 1800 Prozent - bezogen auf die *user*-CPU-Zeit - werden mit dem Zustand *page-waiting* belegt.
- Für das schlechte Abschneiden des *Guided Self-Scheduling* und des *Factoring* sind ähnliche Ursachen verantwortlich: Zu Beginn der Schleifenausführung werden *chunks* mit mehreren Iterationen vergeben; die Größe der *chunks* stimmt im allgemeinen jedoch nur selten mit der Anzahl der Vektoren pro Seite überein. Gegen Ende der Ausführung sind die beiden Algorithmen mit dem *Self-Scheduling* identisch; es wird nur noch jeweils eine Iteration pro Zuweisungsoperation verteilt. Insgesamt sind die Verhältnisse besser als beim *Self-Scheduling*, durch den hohen *page-waiting*-Anteil ist die Ausführung aber immer noch sehr ineffizient.
- Das *Chunk-Scheduling* führt zu nahezu optimaler Effizienz; auch hier spielen die Parametervorgaben eine entscheidende Rolle: Die Wahl der *chunk*-Größe 10 ist der Anzahl der Vektoren pro Seite angepaßt. Die Gruppierung von Iterationen und von Vektoren stimmt hier wieder überein.
- Das *Loop-Blocking* wurde speziell für das *thread*-Scheduling mit Multi-Vektor-Seiten entwickelt [GrWi93]. Der Algorithmus schreibt die Gruppierung der Iterationen in Äquivalenz zur Gruppierung der Vektoren vor. In Verbindung mit dem *Factoring* ergibt sich wiederum eine nahezu optimale Effizienz. Prinzipiell läßt sich mit dem *Loop-Blocking* für alle anderen adaptiven Algorithmen eine ebenso hohe Effizienz erzielen.

Die Anforderungen an das *thread*-Scheduling bei Multi-Vektor-Seiten lassen bewährte lastadaptive Algorithmen sehr schlecht abschneiden. Die beobachteten Verluste sind erheblich und liegen deutlich über den Verlusten durch schlechte Lastbalancierung, so daß auch für sehr schlecht balancierte Arbeitslasten keine wesentlich anderen Verhältnisse zu erwarten sind: Die möglichen Vorteile von lastadaptiven Algorithmen kommen natürlich nur bei unbalancierter Arbeitslast zum Tragen; eine schlechte Lastbalance kann zum Beispiel durch ein ungünstiges Zahlenverhältnis zwischen der Anzahl von Schleifeniterationen und der Prozessorzahl, durch eine große Varianz für die Ausführungszeit der Schleifeniterationen und durch eine variable Prozessorzahl - typischerweise im Multiprogramming-Betrieb - verursacht werden. Abb. 52 zeigt eine zweite Meßreihe für eine jetzt unbalancierte Arbeitslast: Es wird derselbe Programmkern jedoch auf nur 9 Prozessoren ausgeführt. Die schlechte Lastbalance wird in diesem Fall durch eine Nachbildung der Verhältnisse im Multiprogramming-Betrieb erzielt; während die Scheduling-Parameter bezüglich einer optimalen Lastbalance nach wie vor auf 10 Prozessoren eingestellt werden, ergibt sich durch nur 9 tatsächlich verfügbare Prozessoren eine sehr ungünstige Balancierung; damit sind die Vorgaben nach wie vor optimal bezüglich der Anzahl von 10 Vektoren pro Seite, und gleichzeitig ergibt sich eine stark unbalancierte Partitionierung. Die nicht adaptiven Algorithmen verteilen jeweils 10 *chunks* auf 9 Prozessoren. Für die nicht adaptiven Algorithmen ergibt sich durch die schlechte Lastbalance wie erwartet ein relativer Overhead von 80 Prozent, so daß die gesamte Ausführungszeit

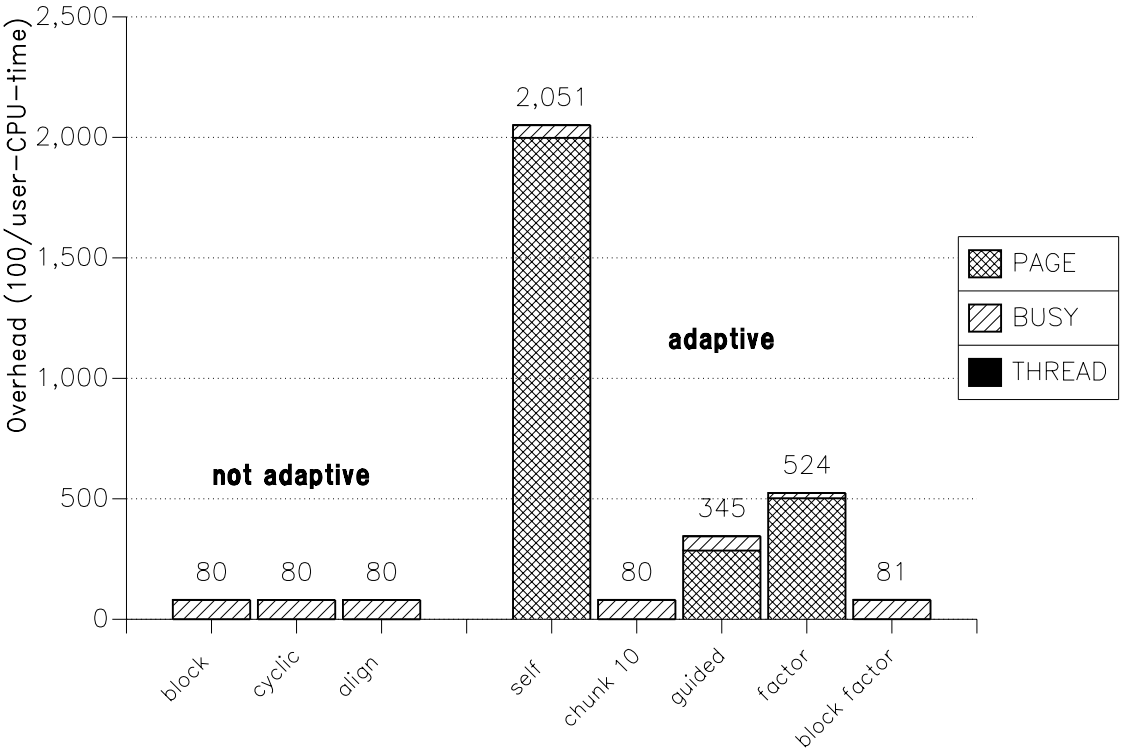


Abb. 52. Unterschiedliche Algorithmen zum thread-Scheduling bei unbalancierter Arbeitslast: Overhead bezogen auf die user-CPU-time in Prozent, Matrixmultiplikation 100*100, Seitengröße 1000 (10 Vektoren/Seite), 9 CPUs, Scheduling für 10 CPUs optimiert,
block, cyclic, align: **nicht-lastadaptiv,**
self, chunk, guided, factor, block factor: **lastadaptiv**

um den Faktor 1.8 ansteigt; dieser Effekt wird nahezu ausschließlich durch den erhöhten *busy-waiting*-Anteil verursacht. Das gleiche Verhalten zeigen auch das *Chunk-Scheduling* und das *Factoring* mit *Loop-Blocking*. Aufgrund der geringen Problemgröße und der nur grobgranularen Adaptionsmechanismen kommt eine Lastadaptierung in dieser Untersuchung nicht zum Tragen. Für günstigere Zahlenverhältnisse von Anzahl der Vektoren pro Seite, Schleifengröße und Prozessorzahl sind durch das *Chunk-Scheduling* und das *Loop-Blocking* jedoch Vorteile gegenüber den nicht adaptiven Algorithmen zu erwarten. Die im balancierten Fall ungünstigen Algorithmen führen jetzt zu folgenden Ergebnissen: Das *Self-Scheduling* schneidet im wesentlichen unverändert schlecht ab. Für das *Guided Self-Scheduling* und das *Factoring* sind die Ergebnisse etwas besser als vorher; insgesamt ist die Ausführung aber immer noch sehr ineffizient. Die Ursache für die leichte Verbesserung ist jeweils verschieden:

- Mit p = Anzahl der Prozesse ergibt die Partionierung der letzten p Schleifeniterationen beim *Guided Self-Scheduling* Partitionen von jeweils nur einer Iteration. Tatsächlich erhält bei der Messung mit 10 Prozessoren mit dem *Guided Self-Scheduling* jeder Prozeß eine dieser Iterationen; damit herrschen in diesem Bereich der Ausführung ähnlich ungünstige Verhältnisse vor wie beim *Self-Scheduling*: Alle Prozesse wollen gleichzeitig auf dieselbe Seite schreiben. Das daraus resultierende *page thrashing* bewirkt, daß etwa die Hälfte der gesamten Ausführungszeit auf die Ausführung der letzten 10 Iterationen der Schleife entfällt. Für das *Guided Self-Scheduling* bei nur 9 Prozessoren ergibt sich eine vollkommen andere Situation: Die geänderten Zahlenverhältnisse für das Scheduling ergeben, daß die letzten p Schleifeniterationen zwar wie vorher einzeln aber jetzt alle einem Prozeß zugewiesen werden; dadurch tritt das vorher beobachtete *page thrashing* hier nicht auf, und die Ausführung wird effizienter.
- Für das *Factoring* kann die Verbesserung folgendermaßen erklärt werden: Aufgrund der Partitionierung ergibt sich zu Beginn der Ausführung ein *page thrashing*, bei dem jeweils 2 Prozesse gleichzeitig auf eine Seite zugreifen wollen. Bei der Konfiguration mit 10 Prozessoren enstehen damit Zugriffskonflikte für insgesamt 5 Seiten, die jeweils von 2 Prozessen zugegriffen werden. Bei 9 Prozessoren entstehen diese Konflikte nur für 4 Seiten, dabei sind 8 Prozesse beteiligt, und der noch übrige Prozeß kann ohne Zugriffskonflikte allein auf eine Seite zugreifen. Dadurch erklärt sich die Verbesserung bei 9 Prozessoren.

Die Ursachen für die Verbesserungen lassen vermuten, daß die Effizienz der Ausführung mit geänderten Zahlenwerten für Prozessorzahl, Anzahl der Schleifeniterationen und Scheduling-Parameter scheinbar regellos schwankt; da die Ursachen nur auf Teilbereiche der Ausführung wirken können, ist in keinem Fall eine Ausführung mit hoher Effizienz zu erwarten.

Zusammenfassend können folgende Aussagen für das *thread*-Scheduling mit Multi-Vektor-Seiten gemacht werden: Die lastadaptiven Algorithmen *Self-Scheduling*, *Guided Self-Scheduling* und *Factoring* sind in der bisher üblichen Form unbrauchbar. Auch für stark unbalancierte Arbeitslasten sind die nicht adaptiven Algorithmen bei geeigneten Parametervorgaben deutlich überlegen. Das *Chunk-Scheduling* erweist sich bei geeigne-

ter Wahl der *chunk*-Größe, die ein Vielfaches der Anzahl von Vektoren pro Seite sein muß, als ein geeigneter lastadaptiver Algorithmus. Eine Anpassung der übrigen lastadaptiven Algorithmen an die Anforderungen durch Multi-Vektor-Seiten kann durch das *Loop-Blocking* erfolgen. Eine Lastbalancierung erfolgt dann jedoch mit einer gröberen Granularität. Die Granularität wird bestimmt durch die Anzahl der Vektoren pro Seite.
An dieser Stelle soll eine Bestimmung der Relevanz von Multi-Vektor-Seiten erfolgen; es gelten hier ähnliche Aussagen wie für die Ausrichtung von Vektoren auf Seitengrenzen: Zunächst läßt sich eine Einschränkung bezüglich der Seitengröße des Rechners vornehmen. Für Rechner mit sehr kleinen Seiten von zum Beispiel 4 Worten (Cray T3D) werden Multi-Vektor-Seiten kaum vorkommen. Bei Rechnern mit mittelgroßen und großen Seiten sind die Aussagen in diesem Abschnitt dagegen von Relevanz: Für relativ zur Seitengröße lange Vektoren treten keine Multi-Vektor-Seiten auf. Anwendungen mit sehr langen Vektoren sind im allgemeinen auch sehr rechenintensiv und damit für *virtual-shared-memory*-Rechner nicht untypisch. Für solche Anwendungen werden bei der Programmentwicklung zu Testzwecken allerdings auch Versionen mit kurzen Vektoren ausgeführt. Außerdem können diese Anwendungen neben Datenfeldern mit langen Vektoren auch Datenfelder mit kurzen Vektoren enthalten. Um einen VSM-Rechner nicht von vorneherein auf lange Vektoren zu beschränken, können die in diesem Abschnitt propagierten Algorithmen genutzt werden.

7.3. Einfache Affinitäten in Schleifennestern

Während die Betrachtungen in den bisherigen Abschnitten für verschiedene Arbeitslasten durchgeführt worden sind, soll bei den folgenden Untersuchungen jeweils eine Klasse von Anwendungen im Vordergrund stehen und anhand von charakteristischen Beispielanwendungen untersucht werden. In diesem Abschnitt wird zunächst eine Klasse von Anwendungen betrachtet, die durch einfache Affinitätsbeziehungen beim Datenzugriff ein besonderes *thread*-Scheduling erfordert. Gemeint sind Schleifennester mit Affinität gemäß der Abb. 53. Durch die mehrfache Ausführung einer parallelen Schleife wird von parallelen Prozessen auf dieselben Daten mehrfach zugegriffen. Herkömmliche Algorithmen zum *thread*-Scheduling beachten diese Affinitäten nicht. Wie die Abb. 54 zeigt, führen die Algorithmen *Block-Scheduling*, *Self-Scheduling*, *Chunk-Scheduling*, *Guided Self-Scheduling* und *Factoring* durch erhebliches *page-waiting* zu einer ineffizienten Ausführung. Durch das *page-waiting* ergibt sich darüber hinaus eine unbalancierte Ausführung, so daß zum Teil auch der *busy-waiting*-Anteil erheblich ist. Der durch *page-waiting* und *busy-waiting* verursachte relative Overhead liegt in dem betrachteten Beispiel zwischen 99 und 143

Abb. 53. Affinität in geschachtelten Schleifen: Parallele Schleifen innerhalb sequentieller Schleifen

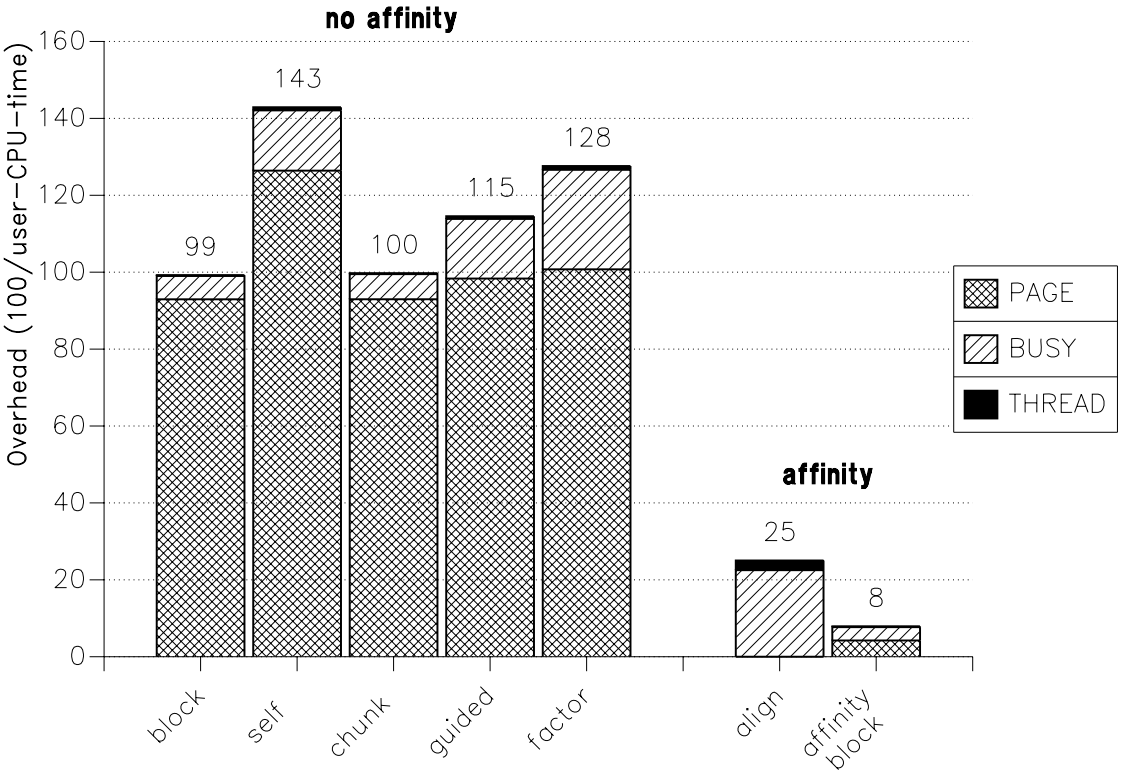


Abb. 54. Unterschiedliche Algorithmen zum thread-Scheduling bei einfachen Affinitäten: Overhead bezogen auf die user-CPU-time in Prozent, Schleifennest mit Affinität 20*100, Seitengröße 128, 10 CPUs, *block, self, chunk, guided, factor***: ohne Beachtung von Affinitäten,** *align, affinity block***: mit Beachtung von Affinitäten**

Prozent. Für das *Affinity-Scheduling* ergeben sich wie erwartet nahezu optimale Verhältnisse. Der geringe *page-waiting*-Anteil wird ausschließlich durch die Verteilung der Seiten während der ersten Ausführung der parallelen Schleife erzeugt; für alle weiteren Ausführungen der parallelen Schleife treten keine weiteren *page faults* auf. Für das *Align-Scheduling* sind die Verhältnisse etwas ungünstiger: Obwohl hier im Vergleich zum *Affinity-Scheduling* das *page-waiting* zu Beginn der Ausführung wegfällt und damit kein einziger *page fault* auftritt, ergibt sich insgesamt ein höherer relativer Overhead. Die Ursache liegt im vergleichsweise höheren Overhead für das *thread*-Scheduling. Die den Simulationen zugrundeliegenden Kosten für das *Align-Scheduling* liegen bei 6700 Maschinentakten für den Eintritt in eine parallele Schleife (*startup*) und bei 70 Takten für jede *dispatch*-Operation. Im Vergleich dazu liegen die veranschlagten Kosten für das *Affinity-Scheduling* mit 330 Takten für den *startup* und 20 Takten für jede *dispatch*-Operation insgesamt deutlich niedriger. Da der *thread*-Scheduler innerhalb einer parallelen Schleife nur exklusiv vergeben wird, müssen alle noch nicht von dieser Instanz bedienten Prozesse warten; damit erklärt sich der gegenüber dem *Affinity-Scheduling* deutlich erhöhte *busy-waiting*-Anteil.

Aufgrund der in diesem Abschnitt beschriebenen Untersuchungen kann geschlossen werden, daß die beobachteten Effekte auch für andere Anwendungen dieser Klasse gültig sind, d.h. das *Affinity-Scheduling* realisiert das effizienteste Scheduling. Ein in dieser

Untersuchung nicht betrachteter Nebeneffekt kann sich durch eine Limitierung des Lokalspeichers ergeben: Wenn die während der ersten Ausführung der parallelen Schleife jeweils lokal bereitgestellten Seiten aufgrund von Speicherengpässen aus dem Lokalspeicher des jeweiligen Prozessors verdrängt werden, dann können die bestehenden Affinitätsbeziehungen teilweise oder ganz verlorengehen, und die für das *Affinity-Scheduling* beobachteten Vorteile können unter Umständen vollständig verschwinden. In diesem Fall kann sich das *Align-Scheduling* als vorteilhaft erweisen.

7.4. Affinität in triangularen Schleifennestern

Die Affinitätsbeziehungen in Schleifennestern stellen sich nicht immer in der einfachen Form wie vorigen Abschnitt dar. Im folgenden werden triangulare Schleifennester mit Affinität am Beispiel der Gaußelimination betrachtet; dabei haben Datenabbildung und Datenzugriff die gleiche Orientierung. Die Abb. 55 beschreibt den relativen Overhead für die Anwendung verschiedener Scheduling-Algorithmen: Die Algorithmen ohne Beachtung von Affinitäten (*Block-Scheduling*, *Cyclic-Scheduling*, *Self-Scheduling*, *Chunk-Scheduling*, *Guided Self-Scheduling* und *Factoring*) führen zu erheblichem *page-waiting*

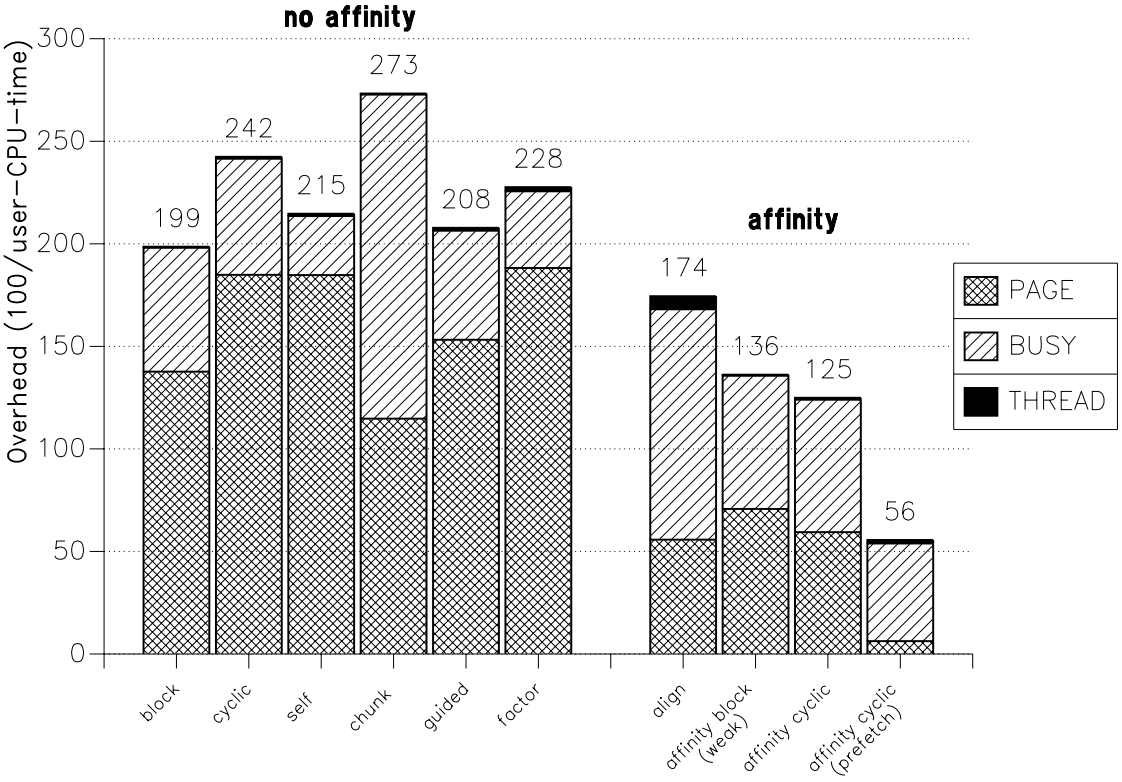


Abb. 55. Unterschiedliche Algorithmen zum thread-Scheduling bei Affinitäten in triangularem Schleifennest:
Overhead bezogen auf die user-CPU-time in Prozent,
Gaußelimination 100*100, Seitengröße 128, 10 CPUs,
block, cyclic, self, chunk, guided, factor: **ohne Beachtung von Affinitäten,**
align, affinity block (weak), affinity cyclic, affinity cyclic (prefetch): **mit Beachtung von Affinitäten**

und, dadurch bedingt, auch zu erheblichem *busy-waiting*; der gesamte relative Overhead wird nahezu ausschließlich durch diese beiden Anteile bestimmt und liegt zwischen 199 und 273 Prozent. Die Algorithmen mit Beachtung von Affintäten bewirken eine günstigere Ausführung; insbesondere der *page-waiting*-Anteil ist jeweils deutlich reduziert. Für das *Align-Scheduling* ergibt sich bezüglich der Algorithmen mit Beachtung von Affinitäten das vergleichsweise schlechteste Ergebnis. Die Ursache dafür liegt an den erhöhten Kosten für das *thread*-Scheduling, die wiederum ein erhöhtes *busy-waiting* zur Folge haben.

Eine effizientere Ausführung wird durch das *Affinity-Scheduling* erzielt; es sind die beiden Algorithmen *Affinity Block-Scheduling* mit abgeschwächter Affinitätsforderung *(affinity block weak)* und das *Affinity Cyclic-Scheduling* untersucht worden. Das *Affinity Cyclic-Scheduling* führt aufgrund des geringeren *page-waiting*-Anteils zu einer etwas effizienteren Ausführung. Die Abb. 56 verdeutlicht die Ursache dieses Effekts für die Gaußelimination. Die Darstellungen beziehen sich auf eine Problemgröße von *N=16*; es werden jeweils die ersten 4 Iterationen der äußeren Schleife betrachtet (*K=1* bis *4*). Abb. 56.a zeigt die Partitionierung für das *Affinity Block-Scheduling* mit abgeschwächter Affinitätsforderung: Für *K=1* werden dem Prozeß 1 die Iterationen *I=2* bis *5* zugeteilt, der Prozeß 2 erhält die Iterationen *I=6* bis *9*, usw. Die Partitionierung für *K=2* ergibt, daß zum Beispiel die Iteration *I=6* nicht wie zuvor dem Prozeß 2 zugeordnet wird, sondern

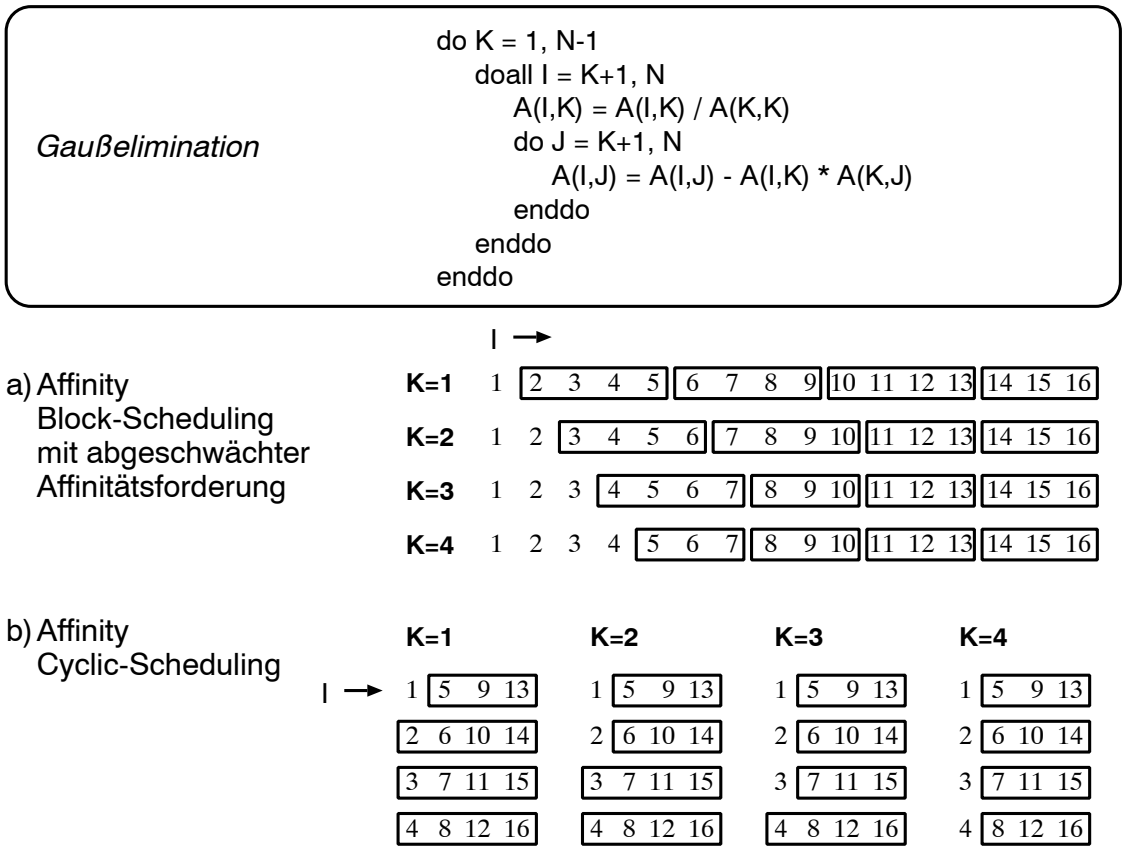


Abb. 56. Affinity-Scheduling am Beispiel der Gaußelimination:
a) Affinity Block-Scheduling mit abgeschwächter Affinitätsforderung
b) Affinity Cyclic-Scheduling

dem Prozeß 1. Eine solche Verschiebung von Iterationen bewirkt, daß die entsprechenden Daten transferiert werden müssen. Für das *Affinity Cyclic-Scheduling* treten keine Verschiebungen auf (siehe Abb. 56.b). Daraus erklärt sich der geringere *page-waiting*-Anteil für diesen Algorithmus. Tatsächlich wird der relative Overhead in diesem Beispiel von 136 auf 125 Prozent reduziert.

Für alle drei betrachteten Algorithmen mit Beachtung der Affinitäten ergibt sich ein zunächst unerwartet hoher *page-waiting*-Anteil (siehe Abb. 55). Die Ursache ist bei allen drei Algorithmen identisch und soll am Beispiel des *Affinity Cyclic-Scheduling* erläutert werden. Die Abb. 57.a zeigt einen Ausschnitt der Prozeßzustandsfolge; dargestellt sind zwei Ausführungen der parallelen Schleife. Wie schon in Abschnitt 7.1.1 beobachtet, ergibt sich durch die Anforderung der jeweils selben Seite durch alle Prozessoren ein Kommunikationsengpaß; das führt zu vergleichsweise langen Wartezeiten bei der Auflösung der *page faults* und, dadurch bedingt, zu einer unbalancierten Ausführung mit erheblichem *busy-waiting*.

Für diese Problemstellung soll der Einsatz von *prefetch*-Operationen untersucht werden. Um den Kommunikationsengpaß zu beheben, wird an geeigneter Stelle eine *prefetch*-Operation eingefügt: Auf die jeweils von allen Prozessoren benötigte Seite wird in der jeweils vorhergehenden Ausführung der parallelen Schleife geschrieben. Dieser Schreibzugriff erfolgt in der ersten Iteration *I=K+1* der parallelen Schleife. Um in allen Prozessen nach Beendigung der Iteration *I=K+1* eine *prefetch*-Operation ausführen zu können, wird diese Iteration vor die parallele Schleife gezogen. Auf diese Weise ergibt sich ein sequentieller Anteil, nach dessen Ausführung die *prefetch*-Operationen erfolgen. Die Abb. 57.b zeigt die Prozeßzustandsfolge für die entsprechende Simulation: Die *page faults* sind im dargestellten Zeitintervall vollständig eliminiert. Die jetzt teilweise sequentielle Ausführung resultiert natürlich in zusätzlichem *busy-waiting* für die jeweils übrigen Prozessoren. Die Summe der Effekte führt aber zu deutlichen Vorteilen für die Ausführung mit *prefetch*-Operationen (siehe Abb. 55). Der *page-waiting*-Anteil ist fast vollständig eliminiert, und auch der *busy-waiting*-Anteil ist insgesamt reduziert. Mit einem relativen Overhead von 56 Prozent ergibt sich die insgesamt effizienteste Ausführung.

Mithilfe der *prefetch*-Operation konnte die Effizienz für die Ausführung der Gaußelimination deutlich gesteigert werden. Es ist jedoch zu erwarten, daß dieser positive Effekt nicht skaliert: Mit einer erhöhten Prozessorzahl steigt auch der Zeitbedarf für die Bereitstellung der jeweiligen Seite auf allen Prozessoren, d.h. die zur Durchführung der *prefetch*-Operationen benötigte Zeit skaliert mit der Prozessorzahl. Wenn die Problemgröße mit der Prozessorzahl in der Weise skaliert, daß die Granularität konstant bleibt, dann ist auch die für die Ausführung der *prefetch*-Operationen zur Verfügung stehende Zeit konstant und damit nicht ausreichend zur Durchführung aller *prefetch*-Operationen. Abb. 58 beschreibt die Ausführung einer Gaußelimination mit doppelter Problemgröße, d.h. der Dimension 200, und mit 40 Prozessoren; für diese Konstellation ergibt sich die gleiche Granularität wie beim zuvor betrachteten Fall für die Dimension 100 und 10 Prozessoren. Wie die Abbildung zeigt, können in der zur Verfügung stehenden Zeit nicht alle *prefetch*-Operationen durchgeführt werden: Im dargestellten Zeitintervall ergeben sich auf einem Teil der Prozessoren *page faults* zum Beispiel für die Seite 4. Auf allen

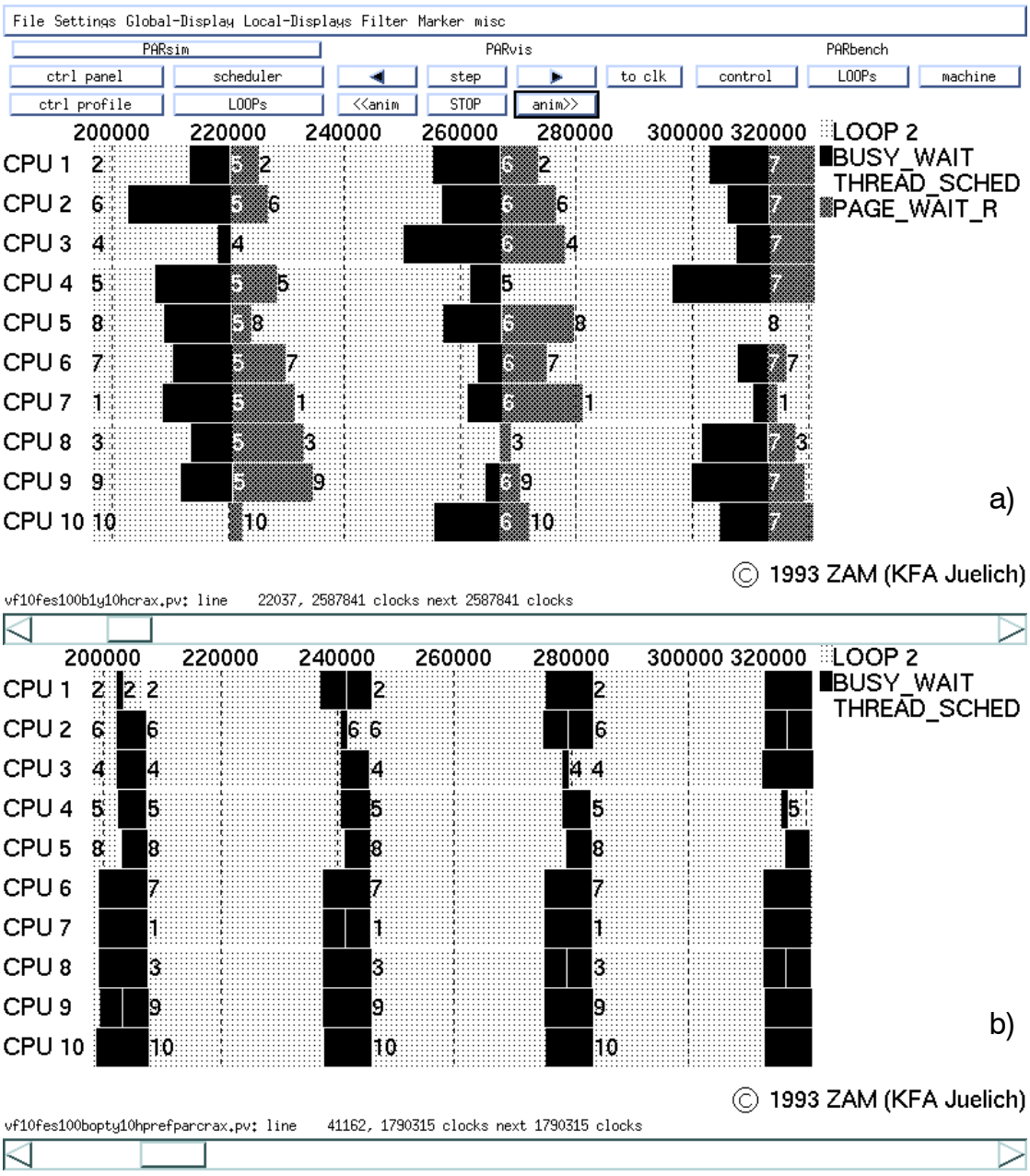


Abb. 57. Gaußelimination bei Affinity Cyclic-Scheduling: Zeitliniendarstellung der Prozeßzustände, Gaußelimination 100*100, Seitengröße 128, 10 CPUs, a) ohne *prefetch*: Kommunikationsengpaß b) mit *prefetch*

anderen Prozessoren konnte die Seite 4 bereits durch *prefetch*-Operationen bereitgestellt werden.

Die Untersuchungen haben gezeigt, daß ein zyklisches *Affinity-Scheduling* für triangulare Schleifennester mit Affinität die vergleichsweise beste Effizienz erzielt. Da dieses Ergebnis im Widerspruch zu Ergebnissen in [LaPr91] steht, sei hier nochmal erwähnt, daß das *Cyclic-Scheduling* nur in Verbindung mit einer geeigneten und damit ebenfalls zyklischen Datenabbildung zu einem günstigeren Verhalten führt als das *Block-Scheduling*. Auch

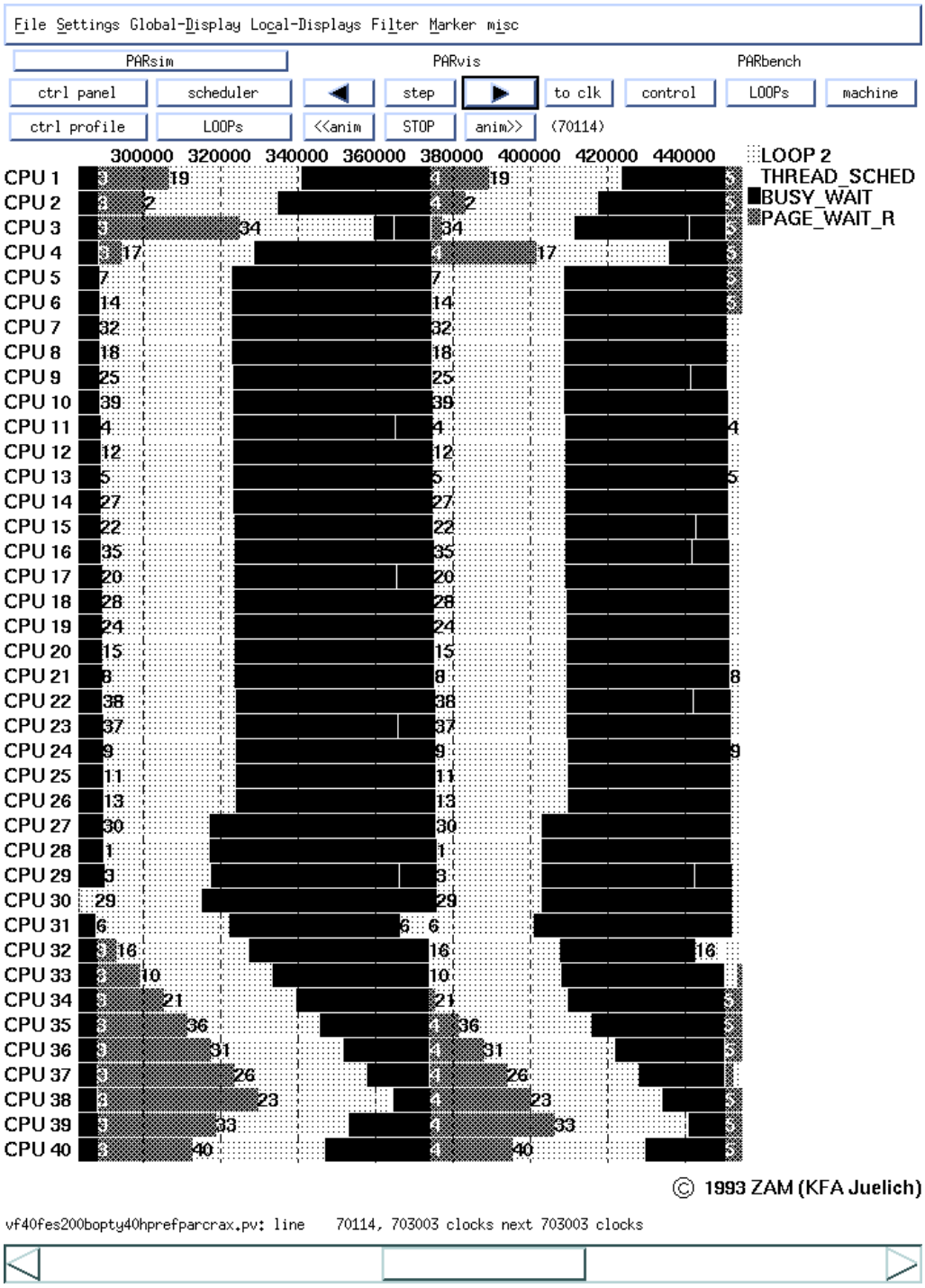


Abb. 58. Gaußelimination bei Affinity Cyclic-Scheduling; Kommunikationsengpaß trotz prefetch-Operation: Zeitliniendarstellung der Prozeßzustände, Gaußelimination 200*200 mit prefetch-Operation, Seitengröße 128, 40 CPUs

für den günstigsten Scheduling-Algorithmus ergab sich jedoch zunächst keine effiziente Ausführung; das charakteristische Verhalten der beispielhaft untersuchten Gaußelimination, d.h. das Schreiben einer Seite durch einen Prozeß mit darauffolgenden Leseanforderungen für diese Seite durch alle übrigen Prozesse, erweist sich auf *virtual-shared-*

memory-Rechnern als problematisch. Für die Konstellation mit 10 Prozessoren konnte der Kommunikationsengpaß mithilfe von *prefetch*-Operationen zwar weitgehend ausgeräumt werden, so daß sich eine effiziente Ausführung ergab; anhand einer Konstellation mit 40 Prozessoren wurde jedoch nachgewiesen, daß dieses günstige Verhalten nicht skaliert.
Aus diesen Ergebnissen kann gefolgert werden, daß Anwendungen mit dem beschriebenen Verhalten für den zugrundegelegten VSM-Rechner ungeeignet sind; diese Aussage gilt jedoch bei gleichwertiger Kommunikationsstruktur für alle Rechner mit physikalisch verteiltem Speicher: In *message-passing*-Systemen ergibt sich die äquivalente Problemstellung für eine *broadcast*-Operation. In bestimmten Fällen kann eine Verbesserung auf algorithmischer Ebene erzielt werden: Wenn das Schreiben auf die Seite auf allen Prozessoren redundant ausgeführt werden kann, dann entfällt die Kommunikation. Anderenfalls kann eine Verbesserung zum Beispiel durch zusätzliche Hardware erzielt werden: Eine hierarchische Kommunikations-Hardware kann die Kosten für das Verschicken einer Seite an alle Prozessoren von linearer Ordnung auf logarithmische Ordnung reduzieren.

7.5. Schleifennester mit Indexadressierung

In den beiden vorigen Abschnitten konnte ein effizientes *thread*-Scheduling durch die Beachtung von Affinitäten beim Datenzugriff erzielt werden. Wenn solche Affinitäten nicht bestehen, dann kann sich das *thread*-Scheduling wesentlich problematischer darstellen. Die Abb. 59 beschreibt vier Typen von Schleifennestern ohne Affinität: Der Datenzugriff innerhalb der parallelen Schleife erfolgt indiziert, und für die wiederholte Ausführung der parallelen Schleife ändert sich das Indexfeld *(Update-Index)*. Dadurch existiert keine feste Zuordnung von Iterationen und Daten. Die vier beschriebenen Typen unterscheiden sich wie folgt: Wenn alle Datenzugriffe über dasselbe Indexfeld indiziert sind, dann handelt es sich um eine *Single-Index*-Schleife. Mit der weiteren Unterscheidung zwischen Lese- beziehungsweise Schreibzugriffen ergeben sich die beiden Schleifentypen *Single Index Read* (SIR) beziehungsweise *Single Index Write* (SIW). Erfolgt die Indizierung über zwei oder mehr Indexfelder, dann ergeben sich die beiden weiteren Schleifentypen *Multiple Index Read* (MIR) und *Mutliple Index Write* (MIW). Für die vier betrachteten Schleifentypen wird vorausgesetzt, daß alle sonstigen Datenzugriffe lokal sind. Die Abb. 60 beschreibt den relativen Overhead für ein SIW-Schleifennest bei unterschiedlichen Scheduling-Algorithmen. Die Algorithmen *Block-Scheduling*, *Self-Scheduling*, *Chunk-Scheduling*, *Guided Self-Scheduling* und *Factoring* resultieren in erheblichem *page-waiting* und damit in einer insgesamt sehr ineffizienten Ausführung; der relative Overhead, der im wesentlichen durch *page-waiting* und *busy-waiting* verursacht wird, liegt zwischen 94 und 108 Prozent. Auch für das *Affinity Block-Scheduling* sind die Verhältnisse mit 75 Prozent relativem Overhead nur wenig besser. Die höchste Effizienz wird durch das *Align-Scheduling* erzielt: Das *page-waiting* ist vollständig eliminiert; der vergleichsweise hohe Scheduling-Overhead und das damit verbundene *busy-waiting* bestimmen den gesamten realtiven Overhead von 32 Prozent.
Die Ursache für diese Effekte ist offensichtlich. Der Wechsel von Indexfeld und Zuordnung Iteration zu Prozeß bei aufeinanderfolgenden Ausführungen der parallelen Schleife führt für die Algorithmen ohne *alignment* sehr häufig zu *page faults*. Für die Algorithmen ohne Beachtung von Affinitäten führen jeweils etwa 90% der Datenzugriffe zu

Single Index Read (SIR)

```
do K=1, N
   doall I=1, M
      ... = A(Index(I)) ... B(Index(I))
   enddo
   Update_Index
enddo
```

Single Index Write (SIW)

```
do K=1, N
   doall I=1, M
      A(Index(I)) = ...
      B(Index(I)) = ...
   enddo
   Update_Index
enddo
```

Multiple Index Read (MIR)

```
do K=1, N
   doall I=1, M
      ... = A(Index1(I)) ... B(Index2(I))
   enddo
   Update_Index
enddo
```

Multiple Index Write (MIW)

```
do K=1, N
   doall I=1, M
      A(Index1(I)) = ...
      B(Index2(I)) = ...
   enddo
   Update_Index
enddo
```

Abb. 59. Variable Indexadressierung:
Single Index Read (SIR), Single Index Write (SIW),
Multiple Index Read (MIR), Multiple Index Write (MIW)

einem *page fault*; beim *Affinity Block-Scheduling* resultieren etwa 50% der Datenzugriffe in einem *page fault*. Das günstigere Abschneiden des *Affinity-Scheduling* begründet sich in dem Zufallsprozeß für die Änderung des Indexfeldes; tatsächlich ändert sich das Indexfeld nicht in jedem Fall, so daß die Beachtung dieser Affinität vorteilhaft ist. Für andere Zufallsprozesse kann der Vorteil für das *Affinity-Scheduling* jedoch ebensogut verschwinden. Das *Align-Scheduling* bezieht sich auf eines der beiden Datenfelder. Alle Datenelemente mit gleichem Index befinden sich auf demselben Prozessorelement. Auf diese Weise ergibt sich durch das *alignment* eine vollständig lokale Ausführung. Der relativ zur Ausführungszeit hohe *busy-waiting*-Anteil ergibt sich aussschließlich aufgrund der hohen Kosten für das *Align-Scheduling*.

Für das untersuchte SIR-Schleifennest ergeben sich qualitativ die gleichen Verhältnisse wie für das SIW-Schleifennest. Bei Scheduling-Algorithmen ohne *alignment* sind die *page-waiting*-Anteile vergleichsweise etwas reduziert. Die Ursache dafür liegt im unterschiedlichen Verhalten bezüglich *write* beziehungsweise *read page faults*: Während bei wiederholter Ausführung der parallelen Schleife mehrere *write page faults* für jede Seite auftreten können, ereignet sich aufgrund des unbegrenzten Lokalspeichers jeweils nur ein *read page fault* für jede Seite.

Die Abb. 61 beschreibt den realtiven Oberhead für ein MIW-Schleifennest bei unterschiedlichen Scheduling-Algorithmen. Die Algorithmen *Block-Scheduling*, *Self-*

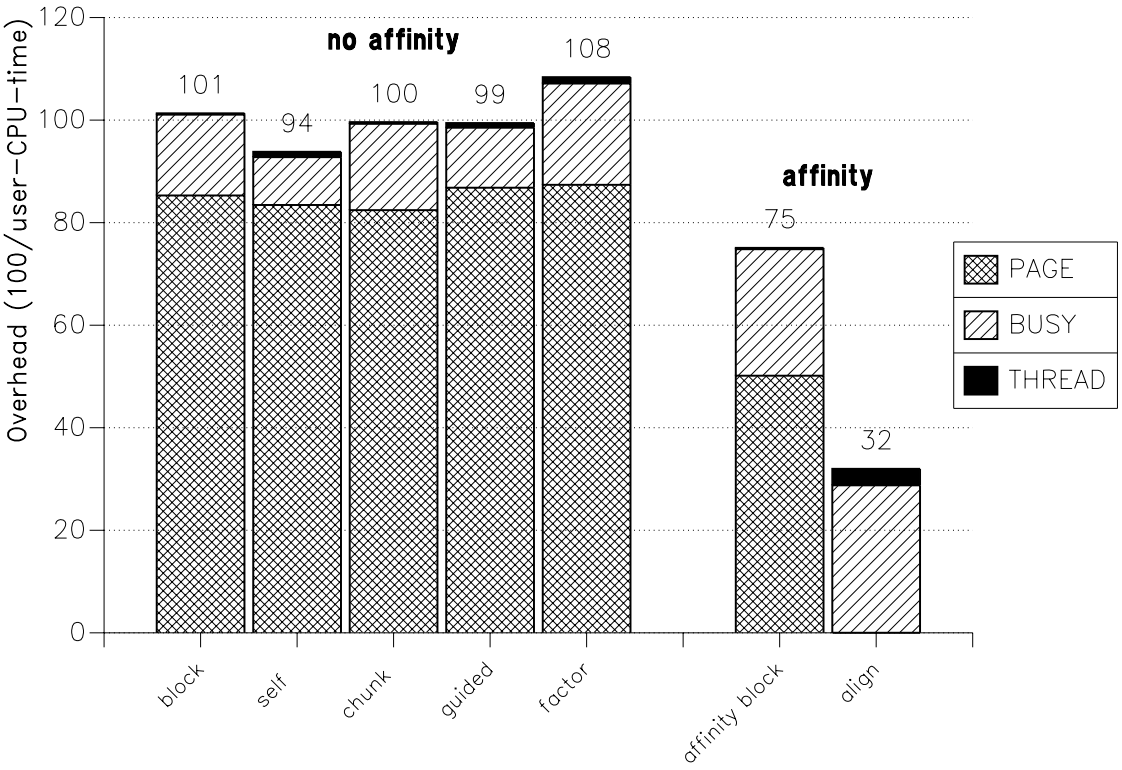


Abb. 60. Unterschiedliche Algorithmen zum thread-Scheduling bei einfacher, variabler Indexadressierung:
Overhead bezogen auf die user-CPU-time in Prozent,
SIW-Schleife (Single Index Write), Seitengröße 128, 10 CPUs,
block, self, chunk, guided, factor: **ohne Beachtung von Affinitäten,**
affinity block, align: **mit Beachtung von Affinitäten**

Scheduling, *Chunk-Scheduling*, *Guided Self-Scheduling* und *Factoring* resultieren wiederum in einer sehr ineffizienten Ausführung; der relative Overhead durch *page-waiting* und *busy-waiting* ist im Vergleich zum SIW-Scheifennest noch erhöht und liegt zwischen 109 und 118 Prozent. Das *Affinity Block-Scheduling* liefert mit etwa 86 Prozent relativem Overhead ein wiederum nur geringfügig besseres Ergebnis. Für das *Align-Scheduling* ergeben sich bei diesem Schleifennest keine deutlichen Vorteile mehr; der relative Overhead liegt bei 72 Prozent. Die Ursache für das schlechte Abschneiden des *Align-Scheduling* ergibt sich wie folgt: Das *alignment* kann sich nur auf ein Datenfeld beziehen. Durch die Existenz verschiedener Indexfelder liegen im Gegensatz zum SIW-Schleifennest die Datenelemente einer Iteration im allgemeinen nicht auf demselben Prozessorknoten. Damit verhindert das *alignment* alle *page faults* bezüglich eines der Datenfelder; für alle Datenfelder, die nicht über dasselbe Indexfeld adressiert werden, können sich im allgemeinen *page faults* ergeben. Tatsächlich beziehen sich die *page faults* in der Simulation ausschließlich auf das Datenfeld ohne *alignment*. Ein MIR-Schleifennest führt wieder zu qualitativ gleichen Ergebnissen; auch hier ist der *page-fault*-Anteil vergleichsweise etwas reduziert.

Die Ergebnisse der Untersuchungen in diesem Abschnitt können wie folgt zusammengefaßt werden: Das *Affinity-Scheduling* resultiert im Vergleich zu den Algorithmen ohne Beachtung von Affintäten in einem etwas günstigeren Zeitverhalten. Für verschiedene

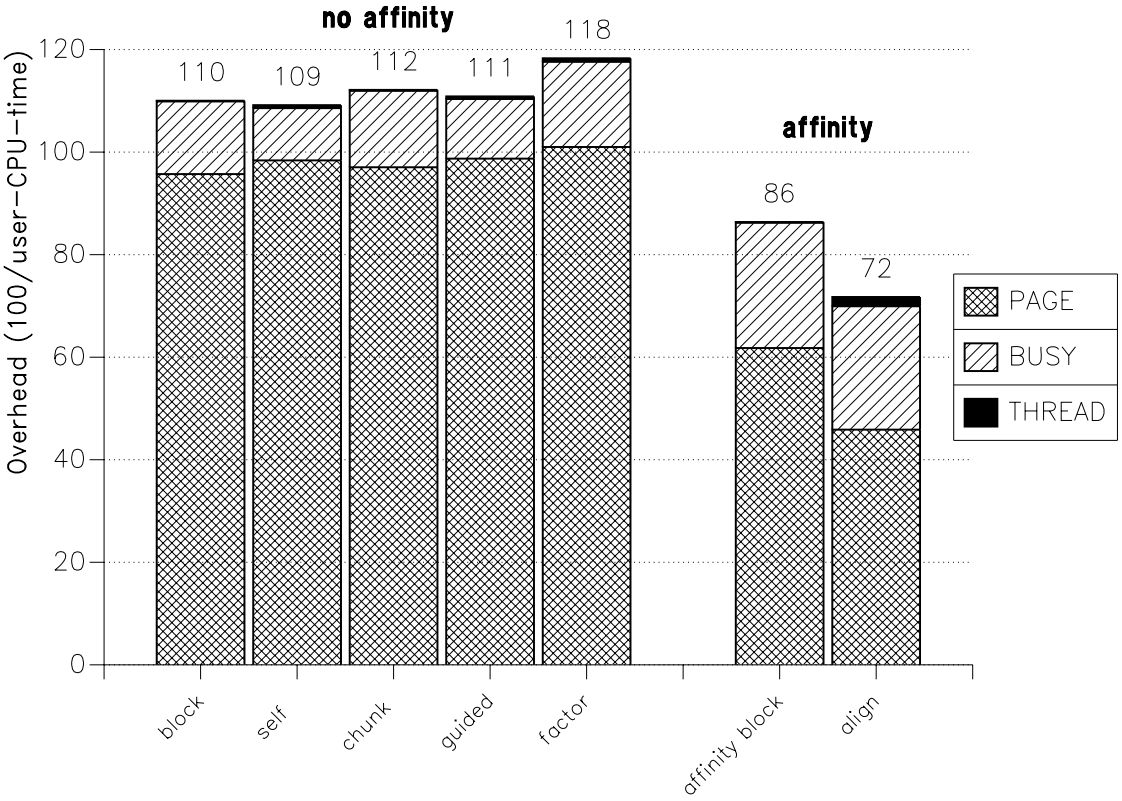


Abb. 61. Unterschiedliche Algorithmen zum thread-Scheduling bei mehrfacher, variabler Indexadressierung:
Overhead bezogen auf die user-CPU-time in Prozent,
MIW-Schleife (Multiple Index Write), Seitengröße 128, 10 CPUs,
block, self, chunk, guided, factor: **ohne Beachtung von Affinitäten,**
affinity block, align: **mit Beachtung von Affinitäten**

Schleifennester mit variabler Indexadressierung wird die Effizienz des *Affinity-Scheduling* von der Häufigkeit der Indexänderungen abhängen; so kann sich die Effizienz bei häufigeren Indexänderungen verschlechtern oder bei selteneren Indexänderungen verbessern. Für sehr seltene Indexänderungen ist zu erwarten, daß das *Affinity-Scheduling* das *Align-Scheduling* bezüglich der Effizienz übertreffen kann.
Bei allen durchgeführten Untersuchungen hat sich das *Align-Scheduling* als jeweils effizienteste Lösung dargestellt. Für die Schleifennester vom *Single-Index*-Typ war die Effizienz durchaus befriedigend; alle *page faults* konnten eliminiert werden. Bei den Schleifennestern vom *Multiple-Index*-Typ sind die Vorteile durch das *Align-Scheduling* jedoch nur begrenzt; der *page-waiting*-Anteil ist gegenüber den übrigen Algorithmen zwar reduziert, aber dennoch erheblich. Damit können alle betrachteten Scheduling-Algorithmen für die untersuchten Schleifennester vom *Multiple-Index*-Typ als ungeeignet bezeichnet werden; dabei ist zu beachten, daß sich dieser Typ von Schleifennestern auf allen Rechnern mit physikalisch verteiltem Speicher problematisch darstellen kann. Eine effiziente Ausführung kann möglicherweise dann erfolgen, wenn die benötigten Daten frühzeitig bereitgestellt werden können; in *virtual-shared-memory*-Systemen kann das durch geeignete *prefetch*-Operationen erreicht werden, sofern die zwischen der Änderung von Indexfeldern und den tatsächlichen Datenzugriffen liegende Zeitspanne hinreichend groß ist.

7.6. Parallelität auf verschiedenen Schleifenebenen

Während die in den bisherigen Untersuchungen in diesem Kapitel betrachteten Schleifennester durch ihr charakteristisches Speicherzugriffsverhalten ein Klasse von Anwendungen repräsentieren sollten, wird die in diesem Abschnitt betrachtete Klasse von Schleifen durch die Art der Schleifenschachtelung charakterisiert: Es werden Schleifennester mit zwei oder mehr ineinander geschachtelten, parallelen Schleifen untersucht (siehe Abschnitt 5.2.3). Bezüglich des Speicherzugriffsverhaltens verschiedener Anwendungen kann eine weitere Klassifizierung erfolgen; in diesem Abschnitt wird jedoch beispielhaft ausschließlich eine Matrixmultiplikation gemäß der Abb. 62 betrachtet.
Die alternative Parallelisierung einer der beiden parallelisierbaren Schleifen stellt den ersten Untersuchungsparameter dar; Abb. 63 beschreibt den relativen Overhead für die Parallelisierung der äußeren Schleife (10/1 = 10 Prozessoren in der äußeren Schleife, mittlere Schleife sequentiell) beziehungsweise für die Parallelisierung der mittleren Schleife (1/10). Bei der Parallelisierung der mittleren Schleife bestehen Affinitäten beim Datenzugriff; deswegen wurde ein *Affinity-Scheduling* durchgeführt. Für die Parallelisierung der äußeren Schleife ergibt sich ein für die Matrixmultiplikation zunächst unerwartet hoher relativer Overhead von 60.2 Prozent; dieser Overhead wird nahezu ausschließlich durch *page-waiting* verursacht. Die Parallelisierung der mittleren Schleife führt zu einer deutlich effizienteren Ausführung; der relative Overhead beträgt nur noch 5.1 Prozent und beruht im wesentlichen auf *busy-waiting*.
Die Erklärung für dieses Ergebnis liegt in der begrenzten Lokalspeichergröße: Bei der Parallelisierung der äußeren Schleife ergibt sich gemäß Abschnitt 5.2.3 ein etwa um den Faktor der Prozessorzahl (10) höherer Speicherbedarf als bei der Vergleichsmessung. Die Größe des Lokalspeichers ist für diese Messungen gerade so gewählt, daß sich für die Parallelisierung der äußeren Schleife ein Speicherengpaß ergibt: Die für den Lesezugriff mehrfach benötigten Datenelemente der Matrix *B* müssen für jeden Zugriff in den lokalen Speicher geladen werden; vor dem nächsten Zugriff werden diese Daten aufgrund von Platzmangel wieder gelöscht. Das mehrfache Bereitstellen derselben Daten erklärt den hohen *page-waiting*-Overhead.
Für die Parallelisierung der mittleren Schleife ist der *page-waiting*-Anteil nahezu vollständig eliminiert; alle Daten werden nur einmal bereit gestellt und können dann mehrfach wiederverwendet werden. Der *busy-waiting*-Anteil ist jedoch gegenüber der Vergleichsmessung erhöht. Die Ursache für die Erhöhung liegt im vergleichsweise erhöhten *thread*-Scheduler-Overhead; der *thread*-Scheduler tritt im betrachteten Fall sehr viel häufiger in Aktion, und es ergibt sich eine Erhöhung beim Overhead. Da der *thread*-

Matrix Multiplication
with nested parallel loops

```
doall I = 1, N
   doall J = 1, N
      do K = 1, N
         C(I,J) = C(I,J) +A(I,K)*B(K,J)
      enddo
   enddo
enddo
```

Abb. 62. Matrixmultiplikation mit geschachtelten, parallelen Schleifen

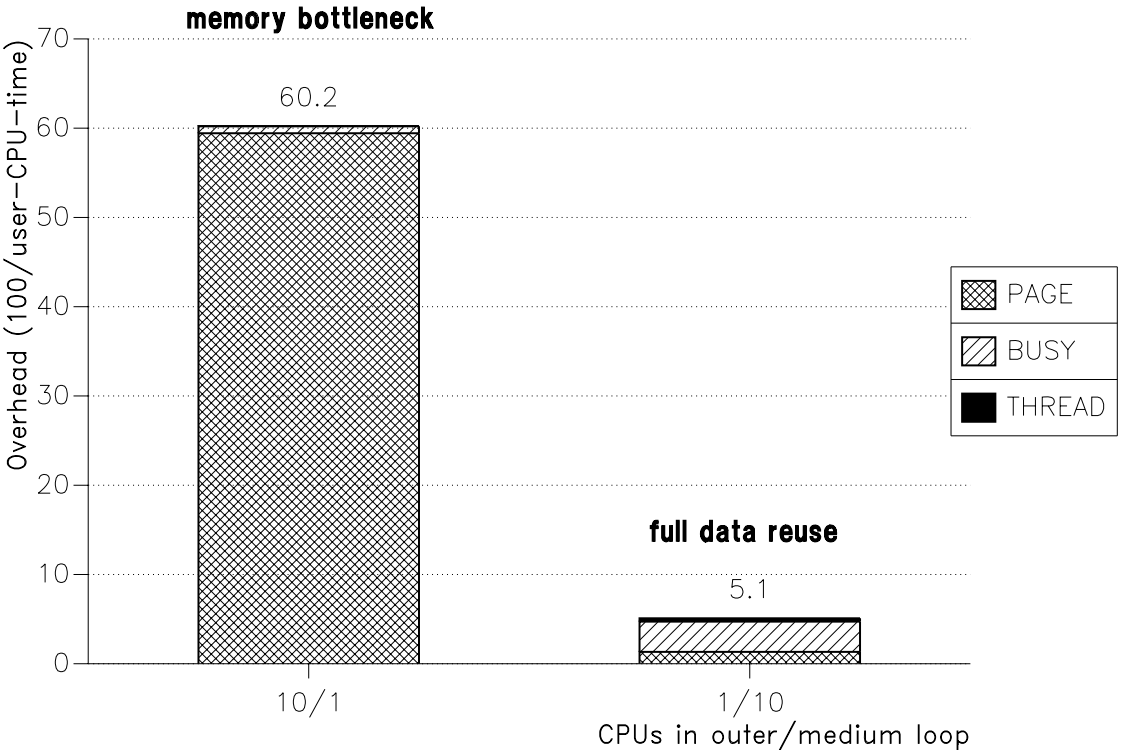


Abb. 63. Parallelisierung der äußeren oder der mittleren Schleife für eine Matrixmultiplikation: Overhead bezogen auf die user-CPU-time in Prozent, Matrixmultiplikation 100*100 mit prefetch, Seitengröße 128, 10 CPUs, Datenabbildung: A zeilenweise, B spaltenweise, C zeilenweise, initiale Seitenverteilung: cyclic-read für alle drei Matrizen

Scheduler nur exklusiv vergeben wird, entsteht ein *busy-waiting*-Anteil - im Vergleich zum *thread*-Scheduler-Overhead in sehr viel größerem Ausmaß. Damit ergibt sich in beiden betrachteten Fällen ein nachteiliger Effekt: Für die Parallelisierung der äußeren Schleife ein Speicherengpaß und für die Parallelisierung der mittleren Schleife ein erhöhter *busy-waiting*-Anteil. Die Abb. 64 zeigt, daß die beste Lösung zwischen den beiden betrachteten Extremfällen liegt, nämlich in der Parallelisierung beider Schleifen: Die Abb. 64 beschreibt zusätzlich zu den beiden schon betrachteten Messungen zwei weitere Messungen, bei denen beide Schleifen parallelisiert wurden, einmal mit 5 Prozessoren bezüglich der äußeren Schleife und mit 2 Prozessoren bezüglich der mittleren Schleife (5/2) und das andere Mal umgekehrt (2/5). Für die Messung 5/2 besteht der Speicherengpaß nach wie vor, d.h. die Reduzierung des Speicherbedarfs um etwa den Faktor 2 durch 2 Prozessoren in der mittleren Schleife war nicht ausreichend. Darüber hinaus ergibt sich für diese Messung gegenüber der Messung 1/10 ein deutlich erhöhter *busy-waiting*-Anteil. Der Grund für diese Erhöhung liegt in der Kombination von unbalancierter Ausführung aufgrund von *page faults* auf der einen Seite und vergleichsweise häufigeren Synchronisationspunkten auf der anderen Seite. Der bei allen betrachteten Messungen niedrigste relative Overhead ergibt sich mit 3.1 Prozent für die Messung 2/5. Die vorliegende Konstellation reduziert den Speicherbedarf soweit, daß kein Speicherengpaß mehr auftritt. Das *thread*-Scheduling findet im Vergleich zu Messung 1/10 auf zwei Ebenen für die beiden Schleifen statt; durch die so realisierte Verteilung des *thread*-Scheduling

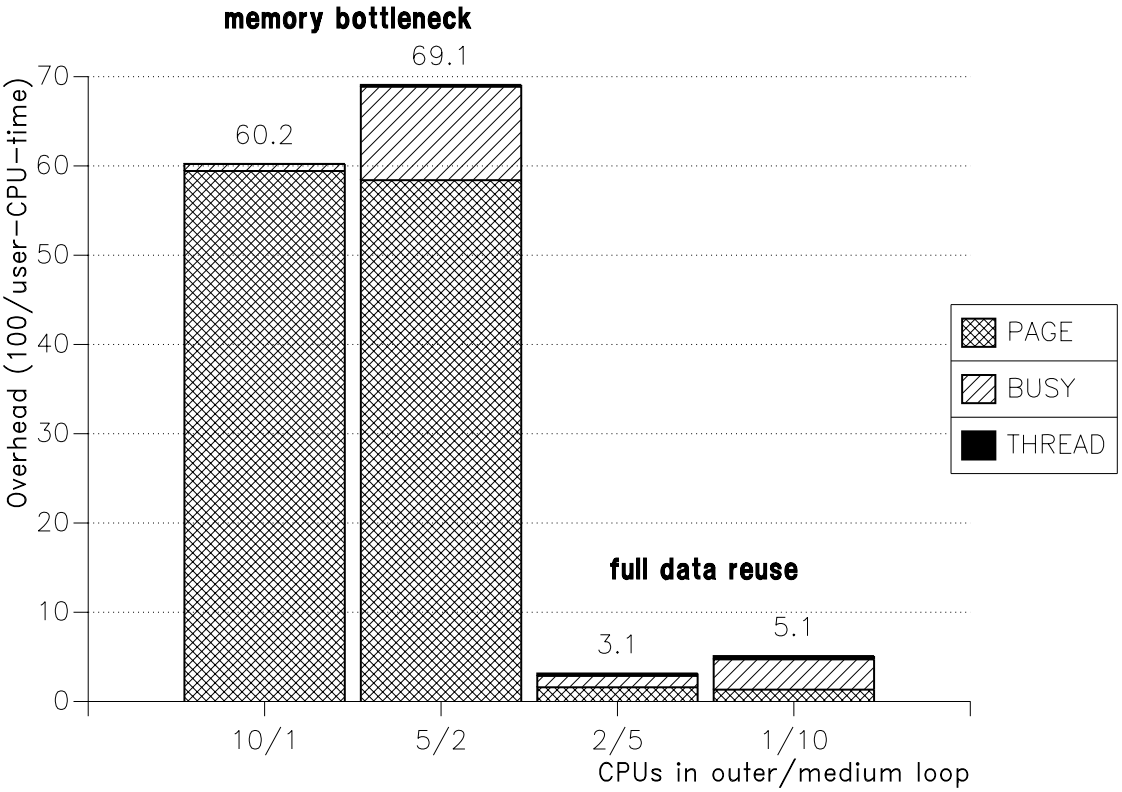


**Abb. 64. Verteilung von 10 CPUs über geschachtelte Schleifen; Speicherengpaß für die Parallelisierung der äußeren Schleife:
Overhead bezogen auf die user-CPU-time in Prozent,
Matrixmultiplikation 100*100 mit prefetch, Seitengröße 128, 10 CPUs,
Datenabbildung: A zeilenweise, B spaltenweise, C zeilenweise,
initiale Seitenverteilung: cyclic-read für alle Matrizen**

ist auch der *busy-waiting*-Overhead in Relation zu Messung 1/10 reduziert.
Um die Stabilität der beobachteten Effekte zu untersuchen und eventuell weitere Effekte zu identifizieren, wurden weitere Messungen mit erhöhter Prozessorzahl durchgeführt. Die Abb. 65 und die Abb. 66 beschreiben die Messungen für dieselbe Matrixmultiplikation jetzt mit der Problemgröße 200 und mit 20 Prozessoren (Abb. 65) beziehungsweise mit 50 Prozessoren (Abb. 66): Bei der Meßreihe mit 20 Prozessoren ergeben die Messungen 20/1, 10/2, 5/4, 4/5 und 2/10 zumindest quantitativ ähnliche Ergebnisse wie die entsprechenden Messungen der zuvor betrachteten Meßreihe; die Messung 10/2 zeigt jedoch in Relation zur Messung 5/2 eine deutlich andere Zusammensetzung der Overhead-Anteile: Während bei der Messung 5/2 der mit Abstand größte Teil des Overhead auf *page-waiting* beruht, wird der Overhead bei der Messung 10/2 zu etwa gleichen Teilen durch *page-waiting* beziehungsweise durch *busy-waiting* verursacht. Der Grund für diesen Unterschied ergibt sich wie folgt: Bei den Aufteilungen 5/2 und 10/2 benötigt gemäß Abschnitt 5.2.3 jeder Prozeß von der Matrix *B* nur einen Teil; durch die Parallelisierung der mittleren Schleife mit jeweils 2 Prozessoren wird jeweils genau die halbe Matrix *B* benötigt. Die Verteilung der Seiten über die Prozessorelemente in Kombination mit der Verteilung der Prozesse ergibt bei der Messung 10/2, daß ein Teil der benötigten Daten der Matrix *B* auf einigen Prozessorelementen bereits lokal verfügbar ist; dadurch

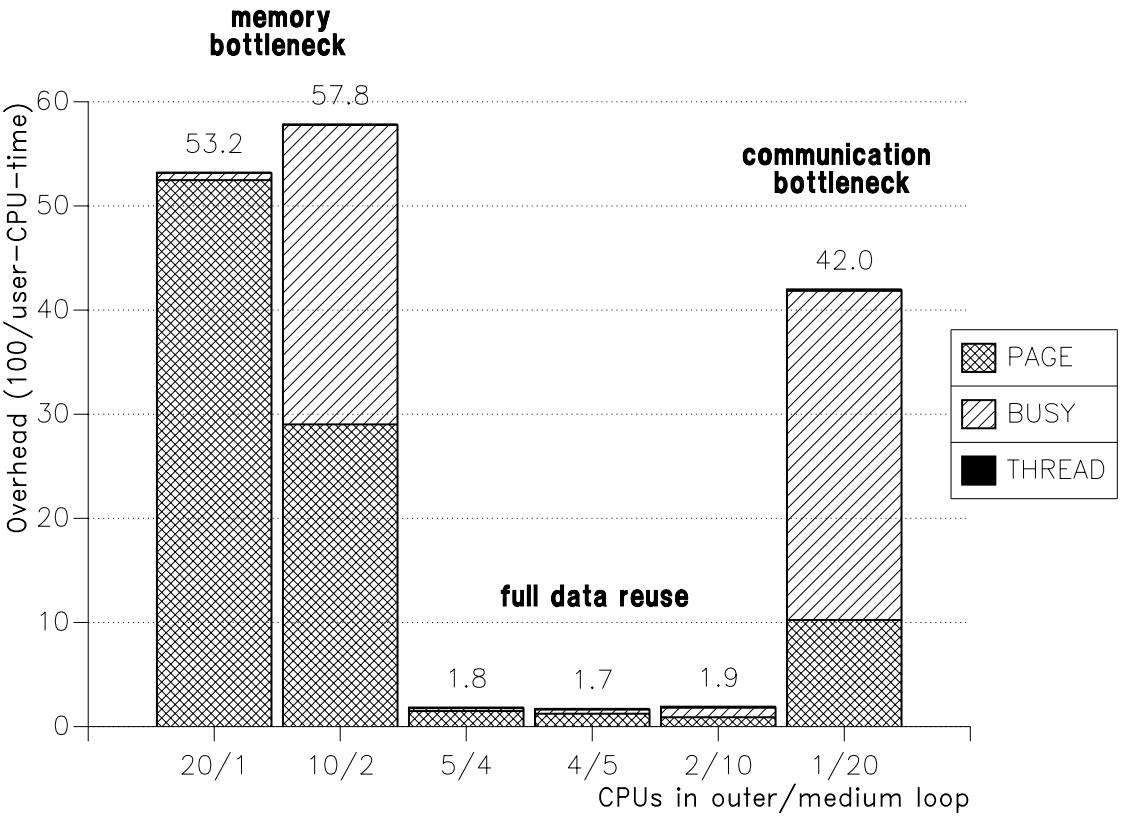


**Abb. 65. Verteilung von 20 CPUs über geschachtelte Schleifen; Speicherengpaß für die Parallelisierung der äußeren Schleife:
Overhead bezogen auf die user-CPU-time in Prozent,
Matrixmultiplikation 200*200 mit prefetch, Seitengröße 256, 20 CPUs,
Datenabbildung: A zeilenweise, B spaltenweise, C zeilenweise,
initiale Seitenverteilung: cyclic-read für alle Matrizen**

ergibt sich ein Speicherengpaß tatsächlich nur bei einem Teil der Prozessoren. Durch die auf diese Weise unbalancierte Ausführung ensteht an den Synchronisationspunkten ein erheblicher *busy-waiting*-Anteil. Bei der Messung 5/2 ist dagegen bei allen Prozessoren ein Speicherengpaß zu beobachten, so daß dort die Ausführung balanciert ist und sich deutlich weniger *busy-waiting* ergibt.
Ein sowohl qualitativer als auch quantitativer Unterschied ergibt sich beim Vergleich der Messungen 1/10 und 1/20: Bei der Messung 1/20 ist der relative Overhead mit 42 Prozent deutlich erhöht. Verursacht wird dieses Ergebnis durch einen Kommunikationsengpaß: Von den Matrizen *A* und *C* werden in aufeinanderfolgenden Iterationen der äußeren Schleife jeweils andere Datenbereiche benötigt; dabei wird jeder Datenbereich insgesamt genau einmal von einem Prozessor geladen. Um die dabei normalerweise auftretenden *page-fault*-Wartezeiten zu eliminieren werden an geeigneter Stelle *prefetch*-Operationen durchgeführt. Bei der Meßreihe mit 10 Prozessoren, der Problemgröße 100 und der Seitengröße 128 war die für die *prefetch*-Operationen zur Verfügung stehende Zeit ausreichend, um die entsprechenden Daten bereits vor dem tatsächlichen Zugriffswunsch bereitzustellen. Bei der hier betrachteten Meßreihe mit 20 Prozessoren, der Problemgröße 200 und der Seitengröße 256 war die zur Verfügung stehende Zeit jedoch so kurz, daß die *prefetch*-Operationen nur teilweise rechtzeitig beendet werden konnten. Durch

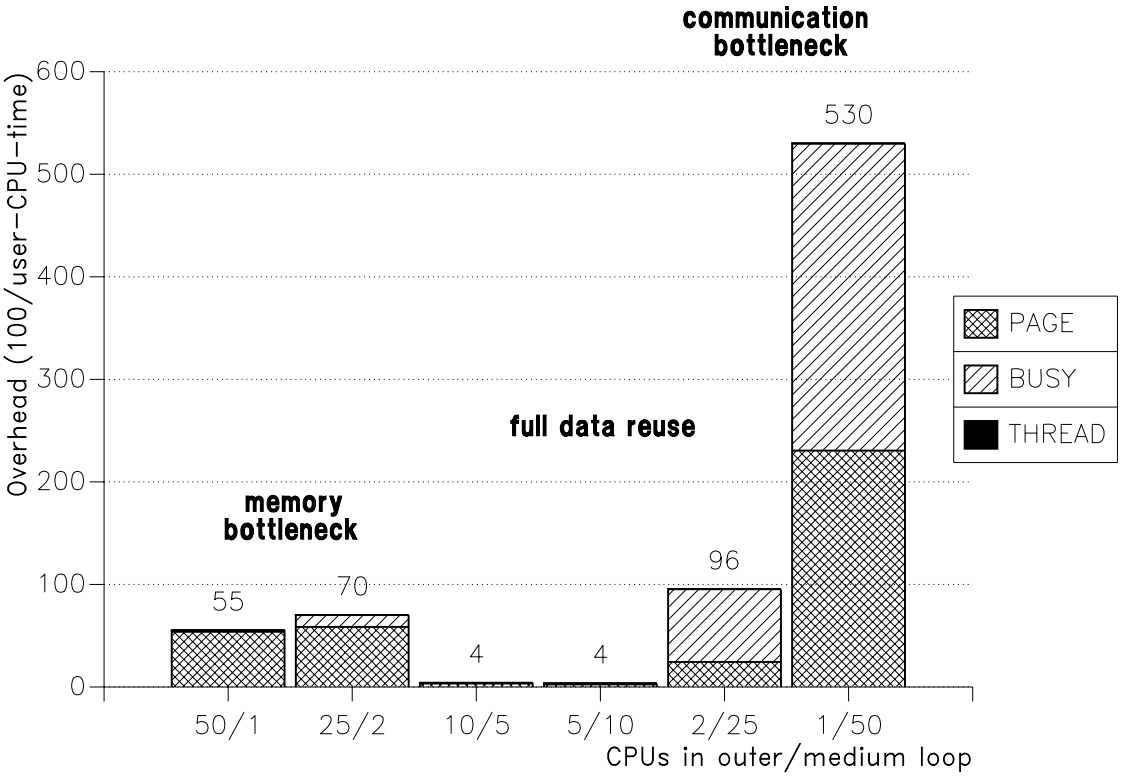


Abb. 66. Verteilung von 50 CPUs über geschachtelte Schleifen; Speicherengpaß versus Kommunikationsengpaß:
Overhead bezogen auf die user-CPU-time in Prozent,
Matrixmultiplikation 200*200 mit prefetch, Seitengröße 256, 50 CPUs,
Datenabbildung: A zeilenweise, B spaltenweise, C zeilenweise,
initiale Seitenverteilung: cyclic-read für alle Matrizen

den Kommunikationsengpaß wird das *page-waiting* erzeugt, das wiederum zu einer unbalancierten Ausführung und damit zu *busy-waiting* führt. Die Messungen 5/4, 4/5 und 2/10 mit einer Verteilung der Prozessoren über die zwei parallelen Schleifen zeigen die deutlich beste Effizienz.

Die Abb. 66 beschreibt die Meßreihe mit 50 Prozessoren: Im Vergleich zu den Messungen 20/1 und 10/2 haben sich hier bei den entsprechenden Vergleichsmessungen nicht etwa deutliche Verbesserungen ergeben, sondern der Maßstab der Darstellung ist um etwa eine Größenordnung größer gewählt. Die Messungen 50/1, 25/2, 10/5 und 5/10 zeigen ein quantitativ ähnliches Verhalten wie die entsprechenden Messungen bei der Meßreihe mit 20 Prozessoren. Für die beiden Messungen 2/25 und 1/50 ist demgegenüber jedoch ein erheblicher Anstieg des relativen Overhead zu verzeichnen; mit 96 und 530 Prozent ergeben sich die mit Abstand ineffizientesten Messungen dieser Reihe. Die Ursache liegt, wie bei der Messung 1/20, an einem Kommunikationsengpaß bezüglich der Matrizen *A* und *B*. Für die Messung 1/50 ist die Granularität bei der Ausführung am feinsten, und der entstehende Kommunikationsengpaß ist am deutlichsten; für eine weitere Erhöhung der Prozessorzahl ist auch eine weitere Verschlechterung der Ergebnisse zu erwarten. Die Kombination von Speicherengpaß und Kommunikationsengpaß in den beiden Meßreihen mit 20 beziehungsweise mit 50 Prozessoren führt zu einer klaren Favorisierung von ge-

schachteltem Parallelismus: Die Messungen mit einer Verteilung der Prozessoren über die zwei parallelen Schleifen zeigt mit Abstand die höchste Effizienz.

Nach diesen eindeutigen Ergebnissen aus den Meßreihen mit Speicherengpaß wird bei den folgenden Untersuchungen ein jeweils hinreichend großer Speicher zur Verfügung stehen, so daß keine Speicherengpässe mehr auftreten können. Die Abb. 67 zeigt eine Vergleichsmessung zu Abb. 66, bei der nur der Speicher vergrößert ist: Eine Veränderung ergibt sich ausschließlich für die Messungen, die zuvor vom Speicherengpaß betroffen waren, das sind die beiden Messungen 50/1 und 25/2. Für beide Messungen konnte der relative Overhead deutlich verringert werden; mit 16 beziehungsweise 8 Prozent liegt dieser aber noch immer um einen Faktor 4 beziehungsweise 2 höher als bei den günstigsten Konstellationen. Die Ursache findet sich wiederum im Datenbedarf je Prozessor. Wie bereits in Abschnitt 5.2.3 erläutert, können bei der Parallelisierung der mittleren Schleife aufgrund des *Affinity-Scheduling* die Daten der Matrix *B* bei aufeinanderfolgenden Iterationen der äußeren Schleife wiederverwendet werden. Tatsächlich ergibt sich dadurch ein im Vergleich zur Parallelisierung der äußeren Schleife reduzierter Datenbedarf je Prozessorelement. Deswegen entstehen bei den Messungen 50/1 und 25/2 mehr *page faults* und damit ein insgesamt höherer *page-fault*-Overhead. Alle übrigen Messungen

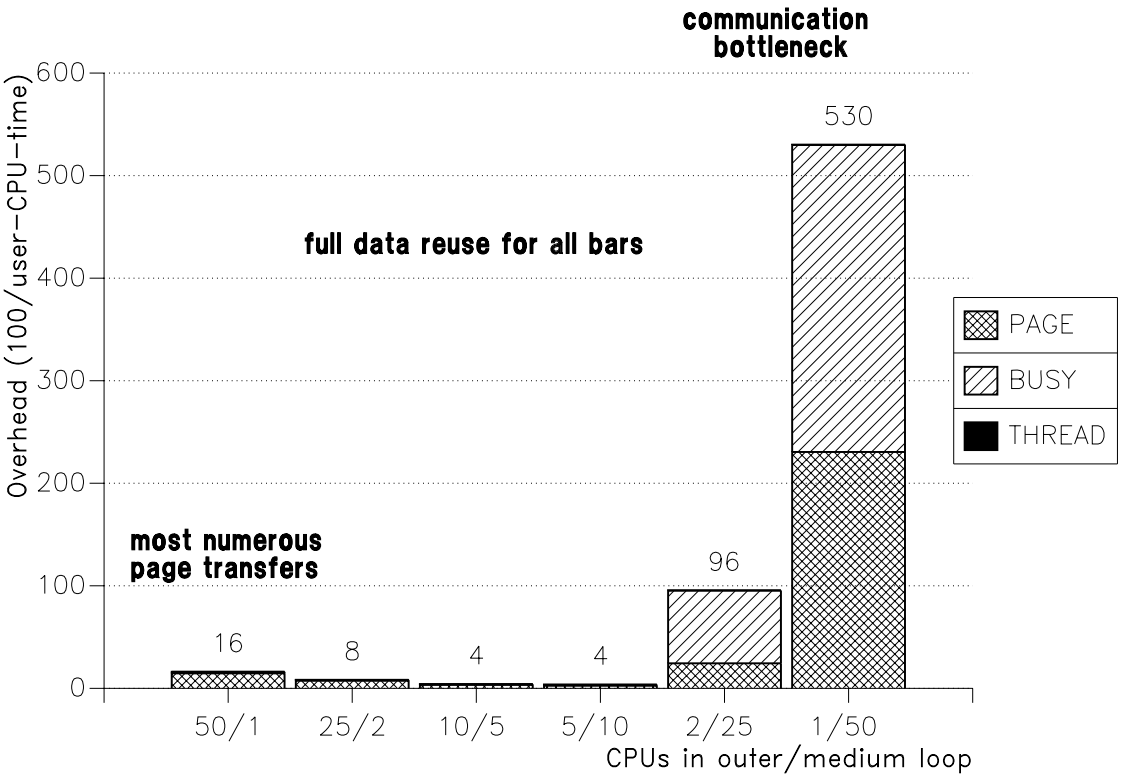


Abb. 67. Verteilung von 50 CPUs über geschachtelte Schleifen; hohes Kommunikationsaufkommen versus Kommunikationsengpaß:
Overhead bezogen auf die user-CPU-time in Prozent,
Matrixmultiplikation 200*200 mit prefetch, Seitengröße 256, 50 CPUs,
Datenabbildung: A zeilenweise, B spaltenweise, C zeilenweise,
initiale Seitenverteilung: cyclic-read für alle Matrizen,
ohne Speicherengpaß durch hinreichend großen Speicher

sind im Vergleich zur Abb. 66 unverändert. Damit ergibt sich auch für diese Messung ohne Speicherengpaß eine Favorisierung von geschachteltem Parallelismus.

Bei allen bisher in diesem Abschnitt betrachteten Messungen war die initiale Verteilung der Matrizen über die Prozessorelemente sehr günstig gewählt; in allen Fällen erfolgte die initiale Verteilung der Matrizen beziehungsweise der zugehörigen Seiten zyklisch. Im folgenden wird eine Meßreihe betrachtet, bei der im Vergleich dazu die Seiten der Matrizen *A* und *C* auf allen Prozessorelementen vorliegen und die Seiten der Matrix *B* initial blockweise verteilt sind. Die Abb. 68 beschreibt diese Meßreihe, die gegenüber der Meßreihe in Abb. 67 ansonsten unverändert ist und damit ebenfalls keinen Speicherengpaß enthält: Die Werte für den relativen Overhead liegen bei dieser Meßreihe ungefähr auf einer Parabel, bei den in der Bildmitte dargestellten Messungen ist der relative Overhead am kleinsten, und nach außen hin steigt der Overhead jeweils an. Der Anstieg auf der linken Seite erreicht für die Messung 50/1 ihr Maximum, das mit 50 Prozent in derselben Größenordnung liegt wie die Overhead-Werte aufgrund von Speicherengpässen. Die Ursache liegt hier jedoch anders: Bei der Parallelisierung der äußeren Schleife benötigt jeder Prozeß die gesamte Matrix *B*, und zwar vollständig bereits für

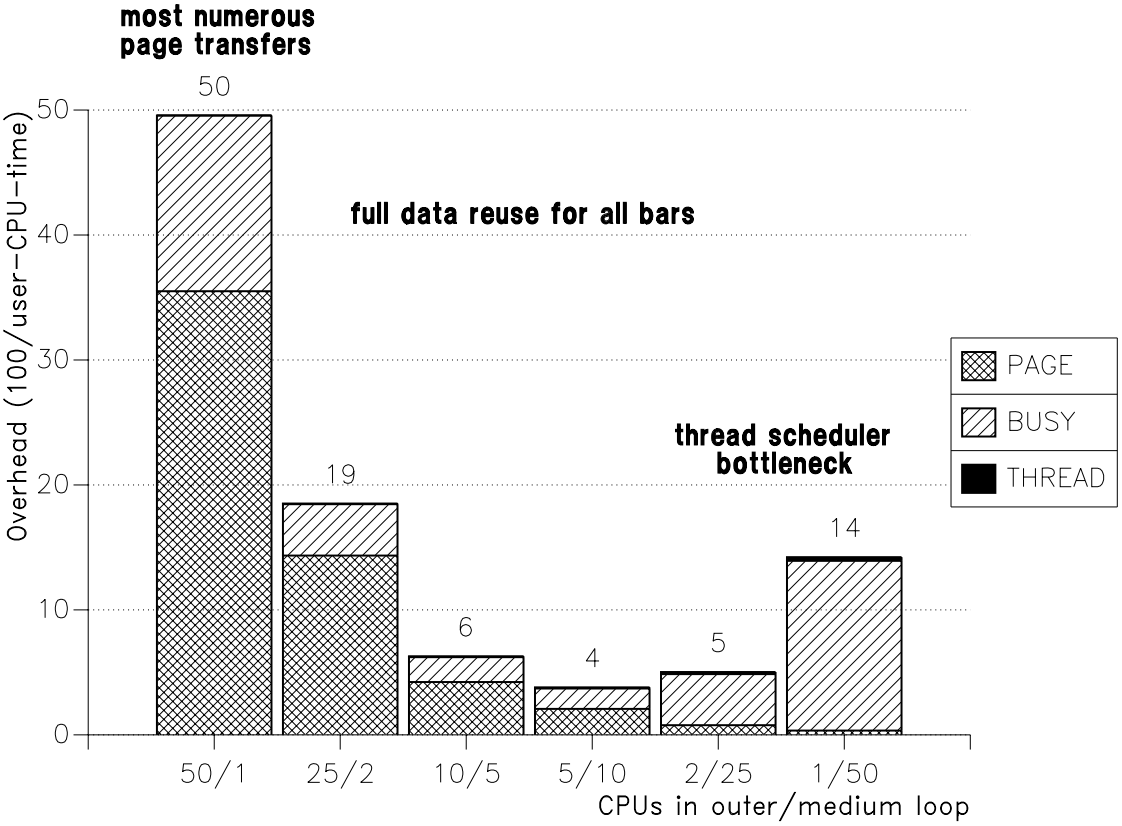


Abb. 68. Verteilung von 50 CPUs über geschachtelte Schleifen; hohe Kommunikationsanforderung versus thread-Scheduler-Engpaß:
Overhead bezogen auf die user-CPU-time in Prozent,
Matrixmultiplikation 200*200 mit prefetch, Seitengröße 256, 50 CPUs,
Datenabbildung: A zeilenweise, B spaltenweise, C zeilenweise,
initiale Seitenverteilung: Matrix A und C each-read, Matrix B block-read,
ohne Speicherengpaß durch hinreichend großen Speicher

die Ausführung der ersten Iteration der äußeren Schleife. In dieser Anfangsphase fordert jeder Prozeß die von ihm benötigten Teile der Matrix *B* an; da die benötigten Datenelemente nicht vom Index der äußeren Schleife abhängen, benötigt jeder Prozeß dieselben Datenelemente etwa zur gleichen Zeit. Weil die Daten nur sequentiell an die Prozessoren verschickt werden können, entsteht dabei ein Kommunikationsengpaß. Darüber hinaus werden alle Datenelemente der Matrix *B* von allen Prozessen in aufsteigender Reihenfolge angefordert. Während bei der zyklischen Verteilung der Seiten aufeinanderfolgende Seiten auf verschiedenen Prozessoren lagen, so daß sich eine Verteilung der Kommunikationslast ergab, wird durch die blockweise Verteilung eine weitere Verschärfung des Kommunikationsengpasses verursacht: Da aufeinanderfolgende Seiten bei der jetzt blockweisen Verteilung meist auf jeweils demselben Prozessor liegen, ist jeweils ein Kommunikationsprozessor mit dem Verschicken von mehreren Seiten an alle anderen Knoten beschäftigt, und alle anderen Kommunikationsprozessoren sind in diesem Zeitintervall *idle*. Der auf diese Weise erhebliche Kommunikationsengpaß verursacht den hohen *page-waiting*-Overhead und als Folge ein ebenfalls erhebliches *busy-waiting*. Die Erhöhung des relativen Overhead bei den Messungen auf der rechten Seite der Abb. 68 ist in Relation zur Abb. 67 nur noch gering; der *page-waiting*-Anteil ist vollständig und der *busy-waiting*-Anteil ist erheblich reduziert. Die Ursache liegt an der geänderten, initialen Verteilung der Matrizen *A* und *C*: Der in der Abb. 67 beobachtete Kommunikationsengpaß beruht auf der Bereitstellung der Matrizen *A* und *C*; da beide Matrizen jetzt auf allen Prozessorelementen vollständig vorliegen - dabei liegt die Matrix *A* mit Lesezugriffsrecht vor und die Matrix *C* mit schwach kohärentem Schreibzugriffsrecht - tritt kein Kommunikationsengpaß mehr auf. Der Anstieg des relativen Overhead auf der rechten Seite beruht jetzt ausschließlich auf einem erhöhten *busy-waiting*-Anteil. Die Bestimmung der Ursache kann anhand der Abb. 69 erfolgen; sie beschreibt den relativen Overhead für die Ausführung einer Matrixmultiplikation auf einem Rechner mit global gemeinsamem Speicher: Bei dieser Meßreihe basiert der Overhead ausschließlich auf dem *thread*-Scheduling und auf dem *busy-waiting*. Während der Overhead im linken Teil der Darstellung sehr niedrig ist, ergibt sich auf der rechten Seite ein schließlich erheblicher Anstieg. Die Ursache für diesen Anstieg liegt einerseits wieder in der wachsenden Häufigkeit der *thread*-Scheduler-Aufrufe. Andererseits wird der *thread*-Scheduler bei der vorliegenden, exklusiven Zuteilung an eine wachsende Zahl von Prozessen zu einem ebenfalls wachsenden Engpaß. Der *busy-waiting*-Anteil wächst dabei deutlich schneller als die Anzahl der Prozesse. Ein Vergleich der Werte für den relativen Overhead mit denen in der Abb. 68 ergibt, daß der Anstieg des *busy-waiting* zumindest bei der Messung 1/50 ausschließlich auf der gleichen Ursache beruht. Bei den übrigen Messungen enthält der dargestellte *busy-waiting*-Overhead außerdem Anteile, die auf einer unbalancierten Ausführung aufgrund von *page faults* basieren.

Zusammenfassend können folgende Forderungen für die Ablaufplanung von geschachtelten, parallelen Scheifen bei *virtual-shared-memory*-Rechnern aufgestellt werden; dabei sind alle Ergebnisse zunächst ausschließlich auf die betrachtete Matrixmultiplikation bezogen: Beim Vorliegen von Speicherengpässen, in der betrachteten Matrixmultiplikation bezüglich der Matrix *B*, muß aus Effizienzgründen die mittlere Schleife parallelisiert werden. Mit wachsender Prozessorzahl kann jedoch, in Abhängigkeit von der Verteilung

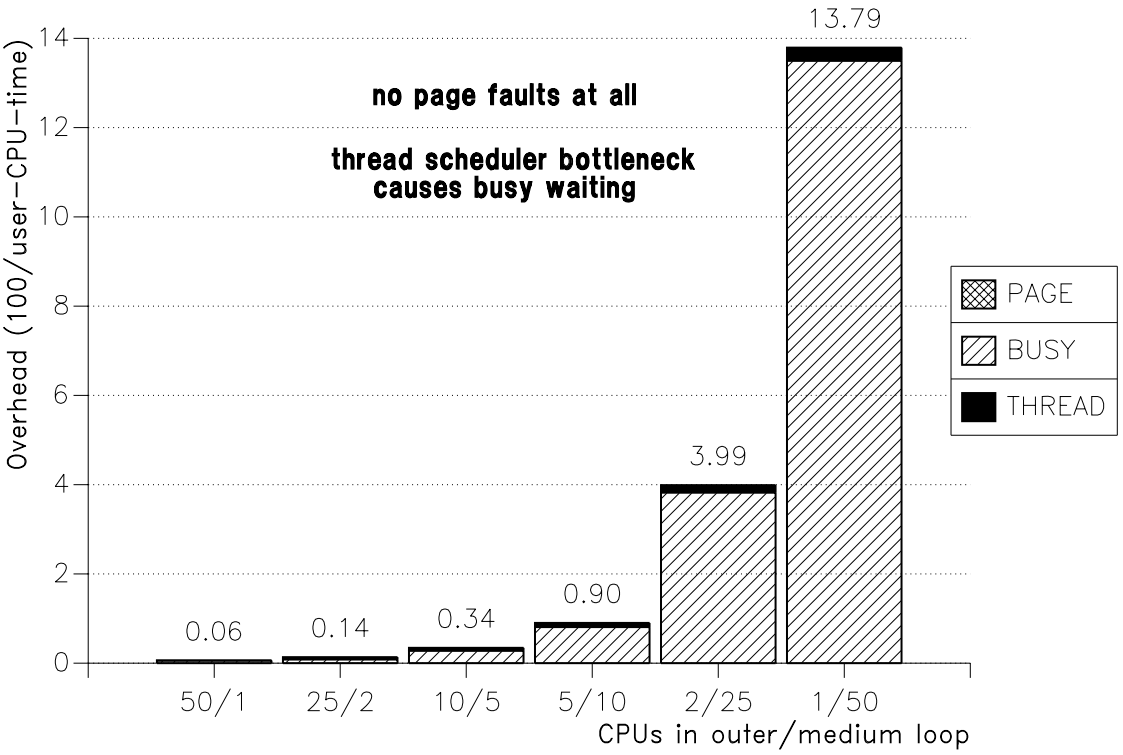


Abb. 69. Geschachtelte parallele Schleifen auf einem global-shared-memory-Rechner; Thread-Scheduler-Engpaß: Overhead bezogen auf die user-CPU-time in Prozent, Matrixmultiplikation 200*200, 50 CPUs, *global shared memory***,**

der Matrizen über die Prozessorelemente, insbesondere bei der Parallelisierung nur der mittleren Schleife ein Kommunikationsengpaß bezüglich der Matrizen A und C auftreten; in diesem Fall führt nur die Parallelisierung sowohl der äußeren als auch der mittleren Schleife bei einer geeigneten, d.h. in den meisten Fällen etwa gleichmäßigen, Verteilung der Prozessoren über die beiden Schleifen zu einer effizienten Ausführung.

Auch ohne Speicherengpaß kann die Parallelisierung nur der äußeren Schleife zum Beispiel aufgrund einer ungünstigen Verteilung der Matrizen zu einer schlechten Effizienz aufgrund von häufigen Datentransfers führen. Darüber hinaus ergibt sich auch ohne Kommunikationsengpaß bei der Parallelisierung nur der mittleren Schleife ein Engpaß, der sich auf den *thread*-Scheduler bezieht. Auch in diesen Fällen führt nur die geschachtelte Parallelisierung zu einer hohen Effizienz.
Selbst bei den bestmöglichen Voraussetzungen, d.h. kein Speicherengpaß, kein Kommunikationsengpaß und eine tatsächlich unrealistische, aber günstige Verteilung der Matrizen über die Prozessorelemente, führt die geschachtelte Parallelisierung zu der effizientesten Ausführung.

8. Folgerungen für die Ablaufplanung in VSM-Systemen

Die in dieser Arbeit vorgestellten Untersuchungen behandeln verschiedene Problembereiche bei der Ablaufplanung in *virtual-shared-memory*-Systemen: Die Abbildung von Daten auf Seiten bildet eine Grundlage für das *thread*-Scheduling, das für einfach parallele Schleifen und, darauf aufbauend, für geschachtelte parallele Schleifen untersucht wurde. Die folgenden Ausführungen erläutern die Wahl der Untersuchungsparameter im Gesamtkontext der Ablaufplanung bei VSM-Systemen; die gewonnenen Ergebnisse werden zusammengefaßt, und es werden Folgerungen für die Ablaufplanung aufgestellt.

Der Schritt vom *global shared memory* zum *virtual shared memory* bedeutet für die Ablaufplanung eine zusätzliche Abhängigkeit: Eine günstige Zuordnung von Schleifeniterationen beim *thread*-Scheduling hängt davon ab, welche Daten bei der Ausführung benötigt werden. Da die Daten zu Seiten gruppiert sind, kann eine Zuordnung nur dann günstig sein, wenn beim Datenzugriff keine Konflikte bezüglich der Seiten auftreten; die Iterationen, die dieselben Seiten benötigen, müssen gruppiert werden. Zunächst ist nur die Zuordnung von Iterationen zu Daten gegeben. Die Zuordnung von Iterationen zu Seiten wird durch die Abbildung von Daten auf Seiten geliefert; damit ist die Abbildung von Daten auf Seiten eine wichtige Grundlage für das *thread*-Scheduling in VSM-Systemen. Abb. 70 faßt die untersuchten Problemstellungen und die dazugehörigen Lösungen zusammen:
Für ein effizientes *thread*-Scheduling hat es sich als günstig erwiesen, wenn die von einer Iteration benötigten Daten auf eine Seite abgebildet werden; in vielen Fällen erfolgt der Datenzugriff innerhalb einer Iteration in Form eines Vektors, so daß im günstigen Fall zum Beispiel ein Vektor auf eine Seite abgebildet wird. Da in VSM-Systemen die Abbildung von Daten auf Seiten bisher üblicherweise ausschließlich spaltenweise erfolgt, ergibt sich eine erste Problemstellung zum Beispiel für einen nicht spaltenorientierten Datenzugriff. Gemäß Abb. 70 liefert die explizte Datenabbildung eine Lösung; dabei kann die Datenabbildung dem Datenzugriff angepaßt werden. Unabhängig davon kann eine zweite Problemstellung durch einen Unterschied zwischen Vektorgröße und Seitengröße

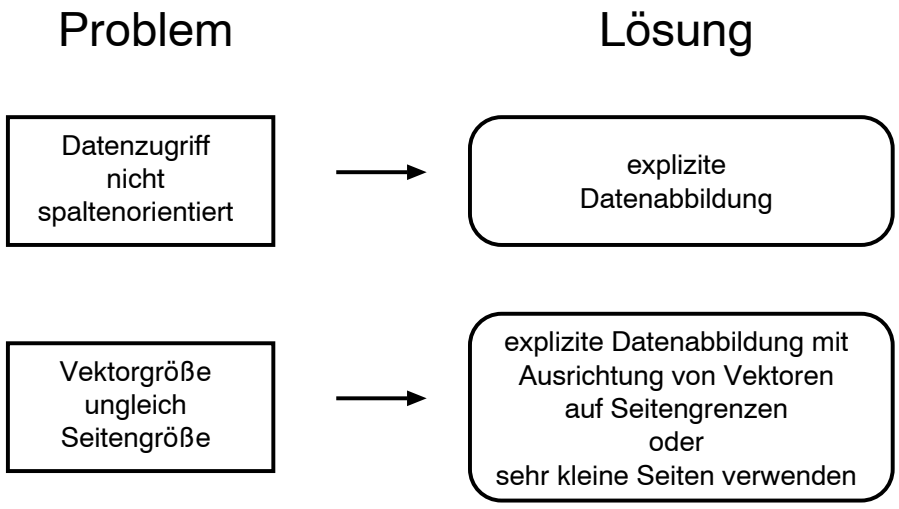


Abb. 70. Folgerungen für die Ablaufplanung: Abbildung von Daten auf Seiten

gegeben sein. Für diesen Fall kann die Ausrichtung der Vektoren auf Seitengrenzen bei der Datenabbildung Abhilfe schaffen. Da bei der Verwendung von relativ zur Vektorgröße sehr kleinen Seiten auch ohne Ausrichtung der Vektoren keine Probleme beobachtet worden sind, kann dies eine Alternativlösung sein (siehe Abb. 70).
Mit einer expliziten Datenabbildung als Basis ist das *thread*-Scheduling zunächst für einfach parallele Schleifen untersucht worden. Abb. 71 faßt die untersuchten Problemstellungen und die dazu gefundenen Lösungen zusammen. Eine erste Problemstellung für das *thread*-Scheduling ergab sich aus den Untersuchungen zur Datenabbildung: Nachdem die Fälle "Vektor größer als Seite" und "Vektor ungefähr gleich groß Seite" zu Problemstellungen geführt haben, die mittels einer geeigneten Datenabbildung gelöst werden konnten, stellt der verbleibende Fall "Vektor kleiner als Seite" (Multi-Vektor-Seiten) besondere Anforderungen an das *thread*-Scheduling. Das als Lösung dieser Problemstellung bereits in [GrWi93] beschriebene *Loop-Blocking* ist durch die in Kapitel 7 durchgeführten Simulationsuntersuchungen bestätigt worden (siehe Abb. 71). Damit sind alle möglichen Fälle für das Verhältnis Vektorgröße zu Seitengröße erfaßt.
Im folgenden soll aufgezeigt werden, welche Problemstellungen sich für das *thread*-Scheduling in Abhängigkeit von der Art der Anwendung ergeben können: Da ein effizientes *thread*-Scheduling von der aktuellen Datenverteilung abhängt, kann die Planung der

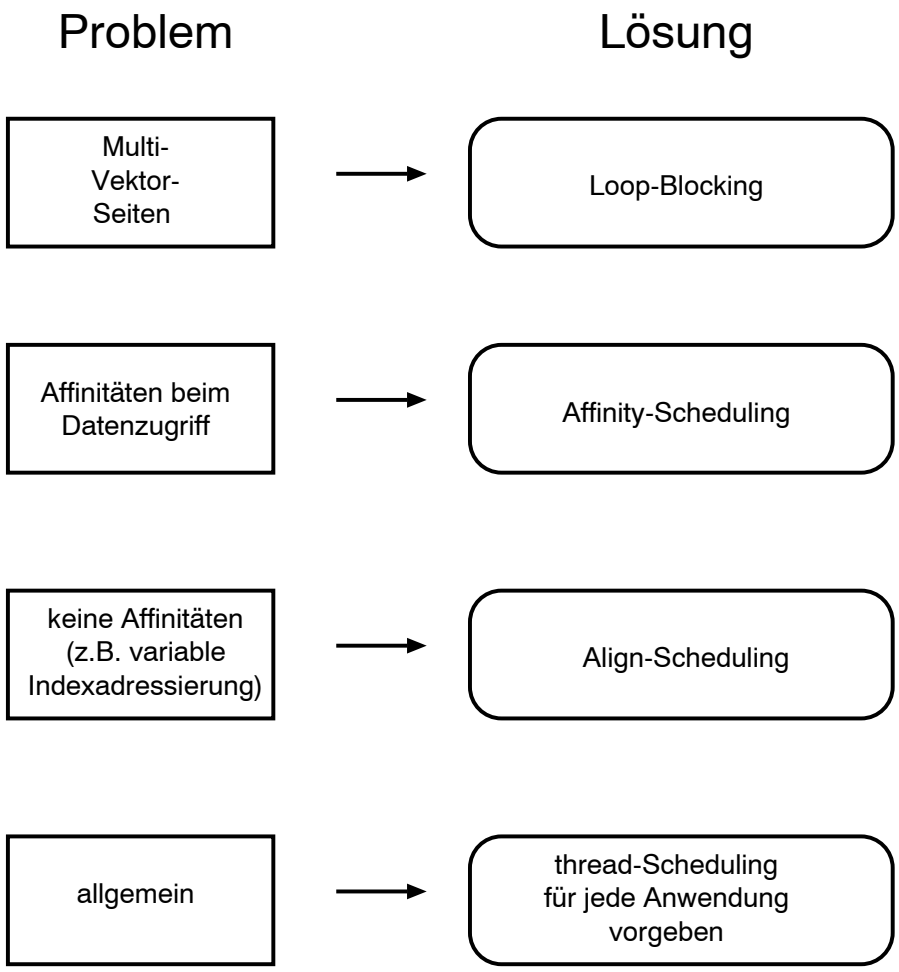


Abb. 71. Folgerungen für die Ablaufplanung: Thread-Scheduling für einfach parallele Schleifen

Abläufe nur dann erfolgreich sein, wenn dabei Informationen über die Datenverteilung verwendet werden, d.h. wenn zum Beispiel Kenntnisse über Vorgänge in der Vergangenheit genutzt werden, um günstige Entscheidungen für zukünftige Abläufe zu treffen. Kenntnisse über Vorgänge in der Vergangenheit liegen zum Beispiel immer dann vor, wenn sich diese Vorgänge wiederholen. Beim *thread*-Scheduling wird die Ausführung von parallelen Schleifen geplant; günstig planbar wird das *thread*-Scheduling also dann, wenn die zu planenden parallelen Schleifen mehrfach ausgeführt werden, wenn sie zum Beispiel innerhalb einer sequentiellen Schleife liegen. Für diese Konstellation sind zwei Fälle zu unterscheiden:

1. Bei der wiederholten Ausführung der parallelen Schleife greifen dieselben Iterationen wieder auf dieselben Daten zu, d.h. es bestehen Affinitäten beim Datenzugriff. Für diese Problemstellung hat sich das *Affinity-Scheduling* als günstig erwiesen (siehe Abschnitte 7.3 und 7.4).
2. Bei der wiederholten Ausführung besteht keine Garantie, daß dieselben Iterationen wieder auf dieselben Daten zugreifen, d.h. es bestehen keine Affinitäten. In diesem Fall kann das *Align-Scheduling* eine effiziente Ausführung gewährleisten (siehe Abschnitt 7.5).

In VSM-Systemen hängt der günstigste Scheduling-Algorithmus von der Art der Anwendung ab. Deshalb ist die Bestimmung eines geeigneten Algorithmus von besonderer Bedeutung: Die Festlegung kann automatisch durch Compiler oder Laufzeit-Bibliotheksroutinen oder manuell durch den Programmierer erfolgen; die Automatisierung einer solchen Festlegung kann in vielen Fällen zu guten Ergebnissen führen und stellt für zukünftige Arbeiten im Compiler-Bereich ein langfristiges Ziel dar.
Aufbauend auf den Untersuchungen zum *thread*-Scheduling bei einfach parallelen Schleifen sind geschachtelte parallele Schleifen betrachtet worden: Die Parallelisierung von mehr als einer Schleife kann besonders bei massiv-parallelen Systemen zu signifikanten Beschleunigungen führen; um das zu verdeutlichen, sind die Simulationsuntersuchungen für wachsende Prozessorzahlen durchgeführt worden. Die durch geschachtelte Parallelisierung möglichen Vorteile sind in Abb. 72 zusammengestellt:

- Die Auflösung eines Speicherengpasses kann durch eine Verschiebung der Parallelisierung zu einer weiter innen liegenden Schleife erzielt werden.
- Auch ein hoher Datenbedarf und damit verbundene Wartezeiten für Datentransfers können durch eine Verschiebung der Parallelisierung nach innen reduziert werden.
- Ein Kommunikationsengpaß aufgrund der gleichzeitigen Anforderung einer Seite durch alle Prozesse einer Schleife kann entschärft werden, indem die Anzahl der Prozesse in dieser Schleife reduziert wird; wenn diese Prozesse zur Parallelisierung einer anderen Schleife verwendet werden, dann entspricht das einer Verteilung der Parallelisierung über mehr Schleifen.
- Die massive Parallelisierung einer weiter innen liegenden Schleife kann einen *thread*-Scheduler-Engpaß bewirken; durch eine Verschiebung der Parallelisierung nach außen kann dieser Engpaß eliminiert werden.

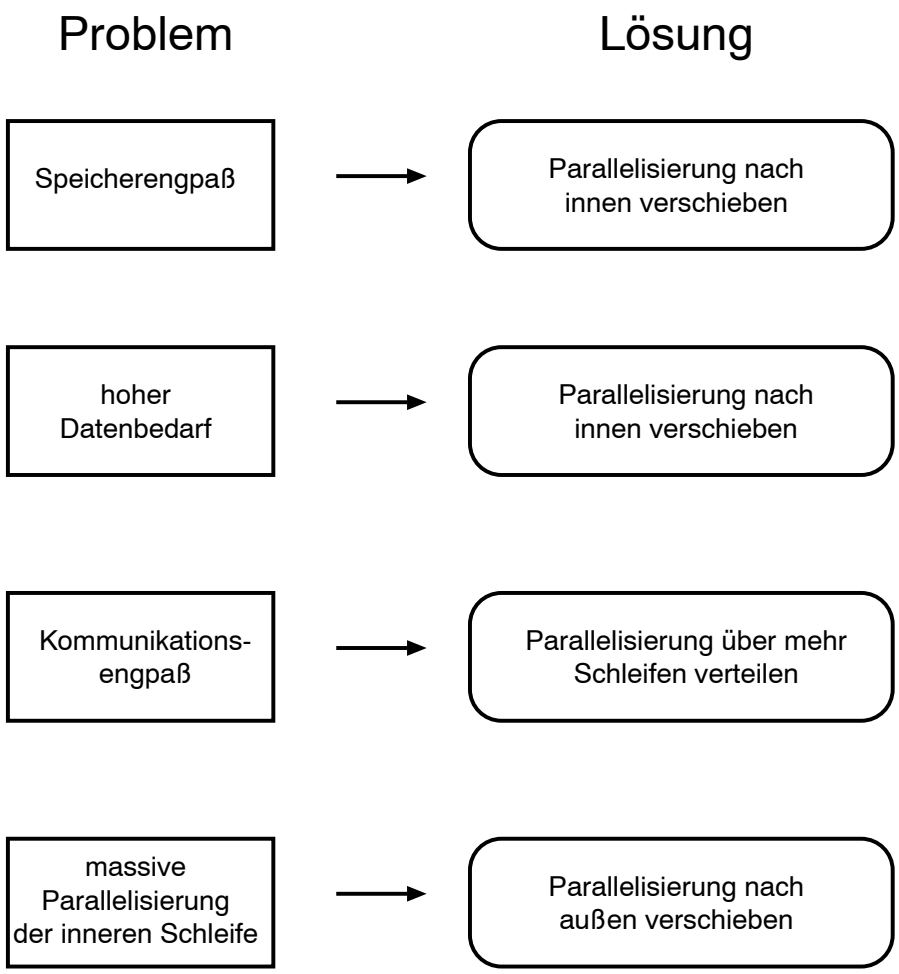


Abb. 72. Folgerungen für die Ablaufplanung: Thread-Scheduling für geschachtelte parallele Schleifen

Die Zusammenfassung der Ergebnisse hat gezeigt, daß die betrachteten Problembereiche durch die Wahl der Untersuchungsparameter jeweils vollständig untersucht worden sind. Die umfangreichen Informationen zu den Simulationsläufen in Verbindung mit der leistungsfähigen Visualisierung haben neben den relevanten Effekten auch die Ursachen dieser Effekte offengelegt. Als Ursache zum Beispiel für einen hohen *busy-waiting*-Anteil ergibt sich mithilfe der Visualisierung der Simulationsdaten ein *thread*-Scheduler-Engpaß. Das Verständnis von komplex zusammenhängenden Abläufen erlaubt die Zuordnung von Effekten zu Ursachen: Wenn die Ursache auftritt, dann ist der zugehörige Effekt zu erwarten, d.h. wenn zum Beispiel ein *thread*-Scheduler-Engpaß vorliegt, dann wird ein erhöhter *busy-waiting*-Anteil verursacht. Im folgenden wird die Verknüpfung von Ursachen und Effekten anhand einer Zusammenstellung der Ergebnisse dieser Arbeit gegeben. Tab. 3 nimmt Bezug auf die Abbildung von Daten auf Seiten:

- Eine Meßreihe wurde ohne Anpassung von Datenabbildung und Datenzugriff durchgeführt, d.h. mit zeilenorientiertem Datenzugriff und der bisher üblichen spaltenorientierten Datenabbildung. Bei einer derartigen, orthogonalen Orientierung benötigt jeder Prozeß jede Seite der entsprechenden Datenfelder. Das so verursachte massive Kommunikationsaufkommen führt zu einem Engpaß mit katastrophalen Folgen. Die-

Untersuchungskriterium	Effekt	Ursache
Abbildung von Daten auf Seiten		
- Datenabbildung und -zugriff nicht angepaßt	massives Kommunikationsaufkommen	jeder Prozeß benötigt jede Seite
- Vektorgröße ungleich Seitengröße	a) ohne Ausrichtung der Vektoren auf Seitengrenzen: page thrashing	Aufteilung der Vektoren über die Seiten
	b) negative Effekte ohne Ausrichtung verschwinden, wenn Seite << Vektor	

Tab. 3. Die Abbildung von Daten auf Seiten: Effekte und Ursachen

ser Effekt ist für alle Anwendungen zu erwarten, bei denen der Datenzugriff in Form von Vektoren erfolgt und Datenabbildung und Datenzugriff nicht angepaßt sind. Wie zwei weitere Meßreihen mit anderen Arbeitslasten gezeigt haben, werden die Ursache und der beobachtete Effekt durch eine Anpassung von Datenabbildung und Datenzugriff beseitigt. Für die Realisierung der Anpassung von Abbildung und Zugriff ergibt sich die Forderung nach einer expliziten Abbildung von Daten auf Seiten.

- Die Relevanz der Ausrichtung von Vektoren auf Seitengrenzen hängt von der Art der Anwendung ab. Um dies zu verdeutlichen, wurden zwei Anwendungen mit diesbezüglich sehr unterschiedlichem Verhalten gegenübergestellt. Bei der gutartigen Anwendung ergibt sich ohne Ausrichtung der Vektoren auf Seitengrenzen der Effekt des *page thrashing*; dabei liegen je zwei Prozessoren bezüglich einer Seite im Konflikt, der dabei verursachte relative Overhead ist mit etwa 25 Prozent bereits signifikant. Für die zweite Anwendung ist der ohne Ausrichtung der Vektoren verursachte relative Overhead um etwa 750 Prozent erhöht. Als Ursache ergibt sich wieder *page thrashing*, diesmal bezüglich jeweils einer Seite, auf die ein Prozessor mehrfach schreiben will, während alle anderen Prozessoren mehrfach von dieser Seite lesen wollen. Aus diesem Ergebnis resultiert die Forderung, die Ausrichtung von Vektoren auf Seitengrenzen in die explizite Datenabbildung einzubeziehen.
  Die beobachtete Erhöhung des Overhead ohne Ausrichtung der Vektoren auf Seitengrenzen gründet sich auf die Aufteilung von Seiten über verschiedene Vektoren. In den zuvor betrachteten Fällen war das Verhältnis von Vektorgröße zur Seitengröße gerade so gewählt, daß alle Seiten aufgeteilt wurden. Bei kleineren Seitengrößen wird nur ein Teil der Seiten aufgeteilt, und es ist zu erwarten, daß die negativen Effekte auch ohne Ausrichtung verschwinden. Deswegen wurden zwei weitere Meßreihen durchgeführt, bei denen die Seitengröße als Untersuchungsparameter reduziert wurde. Bei beiden vorher betrachteten Anwendungen traten die negativen Effekte ohne Ausrichtung bei sehr kleinen Seiten nicht mehr auf. Daraus läßt sich als Alternative zur Ausrichtung von Vektoren auf Seitengrenzen die Forderung nach sehr kleinen Seiten ableiten.

Die Abbildung von Daten auf Seiten beeinflußt die Datenlokalität bei der Programmausführung; daneben nimmt das *thread*-Scheduling Einfluß sowohl auf die Datenlokalität als auch auf die Lastbalancierung. Tab. 4 beschreibt Effekte und Ursachen für das *thread*-Scheduling bei einfach parallelen Schleifen: Der starke Einfluß des Datenzugriffsverhaltens einer Anwendung auf ein geeignetes *thread*-Scheduling bewirkt, daß für unterschiedliche Anwendungen jeweils andere Scheduling-Algorithmen zu favorisieren sind. Es ergeben sich Abhängigkeiten zum Beispiel von der Datenzugriffsstruktur einer Anwendung oder vom Verhältnis Vektorgröße zu Seitengröße. Damit finden sich charakteristische Problemstellungen zum einen für charakteristische Anwendungen und zum anderen, unabhängig von der Art der Anwendung, zum Beispiel für Seiten mit mehreren Vektoren.
Bei der Auswahl der Anwendungsbeispiele wurde versucht, jeweils charakteristische Problemstellungen herauszuarbeiten. Die in der folgenden Liste enthaltenen Unterpunkte beschreiben charakteristische Problemstellungen und die dabei beobachteten Effekte und Ursachen zum *thread*-Scheduling:

- Die bei Seiten mit mehreren Vektoren (Multi-Vektor-Seiten) resultierende Problemstellung ist in [GrWi93] bereits theoretisch untersucht worden: Lastadaptive Scheduling-Algorithmen sind in der herkömmlichen Form ungeeignet und nur in Kombination mit *Loop-Blocking* effizient. Die Ursache liegt in der Gruppierung der Vektoren zu Seiten: Nur die Beachtung dieser Gruppierung durch das *Loop-Blocking* erlaubt eine effiziente Ausführung. Dieses theoretisch gefundene Ergebnis

Untersuchungskriterium	Effekt	Ursache
Thread-Scheduling		
- allgemein	kein universell geeigneter Algorithmus	Abhängigkeit von - Anwendung - Vektorgröße/Seitengröße
- Multi-Vektor-Seiten	lastadaptive Algorithmen sind in der herkömmlichen Form ungeeignet und nur in Kombination mit Loop-Blocking effizient	nur Loop-Blocking beachtet Gruppierung der Vektoren zu Seiten
- Affinitäten beim Datenzugriff	Beachtung der Affinitäten bringt deutliche Vorteile	Affinity-Scheduling erlaubt Wiederverwendung der Daten
- Variable Index-adressierung	beste Effizienz bei Align-Scheduling, bei unterschiedlicher Indizierung aber nur geringe Effizienz	keine systematische Wiederverwendung von Daten möglich

Tab. 4. Das Thread-Scheduling bei einfach parallelen Schleifen: Effekte und Ursachen

wird durch die in Kapitel 7 beschriebenen Simulationsuntersuchungen bestätigt. Für Multi-Vektor-Seiten ergibt sich damit die Forderung nach der Anwendung des *Loop-Blocking* auf die bekannten lastadaptiven Algorithmen.

- Schleifennester mit Affinitäten beim Datenzugriff stellen eine charakteristische Problemstellung dar. Es wurden zwei Anwendungsbeispiele ausgewählt, deren Affinitätsbeziehungen verschieden sind. Eine effiziente Ausführung wird in beiden Fällen nur bei Beachtung der Affinitäten erzielt, wodurch eine Wiederverwendung der Daten ermöglicht wird; dies kann durch ein *Align-Scheduling* oder ein *Affinity-Scheduling* geleistet werden. Da ein *Align-Scheduling* mit vergleichsweise höherem Scheduling-Overhead verbunden ist, führt das *Affinity-Scheduling* zum günstigsten Zeitverhalten. Bei der Untersuchung von triangularen Schleifennestern wurde ein Widerspruch zu Ergebnissen in [LaPr91] gefunden; in [LaPr91] wird das *Affinity Block-Scheduling* favorisiert, während hier das *Affinity Cyclic-Scheduling* zu der besten Effizienz führt. Die Ursache für diesen Widerspruch liegt in der Datenabbildung: Nur bei einer dem zyklischen Scheduling angepaßten Datenabbildung, die ebenfalls zyklisch sein muß, kann das *Cyclic-Scheduling* dem *Block-Scheduling* überlegen sein; da in [LaPr91] keine explizite Datenabbildung durchgeführt wird, ergibt sich dort eine Favorisierung des *Affinity Block-Scheduling*.
- Affinitäten beim Datenzugriff liegen nicht bei allen Anwendungen vor. Deswegen wurde für weitere Untersuchungen ein Schleifennest ausgewählt, dessen Datenzugriffe mit variabler Indexadressierung erfolgen, so daß keine Affinitätsbeziehungen genutzt werden können. In allen betrachteten Fällen hat das *Align-Scheduling* zur besten Effizienz geführt; bei mehrfacher variabler Indexadressierung war die Effizienz jedoch auch beim *Align-Scheduling* eher unbefriedigend. Die Ursache liegt darin, daß durch das Fehlen von Affinitäten keine systematische Wiederverwendung von Daten möglich ist.

Um die besonderen Gegebenheiten bei der Ausführung von geschachtelten parallelen Schleifen in VSM-Systemen zu beleuchten, wurde ein Repräsentant dieser Anwendungsklasse untersucht. Tab. 5 beschreibt die Effekte und Ursachen für das *thread*-Scheduling bei geschachtelten parallelen Schleifen:

- Als eine charakteristische Problemstellung kann bei der Parallelisierung nur der äußeren Schleife ein Speicherengpaß auftreten, der durch einen hohen Datenbedarf je Prozessorelement verursacht wird. Dabei können Daten in aufeinanderfolgenden Schleifeniterationen deswegen nicht wiederverwendet werden, weil sie aufgrund eines Speicherengpasses invalidiert wurden. Durch die zusätzliche Parallelisierung einer weiter innen liegenden Schleife (Verschiebung der Parallelisierung nach innen) kann der Datenbedarf je Schleifeniteration und damit je Prozessorelement reduziert werden, so daß der Speicherengpaß entschärft werden kann.
- Um die Auswirkungen einer Verschiebung der Parallelisierung nach innen auch für den Fall zu untersuchen, daß kein Speicherengpaß vorliegt, wurde eine weitere Meßreihe durchgeführt. Der hohe Datenbedarf je Prozessorelement führt jedoch auch bei hinreichend großem Speicher zu negativen Effekten, d.h. zu häufigen *page faults* und damit zu geringer Effizienz. Auch hier kann eine Verschiebung

Untersuchungskriterium	Effekt	Ursache
Thread-Scheduling bei geschachtelten parallelen Schleifen		
- Speicherengpaß	Auflösung von Speicherengpaß durch Verschiebung der Parallelisierung nach innen	Verschiebung nach innen kann den Datenbedarf je Iteration reduzieren
- häufige page faults durch hohen Datenbedarf	Verschiebung der Parallelisierung nach innen reduziert die Anzahl von page faults	Verschiebung nach innen kann den Datenbedarf je Iteration reduzieren
- gleichzeitige Seitenanforderung sehr vieler Prozesse einer parallelen Schleife	Auflösung von Kommunikationsengpaß durch Verteilung der Parallelisierung über mehr Schleifen	Verteilung über mehr Schleifen reduziert die Anzahl der Prozesse, die innerhalb einer Schleife gleichzeitig arbeiten
- massive Parallelisierung einer inneren Schleife	Auflösung von Thread-Scheduler-Engpaß durch Verschiebung der Parallelisierung nach außen	Verschiebung nach außen erhöht die Granularität der threads und verteilt das thread-Scheduling

Tab. 5. Das Thread-Scheduling bei geschachtelten parallelen Schleifen: Effekte und Ursachen

der Parallelisierung nach innen den Datenbedarf je Iteration beziehungsweise je Prozessorelement reduzieren und so die Anzahl von *page faults* verringern.

- Die beiden zuvor betrachteten Meßreihen resultieren in einer Forderung zur Verschiebung der Parallelisierung nach innen. Eine maximale Verschiebung nach innen, d.h. die Parallelisierung nur einer innen liegenden Schleife, entspricht einer Konzentration aller parallelen Prozesse in dieser Schleife. Dabei tritt ein anderer negativer Effekt auf: Alle Prozesse der parallelen Schleife fordern gleichzeitig jeweils dieselbe Seite an, und es entsteht ein Kommunikationsengpaß. Bei einer Verteilung der Prozesse über zwei parallele Schleifen wird die Anzahl der Prozesse, die innerhalb einer Schleife gleichzeitig arbeiten, reduziert, und der Engpaß wird aufgeweitet.
- Um die Auswirkungen einer maximalen Verschiebung der Parallelisierung nach innen auch ohne Kommunikationsengpaß zu untersuchen, wurden in einer weiteren Meßreihe die entsprechenden Daten bereits zu Beginn der Ausführung allen Prozessorelementen zur Verfügung gestellt. Auch bei dieser Untersuchung hat sich gezeigt, daß eine massive Parallelisierung der mittleren Schleife einen Nachteil birgt: Die Granularität der vom *thread*-Scheduler zugeteilten Arbeitspakete wird so klein, daß

der *thread*-Scheduler zur kritischen Ressource und damit zum Engpaß wird. Die Verschiebung der Parallelisierung nach außen vergrößert die Granularität und bewirkt eine Verteilung des *thread*-Scheduler auf zwei Ebenen.

Die aus den Untersuchungen resultierenden Forderungen für das *thread*-Scheduling sind die Verschiebung der Parallelisierung nach innen beziehungsweise nach außen und die Verteilung der Parallelisierung über die paralleliserbaren Schleifen. Tatsächlich ergab sich in allen betrachteten Fällen die maximale Effizienz bei einer etwa gleichmäßigen Verteilung der Parallelisierung über die Schleifen.

Der Überblick über die Ergebnisse und deren Einbettung in den Gesamtkontext der Ablaufplanung verdeutlichen das logische Aufeinanderfolgen und die Vollständigkeit der beschriebenen Untersuchungen. Das Verständnis um die beschriebenen Effekte und Ursachen läßt verallgemeinernde Folgerungen zu. Das Resultat ist eine Sammlung von Bausteinen, durch deren Einsatz bei der Gestaltung und Nutzung zukünftiger *virtual-shared-memory*-Systeme bisherige Lücken im Fundament geschlossen werden können.

9. Zusammenfassung und Ausblick

Die Entwicklung von zukunftsweisenden Konzepten im Supercomputing bezieht sich heute in vielen Fällen auf Rechner mit virtuell gemeinsamem Speicher: Der Status existierender Implementierungen reicht von prototypischen Forschungsprojekten zur Verbesserung und Erweiterung der verwendeten Modelle bis zu kommerziellen Produkten. Zur Zeit konzentriert sich ein großer Teil der Anstrengungen auf die effiziente Realisierung verschiedener VSM-Modelle; während vielen an der Forschung orientierten Projekten das von K. Li entwickelte VSM-Modell zugrunde liegt, basieren die kommerziellen Produkte (zum Beispiel KSR 1 und Cray T3D) auf neuartigen Modellen. Allen Modellen zum *virtual shared memory* ist gegenüber dem auf Rechnern mit physikalisch verteiltem Speicher verbreiteten *message-passing*-Modell jedoch ein entscheidender Vorteil gemeinsam: Die vergleichsweise komfortable Programmierung auf der Basis des logisch gemeinsamen Speichers. Das VSM-Modell erhält durch diesen Vorteil eine strategische Bedeutung für den Erfolg von massiv-parallelen Rechnern. Die Software-Unterstützung der verwendeten Konzepte ist - wie bei vielen neuartigen Entwicklungen - jedoch noch nicht ausgereift; heute ist die entsprechende Software existierenden Strategien konzeptuell nachempfunden und nicht auf die neuen, zusätzlichen Anforderungen abgestimmt. Die deswegen bei VSM-Rechnern zum Teil nur geringe Effizienz stellt bisher ein Hemmnis für die allgemeine Akzeptanz dar, und die Entwicklung geeigneter Algorithmen für die System-Software wird die Bedeutung von VSM-Rechnern auf dem hart umkämpften Markt für Supercomputer entscheidend beeinflussen.

Zu den nicht ausgereiften Teilen der System-Software bei VSM-Rechnern gehört die Ablaufplanung. Die Ablaufplanung kann insbesondere die Lokalität von Datenzugriffen beeinflussen, die bei allen Rechnern mit verteiltem Speicher von entscheidender Bedeutung für die Effizienz der Ausführung ist. Die Ergebnisse dieser Arbeit weisen einen erheblichen Einfluß der Ablaufplanung auf die Lokalität nach:

- Die Abbildung von Daten auf Seiten beeinflußt den Effekt des *false sharing*.
- Die initiale Verteilung der Seiten auf die Prozessorelemente kann Kommunikationsengpässe hervorrufen beziehungsweise verhindern.
- Die Anpassung des *thread*-Scheduling an das Speicherzugriffsverhalten beeinflußt ebenfalls den Effekt des *false sharing*.
- Die Aufteilung von parallelen Schleifen beziehungsweise die Parallelisierung von zwei oder mehr Schleifen kann das Kommunikationsaufkommen vermindern, Speicherengpässe oder Kommunikationsengpässe vermeiden und den Scheduling-Overhead reduzieren.

Die Untersuchung der propagierten Strategien basiert auf Simulationen. Zu diesem Zweck wurde ein Ereignis-gesteuerter Simulator entwickelt, der drei unabhängige Komponenten für das VSM-Modell, das Multiprozessormodell und das Arbeitslastmodell implementiert. Das VSM-Modell basiert auf dem statisch-verteilten-Manager-Algorithmus nach K. Li [Li86]; in Erweiterung dieses Modells sind die explizite Abbildung von Daten auf Seiten, die schwache Kohärenz und die *prefetch*-Operation untersucht worden. Das Multiprozessormodell kann verschiedene Rechnerkonfigurationen nachbilden, die sich

zum Beispiel bezüglich der Anzahl der Prozessorelemente, der Größe des Lokalspeichers und der Größe der Seiten unterscheiden. Das Arbeitslastmodell verwendet die Speicherzugriffsmuster realer Programmkerne; auf diese Weise kann garantiert werden, daß das Speicherzugriffsverhalten der Realität entspricht. Anhand dieser Modellierung eines VSM-Systems erlaubt der Simulator die alternative Untersuchung von Algorithmen zur Ablaufplanung.

Neben den zeitlichen Anteilen von diskreten Prozeßzuständen am gesamten simulierten Zeitintervall einer Programmausführung generiert der Simulator Informationen über die zeitliche Abfolge der Prozeßzustandswechsel und der Speicherzustandswechsel mit Bezug auf das jeweilige Prozessorelement; diese Informationen werden in einer *Trace*-Datei abgelegt. Die Größe dieser *Trace*-Datei und die Komplexität der enthaltenen Daten erfordern eine leistungsfähige Visualisierung. Zu diesem Zweck ist der Simulator in die Werkzeugumgebung PARtools eingebunden, deren Komponente PARvis speziell für die Visualisierung des Speicherverhaltens erweitert wurde. In Verbindung mit dieser Visualisierung kann ein direkter Bezug zwischen dargestellten Zustandswechseln von Prozessen beziehungsweise Speicher zu den Operationen der zugrundeliegenden Arbeitslast erfolgen; auf diese Weise kann das Verständnis von negativen Effekten gefördert werden, und notwendige Maßnahmen zur Verbesserung können eingeleitet werden.

Der erste Teil der durchgeführten Untersuchungen bezieht sich auf die Abbildung von Daten auf Seiten: Für verschiedene Arbeitslasten wurden der bisher üblichen, spaltenorientierten Datenabbildung ohne Ausrichtung von Vektoren auf Seitengrenzen jeweils geeignete Abbildungen gegenübergestellt. Ein Vergleich der Messungen ergab einen unterschiedlichen Einfluß der herkömmlichen Datenabbildung auf die Effizienz in Abhängigkeit von der jeweiligen Arbeitslast: Der durch *page faults* verursachte relative Overhead *(page-waiting* und *busy-waiting)* lag für verschiedene Arbeitslasten bei 45, 900 bzw. 10,000 Prozent. Durch eine explizite Datenabbildung in jeweils geeigneter Form konnte die Effizienz in allen betrachteten Fällen verbessert werden: Selbst die bei herkömmlicher Datenabbildung sehr hohen Effizienzverluste konnten deutlich gesenkt werden. Aus den durchgeführten Simulationen ergeben sich die folgenden Forderungen an die Ablaufplanung:

- In VSM-Systemen müssen Datenabbildung und Datenzugriff aus Effizienzgründen einander angepaßt sein; die herkömmliche spaltenorientierte Datenabbildung führt zum Beispiel bei zeilenorientiertem Datenzugriff zu erheblichen Effizienzverlusten. Für wechselnde Zugriffsmuster sind durch eine dynamische, explizite Datenabbildung Vorteile gegenüber der herkömmlichen Datenabbildung zu erwarten.
- Darüber hinaus kann - in Abhängigkeit von der Arbeitslast - die Ausrichtung von Vektoren auf Seitengrenzen zu signifikanten Effizienzsteigerungen führen. Für diesen Effekt ergibt sich eine weitere Abhängigkeit vom Verhältnis Vektorgröße zu Seitengröße; nur bei relativ zur Vektorgröße sehr kleinen Seiten kann der Effekt vernachlässigt werden.

Ganz spezielle Anforderungen an die Schleifenpartitionierung ergeben sich bei Seiten mit mehreren Vektoren: Für diesen Fall sind herkömmliche Algorithmen zum *thread*-Scheduling und das bereits theoretisch verifizierte *Loop-Blocking* [GrWi93] untersucht

worden. Es hat sich gezeigt, daß die lastadaptiven Algorithmen *Self-Scheduling*, *Guided Self-Scheduling* und *Factoring* [TaYe86, PoKu87, HSF91] in der üblichen Form zu so geringer Effizienz führen, daß sie als ungeeignet bezeichnet werden können; nur in Verbindung mit dem *Loop-Blocking*-Algorithmus können die lastadaptiven Algorithmen effizient sein: Ohne das *Loop-Blocking* haben selbst bei einer extrem unbalancierten Arbeitslast die nicht-lastadaptiven Algorithmen bei günstiger Parametrisierung zu sehr viel besserer Effizienz geführt. Zunächst hat sich allein das *Chunk-Scheduling* als geeigneter lastadaptiver Algorithmus erwiesen. Die Ursache für die vergleichsweise bessere Effizienz der nicht-adaptiven Algorithmen und des *Chunk-Scheduling* lagen in der Anpassung der Partitionierung an die Gruppierung der Vektoren zu Seiten. Diese Strategie verfolgt auch der neuartige *Loop-Blocking*-Algorithmus: Die Gruppierung der Iterationen bei der Partitionierung erfolgt analog zur Gruppierung der Vektoren durch deren Abbildung auf Seiten. In Verbindung zum Beispiel mit dem vorher ineffizienten *Factoring* ergibt sich eine mindestens ebenso hohe Effizienz wie bei den bisher günstigsten Algorithmen. Ähnliche Ergebnisse sind auch für das *Loop-Blocking* in Verbindung mit dem *Self-Scheduling* und dem *Guided Self-Scheduling* zu erwarten. Es ist jedoch zu beachten, daß bei den adaptiven Algorithmen mit *Loop-Blocking* die Lastbalancierung um den Faktor der Anzahl von Vektoren pro Seite schlechter ist als bei den originären Versionen der adaptiven Algorithmen.

Alle weiteren Untersuchungen beschäftigen sich mit dem *thread*-Scheduling für charakteristische Schleifennester. Für Schleifennester mit wiederholter Ausführung einer parallelen Schleife hat sich das für *multicache*-Rechner entwickelte *Affinity-Scheduling* [MaLe92] als bester Algorithmus gezeigt. Für triangulare Schleifennester mit Affinität führte das *Affinity Cyclic-Scheduling* zur effizientesten Ausführung: Dieses Ergebnis steht im Widerspruch zu Untersuchungen in [LaPr91], wo das *Affinity Block-Scheduling* favorisiert wird. Die Ursache für diesen Widerspruch liegt in der Datenabbildung: Nur bei einer dem zyklischen Scheduling angepaßten Datenabbildung, die ebenfalls zyklisch sein muß, kann das *Cyclic-Scheduling* dem *Block-Scheduling* überlegen sein; da in [LaPr91] keine explizite Datenabbildung durchgeführt wird, ergibt sich dort eine Favorisierung des *Affinity Block-Scheduling*. Für Schleifennester mit wechselnder Indexadressierung hat sich das *Align-Scheduling* als bester Algorithmus erwiesen. Dieses Ergebnis stellt sich zum Beispiel dann ein, wenn alle Datenzugriffe innerhalb einer parallelen Schleife über dasselbe Indexfeld adressiert werden; bei Zugriffen über verschiedene Indexfelder sind die Vorteile durch das *Align-Scheduling* nur noch gering, und die Effizienz der Ausführung ist eher unbefriedigend.

Wichtige Ergebnisse dieser Arbeit resultieren aus Untersuchungen mit geschachtelten parallelen Schleifen, d.h. Schleifennester mit zwei oder mehr parallelen Schleifen. Während bei Rechnern mit global gemeinsamem Speicher - bei hinreichender Parallelität in der äußeren parallelisierbaren Schleife - ausschließlich diese Schleife parallel ausgeführt werden sollte, können sich bei *virtual-shared-memory*-Rechnern vollkommen andere Verhältnisse ergeben: Bei der betrachteten Matrixmultiplikation ergibt sich einerseits für die Parallelisierung der äußeren Schleife ein hohes Kommunikationsaufkommen bzw. ein Speicherengpaß und andererseits für die Parallelisierung der mittleren Schleife ein Kom-

munikationsengpaß und ein hoher *busy-waiting*-Anteil. In allen betrachteten Fällen ist die Effizienz der Ausführung maximal bei der Parallelisierung beider Schleifen und einer etwa gleichmäßigen Verteilung der Prozessoren über die Schleifen. In den meisten betrachteten Fällen ist der erzielte Effizienzgewinn gegenüber der Parallelisierung nur einer Schleife erheblich, Nachteile aufgrund der mehrfachen Parallelisierung sind nicht beobachtet worden.

Die in dieser Arbeit beschriebenen Untersuchungen resultieren in Forderungen für die Ablaufplanung in *virtual-shared-memory*-Systemen, die zum Teil erhebliche Effizienzsteigerungen bewirken können; dabei sind verschiedene Bereiche der Ablaufplanung voneinander abhängig: Die Datenabbildung stellt eine Grundlage für die darauf folgenden Abläufe dar, selbiges gilt für die Verteilung der Seiten auf die Prozessorelemente. Auf dieser Basis ergeben sich Forderungen für ein geeignetes *thread*-Scheduling. Ebenso wie die Datenabbildung und die Verteilung der Seiten hängt auch ein geeignetes *thread*-Scheduling von der jeweiligen Anwendung ab. Insgesamt ergibt sich die Notwendigkeit der Erweiterung bestehender Strategien und damit der Adaption an die Eigenheiten von *virtual-shared-memory*-Systemen.

Die in dieser Arbeit gewonnenen Einsichten beziehen sich zum Teil auf charakteristische Arbeitslasten und führen zu jeweils verschiedenen Strategien zur Erlangung einer bestmöglichen Effizienz. Die in dieser Arbeit beschriebenen Untersuchungen beschränken sich auf ein Modell zum virtuell gemeinsamen Speicher; Entwicklungen mit neuartigen, möglicherweise überlegenen Modellen eröffnen ein weites Feld für ähnliche Untersuchungen zur Ablaufplanung. Auch hier kann der bestehende Simulator als Basis benutzt werden, da eine weitgehende Modularität bezüglich anderer *virtual-shared-memory*-Modelle besteht.

Da es bisher nur wenige Untersuchungen zur Ablaufplanung bei *virtual-shared-memory*-Rechnern gibt, zielt diese Arbeit auch auf die Schaffung von Grundlagen; deswegen werden die Betrachtungen im Bereich des Scheduling auf den *thread*-Scheduler bezogen, und alle Ergebnisse gelten für den Singleprogramming-Betrieb. In Kombination mit den Ergebnissen zur Datenabbildung ergibt sich eine gute Grundlage für weiterführende Arbeiten zur Untersuchung der Ablaufplanung bei *virtual-shared-memory*-Rechnern im Multiprogramming-Betrieb. Hier kann das Job-Scheduling beziehungsweise die Kombination von aktuellen Ergebnissen zum Scheduling auf *global-shared-memory*-Rechnern im Multiprogramming-Betrieb [Nag93] mit Ergebnissen aus dieser Arbeit für das *thread*-Scheduling im Vordergrund stehen.

Als wichtig erscheint an dieser Stelle die Übertragung der gewonnenen Erkenntnisse in die reale Welt der Parallelrechner, d.h. die Implementierung der beschriebenen Strategien für reale Rechner. In vielen Fällen endet die Entwicklung von neuen Strategien im Bereich der Ablaufplanung mit deren theoretischer Untersuchung oder mit der Implementierung im Rahmen von Simulationen; auch von den in Kapitel 4 beschriebenen Algorithmen sind nicht alle auf realen Maschinen implementiert worden. Hier wäre zu wünschen, daß die in dieser Arbeit gewonnenen Einsichten durch die Implementierung der abgeleiteten Verfahren bei realen Parallelrechnern in die wünschenswerten Leistungsgewinne umgesetzt werden.

Literatur

[AlGo89] G.S. Almasi and A. Gottlieb, Highly parallel computing, The Benjamin/Cummings Publishing Company, Redwood City, CA, 1989.

[And91] G.R. Andrews, Concurrent programming, The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[AnSc83] G.R. Andrews and F.B. Schneider, Concepts and notations for concurrent programming, *ACM Comp. Surv. 15 (1)* (1983) 3-43.

[Arn93] A. Arnold, PARvis - eine X-basierte Umgebung zur Visualisierung von Prozeßwechseln in Multiprozessorsystemen, Forschungszentrum Jülich, Jül-2848, 1993.

[ANP86] V.K. Arvind, R. Nikhil, and K. Pingali, I-structures: Data structures for parallel computing, in *Proc. Workshop Graph Reduction*, Santa Fe, NM, 1986, 336-369.

[BKLR90] F. Barriuso, J. Kohn, S. LaCroix, and S. Reinhard, Improvements to non-dedicated performance of autotasking programs on Cray Y-MP computer systems, in *Proc. of Cray User Group Meeting (Fall)*, 1990, 295-300.

[BBN89] BBN TC2000^{TM} product summary, BBN Advanced Computers Inc, (1989).

[BBN90] BBN TC2000, Parallel computing: Past, present, and future, BBN Advanced Computers Inc, (1990).

[BBZ88] M. Beltrametti, K. Bobey, and J.R. Zorbas, The control mechanism for the Myrias parallel computer system, *Computer Architecture News 16 (4)* (1988) 21-30.

[BCZ90a] J.K. Bennet, J.B. Carter, and W. Zwaenepoel, Munin: Distributed shared memory based on type-specific memory coherence, *ACM SIGPLAN 25 (3)* (1990) 168-176.

[Ber92] R. Berrendorf, Memory access in shared virtual memory, in *L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, Parallel Processing: CONPAR 92 - VAPP V, LNCS 634, Springer-Verlag*, Berlin, 1992, 785-786.

[BGNP93] R. Berrendorf, M. Gerndt, W.E. Nagel, and J. Prümmer, SVM FORTRAN, Forschungszentrum Jülich, KFA-ZAM-IB-9322, 1993.

[Bie88] M. Bieterman, Microtasking general purpose partial differential equation software on the Cray X-MP, *Journal of Supercomputing 2 (4)* (1988) 381-414.

[BNR89] R. Bisiani, A. Nowatzyk, and M. Ravishankar, Coherent shared memory on a distributed memory machine, in *Proc. 1989 Int. Conf. Parallel Processing Vol. I*, St. Charles, IL, 1989, 133-141.

[Bla90] D.L. Black, Scheduling support for concurrency and parallelism in the Mach operating system, *IEEE Computer 23 (5)* (1990) 35-43.

[BKP93] F. Bodin, L. Kervella, and T. Priol, FORTRAN-S: A FORTRAN interface for shared virtual memory architectures, in *Proc. Supercomputing '93*, Portland, OR, 1993, 274-283.

[BFS89] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott, Simple but effective techniques for NUMA memory management, *Operating Systems Review 23 (5)* (1989) 19-31.

[BoIs91] L. Borrmann and P. Istavrinos, Store coherency in a parallel distributed-memory machine, in *A. Bode, editor, Distributed Memory Computing, LNCS 487, Springer-Verlag*, Berlin, 1991, 32-41.

[CMZ92] B.M. Chapman, P. Mehrotra, and H. Zima, Vienna FORTRAN - a FORTRAN language extension for distributed memory multiprocessors, in: *J. Saltz, P. Mehrotra, editors, Languages, compilers and run-time environments for distributed memory machines, North Holland*, 1992, 39-62.

[CPME93] Committee on Physical, Mathematical, and Engineering Sciences, Federal Coordinating Council for Science, Engineering, and Technology, Grand Challenges 1993: High performance computing and communications, The FY 1993 U.S. Research and Development Program, Report of the Committee to Supplement the Presidents Fiscal Year 1993 Budget, Office of Science and Technology Policy, Washington, 1992.

[Com89] Supercomputer hardware: An update of the 1983 report's summary and tables, *IEEE Computer 22 (11)* (1989) 63-68.

[CoFo89] A.L. Cox and R.J. Fowler, The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM, in *Proc. 12th ACM Symp. Operating Systems Principles*, The Wigwam Litchfield Park, AR, 1989, 32-44.

[Cra0222] Cray multitasking programmer's manual, Cray Research Inc, SR-0222 F (1989).

[Cra3074] Cray CF77 compiling system, Vol. 4: Parallel processing guide, Cray Research Inc, SG-3074 (1990).

[DMK92] E. Darnell, J.M. Mellor-Crummey, and K. Kennedy, Automatic software cache coherence through vectorization, Rice University, Technical Report CRPC-TR92197, 1992.

[Den68] P.J. Denning, The working set model for program behavior, *CACM 11 (5)* (1968) 323-333.

[DeHo66] J.B. Dennis and E.C. Van Horn, Programming semantics for multiprogrammed computations, *CACM 9 (3)* (1966) 143-155.

[Dij68] E.W. Dijkstra, Cooperating sequential processes, in *F. Gennys, editor, Programming Languages, Academic Press*, New York, NY, 1968, 43-112.

[DiIy90] R.T. Dimpsey and R.K. Iyer, Performance degradation due to multiprogramming and system overheads in real workloads: Case study on a shared memory multiprocessor, in *Proc. 1990 Int. Conf. Supercomputing*, Amsterdam, The Netherlands, 1990, 227-238.

[Dun92] T.H. Dunigan, Kendall Square multiprocessor: Early experiences and performance, Mathematical Sciences Section, Oak Ridge National Lab., Technical Report ORNL/TM-12065, 1992.

[FeRu90] D.G. Feitelson and L. Rudolph, Distributed hierarchical control for parallel processing, *IEEE Computer 23 (5)* (1990) 65-77.

[FlPo89] B.D. Fleisch and G.J. Popek, Mirage: A coherent distributed shared memory design, in *Proc. 12th ACM Symp. Operating Systems Principles*, The Wigwam Litchfield Park, AR, 1989, 211-223.

[Fly66] M.J. Flynn, Very high-speed computing systems, *Proc. IEEE 54 (12)* (1966) 1901-1909.

[Fox90] G. Fox et al., FORTRAN D language specification, Rice University, Technical Report Rice COMP TR90-141, 1990.

[GFF89] A. Garcia, D. Foster, and R. Freitas, The advanced computing environment multiprocessor workstation, IBM Research Division, Research Report RC-14419, 1989.

[GSS87] E.F. Gehringer, D.P. Siewiorek, and Z. Segall, Parallel processing, the Cm* experience, Digital Press, 1987.

[Ger89] M. Gerndt, Automatic parallelization for distributed-memory multiprocessing systems, PhD thesis, Universität Bonn, 1989.

[Ger92c] M. Gerndt, Private communications, 1992.

[Gil88] W. Giloi, SUPRENUM: A trendsetter in modern supercomputer development, *Parallel Computing 7* (1988) 283-296.

[GHSS91] W.K. Giloi, C. Hastedt, F. Schoen, and W. Schroeder-Preikschat, A distributed implementation of shared virtual memory with strong and weak coherence, in *A. Bode, editor, Distributed Memory Computing, LNCS 487, Springer-Verlag*, Berlin, 1991, 23-31.

[Got84] A. Gottlieb et al., The NYU Ultracomputer - designing an MIMD parallel computer, *IEEE Trans. Comp. C-32 (2)* (1984) 175-189.

[GrWi93] E.D. Granston and H.A.G. Wijshoff, Managing pages in shared virtual memory systems: Getting the compiler into the game, in *Proc. 1993 Int. Conf. Supercomputing*, Tokyo, Japan, 1993, 11-20.

[GTU91] A. Gupta, A. Tucker, and S. Urushibara, The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications, in *Proc. 1991 ACM SIGMETRICS Conf. Measurement Modelling Comp. Syst.*, San Diego, CA, 1991, 120-132.

[HaHa89] S. Haridi and E. Hagersten, The cache coherence protocol of the Data Diffusion Machine, in *Proc. PARLE 89 Par. Arch. Lang. Europe Vol. I: Par. Arch. Proc.*, Eindhoven, The Netherlands, 1989, 1-18.

[Her90] M. Herlihy, A methodology for implementing highly concurrent data structures, *SIGPLAN Notices 25 (3)* (1990) 197-206.

[HPF93] High Performance FORTRAN language specification, High Performance FORTRAN Forum, Rice University, 1993.

[Hoa74] C.A.R. Hoare, Monitors: An operating system structuring concept, *CACM 17 (10)* (1974) 549-557.

[Hos91] F. Hoßfeld, "Grand Challenges" - wie weit tragen die Antworten des Supercomputing? in *Proc. Arbeitsgespräch Physik und Informatik - Informatik und Physik*, Munich, Germany, 1991.

[HoMo87] C.D. Howe and B. Moxon, How to program parallel computers, *IEEE Spectrum 24 (9)* (1987) 36-41.

[HSF91] S. Flynn Hummel, E. Schonberg, and L.E. Flynn, Factoring: A practical and robust method for scheduling parallel loops, in *Proc. Supercomputing '91*, Albuquerque, NM, 1991, 610-619.

[HwBr84] K. Hwang and F.A. Briggs, Computer architecture and parallel processing, McGraw-Hill Book Company, New York, NY, 1984.

[IBM23] IBM Parallel FORTRAN language and library reference, IBM, SC23-0431 (1988).

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower, An automatic and symbolic parallelization system for distributed memory parallel computers, in *Proc. 5th Dist. Mem. Comp. Conf.*, Charleston, SC, 1990, 1105-1114.

[Int91] Intel, Paragon XP/S product overview, Intel Corporation (1991).

[Joh87] E.E. Johnson, The virtual port memory multiprocessor architecture, PhD thesis, New Mexico State University, 1987.

[Joh88] E.E. Johnson, Completing an MIMD multiprocessor taxonomy, *Computer Architecture News 16 (3)* (1988) 44-47.

[Kar87] A.H. Karp, Programming for parallelism, *IEEE Computer 20 (5)* (1987) 43-57.

[Kat85] R. Katz et al., Implementing a cache consitency protocol, in *Proc. 12th Annual Int. Symp. Comp. Arch.*, 1985, 276-283.

[KSR92] Kendall Square Research, Technical summary, 1992.

[KMT91] K. Kennedy, K.S. McKinley, and C.-W. Tseng, Analysis and transformation in the ParaScope editor, in *Proc. 1991 Int. Conf. Supercomputing*, Cologne, Germany, 1991, 433-447.

[KeZi89] K. Kennedy and H.P. Zima, Virtual shared memory for distributed-memory machines, in *Proc. Fourth Conf. Hypercubes, Concurrent Computers and Applications Vol. 1,* Monterey, CA, 1989, 361-366.

[KeSc93] R.E. Kessler and J.L. Schwarzmeier, Cray T3D: A new dimension for Cray Research, to appear in *COMPCON '93*, 1993.

[KnNa86] S. Knecht and W.E. Nagel, Multitasking on Cray X-MP/22 - experiences in macrotasking and microtasking, in *Proc. of Cray User Group Meeting (Fall)*, 1985, 153-166.

[Knu73] D.E. Knuth, The art of computer programming, Vol. III, Addison Wesley, Reading, MA, 1973.

[KVW87] A.M. Kobos, R.E. VanKooten, and M.A. Walker, The Myrias parallel computer system, in *H.J.J. Riele, T.J. Dekker, and H.A. Vorst, editors, Algorithms and Applications on Vector and Parallel Computers, Elsevier Science Publishers B.V., North-Holland*, Amsterdam, The Netherlands, 1987, 103-118.

[KMR90] C. Koelbel, P. Mehrotra, and J. Van Rosendale, Supporting shared data structures on distributed memory architectures, *SIGPLAN Notices 25 (3)* (1990) 177-186.

[KMSB90] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman, Parallel loops on distributed machines, in *Proc. 5th Dist. Mem. Comp. Conf.*, Charleston, SC, 1990, 1097-1104.

[KrWe85] C.P. Kruskal and A. Weiss, Allocating independent subtasks on parallel processors, *IEEE Trans. Soft. Eng. SE-11* (1985) 1001-1016.

[LaPr91] Z. Lahjomri and T. Priol, KOAN: A shared virtual memory for the iPSC/2 hypercube, IRISA, Campus Universitaire de Beaulieu, Internal Publication PI 597, 1991.

[Led90b] C.S. Ledbetter, A historical perspective of scientific computing in Japan and the United States, Part II, *Supercomputing Review 3 (12)* (1990) 48-58.

[Len90] D. Lenoski et al., The directory-based cache coherence protocol for the DASH multiprocessor, in *Proc. 17th Annual Int. Symp. Comp. Arch.*, Seattle, WA, 1990, 148-159.

[LeVe90] S.T. Leutenegger and M.K. Vernon, The performance of multiprocessor scheduling policies, in *Proc. 1990 ACM SIGMETRICS Conf. Measurement Modelling Comp. Syst.*, Boulder, CO, 1990, 226-236.

[LiCh90a] J. Li and M. Chen, Generating explicit communication from shared-memory program references, in *Proc. Supercomputing '90*, New York, NY, 1990, 865-876.

[Li86] K. Li, Shared virtual memory on loosely coupled multiprocessors, PhD thesis, Yale University, Technical Report YALEU-RR-492, 1986.

[Li88] K. Li, IVY: A shared virtual memory system for parallel computing, in *Proc. 1988 Int. Conf. Parallel Processing Vol. II*, St. Charles, IL, 1988, 94-101.

[LiHu89] K. Li and P. Hudak, Memory coherence in shared virtual memory systems, *ACM Trans. Comp. Syst. 7 (4)* (1989) 321-359.

[LiSc89a] K. Li and R. Schaefer, A hypercube shared virtual memory system, in *Proc. 1989 Int. Conf. Parallel Processing Vol. I*, St. Charles, IL, 1989, 125-132.

[LiSc89b] K. Li and R. Schaefer, Shared virtual memory for a hypercube multiprocessor, in *Proc. 4th Conf. Hypercubes*, Monterey, CA, 1989, 371-378.

[MPM92] T. MacDonald, D.M. Pase, and A. Meltzer, Addressing in Cray Research's MPP FORTRAN, in *Proc. Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, 1992, 161-172.

[MEB88] S. Majumdar, D.L. Eager, and R.B. Bunt, Scheduling in multiprogrammed parallel systems, in *Proc. 1988 ACM SIGMETRICS Conf. Measurement Modelling Comp. Syst.*, Santa Fe, NM, 1988, 104-113.

[MaLe92] E.P. Markatos and T.J. LeBlanc, Using processor affinity in loop scheduling on shared-memory multiprocessors, in *Proc. Supercomputing '92*, Minneapolis, MN, 1992, 104-113.

[MeRo91] P. Mehrotra and J. Van Rosendale, Programming distributed memory architectures using Kali, in *A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, Advances in Languages and Compilers for Parallel Processing, MIT Press*, Cambridge, MA, 1991, 364-384.

[MSMB90] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman, A scheme for supporting automatic data migration on multicomputers, in *Proc. 5th Dist. Mem. Comp. Conf.*, Charleston, SC, 1990, 1028-1037.

[Moh93] M. Mohring, Mach 3 shared virtual memory implementation, Diplomarbeit, Technische Universität München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, 1993.

[Mul94] C. Müllender, Visualisierung der Speicheraktivitäten paralleler Programme in Systemen mit virtuell gemeinsamem Speicher, erscheint als Diplomarbeit, RWTH Aachen, 1994.

[Nag88] W.E. Nagel, Using multiple CPUs for problem solving: Experiences in multitasking on the Cray X-MP/48, *Parallel Computing 8* (1988) 223-230.

[Nag90e] W.E. Nagel, Prinzipien der Parallelverarbeitung auf Rechnern mit gemeinsamem Speicher, in *A. Reuter, editor, Proc. 20. GI-Jahrestagung 1990, Informatik Fachbericht 257, Springer-Verlag*, 1990, 403-417.

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

[NaAr93] W.E. Nagel and A. Arnold, PARvis: Ein Werkzeug zur Visualisierung von parallelen Prozessen auf Mehrprozessorsystemen, in *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, 7. ITG/GI-Fachtagung*, Aachen, Germany, 1993, 178-187.

[NaLi91] W.E. Nagel and M.A. Linn, Parallel programs and background load: Efficiency studies with the PAR-Bench system, in: *Proc. 1991 Int. Conf. Supercomputing*, 1991, 365–375.

[NaLi93] W.E. Nagel and M.A. Linn, Benchmarking parallel programs in a multiprogramming environment: The PAR-Bench system, in: *Dongarra/Gentzsch, editors, Benchmarking for high performance computers, Elsevier Science Publishers B.V.*, 1993, 303-320.

[Oed91] W. Oed, Cooperative parallel interface für Cray X-MP und Cray Y-MP Systeme, Presentation at KFA Jülich (Germany) by W. Oed, Cray Research Inc, 1991.

[Ols85] R. Olson, Parallel processing in a message-based operating system, *IEEE Software 2 (1)* (1985) 39-49.

[Ous82] J.K. Ousterhout, Scheduling techniques for concurrent systems, in *Proc. 3rd Int. Conf. Dist. Comp. Syst.*, Miami, FL, 1982, 22-30.

[Pal88] J.F. Palmer, The nCUBE family of high-performance parallel computer systems, in *Proc. Hypercube Concurrent Computers and Applications Vol. I*, Pasadena, CA, 1988, 847-851.

[PMM93] D.M. Pase, T. MacDonald, and A. Meltzer, MPP FORTRAN programming modell, CRAY internal report, 1993.

[Pol89b] C.D. Polychronopoulos et al., Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multipro-

cessors, in *Proc. 1989 Int. Conf. Parallel Processing Vol. II*, St. Charles, IL, 1989, 39-48.

[Pol91] C.D. Polychronopoulos, The hierarchical task graph and its use in auto-scheduling, in *Proc. 1991 Int. Conf. Supercomputing*, Cologne, Germany, 1991, 252-263.

[PoKu87] C.D. Polychronopoulos and D.J. Kuck, Guided self-scheduling: A practical scheduling scheme for parallel supercomputers, *IEEE Trans. Comp. C-36 (12)* (1987) 1425-1439.

[PrLa92] T. Priol and Z. Lahjomri, Trade-offs between shared virtual memory and message passing on an iPSC/2 hypercube, IRISA, Technical Report 637, 1992.

[Prz92] F. Przybylski, Scheduling-Verfahren für Rechner mit verteiltem Speicher - Eine vergleichende Übersicht, Forschungszentrum Jülich, Jül-2598, 1992.

[RSRM93] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy, Scalability study of the KSR-1, in *Proc. 1993 Int. Conf. Parallel Processing Vol. I*, St. Charles, IL, 1993, 237-240.

[RaKh91] U. Ramachandran and M.Y.A. Khalidi, An implementation of distributed shared memory, *Software - Practice and Experience 21 (5)* (1991) 443-464.

[ReFu87] D.A. Reed and R.M. Fujimoto, Multicomputer networks: Message-based parallel processing, MIT Press, Cambridge, MA, 1987.

[ReKa79] D.P. Reed and R.K. Kanodia, Synchronization with eventcounts and sequencers, *CACM 22 (2)* (1979) 115-123.

[RoPi89] A. Rogers and K. Pingali, Process decomposition through locality of reference, in *Proc. SIGPLAN'89 Conf. Programming Language Design and Implementation*, Portland, OR, 1989, 69-80.

[Roz88] M. Rozier et al., Chorus distributed operating systems, *Computing Systems 1 (4)* (1988) 305-370.

[SPG91] A. Silberschatz, J.L. Peterson, and P.B. Galvin, Operating system concepts, Addison Wesley, Reading, MA, 1991.

[SHH90] J.E. Smith, W.-C. Hsu, and C. Hsiung, Future general purpose supercomputer architectures, in *Proc. Supercomputing '90*, New York, NY, 1990, 796-804.

[SqNe91] M.S. Squillante and R.D. Nelson, Analysis of task migration in shared-memory multiprocessor scheduling, in *Proc. 1991 ACM SIGMETRICS Conf. Measurement Modelling Comp. Syst.*, San Diego, CA, 143-155.

[SBK77] H. Sullivan, T.R. Bashkow, and D. Klappholz, A large-scale, homogeneous, fully distributed parallel machine, II, in *Proc. 4th Ann. Int. Symp. Comp. Arch.*, 1977, 118-124.

[TaYe86] P. Tang and P.C. Yew, Processor self-scheduling for multiple-nested parallel loops, in *Proc. 1986 Int. Conf. Parallel Processing*, St. Charles, IL, 1986, 528-534.

[ThCr88] R.H. Thomas and W. Crowther, The uniform system: An approach to runtime support for large scale shared memory parallel processors, in

Proc. 1988 Int. Conf. Parallel Processing Vol. II, St. Charles, IL, 1988, 245-254.

[TiWi81] A.M. van Tilborg and L.D. Wittie, Wave scheduling: Distributed allocation of task forces in network computers, in *Proc. 2nd Int. Conf. Dist. Comp. Syst.*, Paris, France, 1981, 337-347.

[TiWi84] A.M. van Tilborg and L.D. Wittie, Wave scheduling - decentralized scheduling of task forces in multicomputers, *IEEE Trans. Comp. C-33 (9)* (1984) 835-843.

[TuGu89] A. Tucker and A. Gupta, Process control and scheduling issues for multiprogrammed shared-memory multiprocessors, in *Proc. 12th ACM Symp. Operating Systems Principles*, The Wigwam Litchfield Park, AR, 1989, 159-166.

[TuSi85] D.L. Tuomenoksa and H.J. Siegel, Task scheduling on the PASM parallel processing system, *IEEE Trans. Soft. Eng. SE-11 (2)* (1985) 145-157.

[Wil89] K.G. Wilson, Grand challenges to computational science, *Future Generation Computer Systems 5 (2-3)* (1989) 171-189.

[WiTi80] L.D. Wittie and A.M. van Tilborg, Micros, a distributed operating system for Micronet, a reconfigurable computer, *IEEE Trans. Comp. C-29 (12)* (1980) 1133-1144.

[Wol92] M.E. Wolf, Improving locality and parallelism in nested loops, PhD thesis, Stanford University, 1992.

[You87] M. Young et al., The duality of memory and communication in the implementation of a multiprocessor operating system, *Operating Systems Review 21 (5)* (1987) 63-76.

[ZaMc90] J. Zahorjan and C. McCann, Process scheduling in shared memory multiprocessors, in *Proc. 1990 ACM SIGMETRICS Conf. Measurement Modelling Comp. Syst.*, Boulder, CO, 1990, 214-225.

[ZhBr91] S. Zhou and T. Brecht, Processor pool-based scheduling for large-scale NUMA multiprocessors, in *Proc. 1991 ACM SIGMETRICS Conf. Measurement Modelling Comp. Syst.*, San Diego, CA, 1991, 133-142.

[ZBG88] H.P. Zima, H.-J. Bast, and M. Gerndt, SUPERB: A tool for semi automatic MIMD/SIMD parallelization, *Parallel Computing 6* (1988) 1-18.