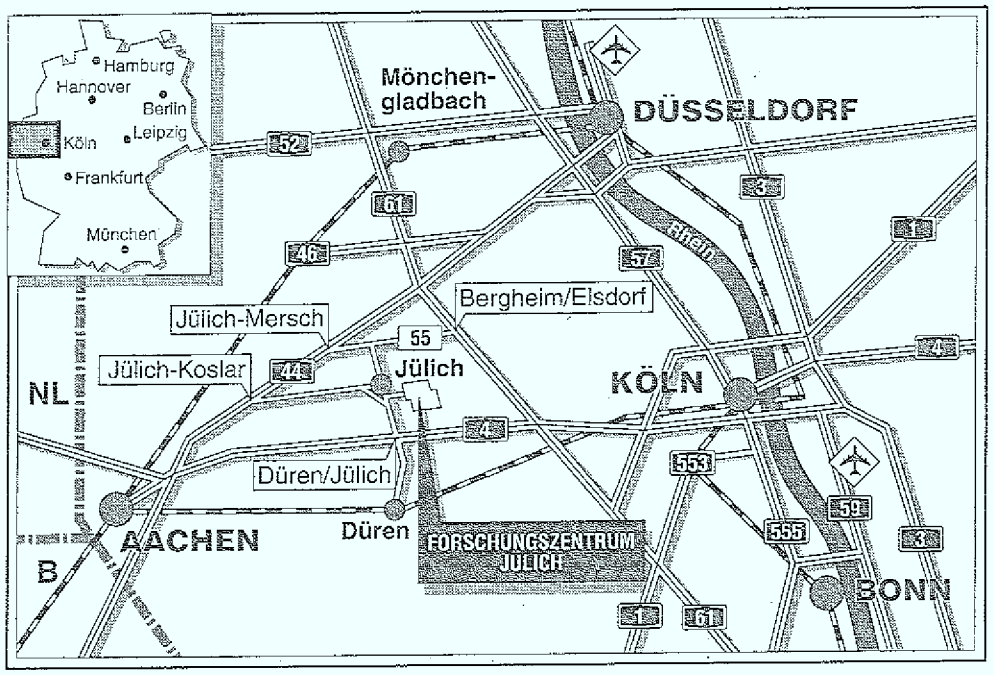
FORSCHUNGSZENTRUM JÜLICH GmbH

Zentralinstitut für Angewandte Mathematik

Leistungsuntersuchung von Multiprozessorsystemen auf der Basis des parametergesteuerten Lastbeschreibungssystems PAR-Bench unter besonderer Berücksichtigung von parallel ablaufbaren Teillasten

Klaus Fabian

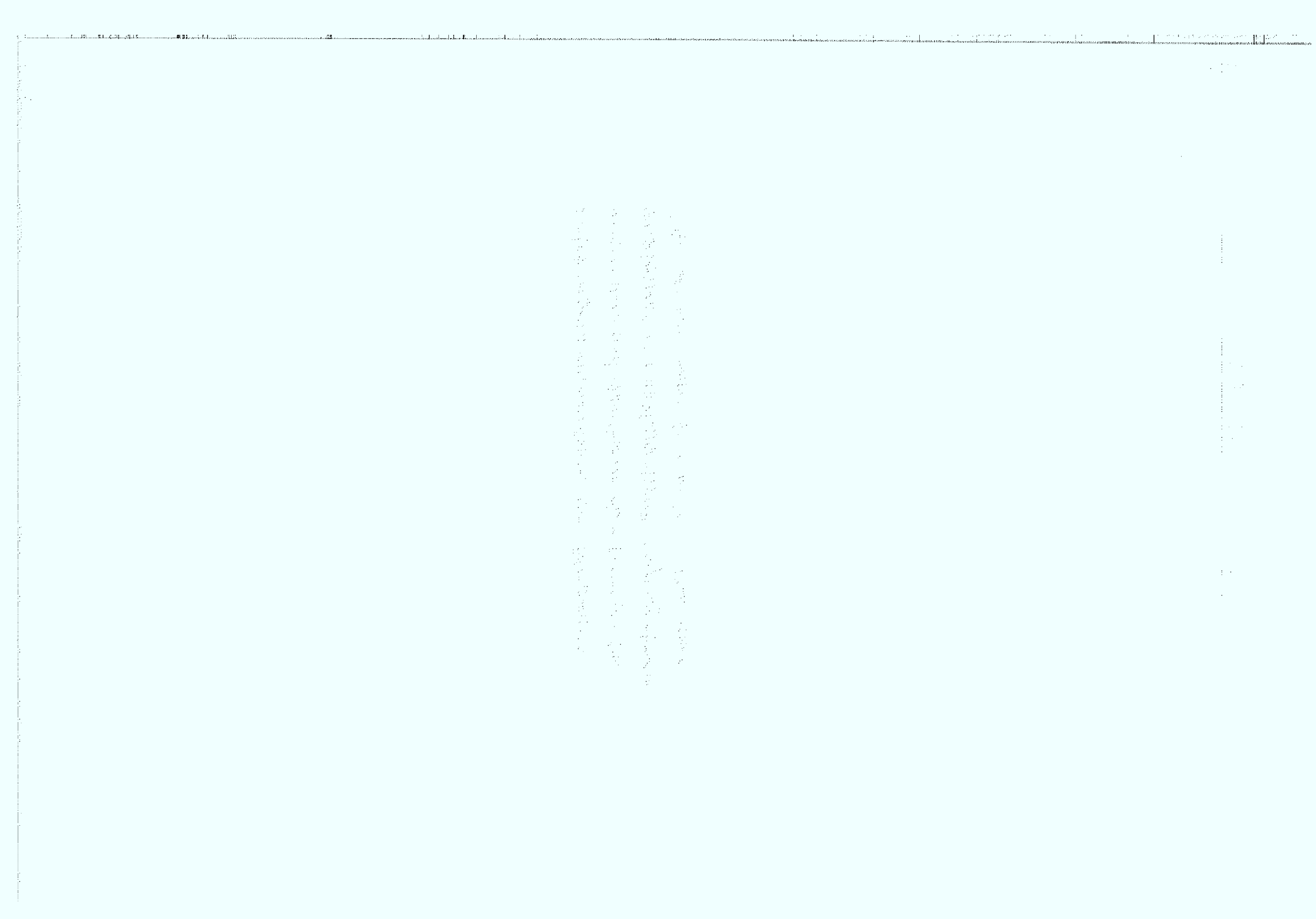


Berichte des Forschungszentrums Jülich ; 2671
ISSN 0366-0885
Zentralinstitut für Angewandte Mathematik Jül-2671

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
Postfach 1913 · D-5170 Jülich · Bundesrepublik Deutschland
Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Leistungsuntersuchung von Multiprozessorsystemen auf der Basis des parametergesteuerten Lastbeschreibungssystems PAR-Bench unter besonderer Berücksichtigung von parallel ablaufbaren Teillasten

Klaus Fabian



Inhalt

1 Einleitung 3

2 Leistungsanalyse 6
2.1 Rechnerarchitektur . . . 7
2.1.1 Prozessoren . . . 8
2.1.2 Hauptspeicher . . . 10
2.1.3 Cache-Speicher . . . 14
2.1.4 Virtueller Speicher . . . 19
2.1.5 Ein-/Ausgabesystem . . . 21
2.1.6 Synchronisation . . . 22
2.2 Betriebssystem . . . 26
2.2.1 Speicherverwaltung . . . 27
2.2.2 Scheduling . . . 29
2.3 Compiler . . . 36
2.3.1 Skalare Optimierung . . . 37
2.3.2 Vektorisierung . . . 39
2.3.3 Parallelisierung . . . 41
2.3.4 Sonstige Optimierungen . . . 43
2.4 Arbeitslast . . . 44
2.4.1 Parallele Programme im Einprogrammbetrieb . . . 46
2.4.2 Parallele Programme im Mehrprogrammbetrieb . . . 48
2.4.3 Wechselwirkungen von Programmen im Mehrprogrammbetrieb 50
2.5 Meßmethoden . . . 52
2.5.1 Zeitmessung . . . 53
2.5.2 Software-Monitore . . . 55
2.5.3 Hardware-Monitore . . . 59

3 Multiprozessor-Vektorrechner mit gemeinsamem Speicher 62
3.1 Die ALLIANT FX/2800-Serie . . . 64
3.2 Die CONVEX C3-Serie . . . 68
3.3 Die CRAY X-MP/Y-MP Rechnerfamilien . . . 69
3.3.1 Die CRAY X-MP/416 . . . 71
3.3.2 Die CRAY Y-MP8/832 . . . 71
3.4 Vergleichende Darstellung . . . 71

4 Meßmöglichkeiten des PAR-Bench-Systems 74
4.1 Generierung der synthetischen Arbeitslast 75
4.2 Messung und Auswertung 77
4.3 Portierung des Systems PAR-Bench 80

5 Leistungsmessung 84
5.1 Messungen auf der Alliant FX/2816 85
5.1.1 Arbeitslasten mit günstigen Eigenschaften 86
5.1.2 Arbeitslasten mit niedriger Parallelität 92
5.1.3 Arbeitslasten mit hoher Speicherbelastung 96
5.1.4 Zusammenfassung der Meßergebnisse 100
5.2 Messungen auf der Convex C3840 102
5.2.1 Arbeitslasten mit günstigen Eigenschaften 103
5.2.2 Arbeitslasten mit niedriger Parallelität 109
5.2.3 Arbeitslasten mit hoher Speicherbelastung 113
5.2.4 Zusammenfassung der Meßergebnisse 117
5.3 Messungen auf der CRAY X-MP/416 118
5.3.1 Arbeitslasten mit günstigen Eigenschaften 119
5.3.2 Arbeitslasten mit niedriger Parallelität 125
5.3.3 Arbeitslasten mit hoher Speicherbelastung 129
5.3.4 Zusammenfassung der Meßergebnisse 131
5.4 Messungen auf der CRAY Y-MP 132
5.4.1 Arbeitslasten mit günstigen Eigenschaften 132
5.4.2 Arbeitslasten mit niedriger Parallelität 141
5.4.3 Arbeitslasten mit hoher Speicherbelastung 146
5.4.4 Zusammenfassung der Meßergebnisse 148
5.5 Gegenüberstellender Vergleich der Meßergebnisse 149

6 Zusammenfassung 156

Literaturverzeichnis 157

Kapitel 1

Einleitung

Für die Untersuchung von technisch-wissenschaftlichen Problemen, deren analytische Lösungen nicht bekannt sind, eröffnet die Simulation und der damit verbundene Einsatz von numerischen Methoden neue Perspektiven. Derartige Methoden werden erfolgreich z.B. in der Seismik oder der Klimaforschung angewendet. Während der Rechenaufwand für einige dieser Probleme schon mit heutigen Vektorsupercomputern in hinreichend kurzer Zeit bewältigt werden kann, ist der Rechenaufwand für die Lösung anderer Probleme noch um Größenordnungen zu hoch. Da die Leistung von Einprozessorsystemen aufgrund von physikalischen Gesetzen nicht mehr im erforderlichen Maß erhöht werden kann, erscheint eine drastische Leistungssteigerung nur mit Hilfe der Parallelverarbeitung auf Multiprozessorsystemen möglich [Hoß91].

Für die Parallelverarbeitung werden zur Zeit zwei verschiedene Arten von Multiprozessorsystemen angeboten, Systeme mit verteiltem Speicher (*distributed memory*) und Systeme mit gemeinsamem Speicher (*shared memory*). Die Programmierung von Computersystemen mit verteiltem Speicher erfolgt nahezu ausschließlich nach dem Konzept des *message passing*. Beim *message passing* ist der Programmierer für die Kommunikation der kooperierenden Prozesse eines parallelen Programms verantwortlich und muß diese explizit durch Anweisungen formulieren. Die Kommunikation erfolgt auf diesen Rechnern über ein Prozessorverbindungsnetzwerk und ist auf den heute kommerziell verfügbaren Computersystemen mit verteiltem Speicher sehr viel langsamer als auf den Computersystemen mit gemeinsamem Hauptspeicher. In dieser Arbeit stehen Computersysteme mit gemeinsamem Speicher im Vordergrund. Derartige Rechner besitzen einen globalen Hauptspeicher, auf den jeder Prozessor unmittelbar zugreifen kann. Ein gemeinsamer Hauptspeicher wird insbesondere in Computersystemen mit wenigen sehr leistungsfähigen Prozessoren verwendet. Die Hersteller traditioneller Vektorsupercomputer nutzen einige wenige Prozessoren, um den Durchsatz ihrer Systeme zu erhöhen. Die Anzahl der in solchen Computersystemen genutzten Prozessoren ist zwar noch gering, steigt aber von Rechnergeneration zu Rechnergeneration stetig an. Während in der CRAY X-MP und der NEC SX-3 noch maximal vier Prozessoren eingesetzt werden, besitzt die CRAY Y-MP bis zu

acht und die in ersten Exemplaren verfügbare CRAY Y-MP C90 sogar sechzehn Prozessoren. Die für Multiprozessorsysteme mit gemeinsamem Hauptspeicher entwickelten Programmierwerkzeuge vereinfachen die Nutzung der Parallelverarbeitung zur Leistungssteigerung von Programmen. Mit dem *microtasking* stellte die Firma CRAY bereits 1986 ein Konzept vor, bei dem die Parallelprogrammierung allein durch die Nutzung von Compiler-Direktiven ermöglicht wird [HKN89]. Ein durch derartige Compiler-Direktiven instrumentiertes Programm ist sowohl auf Ein- als auch auf Mehrprozessorsystemen mit gemeinsamem Hauptspeicher ablauffähig. Die Beschleunigung bestehender FORTRAN-Programme durch Parallelverarbeitung erfordert bei diesem Programmierkonzept einen vergleichsweise geringen Aufwand. In vielen Fällen kann der Compiler dem Benutzer diese Arbeit sogar abnehmen. Das *autotasking* von CRAY arbeitet nach diesem Prinzip und ermöglicht so eine automatische Parallelisierung [Reg89]. Automatisch parallelisierende FORTRAN-Compiler sind z.B. für die Multiprozessorsysteme der Hersteller Alliant, CONVEX, CRAY und IBM verfügbar.

Aus unterschiedlichen Gründen wird heute gerade in Rechenzentrumsumgebungen die Integration von parallelen Programmen in den Mehrprogrammbetrieb immer wichtiger. Zum einen ist der Wechsel zwischen einer Mehrprogrammumgebung für den Produktionsbetrieb und einer dedizierten Umgebung für parallele Programme zu aufwendig. Zum anderen wird es immer schwieriger, die steigende Zahl von Prozessoren in Multiprozessorsystemen nur mit sequentiellen Programmen auszulasten. Dies gilt insbesondere für den Nacht- und den Wochenendbetrieb, wenn die Rechner nur in geringem Maß interaktiv genutzt werden. Ein Leerlaufen von Prozessoren kann jedoch auch bei einer hohen Nutzung auftreten, wenn ein speicherintensives Programm den gesamten Realspeicher eines Rechners zur Ausführung benötigt und deshalb kein Speicherplatz für weitere Jobs zur Verfügung steht. In diesen Fällen kann durch die Parallelisierung von Programmen eine bessere Nutzung der Prozessoren und damit eine signifikante Erhöhung des Durchsatzes erreicht werden. Unter diesen Randbedingungen steht der zusätzliche Aufwand, der durch die Abarbeitung von parallelen Programmen entsteht, im Vordergrund. Leider gibt es generell weder genaue Herstellerinformationen noch Programmierwerkzeuge, die sowohl qualitativ als auch quantitativ eine Abschätzung dieses Aufwands erlauben. Deshalb werden im Rahmen dieser Arbeit für vier sehr unterschiedliche Multiprozessorsysteme Kostenabschätzungen für die Bearbeitung von parallelen Programmen im Mehrprogrammbetrieb vorgenommen.

Mit dem System PAR-Bench (PARallel Benchmark, [Nag90a, Lin90]) wurde im Zentralinstitut für Angewandte Mathematik ein parametergesteuerter Lastgenerator entwickelt, mit dem beliebige synthetische Arbeitslasten generiert werden können. In Abhängigkeit von den Parametern der Arbeitslast kann dann die Leistung eines Multiprozessorsystems detailliert analysiert werden. Das System PAR-Bench wurde zunächst auf den CRAY-Rechner X-MP/416 und Y-MP8/832 implementiert und ermöglicht auf diesen Computersystemen nach Vorgabe von charakteristischen Lastmerkmalen die Generierung beliebiger Arbeitslasten. Es zeichnet sich

insbesondere dadurch aus, daß Arbeitslasten parallel ablauffähige Teillasten enthalten können. Bei der Generierung der parallelen Programme einer derartigen Arbeitslast können die Parameter Parallelisierungsgrad und Lastverteilung vorgegeben werden. Dadurch wird eine Leistungsmessung von parallelen Programmen im Mehrprogrammbetrieb und damit eine Untersuchung des Zusammenspiels der *multitasking*-Implementation mit dem Betriebssystem möglich.

Durch Wahl der PAR-Bench-Parameter sind die Laufzeitexperimente zunächst so angelegt, daß durch die sequentielle Abarbeitung der Programme einer Arbeitslast alle Prozessoren ständig belegt sind und damit das Multiprozessorsystem voll ausgelastet ist. In einem solchen Fall sind durch Parallelverarbeitung Performance-Gewinne, wie sie oben beschrieben sind, nicht zu erzielen. Durch schrittweise Parallelisierung von einzelnen Arbeitslastprogrammen kann dann jedoch der zusätzliche Aufwand, der durch die Parallelisierung entsteht, sehr genau bestimmt werden. Gelingt der Nachweis, daß die heutigen *multitasking*-Implementationen, in Zusammenarbeit mit den Betriebssystemen und der Hardware, effiziente Parallelverarbeitung auch in diesen Situationen ermöglichen, so hat dies zur Konsequenz, daß zukünftig in Rechenzentrumsumgebungen mehrere der Produktionsprogramme (bei entsprechender Eignung) generell parallel abgearbeitet werden können. Kommt es dann z.B. am Wochenende zu Lastumgebungen, wie sie oben beschrieben wurden, so können diese Programme von den verfügbaren Ressourcen wie z.B. der Prozessorleistung profitieren, die anderenfalls nicht genutzt werden könnte. Damit werden derartige Programme besser bedient, ohne daß ein anderes Programm benachteiligt wird, und darüber hinaus steigt die Auslastung des Gesamtsystems.

Im Rahmen dieser Arbeit wird nun auf verschiedenen Multiprozessor-Vektorrechnern das Zusammenspiel von *multitasking*-Implementation, Betriebssystem und Hardware mit Hilfe von vergleichenden Leistungsmessungen analysiert und quantifiziert. Dazu wird in einem ersten Schritt überprüft, welche Eigenschaften der Computersysteme und der Arbeitslast den gemessenen Durchsatz beeinflußen. In einem zweiten Schritt erfolgt eine Portierung und Adaption des in FORTRAN 77 implementierten PAR-Bench auf die Multiprozessorsysteme Alliant FX/2816 und CONVEX C3840. Die darauf folgenden vergleichenden Leistungsmessungen zeigen, daß die Implementierung des *multitasking* auf den vier untersuchten Computersystemen nach unterschiedlichen Prinzipien erfolgte, die einen gravierenden Einfluß auf die Ausführung der parallelen Programme einerseits und den Durchsatz der Computersysteme andererseits haben.

Kapitel 2

Leistungsanalyse

Eine absolute Leistungsangabe mit universellem Anspruch ist aufgrund der unterschiedlichen Rechnerarchitekturen sowie der verschiedenen Anforderungen, die an ein Computersystem gestellt werden, unabhängig von den Eigenschaften einer Applikation nicht möglich. Bei einer Leistungsanalyse werden deshalb die leistungsrelevanten Faktoren qualitativ und quantitativ erfaßt und in bezug auf ihre Ursachen und Wechselwirkungen analysiert. Die so gewonnenen Ergebnisse werden sowohl für Entwurfsentscheidungen als auch für den Vergleich und die Optimierung des Betriebs von Computersystemen eingesetzt [Jai91, Rei90].

Die in dieser Arbeit betrachteten Multiprozessor-Vektorrechner mit gemeinsamem Hauptspeicher zeichnen sich sowohl durch einfache Programmierbarkeit als auch durch die Verfügbarkeit von leistungsfähigen, automatisch parallelisierenden Compilern aus. Diese Rechner werden in Rechenzentren meist im Multiprogramming-Betrieb genutzt, um eine möglichst wirtschaftliche Verwendung der Ressourcen zu erreichen. Bei der parallelen Ausführung von Programmen kommt es in derartigen Umgebungen zu einer Wechselwirkung des parallelen Programms mit der Hintergrundlast, die einerseits die Ausführung des parallelen Programms und andererseits die der Hintergrundlast beeinflußt [Bie87, DI90, NL91a]. Der Einfluß dieser Wechselwirkungen auf die Leistung von verschiedenen Computersystemen wird im folgenden durch eine Leistungsanalyse bestimmt.

Dazu werden zunächst alle leistungsrelevanten Eigenschaften der Computersysteme untersucht. Die Leistung eines Computersystems wird von der Rechnerarchitektur, dem Betriebssystem und dem Compiler beeinflußt; in den folgenden Abschnitten werden die einzelnen Aspekte systematisch dargelegt. Diese Untersuchung erfolgt zunächst allgemein und damit unabhängig von den in dieser Arbeit genauer untersuchten Computersystemen, und daher sind die geschilderten Aspekte auch für weitere Untersuchungen mit anderen Systemen von Bedeutung. Darüber hinaus hat die Wahl der Arbeitslast sowie die verwendete Meßmethode einen starken Einfluß auf die Leistungsresultate. Eine Untersuchung dieser Aspekte erfolgt in den weiteren

Abschnitten dieses Kapitels.

2.1 Rechnerarchitektur

Da bei der Erhöhung der Schaltgeschwindigkeit nur noch geringe Fortschritte erreichbar sind, werden Möglichkeiten zur Leistungssteigerung zunehmend durch neue Rechnerarchitekturen untersucht. Basierend auf Annahmen über das Verhalten von Programmen werden architektonische Designentscheidungen vorgenommen (z.B. *pipelines*, Cache-Speicher, Speicherverschränkung), die für entsprechend geeignete Programme zu einem deutlichen Leistungsgewinn führen. Programme, die diesen Anforderungen nicht entsprechen, profitieren von diesen Entwicklungen jedoch nur in geringem Maße. Dies hat zur Folge, daß der Unterschied zwischen theoretisch möglicher Spitzenrechenleistung (*peak performance*) und effektiv erreichbarer Leistung durch die Spezialisierung der Architektur für immer mehr Programme immer größer wird. Von solchen Spezialisierungen sind alle Komponenten der hier betrachteten Multiprozessor-Vektorrechner mit gemeinsamem Speicher betroffen:

- Die Prozessoren
- Der Hauptspeicher
- Das Ein-/Ausgabesystem

Die Prozessoren stellen eine Rechenleistung zur Verfügung, deren volle Nutzung nur möglich ist, wenn die zu bearbeitenden Daten schnell genug von den verschiedenen Stufen der Speicherhierarchie zur Verfügung gestellt werden. Von zentraler Bedeutung für die Speicherhierarchie der hier betrachteten Rechnerklasse ist der gemeinsame Hauptspeicher, der Datenanforderungen von allen Prozessoren befriedigen und deshalb eine hohe Datenübertragungsrate (*Bandbreite*) bieten muß. Zur Entlastung des gemeinsamen Hauptspeichers werden Vektorregister, Cache-Speicher und lokale Speicher eingesetzt, die von jedem Prozessor lokal genutzt werden. Mischformen von globalen und lokalen Speichern ergeben sich, wenn die Speicher jeweils von einer Gruppe von Prozessoren (*cluster*) gemeinsam genutzt werden und von Prozessoren aus anderen *clustern* nicht angesprochen werden können; so werden z.B. bei der Alliant FX/8 jeweils 4 Prozessoren einem Cache-Speicher zugeordnet.
Übersteigt die bearbeitete Datenmenge die Hauptspeicherkapazität, so müssen die Daten über das Ein-/Ausgabesystem mit dem Sekundärspeicher ausgetauscht werden. Da der Zugriff auf den Sekundärspeicher um Größenordnungen mehr Zeit erfordert als der Zugriff auf den Hauptspeicher, wird die Leistung bei der Verarbeitung von großen Datenmengen oft ausschließlich von der Leistung des Ein-/Ausgabesystems bestimmt. Der große Leistungsabfall beim Übergang vom Hauptspeicher auf übliche Sekundärspeicher wie Plattenspeicher kann bei geeigneten Anwendungen durch die Verwendung eines zusätzlichen großen Halbleiterspeichers (z.B. SSD) vermieden werden.

Beim Zeitscheibenbetrieb können zusätzliche Ein/-Ausgabeoperationen notwendig werden, wenn die zeitlich verzahnt bearbeiteten Jobs nicht gleichzeitig in den Hauptspeicher passen und abwechselnd auf den Sekundärspeicher ausgelagert werden müssen. Ein virtueller Speicher hält nur die aktuell von den Prozessoren angesprochenen Daten im Hauptspeicher und nutzt so den vorhanden Speicher besser aus. Dadurch kann das Ein- und Auslagern von ganzen Jobs vermieden oder auf den Austausch kleinerer Datenmengen (Seiten) reduziert werden. Im Vergleich zum realen Hauptspeicher können beim virtuellen Speicher mehr Jobs zur Ausführung bereitstehen, was zu einer besseren Auslastung des Rechners und zu einer Steigerung des Durchsatzes führen kann.
Bei der Verarbeitung von parallelen Programmen wird die Leistungsfähigkeit der Rechnerarchitektur zusätzlich von der notwendigen Kommunikation und der dazu benötigten Synchronisation beeinflußt, die in vielen Fällen hinreichend effizient über den gemeinsamen Hauptspeicher durchgeführt werden kann. Bei der Bearbeitung feingranularer Probleme kann eine Leistungssteigerung durch Parallelverarbeitung von einer möglichst leistungsfähigen Prozessor-Prozessor-Kommunikation (z.B. gemeinsam genutzte Register) abhängen.

2.1.1 Prozessoren

Aufgrund der hohen Rechenleistung, die technisch-wissenschaftliche Programme benötigen, werden in Supercomputern häufig Vektorprozessoren zur Steigerung der Rechenleistung eingesetzt, die auf die effiziente Berechnung von Vektoroperationen spezialisiert sind. Die maximal mögliche Rechenleistung eines Prozessors (*peak performance*) wird in Millionen Gleitkommaoperationen pro Sekunde (MFLOPS = *million floating-point operations per second*) angegeben. Da die Rechenleistung vom verwendeten Gleitkommazahlenformat beeinflußt wird und die Prozessoren oft mehrere solcher Formate (einfache und doppelte Genauigkeit) bereitstellen, gehört die Angabe dieses Formats zur Leistungsangabe. Die Rechenleistung eines Prozessors wird durch die im folgenden beschriebenen Komponenten (Rechenwerk, Register und Speicherzugang) bestimmt.

Die Leistungsfähigkeit des Rechenwerkes bestimmt die *peak performance* des Computersystems. Um die Leistung des Rechenwerkes zu steigern, werden in Vektorprozessoren jeweils mehrere arithmetische Operationen gleichzeitig bearbeitet. Dazu können einerseits mehrere unabhängige, gleichzeitig arbeitende Funktionseinheiten verwendet werden; andererseits kann durch *pipelining* die Leistung der einzelnen Funktionseinheiten erhöht werden [HJ88]. Eine arithmetische *pipeline* beschleunigt die Verarbeitung von Vektoren durch die gleichzeitige Bearbeitung mehrerer Vektorelemente nach dem Fließbandprinzip. Auf diese Weise kann eine *pipeline* nach einer Anlaufzeit von mehreren Takten (*startup time*) in jedem Takt ein Ergebnis bereitstellen. Der Einfluß der *startup time* auf die Rechenleistung ist bei Vektoren mit vielen Elementen geringer, so daß eine hohe Rechenleistung nur bei der Verarbeitung

von hinreichend langen Vektoren erreicht wird. Besitzt der Prozessor unabhängige, gleichzeitig arbeitende Funktionseinheiten, so können während der Verarbeitung einer Rechenoperation auf einer Funktionseinheit bereits weitere Rechenoperationen auf anderen Funktionseinheiten gestartet werden. Die Verknüpfung der beiden Prinzipien erfolgt beim *chaining*. Dabei wird das Ergebnis einer arithmetischen *pipeline* (z.B. der Multiplikationseinheit) direkt als Operand für eine weitere arithmetische *pipeline* (z.B. die Additionseinheit) verwendet. Dadurch addieren sich zwar die Anlaufzeiten der beiden *pipelines*, es werden jedoch in jedem Takt zwei Rechenoperationen durchgeführt; damit kann für lange Vektoren die Rechenleistung verdoppelt werden.

Die von den Rechenwerken verarbeiteten Daten müssen von Speichern im oder außerhalb des Prozessors bereitgestellt werden. Die Register sind schnelle Speicher im Prozessor und werden zum Zwischenspeichern von Operanden und Ergebnissen verwendet, die für die Weiter- oder Wiederverwendung vorgesehen sind. In Vektorprozessoren werden neben Skalarregistern für Skalare auch Vektorregister verwendet, die kurze Vektoren oder Teile langer Vektoren aufnehmen können. Die Prozessoren unterscheiden sich in der Anzahl der Skalar- und Vektorregister sowie der Anzahl der Elemente, die ein Vektorregister zwischenspeichern kann; in einigen Architekturen existiert ein zusammenhängender Registerspeicher, der dynamisch auf die Vektorregister verteilt werden kann (z.B. FUJITSU VP2000, [BEK90]). Je größer die Anzahl der Elemente eines Vektorregisters ist, desto mehr Daten können mit einer Vektoroperation verarbeitet werden, und um so effizienter können die arithmetischen *pipelines* genutzt werden. Mit steigender Registerzahl können mehr unabhängige Daten im Prozessor zwischengespeichert und wiederverwendet werden. Die Nutzung einer großen Anzahl von Vektorregistern ist jedoch schwierig und muß von der gesamten Architektur unterstützt werden. Ist dies nicht der Fall, so kann sich die Leistung bei der Erhöhung der Registerzahl sogar verringern. Dieser Effekt wurde z.B. von der Firma CRAY an einer Simulation von Speicher-Verbindungsnetzwerken beobachtet und quantifiziert [ST91].
Der Aufwand, der durch den Austausch des Registersatzes mit dem Hauptspeicher verursacht wird, bestimmt die minimale Zeitdauer eines *task*-Wechsels und damit wie effizient ein frei werdender Prozessor neuen Aufgaben zugeteilt werden kann. Eine Beschleunigung dieses Vorganges läßt sich durch die Verwendung mehrerer Registersätze erreichen, da ein Umschalten des Registersatzes erheblich schneller durchführbar ist als ein Austausch der Register mit dem Speicher. Der Zeitaufwand, der zum Registeraustausch über den Speicher benötigt wird, steigt proportional mit der Speicherkapazität des Registersatzes, also mit der Anzahl der Register und der Anzahl der Elemente, die ein Vektorregister aufnehmen kann.

Aus dem Hauptspeicher gelangen Daten über die Speicherzugänge (*ports*) in den Prozessor, wo sie entweder in Registern zwischengespeichert oder direkt von den Rechenwerken verarbeitet werden. Da bei einer dyadischen Rechenoperation zwei Operanden benötigt und ein Ergebnis berechnet wird, können für eine Rechenoperation drei Speicheroperationen erforderlich sein; bei mehreren parallel arbeitenden

Funktionseinheiten kann sich die Anzahl der Speicheroperationen, die notwendig sind, um das Rechenwerk auszulasten, vervielfachen. Die Anzahl der von einer Architektur durchführbaren Speicheroperationen pro Zeiteinheit läßt sich entweder durch Steigerung der Bandbreite eines *ports* oder durch Erhöhung der Anzahl der *ports* erreichen. Einen Spezialfall unter den Architekturen mit mehreren *ports* stellen die Harvard-Architekturen dar (z.B. Motorola M88000) , die für die Übertragung von Instruktionen und Daten unterschiedliche *ports* verwenden. Die Bandbreite eines *ports* wird einerseits durch die Bitbreite (z.B. 32, 64 Bit) und andererseits durch die Zeit, die zur Durchführung eines *port*-Zugriffs benötigt wird, bestimmt. Die Geamtbandbreite der *ports* begrenzt die maximal erreichbare Rechenleistung, wenn für die Berechnungen mehr Datentransfer mit dem Hauptspeicher benötigt wird als durch die *ports* geleistet werden kann. Die Rechenleistung des Intel-Prozessors i860 wird in Anwendungen häufig durch die Speicherbandbreite begrenzt, da die 40MHz-Version des Prozessors eine *peak performance* von 40 MFLOPS besitzt, die Speicherbandbreite des *ports* aber nur 20 MW/s (a 64 Bit) beträgt [Int90, BH90].

2.1.2 Hauptspeicher

Die hohe Rechenleistung von Vektorprozessoren kann nur dann genutzt werden, wenn die zu verarbeitenden Daten schnell genug von dem gemeinsamen Hauptspeicher zur Verfügung gestellt werden können. Während verteilter (lokaler) Speicher nur von jeweils einem Prozessor angesprochen werden kann, steht der gemeinsame (globale) Speicher im Zugriff aller Prozessoren. Er kann deshalb den von den Prozessoren bearbeiteten Jobs flexibel zugeteilt werden und stellt eine leistungsfähige Kommunikationsressource für die Parallelverarbeitung dar.
Da sich die Speicherbandbreite eines globalen Speichers auf alle Prozessoren aufteilt, muß sie erheblich höher dimensioniert werden als die Bandbreite eines lokalen Speichers. Sie wird einerseits von der Geschwindigkeit des Speichers und andererseits von der Datenmenge bestimmt, auf die gleichzeitig zugegriffen werden kann. Die Geschwindigkeit des Speichers wird durch die Speicherzugriffszeit (*access time*) und die Speicherzykluszeit (*cycle time*) bestimmt. Bei den in Supercomputern überwiegend verwendeten statischen Speichern sind die Speicherzugriffszeit und die Speicherzykluszeit identisch. Im Gegensatz dazu weist der dynamische Speicher, der in der CRAY-2 verwendet wird [CB88], eine im Vergleich zur Speicherzugriffszeit erheblich erheblich höhere Speicherzykluszeit auf, da der Speicher nach einem Zugriff wieder aufgefrischt werden muß und deshalb für eine gewisse Zeitspanne (*reservation time*) nicht angesprochen werden kann [Hay88]. Neuere dynamische Speicher bieten allerdings die Möglichkeit, die nachfolgenden Adressen eines Zugriffs mit erheblich verkürzter Zykluszeit anzusprechen (*burst mode*), so daß zwischen der minimalen und der maximalen Speicherzykluszeit unterschieden werden muß.

Da die Speicherzykluszeit heutiger Supercomputer ein Mehrfaches der Zykluszeit der Prozessoren beträgt, kann eine hohe Speicherbandbreite nur durch eine hohe Par-

allelität beim Datenzugriff erreicht werden. Die in einem Speicherzyklus verfügbare Datenmenge setzt sich aus der Datenmenge eines Speicherzugriffs (Datenbusbreite) und der Anzahl der gleichzeitig ansprechbaren Speicherbänke zusammen. Bei einem gemeinsamen Speicher mit mehreren Speicherbänken (Verschränkung oder *interleaving*) werden die Speicherbänke über die Hauptspeicheradresse angesprochen, und es muß eine Zuordnung der Hauptspeicheradressen auf die Speicherbänke und die Adressen innerhalb der Speicherbänke erfolgen. Das *low order interleaving* wird in Vektorrechnern verwendet, da es aufeinanderfolgende Adressen auf unterschiedliche Speicherbänke abbildet und so einen gleichzeitigen Zugriff auf Ausschnitte eines Vektors ermöglicht. Die Verbindung eines Prozessor-*ports* mit der durch das *interleaving* ausgewählten Speicherbank wird von einem Speicherverbindungsnetzwerk (*memory interconnection network*) hergestellt.
Der gleichzeitige Zugriff der Prozessor-*ports* auf unterschiedliche Adressen des Speichers ist nicht möglich, wenn es im Verbindungsnetzwerk oder auf eine Speicherbank zu Zugriffskonflikten kommt. *Bankkonflikte* entstehen, wenn innerhalb einer Speicherzykluszeit mehrere Zugriffe auf eine Speicherbank erfolgen; sie werden von der Wechselwirkung des Speicherzugriffverhaltens des Programms mit der Art des *interleavings* bestimmt und sind unabhängig von dem verwendeten Verbindungsnetzwerk. *Netzwerkkonflikte* entstehen durch die Konkurrenz von Zugriffen um eine gemeinsame Verbindung innerhalb des Netzwerkes und sind deshalb netzwerkspezifisch. Die Speicherverbindungsnetzwerke werden nach Enslow [HB84] in drei Klassen eingeteilt:

- Gemeinsame Busse (*common bus*);
- Kreuzschienenschalter (*crossbar switch*);
- mehrstufige Verbindungsnetzwerke (*multistage interconnection network*).

Bei einem System mit einem *gemeinsamen Bus* werden die Prozessoren mit den Speicherbänken über einen gemeinsamen, möglichst schnellen Bus miteinander verbunden (siehe Abbildung 2.1 (a)). Die Prozessoren können den Bus nur zeitlich versetzt bzw. alternativ benutzen; erfolgen gleichzeitige Zugriffe auf den Bus, so kommt es zu Buskonfliken und damit zu Zugriffsverzögerungen. Die Buszuteilungsstrategie entscheidet, welcher Buszugriff ausgeführt und welcher Buszugriff abgewiesen wird. Da alle Daten nur über den gemeinsamen Bus zwischen Speicherbänken und Prozessoren transportiert werden können, stellt die Bandbreite des Busses die obere Grenze für die Speicherbandbreite des Computersystems dar. Aufgrund der einfachen Struktur wird ein gemeinsamer Bus in einigen kommerziellen Multiprozessorsystemen wie der FX8 und FX80 von Alliant, der Ballance und Symmetry von Sequent und dem Encore Multimax verwendet; diese Rechner werden zu den Minisupercomputern bzw. zu den Echtzeitsystemen gezählt.

Ein Verbindungsnetzwerk, bei dem keine Netzwerk-Konflikte entstehen können, ist der Kreuzschienenschalter (*crossbar switch*, siehe Abbildung 2.1 (b)). Ein Kreuzschienenschalter besteht aus einer Schaltermatrix, die jeden Eingang mit

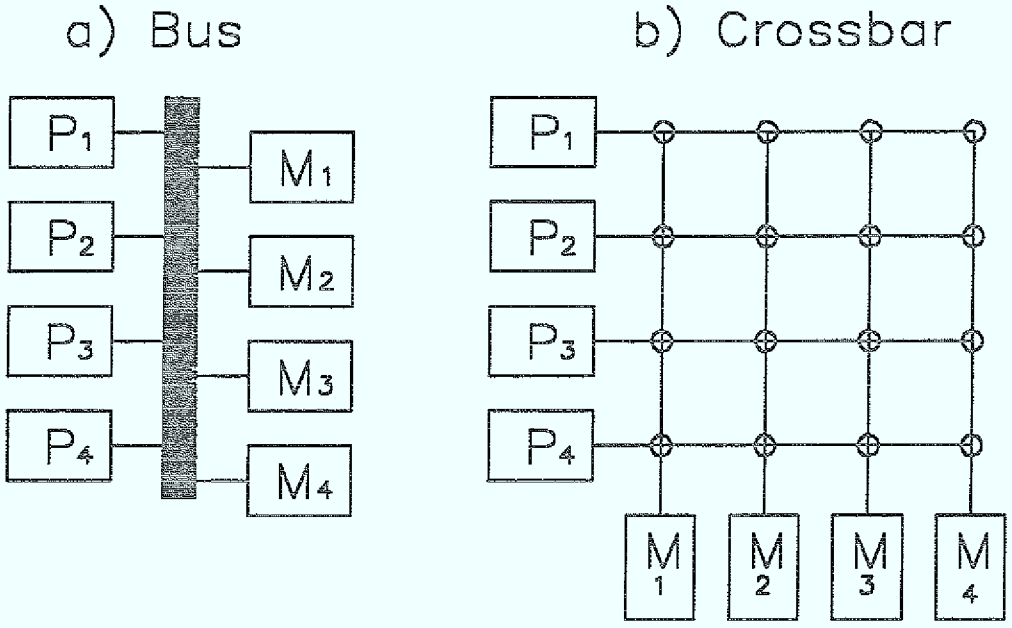


Abbildung 2.1: Computersystem mit a) gemeinsamen Bus b) Crossbar als Speicherverbindungsnetzwerk. P_i bezeichnet die Prozessoren, M_i die Speicherbänke

jedem Ausgang des Schalters verbinden kann. Bei einem Kreuzschienenschalter-Verbindungsnetzwerk kann also jeder Prozessor-*port* durch einen Schalter direkt mit jeder Speicherbank verbunden werden. Für einen Kreuzschienenschalter, der n Prozessor-*ports* mit m Speicherbänken verbindet, sind genau $n * m$ Schalter erforderlich. Dieser Aufwand führt für große n und m zu hohen Kosten und Problemen bei der technischen Realisation. Die CONVEX C3-Serie verwendet einen *crossbar*, der im Spitzenmodell C-3800 in Gallium-Arsenid-Technologie realisiert ist und acht Prozessoren sowie einen I/O-*port* mit bis zu 256 Speicherbänken verbindet.

Eine häufig verwendete Alternative zu den Kreuzschienenschaltern stellen die mehrstufigen Verbindungsnetzwerke dar, die ebenfalls eine direkte Verbindung zwischen den Prozessor-*ports* und den Speicherbänken herstellen, diese Verbindung aber über mehrere Schaltstufen aufbauen (siehe Abbildung 2.2). Bei mehrstufigen Verbindungsnetzwerken verzichtet man darauf, alle eindeutigen (injektiven) Abbildungen von Prozessor-*ports* auf Speicherbänke gleichzeitig durch das Netzwerk zu realisieren und erreicht dadurch eine Reduzierung des Schaltungsaufwands auf $n * log(n)$. Die Einschränkung auf eine Teilmenge aller möglichen Abbildungen bedeutet jedoch, daß selbst beim Zugriff auf unterschiedliche Speicherbänke Konflikte innerhalb des Verbindungsnetzwerkes entstehen können. Zu diesen Netzwerkkonflikten kommt es, wenn sich die Wege mehrerer Zugriffe im Netzwerk überschneiden

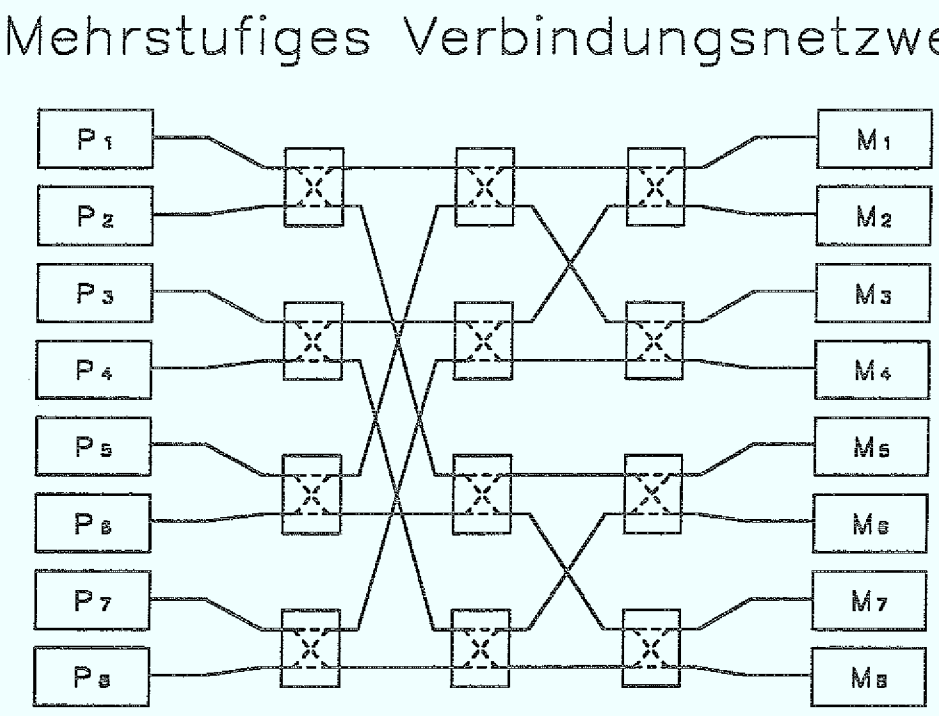


Abbildung 2.2: Computersystem mit Delta-Netzwerk als Beispiel für ein mehrstufiges Speicherverbindungsnetzwerk. P_i bezeichnet Prozessoren, M_i die Speicherbänke

und deshalb eine Leitung gleichzeitig für mehrere Verbindungen benötigt wird. Bei diesen Konflikten muß über eine Strategie ein Zugriff ausgewählt werden, der dann das Zugriffsrecht bekommt und ausgeführt wird. Die restlichen Zugriffswünsche können entweder vom Netzwerk abgewiesen oder im Netzwerk zwischengespeichert werden.

Werden Zugriffswünsche zwischengespeichert, so spricht man von blockierenden Netzwerken, da nicht sofort erfüllbare Zugriffswünsche Ressourcen des Netzwerkes belegen. Im Gegensatz dazu werden Netzwerke, die nicht sofort erfüllbare Zugriffswünsche zurückweisen, nicht-blockierende Netzwerke genannt. Blockierende Netzwerke besitzen meist eine höhere Durchsatzleistung [ST91], reagieren aber empfindlich auf eine ungleichmäßige Verteilung der Zugriffe auf die Speicherbänke (*hot spots*, [PN85, RT86]). Die Zwischenspeicher des Netzwerkes können sich bei Bankkonflikten sehr schnell füllen und damit das Netzwerk „sättigen". Dadurch werden Zugriffe auf andere Speicherbänke ebenfalls behindert. Die Leistung von blockierenden Netzwerken ist deshalb stärker von der Arbeitslast abhängig als die von nicht-blockierenden Netzwerken. Ein blockierendes Verbindungs-Netzwerk wurde im RP3-Projekt [PBG+85] verwendet, während im BBN Butterfly [RT86] ein nicht-blockierendes Netzwerk benutzt wird. Eine Untersuchung mehrstufiger Speicher-

verbindungsnetzwerke am Beispiel der CRAY X-MP wird in [Oed85] beschrieben; einen ausführlichen Vergleich von mehrstufigen Verbindungsnetzwerken findet man in [Hof90] (CRAY X-MP/Y-MP) und [CB88] (CRAY X-MP, CRAY 2). Eine allgemeine Einführung zum Thema Speicherverbindungsnetzwerke wird in [HJ88] gegeben.

2.1.3 Cache-Speicher

Die Leistungsfähigkeit von Rechnern mit Cache-Speicher basiert auf der Tatsache, daß die Speicherzugriffe eines Programms häufig sowohl zeitlich als auch räumlich lokal erfolgen. Die Cache-Speicher sind schnelle prozessornahe Speicher, die einen Teil der zuletzt angesprochenen Hauptspeicherplätze puffern und damit weitere Zugriffe auf diese Daten beschleunigen. Dazu werden bei einem Hauptspeicherzugriff des Prozessors sowohl die angesprochene Adresse als auch das dazugehörige Datum im Cache-Speicher zwischengespeichert. Fordert der Prozessor ein Datum aus einer der im Cache-Speicher enthaltenen Adressen an, so wird es nun aus dem schnelleren Cache-Speicher geliefert, ohne daß der Hauptspeicher durch einen Zugriff belastet wird. Für Multiprozessorsysteme mit gemeinsamem Speicher bedeutet dies weniger Zugriffskonflikte und somit eine höhere Systemleistung. Das Ziel eines Cache-Speicher-Designs ist es, einen möglichst hohen Anteil der Speicheranforderungen aus dem Cache-Speicher zu bedienen. Dieser Anteil wird durch die Cache-Fehlerrate (*cache miss ratio*) wiedergegeben, die durch den Quotient aus den erfolgten Hauptspeicherzugriffen und der Anzahl der Speicheranforderungen definiert ist. Dabei muß berücksichtigt werden, daß Cache-Speicher in Multiprozessorsystemen insbesondere die Belastung des gemeinsamen Speichers reduzieren sollen, und daß Designentscheidungen, die die Cache-Fehlerrate auf Kosten der Hauptspeicherbelastung verringern (*prefetch*), die Leistung des Computersystems reduzieren können. Die Cache-Fehlerrate, die mit einem Cache-Speicher bei der Bearbeitung einer Anwendung erreicht wird, hängt von der Wechselwirkung zwischen der Organisation des Cache-Speichers und dem Speicherzugriffsverhalten der Anwendung ab und kann deshalb nicht auf andere Anwendungen verallgemeinert werden.

Ein Cache-Speicher enthält ein Adresseninhaltsverzeichnis, das die Adressen der zwischengespeicherten Hauptspeicherplätze enthält, und einen Datenspeicher, in dem die dazugehörigen Daten gespeichert sind. Jeder Eintrag im Adresseninhaltsverzeichnis korrespondiert mit einem Ausschnitt des Datenspeichers, der in dieser Arbeit als Cache-Sektor bezeichnet wird, und für den in der Literatur auch häufig der Begriff der Cache-Zeile (*line*) verwendet wird. Da eine Cache-Zeile aber als die Datenmenge definiert ist, die bei einem Transfer zwischen Hauptspeicher und Cache-Speicher ausgetauscht wird [HB84, Smi82] und die beiden Definitionen nicht identisch sind, wird hier zwischen den beiden Begriffen differenziert. Ein Cache-Sektor ist in den meisten realisierten Cache-Speicher-Architekturen mit einer Cache-Zeile identisch, er kann aber auch mehrere Cache-Zeilen enthalten (Zilog Z800/80000

Prozessor). Die Größe des Cache-Datenspeichers wird durch die Anzahl der Cache-Sektoren und der Größe eines Cache-Sektors bestimmt.

Die Cache-Fehlerrate und die Hauptspeicherentlastung durch einen Cache-Speicher wird von der Organisation und der Kapazität des Datenspeichers und des Adresseninhaltsverzeichnisses sowie den Strategien zur Steuerung des Cache-Speichers beeinflußt. Die Datenspeicherkapazität hat durch die Anzahl der Cache-Sektoren einen unmittelbaren Einfluß auf die Cache-Fehlerrate. Werden zwei Cache-Speicher, die sich nur hinsichtlich der Anzahl der Cache-Sektoren unterscheiden, miteinander verglichen, so läßt sich zeigen, daß alle Daten des Cache-Speichers mit der geringeren Sektorenzahl auch in dem Cache-Speicher mit höheren Sektorenzahl enthalten sind. Durch eine Erhöhung der Sektorenanzahl eines Cache-Speichers kann die Fehlerrate daher in keinem Fall nachteilig beeinflußt werden; im allgemeinen ist dagegen mit einem deutlichen Rückgang der Fehlerrate zu rechnen. So wird in [Sto90] eine Erfahrungsregel genannt, wonach eine Verdoppelung des Cache-Datenspeichers häufig ein Reduzierung der Cache-Fehlerrate um 30% bewirkt.
Durch die Erhöhung der Cache-Zeilenlänge über die Datenbusbreite hinaus werden bei einem Hauptspeicherzugriff zusätzlich Daten aus Hauptspeicherplätzen mit darauf folgenden Adressen miteingelesen (*prefetch*). Da diese Speicherplätze mit hoher Wahrscheinlichkeit als nächstes benötigt werden und sie durch den *prefetch* schon im Cache-Speicher zur Verfügung stehen, kann die Cache-Fehlerrate dadurch verringert werden. Ist die Cache-Zeilenlänge größer als die Breite des Datenbusses, so müssen beim *prefetch* zusätzliche Speicherzugriffe durchgeführt werden. Da u.U. nicht alle der beim *prefetch* eingelesenen Daten auch benötigt werden, kann durch den *prefetch* die Belastung des Hauptspeichers erhöht werden [HB84].

Die Organisation des Adresseninhaltsverzeichnisses entscheidet, wie gut die Kapazität des Cache-Datenspeichers genutzt werden kann. Die Anzahl der Einträge dieses Inhaltsverzeichnisses ist identisch mit der Anzahl der Cache-Sektoren und wird bei gleicher Kapazität des Datenspeichers durch die Sektorlänge bestimmt. Die Anzahl der Cache-Sektoren bestimmt die maximale Anzahl von Speicherbereichen, die im Cache-Speicher gehalten werden können. Je mehr solcher Speicherbereiche im Cache-Speicher gehalten werden, desto effizienter kann der Datenspeicher des Cache genutzt werden. Durch die Zuordnung von Adressen auf Cache-Sektoren (*address mapping*) kann die Anzahl der effektiv speicherbaren Adreßbereiche eines Cache-Speichers jedoch reduziert werden. Das *address mapping* bestimmt, wie bei einer Anforderung einer Adresse durch den Prozessor nach dieser Adresse im Inhaltsverzeichnis gesucht wird. Man unterscheidet zwischen assoziativen, eindeutig abbildenden (*direct mapped*) und mengenassoziativen Inhaltsverzeichnissen.
Bei einem assoziativen Inhaltsverzeichnis wird jeder Eintrag im Inhaltsverzeichnis mit der vom Prozessor angeforderten Adresse verglichen. Der Cache-Datenspeicher wird von diesem Verfahren zwar optimal genutzt, jedoch ist die Realisierung eines assoziativen Inhaltsverzeichnisses für eine große Anzahl von Cache-Sektoren technisch aufwendig. Eine Realisierung eines assoziativen Inhaltsverzeichnisses mit nur vier Einträgen erfolgte bei den Cache-Speichern der ECLIPSE von Data General

[Hay88].
Den geringsten Aufwand verursacht ein eindeutig abbildendes Inhaltsverzeichnis, das jedem Bereich im Hauptspeicher über eine *hash*-Funktion eine Position im Adresseninhaltsverzeichnis zuordnet. Dadurch kann es vorkommen, daß mehrere benötigte Hauptspeicheradressen auf eine Position im Inhaltsverzeichnis abgebildet werden. Diese Adressen können dann unabhängig von der Kapazität des Cache-Datenspeichers nur alternativ im Cache-Speicher gehalten werden, so daß der restliche Datenspeicher dafür nicht genutzt werden kann.
Um die Nutzung des Datenspeichers mit vertretbarem Aufwand zu verbessern werden deshalb mengenassoziative Inhaltsverzeichnisse verwendet, die aus der Kombination von eindeutig abbildendem und assoziativem Inhaltsverzeichnis entstanden sind. Die *hash*-Funktion eines mengenassoziativen Inhaltsverzeichnisses bildet die Hauptspeicheradressen nicht auf einen Eintrag , sondern auf eine Menge (*set*) von Einträgen ab. Die Adresse wird dann wie beim vollassoziativen Inhaltsverzeichnis mit allen Einträgen der Menge verglichen. Der Nutzungsgrad des Cache-Datenspeichers hängt damit beim mengenassoziativen Inhaltsverzeichnis von der Anzahl der Einträge einer Menge ab. In üblichen Cache-Speichern werden 2-16 Einträge verwendet [HB84].

Die oben beschriebenen Organisationsformen des Adresseninhaltsverzeichnisses werden auch als Plazierungsstrategien bezeichnet. Neben der Plazierungsstrategie hat auch die Speicherersetzungsstrategie und die Speicheraktualisierungsstrategie einen Einfluß auf die Cache-Fehlerrate. Die Speicherersetzungsstrategie entscheidet, welcher Cache-Sektor eines Sets überschrieben wird, wenn neue Daten aus dem Hauptspeicher eingelesen werden. Eine optimale Speicherersetzungsstrategie würde die Cache-Zeile ersetzen, die in Zukunft von allen als letzte wieder benötigt wird. Da sich dies nicht a priori entscheiden läßt, müssen suboptimale Speicherersetzungsstrategien verwendet werden. Am häufigsten wird die Strategie *least recently used* (LRU) verwendet, die den Cache-Sektor ersetzt, der am längsten nicht mehr angesprochen wurde. Weitere Strategien sind *first in first out* (FIFO) und *random*; bei der Strategie FIFO wird der Cache-Sektor ersetzt, der sich am längsten im Cache-Speicher befindet, und in der Strategie *random* wird der zu ersetzende Cache-Sektor zufällig ausgewählt [Smi82]. Die mit der Strategie LRU bei realen Anwendungen erreichte Cache-Fehlerrate kann durch verbesserte Strategien nur noch um 10-30 % verringert werden. Eine ähnliche Steigerung läßt sich auch durch eine Verdopplung der Anzahl der Cache-Sektoren erreichen, so daß der Aufwand für neue Speicherersetzungsstrategien nicht den einer Verdopplung der Cache-Sektoren übersteigen sollte [Sto90].
Die Speicheraktualisierungsstrategie entscheidet, wann im Cache-Speicher veränderte Daten in den Hauptspeicher zurückgeschrieben werden. Bei der Strategie *write through* werden die Daten bei jedem Schreibzugriff des Prozessors sowohl in den Cache-Speicher als auch in den Hauptspeicher geschrieben. Eine geringere Belastung des Hauptspeichers erreicht man durch die Verwendung der *write-in*-Strategie. Ein *write-in*-Cache-Speicher speichert die durch einen Schreibzugriff veränderten Daten solange zwischen, bis der Cache-Sektor mit den veränderten Daten für neue Daten

benötigt wird; erst dann werden die Daten in den Hauptspeicher geschrieben. Ein *write-in*-Cache-Speicher wird z.B. in den IBM-Rechnern der Serien 308X und 3090 verwendet [Tuc86, Smi82].

Durch eine Aufteilung des Cache-Speichers kann eine Spezialisierung der separaten Cache-Speicher erfolgen, die dann die Cache-Fehlerrate senken soll. Es bietet sich eine Aufteilung in Instruktions- und Daten-Cache sowie System- und Benutzer-Cache an [HB84]. Die Aufteilung in Instruktions- und Daten-Cache ist sinnvoll, da der Befehlsfluß im allgemeinen eine höhere Lokalität als der Datenfluß aufweist und Befehle nur aus dem Hauptspeicher gelesen, nicht aber zurückgeschrieben werden, wenn die Verwendung von sich selbst modifizierendem Code ausgeschlossen wird. Die Einführung eines System-Caches für die Systemdaten des Betriebssystems soll die Leistung des Cache-Speicher für Systemaufgaben erhöhen. Die Systemaufgaben werden mit den Benutzerprogrammen zeitlich verzahnt bearbeitet, wodurch die Systemdaten bei einem gemeinsamen Cache-Speicher überschrieben werden und deshalb neu aus dem Hauptspeicher gelesen werden müssen. Eine Leistungssteigerung durch Unterteilung des Cache-Speichers in System- und Benutzerbereich konnte in [AHH88] jedoch nicht nachgewiesen werden.
Ein besonders schneller Zugriff auf den Cache-Speicher wird erreicht, wenn sich Cache-Speicher und Prozessor auf demselben *chip* befinden. Da der *on-chip*-Cache-Speicher stärker in der Größe eingeschränkt ist als ein externer Cache-Speicher, werden auch Hierarchien von Cache-Speichern verwendet. Diese bestehen dann aus kleinen, besonders schnellen Cache-Speichern, die sich sehr nahe an den Prozessoren befinden, und etwas langsameren und größeren Cache-Speicher näher am Speicher.

Für den Mehrprogrammbetrieb eines Cache-Speichers ist es von Bedeutung, ob bei der Verwendung eines virtuellen Speichers die realen oder die virtuellen Adressen im Adresseninhaltsverzeichnis verwendet werden. Die Verwendung von virtuellen Adressen beschleunigt den Zugriff auf den Cache-Speicher, führt jedoch zu dem Problem, daß verschiedene Jobs gleiche virtuelle Adressen für unterschiedliche private Speicherbereiche verwenden. Um eine fehlerhafte Funktionsweise des Cache-Speichers auszuschließen, muß der Inhalt des Cache-Speichers bei einem *task*-Wechsel für ungültig erklärt und bei einem *write-in*-Cache-Speicher zusätzlich gerettet werden (*cache flush*). Der Aufwand entfällt, wenn jeder Eintrag im Adresseninhaltsverzeichnis um eine *task*-Nummer ergänzt wird, über die der Cache-Speicher erkennen kann, ob die Daten zu der ausgeführten *task* gehören [Ste90]. Eine interessante Variante dieses Verfahrens wird in den SPARC, MIPS R4000 und den i860 Prozessoren verwendet: Die Auswahl der Cache-Zeile durch die *hash*-Funktion erfolgt unter Benutzung der virtuellen Adresse, während die eindeutige physikalische Adresse als „*task*-Nummer" verwendet wird, um zu überprüfen, ob die Daten zu der ausgeführten *task* gehören.

In einem Multiprozessor-System können die Cache-Speicher privat von jedem Prozessor oder auch von mehreren Prozessoren gemeinsam genutzt werden. Gibt es in einem Rechnersystem mehr als einen Cache-Speicher, so muß die Veränderung von

Daten in einem Cache-Speicher bei allen Cache-Speichern, die denselben Speicherbereich puffern, durchgeführt werden, um eine korrekte Programmausführung sicherzustellen. Ein Ein-/Ausgabesystem, das selbständig mit dem Hauptspeicher kommuniziert, kann ebenfalls Hauptspeicherplätze verändern, ohne daß diese in den Cache-Speichern aktualisiert werden und muß deshalb in die Konsistenzstrategie mit einbezogen werden. Die Cache-Speicher-Konsistenzstrategie hat einen besonderen Einfluß auf die Ausführung von parallelen Programmen, da die Kommunikation der *tasks* über gemeinsame Speicherplätze abläuft.
Die Cache-Speicher-Konsistenz kann entweder transparent für die Software durch Hardware erfolgen oder durch die Software unterstützt bzw. sichergestellt werden. Um eine Durchführung der Cache-Speicher-Konsistenz durch Software zu ermöglichen, muß die Hardware spezielle Instruktionen zur Verfügung stellen, mit denen Daten im Cache als ungültig markiert werden können und die es erlauben direkt auf den Hauptspeicher zuzugreifen.

Die Hardware-Konsistenzstrategien lassen sich in entwertende und aktualisierende Strategien unterteilen. Die entwertenden Strategien (*write invalidate*) markieren die Cache-Zeilen mit nicht mehr aktuellen Daten als ungültig, so daß beim nächsten Zugriff auf eine solche Cache-Zeile ein Cache-Fehler und somit ein Hauptspeicherzugriff ausgelöst wird. Eine aktualisierende Strategie (*write-update*) überträgt bei einem Schreibzugriff die Daten direkt zu allen anderen Cache-Speichern, die den Speicherplatz zwischenspeichern. Im Vergleich zu einer entwertenden Strategie verringert eine aktualisierende Strategie die Hauptspeicherbelastung, führt aber zu mehr Belastung der Kommunikationsressourcen der Cache-Speicher.
Die Kommunikationsstrategie entscheidet, ob eine Aktualisierungs-/Entwertungsmeldung allen (mittels *broadcast*) oder nur den betroffenen Cache-Speichern (mittels *multicast*) mitgeteilt wird. Die Kommunikationsstrategien, die alle Cache-Speicher informieren, sind typisch für Computersysteme mit einem gemeinsamen Bus zum Hauptspeicher, da dieser für *broadcasts* besonders gut geeignet ist. Für Computersysteme mit mehrstufigen Verbindungsnetzwerken ist der Kommunikationsaufwand eines *broadcasts* sehr hoch. Deshalb sind Kommunikationsstrategien entwickelt worden, die ein Inhaltsverzeichnis aller Cache-Speicher mit gemeinsamen Speicherbereichen führen. Mit dem Inhaltsverzeichnis können die Cache-Speicher ermittelt werden, die über die Veränderung von zwischengespeicherten Speicherbereichen mit einem *multicast* informiert werden müssen. Wegen des hohen Hardware-Aufwands sind Inhaltsverzeichnisbasierte Konsistenz-Strategien bisher noch nicht in kommerziellen Supercomputern verwendet worden [Ste90].
In der CONVEX C3-Serie wird eine Cache-Speicher-Kommunikation mittels *broadcast* verwendet. Jeder Cache-Speicher überwacht dabei alle Prozessor-*ports* sowie das Ein-/Ausgabesystem und markiert die angesprochenen Speicherplätze im eigenen Cache-Speicher als ungültig (*write invalidate*). Werden den Cache-Speichern feste disjunkte Speicherbereiche zugewiesen, so wird keine Konsistenzstrategie benötigt. Auf Multiprozessorsystemen mit gemeinsamem Speicher ist dies der Fall, wenn sich die Cache-Speicher nicht bei den Prozessoren, sondern bei den Speicherbänken befinden. Ein solches Cache-Speicher-Design reduziert nur die Zugriffszeit auf die

Speicherbänke nicht aber die Häufigkeit von Zugriffskonflikten. In der Alliant FX/2800 werden deshalb sowohl an den Prozessoren als auch an den Speicherbänken Cache-Speicher eingesetzt.

2.1.4 Virtueller Speicher

Ein Computersystem mit virtuellem Speicher unterscheidet zwischen dem Speicher aus der Sicht des Programms, dem sogenannten logischen oder virtuellen Speicher, und dem Speicher aus der Sicht der Maschine, dem physikalischen oder realen Speicher. Eine Architektur, die virtuellen Speicher unterstützt, braucht nur die aktiven Teile des Jobspeicherplatzes im realen Hauptspeicher zu halten und kann deshalb mehr Jobs im Hauptspeicher halten und diesen somit effektiver nutzen. Dies wirkt sich besonders auf den Zeitscheibenbetrieb (*time sharing*) aus, da speicherintensive Jobs bei einem realen Hauptspeicherkonzept nur alternativ im Hauptspeicher ausgeführt werden können und daher abwechselnd auf den Sekundärspeicher ausgelagert werden müssen (*swapping*).
Die Hardware des virtuellen Speichers muß die Adressen des virtuellen Speichers auf die Adressen des realen Speichers abbilden. Dazu unterteilt man sowohl den virtuellen als auch den realen Speicher in Speicherbereiche einer festen Größe (z.B. 4096 Byte). Diese Speicherbereiche werden im virtuellen Speicher als Seiten (*pages*) und im realen Speicher als Rahmen (*page frames*) bezeichnet.
Eine Adresse in einem virtuellen Speicher besteht dann aus einer Seitennummer und einem *offset*. Der *offset* gibt die Adresse innerhalb der Seite an. Die Seiten des virtuellen Speichers werden durch eine Übersetzungstabelle (*page table*) auf die Rahmen des realen Speichers abgebildet. Der *offset* wird dann auf die Adresse des Rahmen addiert und bildet so die Adresse des Realspeichers [HB84].

Da jedes Programm im Mehrprogrammbetrieb seinen eigenen virtuellen Speicher besitzt und der virtuelle Speicher eines Programms selbst schon größer als der Realspeicher sein kann, gibt es i. allg. mehr Seiten in den virtuellen Speichern als Rahmen im Realspeicher. Wird eine Seite A des virtuellen Speichers benötigt, die sich nicht im realen Speicher befindet, so kommt es zu einem *Seitenfehler*, und der Prozeß wird angehalten. Befindet sich bei der Behandlung des Seitenfehlers kein freier Rahmen mehr im realen Speicher, so wird durch die Seitenersetzungsstrategie ein Seite B ausgewählt, die ersetzt und deren Rahmen für die Seite A verwendet werden soll. Dazu muß die Seite B in der Übersetzungstabelle, in der sie bisher eingetragen war, als ausgelagert markiert werden, und die Seite B muß, falls sie verändert wurde, auf den Sekundärspeicher gerettet werden. Wurde die benötigte Seite A des virtuellen Speichers schon einmal auf den Sekundärspeicher gerettet, so muß sie wieder eingelagert werden. Befindet sie sich nun im realen Speicher, so kann der neue Rahmen in die Übersetzungstabelle des Prozesses eingetragen werden, und der Prozeß kann seine Arbeit fortsetzen.

Die Strategien, die in virtuellen Speichern zur Seitenersetzung verwendet werden,

ähneln denen, die in Cache-Speichern zur Speicherersetzung der Cache-Sektoren eingesetzt werden. Da ein Seitenfehler bei einem virtuellen Speicher aber einen zeitaufwendigen Zugriff auf den Sekundärspeicher nach sich zieht, ist der für die Seitenersetzungsstrategie benötigte Zeitaufwand nicht so bedeutend wie bei einem Cache-Speicher. Die Seitenersetzungsstrategie eines virtuellen Speichers wird deshalb vom Betriebssystem, also durch Software, durchgeführt.
Die Ausführungsgeschwindigkeit eines Programms hängt in einem System mit virtuellem Speicher stark von der Häufigkeit der Seitenfehler ab. Je mehr Seiten eines Prozesses sich im realen Speicher befinden, desto weniger Seitenfehler können auftreten. Um unnötige Wartezeiten für die Prozessoren zu vermeiden und einen maximalen Durchsatz zu erreichen, kann bei einem Seitenfehler vom Betriebssystem ein *task*-Wechsel durchgeführt und ein anderer ablauffähiger Prozeß bearbeitet werden. Durch den *task*-Wechsel entsteht dann jedoch ein zusätzlicher Aufwand. Je mehr Prozesse sich im Speicher befinden, desto mehr ausführbare Prozesse existieren, aber desto weniger Seiten des realen Speichers stehen für jeden einzelnen Prozeß zur Verfügung. Die Leistung eines Computersystems mit virtuellem Speicher hängt also von der Kapazität des realen Hauptspeichers ab. Befindet sich kein Prozeß im Hauptspeicher, der in der Wartezeit gestartet werden kann, so wird die Leistung von der Geschwindigkeit des Ein-/Ausgabesystems bestimmt.

Die Übersetzungstabelle ist im allgemeinen zu groß, um sie komplett im Prozessor zu halten. Sie befindet sich deshalb im Hauptspeicher und wird oft sogar mehrstufig, Baum-ähnlich organisiert. Dadurch wird für jeden Hauptspeicherzugriff des Prozessors ein zweiter Hauptspeicherzugriff auf die Übersetzungstabelle nötig. Dieses Problem verschärft sich bei mehrstufigen Übersetzungstabellen, verringert die Leistung der Prozessoren und führt zu einer stärkeren Belastung des Hauptspeichers. Ein *translation lookaside buffer* (TLB) ist ein spezieller Cache-Speicher, der Einträge der Übersetzungstabelle lokal beim Prozessor zwischenspeichert. Da der TLB ein Cache-Speicher ist, gibt es ebenso viele Design-Alternativen wie für einen Daten-Cache-Speicher; da ein TLB aber eine besondere Aufgabe erfüllt, können diese Besonderheiten beim Entwurf berücksichtigt werden. Ein TLB ist meist erheblich kleiner, da jeder Eintrag des TLB für den Zugriff auf eine ganze Seite des Hauptspeichers verwendet werden kann. Außerdem erfolgen Schreibzugriffe auf die Übersetzungstabelle nur bei Seitenfehlern oder bei der Änderung der Markierungen, die die Benutzung (*referenced bit*) und Veränderung (*dirty bit*) der Seiten anzeigen; sie erfolgen somit nur selten. Da die Einträge im TLB zu der Übersetzungstabelle der aktuellen *task* gehören, muß der TLB bei einem *task*-Wechsel für ungültig erklärt werden (*TLB flush*), es sei denn, jeder Eintrag im TLB ist mit einer *task*-Nummer gekennzeichnet, die eine Überprüfung der Zugehörigkeit zur aktuellen *task* ermöglicht.

Auch TLBs können inkonsistent werden, wenn die Information in einer Übersetzungstabelle von einem Prozessor aktualisiert wird und ein anderer Prozessor noch den alten Eintrag der Übersetzungstabelle enthält. Da ein Computersystem mit virtueller Speicherstruktur nur effizient arbeiten kann, wenn Seitenfehler und da-

mit selten Veränderungen der Übersetzungstabelle auftreten, hat der Zeitaufwand, der zur Herstellung der TLB-Konsistenz benötigt wird, keinen großen Einfluß auf die Leistung des Computersystems. Erst bei großen Prozessorzahlen, wie z.B. beim RP3-Projekt [PBG+85] wirkt sich der Aufwand der Konsistenzstrategie aus. Die bisher am häufigsten verwendete Technik des *TLB shootdown* wird ohne Hardware-Unterstützung durchgeführt und wird im RP3-Projekt, dem BBN Butterfly und der Encore Multimax eingesetzt [Tel90]. Bei dieser Technik kennzeichnet ein Prozessor die Übersetzungstabelle, die er modifizieren will, schickt einen globalen Interrupt an alle anderen Prozessoren, wartet bis alle Prozessoren den Interrupt bestätigt haben, modifiziert die Übersetzungstabelle und gibt sie wieder frei. Die anderen Prozessoren bestätigen zuerst den Interrupt und testen dann, ob sie die zu modifizierende Übersetzungstabelle benötigen. Wird die Übersetzungstabelle von dem Prozessor nicht benutzt, so kann er normal weiterarbeiten, benutzt er sie aber, so muß er warten, bis die Übersetzungstabelle wieder freigegeben wird, und muß dann die veränderten TLB-Einträge neu laden. In [Tel90] werden diese und andere Möglichkeiten zur Herstellung der TLB-Konsistenz vorgestellt und analysiert.

Ein virtuelles Speicherkonzept wird z.B. in der Alliant FX-2800 und der Convex C3 verwendet. Es hat sich insbesondere auch in dem Bereich der Großrechner (z.B. IBM 3090, ES/9000) durchgesetzt. Da die Leistung eines Supercomputers essentiell von der Speicherbandbreite abhängt, kann ein virtuelles Speicherkonzept die Leistung eines Supercomputers bei ungünstigen Arbeitslasten drastisch reduzieren. Da die Leistungseinbußen, die durch die Übersetzung der virtuellen Adressen und dem Transfer von Seiten zwischen Haupt- und Sekundärspeicher verursacht werden, in einem Computersystem mit virtuellem Speicher unvermeidbar sind, verwenden einige Computer ein reales Hauptspeicherkonzept. Zu diesen Vektorrechner mit realer Adressierung des Hauptspeichers gehören z.B. alle Systeme der Firma CRAY, die FUJITSU VP2000 und die NEC SX-3 [BEK90].

2.1.5 Ein-/Ausgabesystem

Das Ein-/Ausgabesystem (kurz EA-System) ermöglicht den Zugriff auf den Sekundärspeicher und bestimmt die Leistung der Jobs, die große Datenmengen verarbeiten und deshalb auf die Nutzung des Sekundärspeichers angewiesen sind. Um die Vektorprozessoren für Rechenoperationen freizuhalten, werden die EA-Operationen durch spezielle EA-Prozessoren durchgeführt, die über EA-Kanäle mit dem gemeinsamen Hauptspeicher kommunizieren. Die EA-Bandbreite des Computersystems ist die maximale Datenrate, mit der Daten zwischen dem Hauptspeicher und dem EA-System transferiert werden können, und sie ist durch die Summe der Bandbreiten aller EA-Kanäle beschränkt. Da die CRAY-1 als einer der ersten Vektorsupercomputer 1976 noch nicht über ein eigenständigen EA-System mit unabhängigen EA-Prozessoren verfügte, wurde ihre Leistung durch häufige EA-Operationen signifikant reduziert. Seit der CRAY-1S, dem Nachfolger der CRAY 1, besitzen die Vektorsu-

percomputer der Firma CRAY deshalb ein eigenständiges EA-System [HJ88]. Die Daten müssen dann noch vom EA-System auf die Sekundärspeicher übertragen werden. Neben der Datenübertragungsrate wirkt sich bei üblichen Sekundärspeichern auch die Zugriffsverzögerung leistungsmindernd aus. Deshalb werden heute zum Teil Halbleiter-Pufferspeicher benutzt, in denen die Daten vom EA-System zwischengespeichert werden, bis diese auf den Sekundärspeicher übertragen werden können. Diese müssen eine hinreichend große Kapazität besitzen, um die Daten, die innerhalb der Zugriffszeit des Sekundärspeichers anfallen, aufnehmen zu können. Um eine hohe Datenübertragungsrate zu erzielen, werden heute üblicherweise mehrere Magnetplatten mit jeweils mehreren gleichzeitig arbeitenden Schreib-/Leseköpfen (PTD = *parallel transfer disk*) verwendet. Höhere Datenraten sollen in Zukunft durch massiv parallelen Zugriff auf Magnetplatten-Arrays (RAID = *redundant arrays of inexpensive disks*) erreicht werden, jedoch liegt die Zugriffszeit dieser Platten-*arrays* über der von PTD-Platten. Die RAID-Technologie zeichnet sich außerdem durch eine besondere Berücksichtigung der Fehlererkennung und Fehlerkorrektur aus. In der Wartezeit auf den Sekundärspeicherzugriff erfolgt im Mehrprogrammbetrieb üblicherweise ein *task*-Wechsel. Ist dies jedoch nicht möglich, da die gesamten Ressourcen von einem oder wenigen Jobs genutzt werden, so wird die Leistung des Computersystems u.U. unabhängig von der EA-Bandbreite deutlich verringert.

Der Leistungsabfall durch Zugriffsverzögerungen entfällt für geeignete Anwendungen durch eine Erweiterung der Speicherhierarchie um einen zusätzlichen großen Halbleiterspeicher, der unter unterschiedlichen Bezeichnungen (SSD, ECS oder LCS) in kommerziellen Computersystemen zur Anwendung kommt. Da dieser Speicher zu einem niedrigen, wirtschaftlichen Preis möglichst groß sein soll, wird hierfür ein besonders günstiger Halbleiterspeicher verwendet, der meist eine erheblich höhere Zugriffszeit als der Hauptspeicher aufweist. Die Zugriffszeit eines solchen Speichers liegt damit aber immer noch um Größenordnungen unter der Zugriffszeit üblicher Sekundärspeicher und führt deshalb zu einer deutlichen Leistungssteigerung, falls die zur Verfügung gestellte Speicherkapazität für die EA-intensiven Jobs ausreicht. Unter anderem kann auch die CRAY X-MP mit einem solchen Halbleiterspeicher (SSD = *solid-state storage device*) ausgestattet werden [HJ88, HB84]; die CRAY Y-MP ist serienmäßig mit einem SSD ausgerüstet [BEK90]. Die damit erreichte Leistungssteigerung wird in [AGS88] beschrieben und mit der Leistung der CRAY 2, eines Supercomputers mit großer Hauptspeicherkapazität, verglichen.

2.1.6 Synchronisation

Da für die korrekte Ausführung eines Programms die Reihenfolge der Berechnungen von Bedeutung ist, müssen parallele Programme synchronisiert werden, d.h. bei vorliegenden Datenabhängigkeiten muß sichergestellt werden, daß eine Berechnung beendet ist, bevor die nächste begonnen wird. Die Synchronisationsleistung eines Computersystems kann analog zur Rechenleistung in Millionen Synchronisations-

operationen pro Sekunde (MSYPS = *million synchronisation operations per second*) angegeben werden und ist für feingranulare Programme ebenso leistungsbegrenzend wie die Rechenleistung [Sto90].
Auf Computersystemen mit gemeinsamem Speicher kann die Synchronisation effektiv mittels nicht-unterbrechbarer *read-modify-write*-Operationen durchgeführt werden, die einen Wert aus einem Speicherplatz auslesen, diesen verarbeiten und zurückschreiben, ohne daß diese Sequenz unterbrochen werden kann. Typische Vertreter dieser zur Synchronisation verwendeten Operationen sind die *test-and-set-*, die *compare-and-swap-* und die *fetch-and-operator*-Instruktionen [Sto90]. Ein Vertreter der *fetch and operator*-Operationen ist die *fetch-and-add*-Operation, die einen Wert aus einem Speicherplatz liest, einen zweiten Wert dazuaddiert und das Ergebnis in den Speicherplatz zurückschreibt. Die zur Synchronisation verwendeten Operationen sind entweder blockierend oder nicht-blockierend. Eine blockierende Operation hält den Prozessor solange an, bis die Operation durchgeführt werden kann, während eine nicht-blockierende Operation ein Resultat liefert und bei einem ungünstigen Ergebnis eventuell wiederholt werden muß. Blockierende Operationen halten den Prozessor an (*busy waiting*) und verhindern im Gegensatz zu nichtblockierenden Operationen eine alternative Nutzung des Prozessors.
Da alle zu synchronisierenden Prozessoren bei den üblichen Synchronisierungsstrategien auf einen gemeinsamen Speicherplatz bzw. ein gemeinsames Register zugreifen, entstehen Zugriffskonflikte, deren Anzahl mit der Prozessorzahl steigt. Ist das Computersystem mit Cache-Speichern ausgestattet, so kann die Cache-Speicher-Konsistenzstrategie einen Einfluß auf die Synchronisationsleistung des Computersystems haben. Bei Software-Konsistenzstrategien und entwertenden Hardware-Konsistenzstrategien erfolgt die Synchronisation über den gemeinsamen Hauptspeicher. Im Gegensatz dazu erfolgt die Synchronisation bei aktualisierenden Hardware-Konsistenzstrategien über die Kommunikationsressource, über die die Cache-Speicher aktualisiert werden (meistens ein Bus-System), und wird deshalb nicht von der Leistung der Hauptspeicherarchitektur begrenzt.

Die Verwendung von gemeinsamen Registern ermöglicht eine sehr schnelle Prozessor-Prozessor Kommunikation und Synchronisation. Die gemeinsamen Register, die zur Synchronisation dienen sollen, müssen durch nicht-unterbrechbare *read-modify-write*-Operationen ansprechbar sein. Im Mehrprogrammbetrieb mit mehreren parallelen Programmen muß sichergestellt werden, daß die verschiedenen parallelen Programme nicht dieselben gemeinsamen Register benutzen. Eine Möglichkeit dazu besteht in der Verfügbarkeit mehrerer Registersätze, von denen einer für jedes parallele Programm von allen seinen parallelen Prozessen gemeinsam genutzt wird; diese Möglichkeit wird in der CRAY X-MP/Y-MP verwendet. Die andere Möglichkeit besteht in einem indirekten Zugriff auf die gemeinsamen Register mittels eines Basisregisters und einer relativen Adressierung; das Basisregister der Prozessoren eines parallelen Jobs zeigt dann auf denselben Registerbereich, während die Basisregister der Prozessoren, die unterschiedliche Jobs bearbeiten, auf unterschiedliche Bereiche zeigen. In der Convex C-2 Serie wird ein solcher gemeinsamer Registerbereich von 1024 Werten über ein Basisregister (*communication index register*) an-

gesprochen. Die Prozessorkommunikation verläuft jedoch auch bei gemeinsamen Registersätzen über ein Verbindungsnetzwerk. Mit steigender Prozessorenzahl wird es immer schwieriger eine schnelle Kommunikation der Prozessoren durch ein solches Netzwerk zu ermöglichen. Bei der CRAY X-MP/Y-MP Serie ist dieser Effekt deutlich zu beobachten (siehe Tabelle 2.1, [Rei88]). Außerdem entstehen bei einer hohen Anzahl von Synchronisationsoperationen Zugriffskonflikte auf die gemeinsamen Register, die sequentialisiert werden müssen.

CRAY-Rechner	X-MP/22	X-MP/416	Y-MP/832
Prozessoren	2	4	8
Test-and-Set	1	1	3
Lade *shared register*	1	3	10
shared register reservation time	1	1	3–7

Tabelle 2.1: Anstieg der für verschiedene Operationen auf gemeinsamen Registern benötigten Taktzyklen mit der Prozessorzahl auf der CRAY X-MP/Y-MP [Rei88].

Die Vermeidung von Konflikten beim Zugriff auf einen gemeinsamen Speicherplatz bei der Synchronisation wird durch ein kombinierendes Verbindungsnetzwerk (*combining network*) erreicht. Das kombinierende Netzwerk ist ein spezielles mehrstufiges Verbindungsnetzwerk (Omega-Netzwerk), das die Ausführung der *fetch-and-add*-Operation mehrerer Prozessoren nach dem Prinzip des rekursiven Doppelns parallelisiert und in logarithmischer Zeitkomplexität durchführt. Dazu werden die *fetch-and-add*-Operationen, die denselben Speicherplatz ansprechen, in den Schaltern einer Stufe zu einer kombinierten *fetch-and-add*-Operation zusammengefaßt und diese dann an die nächste Stufe des Netzes weitergegeben. Wenn das Ergebnis der Operation aus dem Netz zu dem Schalter zurückkommt, werden daraus die Ergebnisse abgeleitet, die an die Eingänge zurückgegeben werden. Dadurch kann eine gleichzeitig angeforderte *fetch-and-add*-Operation aller Prozessoren auf dieselbe Speicherstelle auf eine einzige Anfrage an den Speicher reduziert werden und in einem Durchlauf durch das Netzwerk für alle Prozessoren durchgeführt werden. Ein kombinierendes Netzwerk war für das RP3-Projekt geplant, wurde aber wegen des großen Aufwands verworfen [PN85, Sto90].
Eine technisch weniger aufwendige Methode für die gleichzeitige Synchronisation aller Prozessoren eines parallelen Programms bietet die *barrier*-Anweisung. Eine *barrier*-Anweisung stellt sicher, daß alle parallelen Prozesse eines parallel zu bearbeitenden Bereiches ihre Arbeit beendet haben, bevor neue Berechnungen begonnen werden. In einer parallelen Region werden durch die *barrier*-Anweisung die angekommenen Prozesse solange angehalten, bis alle Prozesse an der *barrier* angekommen sind. Daher ist diese Anweisung auch nur für Aufgaben geeignet, die eine globale Synchronisation benötigen. Die Hardware-Unterstützung von *barrier*-Anweisungen

erfordert wenig Aufwand [Sto90, HB84]. Für den Mehrprogrammbetrieb von parallelen Programmen müssen mehrere Hardware-*barrier* (*multiple barrier*) vorhanden sein, da diese ebenso wie die gemeinsamen Registersätze für mehrere parallele Programme zur Verfügung stehen.

2.2 Betriebssystem

Das Betriebssystem eines Computers soll den Benutzer bei seiner Arbeit unterstützen und eine möglichst gute Nutzung des Computers ermöglichen. Die Anforderungen, die an ein Betriebssystem in Bezug auf die Benutzerfreundlichkeit, Fairneß und Sicherheit gestellt werden, steigen stetig. Um diesen Anforderungen gerecht zu werden, benötigt ein Betriebssystem jedoch einen Teil der Leistung des Computersystems (*operating system overhead*) und verringert die Leistung, die für den Benutzer zur Verfügung steht. Eine effiziente Implementierung des Betriebssystems erhöht den Anteil der Leistung, der den Benutzern zur Verfügung gestellt wird. Im Bereich der Supercomputer ist die Leistung von so zentraler Bedeutung, daß die Effizienz des Betriebssystems häufig gegenüber anderen Anforderungen den Vorrang besitzt [Wil88].

Bei den überwiegend im Einprogrammbetrieb durchgeführten Leistungsmessungen auf Supercomputern ist die Bedeutung des Betriebssystems für die gemessene Leistung nur sekundär. Eine wirtschaftliche Nutzung im Produktions- oder im Rechenzentrumsbetrieb setzt jedoch oft den Mehrprogrammbetrieb des Computersystems voraus. Im Mehrprogrammbetrieb verteilt das Betriebssystem die Ressourcen des Computersystems an die Jobs [SPG91]. Diese Aufgabe kann eine hohe Relevanz für die Leistung eines Computersystems besitzen. Der Mehrprogrammbetrieb kann im Batch- oder im interaktiven Betrieb erfolgen. Der Batch-Betrieb favorisiert eine gute Nutzung des Computersystems und damit einen hohen Durchsatz. Er bietet sich deshalb insbesondere für längere Produktionsrechnungen und den Nachtbetrieb von Computersystemen an. Im Gegensatz dazu ist im interaktiven Betrieb eine kurze Antwortzeit des Systems gegenüber der Durchsatzleistung vorrangig. Der interaktive Betrieb von Computersystemen ist deshalb besonders vorteilhaft für die Programmentwicklung und kurze experimentelle Programmläufe. Die meisten der heute verwendeten Betriebssysteme für Supercomputer wie UNICOS (CRAY), CONCENTRIX (Alliant) und ConvexOS (Convex) basieren auf dem Betriebssystem UNIX [Gul88, Tan90] und stellen dem Benutzer somit eine standardisierte Umgebung zur Verfügung, die insbesondere den interaktiven Betrieb unterstützt. Diese Betriebssysteme werden durch Erweiterungen wie das NQS (*network queueing system*) ergänzt, wodurch sie gleichermaßen gut für den Batch-Betrieb geeignet sind.

Der Mehrprogrammbetrieb von parallelen Programmen wird insbesondere durch die Zuteilung der Prozessoren (*scheduling*) und des Hauptspeichers (*memory management*) beeinflußt. Da die hier betrachteten Computersysteme mit einem gemeinsamen Hauptspeicher ausgestattet sind und somit jeder Prozessor auf jeden Speicherbereich zugreifen kann, können Scheduling und Speicherverwaltung im wesentlichen unabhängig voneinander betrachtet werden.

2.2.1 Speicherverwaltung

Die Speicherverwaltung (*memory management*) verteilt den zur Verfügung stehenden Hauptspeicher auf die auszuführenden Jobs. Die Strategien zur Speicherverwaltung beeinflußen die Durchsatzleistung einerseits über die erreichte Speicherauslastung und andererseits über den induzierten Aufwand. Eine gute Speicherauslastung kann im Mehrprogrammbetrieb zu einer größeren Anzahl von ausführbereiten Programmen im Hauptspeicher und damit zu einer besseren Prozessornutzung führen. Im Zeitscheibenbetrieb wird der für das Ein- und Auslagern von Jobs benötigte Aufwand ebenfalls von der Effektivität der Speichernutzung bestimmt. Da der für die Speicherverwaltung benötigte Aufwand die für die Bearbeitung der Jobs zur Verfügung stehende Prozessorzeit reduziert, müssen Aufwand und Speichernutzung gegeneinander abgewogen werden. Grundsätzlich muß zwischen der Speicherverwaltung von Computersystemen mit realem und virtuellem Speicherkonzept unterschieden werden, da in den beiden Konzepten völlig unterschiedliche Strategien zum Einsatz kommen.

Bei einer Speicherverwaltung für ein reales Hauptspeicherkonzept kann eine gute Nutzung des Hauptspeichers nur mit großem Aufwand garantiert werden. Man unterscheidet zwischen Strategien mit festen und variablen Speicherbereichen (Partitionen). Bei einer Speicherverwaltung mit festen Partitionen wird der Realspeicher des Computers fest in verschiedene Speicherbereiche von möglicherweise unterschiedlicher Größe eingeteilt. Jedes Programm kann dann nur eine Partition belegen und deshalb also nur innerhalb einer Partition mit ausreichender Speicherkapazität ausgeführt werden. Dadurch wird der Speicherplatz einer Partition, die größer als der von dem Programm benötigte Hauptspeicher ist, nicht genutzt (*interne Fragmentierung*). Programme, die mehr Speicherplatz benötigen als durch jede einzelne Partition zur Verfügung gestellt wird, können bei diesem Konzept nicht in den Speicher geladen werden, auch wenn die Gesamtkapazität aller freien Partitionen für das Programm ausreichen würde. Eine Speicherverwaltung mit festen Partitionen nutzt den Hauptspeicher daher relativ schlecht, benötigt aber nur geringen Verwaltungsaufwand. Da die Anzahl der ausführbereiten Jobs bei diesem Verfahren durch die Anzahl der Partitionen beschränkt wird, bezeichnet man dieses Verfahren auch als MFT (*multiprogramming with a fixed number of tasks*) [SPG91].
Eine Speicherverwaltung mit variablen Partitionen belegt dagegen immer genau den Speicherplatz, der von einem Job angefordert wird. Dazu muß die Speicherverwaltung für jede Speicheranforderung eine möglichst geeignete Stelle im Hauptspeicher auswählen. In der Standardliteratur über Betriebssysteme werden die dazu verwendeten Verfahren vorgestellt [SPG91, Tan90]: *first fit, best fit, worst fit* etc. Bei der Speicherverwaltung mit variablen Partitionen entstehen durch die unterschiedliche Größe der angeforderten Speicherbereiche immer häufiger Lücken zwischen den genutzten Speicherbereichen, die dazu führen, daß immer mehr kleine, aber kaum noch größere zusammenhängende Partitionen existieren (*externe Fragmentierung*). Durch verschiedene Verfahren können aufeinander folgende Speicherblöcke wieder zu größeren zusammengefaßt werden, so daß der Prozeß der Fragmentie-

rung verlangsamt, aber nicht gestoppt wird. Gelöst werden kann dieses Problem durch eine Speicherkompaktifizierung, bei der alle genutzten Speicherbereiche zusammengeschoben werden, wodurch ein großer Freispeicherbereich entsteht. Die Kompaktifizierung ist aber so aufwendig, daß sie nur selten verwendet wird [HB84]. Da die Anzahl der ausführbereiten Jobs in einem Computersystem mit variablen Partitionen nur von ihrem Speicherplatzbedarf abhängt und somit ebenfalls variabel ist, bezeichnet man ein solches Verfahren als MVT (*multiprogramming with a variable number of tasks*)
Ein System mit realem Speicherkonzept muß im Zeitscheibenbetrieb Jobs auf den Sekundärspeicher auslagern, wenn die gleichzeitig zu bearbeitenden Jobs nicht in den Hauptspeicher passen. Da ein ständiges Ein- und Auslagern der Jobs für jeweils eine Zeitscheibe zu nicht vertretbaren Leistungseinbußen führen würde, werden die ausgelagerten Jobs meist nicht vor Ablauf einer bestimmten Zeit (*on disk trash lock*) wieder eingelagert und eingelagerte Jobs nicht vor Ablauf einer Residenz-Zeitspanne (*in memory trash lock*) ausgelagert.

Die Speicherausnutzung eines Systems mit virtuellem Speicherkonzept ist im allgemeinen erheblich besser als die eines Systems mit Realspeicher. In einem Computersystem mit virtuellem Speicherkonzept tritt ein ungenutzter Speicherplatz nur innerhalb der letzten Seite eines Speicherbereichs auf, so daß die Speichernutzung kein Problem darstellt. Die Durchsatzleistung auf einem Computersystem mit virtuellem Speicher hängt dagegen stark von der Seitenfehlerrate und in geringerem Maß von dem *overhead* der Speicherverwaltung ab.
Die Seitenfehlerrate wird direkt von der Seitenersetzungsstrategie bestimmt, wobei durch die Ersetzung einer aktuell benötigten Seite ein großer *overhead* entsteht, der möglichst vermieden werden muß. Da die in Cache-Speichern verwendete und angestrebte Seitenersetzungsstrategie LRU nur mit großem Aufwand realisierbar ist, werden in kommerziellen Supercomputern Strategien verwendet, die eine Approximation der LRU-Strategie darstellen. Diese Verfahren basieren zumeist auf der Strategie *not recently used* (NRU), die eine der innerhalb eines bestimmten Zeitraums nicht genutzten Seiten ersetzt. Es werden auch Varianten dieses Verfahrens mit den Bezeichnungen *aging* und *second chance* eingesetzt, die in [SPG91, Tan90] beschrieben werden und eine bessere Approximation der LRU-Strategie darstellen. Andere Varianten generieren mit Hilfe der NRU Strategie in periodischen Abständen einen Pool von zu ersetzenden Seitenrahmen, so daß die Routine zur Seitenersetzung nicht bei jedem Seitenfehler aktiviert werden muß. Ebenso wie beim Cache-Speicher-Entwurf kann bei einem Seitenfehler ein *prefetch* erfolgen. Es wird dann nicht nur die beim Seitenfehler referenzierte Seite, sondern auch eine bestimmte Anzahl der logisch folgenden Seiten miteingelesen. Alternativ können Gruppen von aufeinanderfolgenden Seiten gebildet werden, die nur dann gemeinsam eingelesen werden, wenn sie auch auf dem Sekundärspeicher zusammenhängend sind und deshalb wenig Ein-/Ausgabeaufwand erzeugen (*clustering*). Die modifizierten Seiten müssen zuerst auf den Sekundärspeicher zurückgeschrieben werden, bevor sie ersetzt werden können. Um zu vermeiden, daß bei einem Seitenfehler erst das Auslagern einer Seite nötig wird, bevor eine neue gelesen wird, kann die Speicherverwaltung in Pha-

sen geringer Seitenersetzungsaktivität modifizierte Seiten auf den Sekundärspeicher kopieren.

Da die Jobs im Hauptspeicher um die Seiten des Computersystems konkurrieren, wird die Seitenfehlerrate zusätzlich von der Anzahl der im Hauptspeicher vorhandenen Jobs beeinflußt. Die Strategie, die entscheidet, wie viele Jobs den Hauptspeicher nutzen dürfen und somit gleichzeitig bearbeitet werden, nennt man *job scheduling*. Diese Strategie soll eine möglichst gute Nutzung des virtuellen Speichers erzielen und dabei das Entstehen von *thrashing* oder *page flattering* vermeiden. Zu *thrashing* kommt es, wenn der im Ablauf der Jobs genutzte Speicherplatz größer ist als der reale Hauptspeicher des Computersystems. Durch die Bereitstellung eines neuen Seitenrahmens für einen Job wird dabei eine noch benötigte Seite desselben oder eines anderen Jobs aus dem Hauptspeicher verdrängt. Da die verdrängte Seite kurz darauf wieder angesprochen wird, muß sie wieder eingeladen werden, wodurch sie eine weitere von einem Job benötigte Seite verdrängt. Dadurch werden die Jobs im Computersystem ständig durch Seitenfehler blockiert und die Durchsatzleistung fällt gravierend ab. Entdeckt die Speicherverwaltung ein *thrashing* des Systems, so kann durch das Auslagern von Jobs wieder ein effizientes Arbeiten und damit eine hohe Durchsatzleistung erreicht werden. Es gibt verschiedene Strategien, durch die ein *thrashing* des virtuellen Speichers entdeckt werden soll. Diese Strategien verfolgen dazu entweder die Seitenfehlerrate oder sie überprüfen die Anzahl der als benutzt markierten Seiten. Wird eine zu hohe Seitenfehlerrate registriert oder wurden mit der NRU-Strategie zuwenig freie Seiten entdeckt, so werden solange Jobs ausgelagert bis eine bestimmte Anzahl von Seiten frei sind.

2.2.2 Scheduling

Im Mehrprogrammbetrieb wird die zur Verfügung stehende Prozessorzeit durch eine Scheduling-Strategie auf die einzelnen zur Ausführung bereitstehenden Programme verteilt. Zum Scheduling von sequentiellen Programmen sind für Uniprozessorsysteme verschiedene Strategien wie First-Come-First-Served (FCFS) und Shortest-Job-First (SJF) bekannt. Für den Zeitscheibenbetrieb werden verschiedene Round-Robin-Strategien (RR) mit einer oder mehreren Warteschlangen und verschiedenen Prioritäten verwendet. Diese Verfahren werden ausführlich in der einführenden Literatur über Betriebssysteme beschrieben [SPG91, Tan90] und bilden die Basis für Scheduling-Strategien auf Multiprozessorsystemen. Beim Scheduling auf Multiprozessorsystemen kann eine optimale Nutzung der Prozessoren nur dann erreicht werden, wenn die auszuführenden Prozesse dynamisch auf alle Prozessoren verteilt werden, so daß ein Leerlaufen von Prozessoren nicht möglich ist, solange noch ausführbereite Prozesse vorhanden sind. Daher sollten selbst im Mehrprogrammbetrieb von sequentiellen Programmen die zur Verfügung stehenden Prozessoren nicht wie unabhängige Computersysteme betrachtet werden.

Das Scheduling von parallelen Programmen auf Multiprozessorsystemen kann prin-

zipiell auf das Scheduling von sequentiellen Programmen zurückgeführt werden, da die kooperierenden Prozesse eines parallelen Programms von der Scheduling-Strategie genauso wie die unabhängigen Prozesse von verschiedenen sequentiellen Programmen behandelt werden können. Diese Scheduling-Strategien, die nicht zwischen kooperierenden und unabhängigen Prozessen unterscheiden, wurden in den Betriebssystemen der ersten parallelen Vektorsupercomputer von z.B. CRAY und Alliant verwendet [Pol89]. Diese Scheduling-Strategien verhalten sich aber einerseits unfair, da sie parallelen Programmen unverhältnismäßig viel Prozessorzeit zuteilen, und sie erzeugen andererseits unnötigen Aufwand, da sie die besonderen Anforderungen von parallelen Programmen nicht berücksichtigen.
So müssen die Ressourcen von unabhängigen sequentiellen Programmen voreinander geschützt werden, während die Ressourcen von kooperierenden Prozessen gemeinsam genutzt werden. Daher wird bei neueren Konzepten (z.B. beim Mach*kernel*), zwischen der *task* als Gesamtheit der Ressourcen (virtueller Speicher, Zugriffsrechte) und dem *thread* als Befehlsstrom mit eigenem Ausführungszustand (Befehlszeiger, *stack pointer*) unterschieden [ABB+86, TR86]. Auf einem Computersystem mit gemeinsamem Speicher kann die Parallelverarbeitung dann erfolgen, indem mehrere *threads* parallel innerhalb einer *task* ablaufen. Die kooperierenden Prozesse eines parallelen Programms werden dann durch *threads* realisiert, die auf einer gemeinsamen *task* arbeiten, so daß der Aufwand, der durch die Ressourcenverwaltung bei der Erzeugung einer *task* anfällt, bei der Generierung von weiteren kooperierenden Prozessen vermieden werden kann.
Der für das Scheduling von *threads* benötigte Aufwand kann durch eine Aufgabenteilung zwischen Betriebssystem und Applikation reduziert werden. Das Betriebssystem verteilt dazu die Prozessoren an die Applikationen, die für die Verteilung und Bearbeitung der *threads* durch die Prozessoren verantwortlich sind. Dieses Scheduling der Prozessoren auf die *threads* kann durch einen *library scheduler* (CRAY [Nag90c]) durchgeführt werden, der alternativ auch *server* (Mach [Bla90]) oder *task generator* (BBN [TC88]) genannt wird. Durch die Integration des *library schedulers* in die Applikation und die gemeinsame Nutzung der Ressourcen wird bei der Generierung eines *threads* kein aufwendiger Systemaufruf erforderlich. Dadurch wird der Aufwand bei der Erzeugung von *threads* reduziert, so daß sich auch Programme mit feiner Granularität effizient parallelisieren lassen. Die Verwendung eines *library schedulers* erlaubt zusätzlich die Nutzung unterschiedlicher Scheduler für verschiedene Programme oder Programmabschnitte. Das *uniform system* von BBN stellt deshalb mehrere solcher *library scheduler* bereit, die auf die Parallelisierung von Schleifen, Rekursion etc. spezialisiert sind und innerhalb eines Programms genutzt werden können [TC88].

Da die Prozessoren eines Supercomputers bei der Parallelverarbeitung zur Beschleunigung eines einzelnen Programms einsetzbar sein sollen, wurde die Leistung des Computersystems zunächst für den dedizierten Betrieb optimiert. Nach *Amdahl's Law* wird die durch Parallelverarbeitung maximal mögliche Leistungssteigerung aber durch den sequentiellen Anteil des Programms beschränkt. Bei jedem Übergang zwischen parallelen und sequentiellen Bereichen müssen die kooperierenden Pro-

zesse synchronisiert werden. Diese Synchronisation verringert den parallelen Anteil des Programms, da bei der Synchronisation keine produktive parallele Arbeit verrichtet wird. Die Synchronisation erfolgt auf Computersystemen mit gemeinsamem Speicher entweder explizit in Form von kritischen Sektionen, Vorgängerrelationen mittels *barrier*-Anweisungen, sequentiellen und parallelen Regionen oder implizit mittels *message passing*. Die explizite Synchronisation kann auf Vektorrechnern effizienter durchgeführt werden und bildet die Grundlage für die automatische Parallelisierung von Programmen.
Die explizite Synchronisation über den gemeinsamen Speicher bzw. gemeinsame Register kann mit geringem Aufwand durch die Verwendung von Semaphor-Variablen erreicht werden. Dabei wird eine kurze Schleife ausgeführt, in der solange eine *read-modify-write*-Operation auf eine Semaphor-Variable gestartet wird, bis diese erfolgreich ausgeführt werden kann. Diese Schleife kann auch durch einen einzigen Befehl durchgeführt werden, wie dies beim Test-and-Set-Befehl auf der CRAY X-MP/Y-MP Familie der Fall ist. Nach erfolgter Synchronisation stehen die wartenden Prozessoren direkt für weitere parallele Berechnungen zur Verfügung. Dadurch entstehen bei der Aktivierung der Prozessoren nur kurze Verzögerungen, so daß mit Hilfe von Semaphoren insbesondere bei feingranularen Programmen minimale Ausführungszeiten erreicht werden. Die nicht benötigten bzw. wartenden Prozessoren führen dabei allerdings unproduktive Warteschleifen aus (*spinwaiting* oder *busy waiting*). Im dedizierten Betrieb können diese unproduktiven Warteschleifen für Prozessoren, die nicht zur Bearbeitung der Applikation beitragen können, durchaus hingenommen werden. Vor der Version 6.0 verwendete das Betriebssystem UNICOS von CRAY bei den feingranularen Parallelisierungskonzepten (Microtasking und Autotasking) zur Synchronisation ausschließlich Semaphoren [Nag90c]. Da die Synchronisation einer großen Anzahl von Prozessoren über den gemeinsamen Hauptspeicher zu Leistungsverlusten durch Zugriffskonflikte auf eine Speicherstelle führen können (*hot spots*), werden auch modifizierte Verfahren verwendet, die eine geringere Speicherbelastung erzeugen [GT90]. Eine Synchronisierung an kritischen Sektionen mit Hilfe von Semaphoren nennt man blockierend, da die Prozessoren, die nicht in die kritische Sektion dürfen, in einer Schleife warten müssen. In [Her90] wird ein Konzept vorgestellt, daß im Sonderfall von kritischen Sektionen das *spin waiting* mit Hilfe der Compare-und-Swap Anweisung vermeiden kann.
Die unproduktive *spin-waiting*-Zeit steigt an, wenn die Arbeit ungleichmäßig auf die Prozessoren verteilt wird und einige Prozessoren erheblich früher als andere die Synchronisationspunkte erreichen. Die Aufteilung der Arbeit (*Partitionierung*) auf die einzelnen *threads* läßt sich schon bei der Übersetzung durch den Compiler durchführen (*statisches Scheduling*). Ein solches statisches Scheduling kann aber oft keine optimale Leistung erzielen, da eine statische Analyse eines Programms in vielen Fällen nicht genügend Information bereitstellt, um eine gute Aufteilung der Arbeit (*load balance*) zu ermöglichen. Eine günstigere *load balance* wird oft durch Strategien erreicht, die die Partitionierung erst zur Laufzeit mit einem *library scheduler* durchführen (*dynamisches Scheduling*). Zur Partitionierung von parallelen Schleifen stehen verschiedene Strategien zur Verfügung (*self scheduling, guided self scheduling, chunk scheduling*). Eine Unterstützung des dynamischen Schedulings

durch vom Compiler generierte Information (*prescheduling*) kann die Effizienz noch weiter erhöhen [Pol88a, Pol88b].

Bei Blockierung	Probleme	Lösungsansätze
spin waiting	Ungünstige *load balance*	*self scheduling*
		guided self scheduling
		prescheduling
	Ungünstige Suspendierung von kritischen Prozessen	*coscheduling*
		hints / give up flag
task-Wechsel	*overhead*	Hardware-Scheduler: ASAP
		slave hold time
	cache corruption	*dynamic partitioning*

Tabelle 2.2: Probleme und Lösungsansätze beim Scheduling

Bei der Auswahl von Scheduling-Strategien für den Mehrprogrammbetrieb ist im Gegensatz zum dedizierten Betrieb nicht nur eine niedrige Ausführungszeit für ein bestimmtes paralleles Programm, sondern gleichzeitig eine hohe Auslastung des Computersystems und damit eine hohe Durchsatzleistung von Interesse. Für die bei einer bestimmten Scheduling-Strategie erreichte Durchsatzleistung sind zum einen der Verwaltungsaufwand der Scheduling-Strategie sowie der für einen *task*-Wechsel benötigte Aufwand verantwortlich. Zum anderen wird die Durchsatzleistung durch Wartezeiten verringert, die durch eine Blockierung von Prozessen in Folge von fehlenden Ressourcen oder Synchronisationsanforderungen von kooperierenden Prozessen entstehen können.
Das *spin waiting* bei der Synchronisation führt im Mehrprogrammbetrieb unter Umständen zu einer schlechten Nutzung der Prozessoren, da die Prozessoren alternativ zum *spin waiting* auch andere Programme bearbeiten könnten. Besonders im Zeitscheibenbetrieb können durch das *spin waiting* unproduktive Wartezeiten entstehen, wenn einem Prozeß der Prozessor entzogen wird, mit dem andere Prozesse synchronisiert werden sollen und deshalb mittels *spin waiting* warten. Die Prozessoren der zu synchronisierenden Prozesse führen dann solange keine produktive Arbeit aus, bis der Prozeß, auf den die anderen Prozessoren warten, wieder aktiviert wird und den Synchronisationspunkt erreicht.

Diese Wartezeiten werden durch das *coscheduling*, bei dem alle Prozesse eines parallelen Programms in einer Zeitscheibe gleichzeitig zur Ausführung gebracht werden, vermieden. Für das Cm*-Projekt wurden von Ousterhout drei Verfahren untersucht, die einen hohen Grad von *coscheduling* erreichen [GSS87]. Das *coscheduling* aller kooperierenden Prozesse ist aber nur möglich, wenn keiner der Prozesse aufgrund fehlender Ressourcen blockiert ist. In einem System mit virtuellem Speicher kann eine solche Blockade z.B. durch einen Seitenfehler verursacht werden. Bei einer konsequent auf die Maximierung des *coscheduling* ausgerichteten Scheduling-Strategie

müssen bei der Blockade eines Prozesses alle kooperierenden Prozesse des parallelen Programms suspendiert werden. Eine solche synchrone Scheduling-Strategie wird auf der Alliant FX-8 unter Concentrix verfolgt [Bla90]. Ein Problem besteht beim synchronen *coscheduling* darin, daß durch das gleichzeitige Starten aller Prozesse eines Jobs ein vermehrtes Auftreten von Zugriffskonflikten provoziert wird. Ein geringerer Grad an *coscheduling* wird bei asynchronen Scheduling-Strategien erreicht, da die Zuteilung der Prozessoren auf die Prozesse asynchron erfolgt und somit ein gleichzeitiges Starten aller kooperierenden Prozesse eines Programms nicht unterstützt wird.

Alternativ zum *coscheduling* kann unnötiges *spin waiting* dadurch verhindert werden, daß die Scheduling-Strategie mit Informationen ausgestattet wird, durch die es möglich wird Prozesse nicht an ungünstigen Stellen zu suspendieren oder zu aktivieren. Befindet sich ein zu suspendierender Prozeß gerade in einer kritischen Sektion, so kann stattdessen ein anderer Prozeß des Programms, der möglicherweise wegen *spin waiting* unproduktiv ist, suspendiert werden. Ebenso kann die Aktivierung von Prozessen, die aufgrund von *spin waiting* blockiert sind, zugunsten von anderen Prozesse ausgesetzt werden. Im Mach-Kernel kann der Benutzer deshalb das Betriebssystem beim Scheduling durch sogenannte *hints* unterstützen, durch die er das Betriebssystem darüber informieren kann, welcher Prozeß zur Suspendierung geeignet ist und welcher nicht [Bla90]. Eine andere Variante dieses Prinzips wird im *cooperative parallel interface* (CPI) der CRAY UNICOS Version 6.0 verwendet. Dabei wird einem kooperierenden Prozeß nach Ablauf einer Zeitscheibe eine einmalige Verlängerung gewährt, in der der Prozeß durch das gesetzte *give up flag* dazu aufgefordert wird, den Prozessor an der nächsten für ihn günstigen Stelle freiwillig abzugeben. Gibt der Prozeß den Prozessor innerhalb dieses Zeitraums nicht freiwillig ab, so wird ihm dieser nach Ablauf der zugeteilten Nachlaufzeit entzogen [BLR91]. Wird innerhalb des Programms in hinreichend kurzen Abständen das *give up flag* abgefragt, so kann der Prozessor immer an geeigneten Stellen abgegeben und unnötiges *spin waiting* vermieden werden.

Eine weitere unnötige Reduzierung der Durchsatzleistung entsteht, wenn aufgrund einer ungünstigen *load balance* oder einem zu geringen Parallelitätsgrad des Programms nicht alle dem Programm zugeteilten Prozessoren genutzt werden können und die nicht benötigten Prozessoren ein *spin waiting* durchführen. Um dies zu vermeiden, muß eine Scheduling-Strategie einem Programm die von ihm nicht genutzten Prozessoren entziehen und für andere Aufgaben freisetzen. So werden vom CRAY CPI Prozessoren, die länger als eine bestimmte Zeit (*slave hold time*) mit *spin waiting* beschäftigt sind, wieder an das Betriebssystem zurückgegeben.
Eine Steigerung der Durchsatzleistung durch den Entzug von nicht genutzten Prozessoren ist jedoch nur möglich, wenn die Hardware des Computersystems eine dynamische Verteilung der Prozessoren auf die parallelen Prozesse unterstützt. Dies ist nicht möglich, wenn die Prozessoren eines Computersystems in Gruppen (*cluster*) eingeteilt sind und einem Job nur als ganze Gruppe zugeordnet werden können. Dies ist zum Beispiel bei den Rechnern der Firma Alliant der Fall. Während alle

Prozessoren einer FX/8 bzw. FX/80 ein *cluster* bilden können die Prozessoren einer FX/2800 bis auf geringe Einschränkungen frei auf die *cluster* verteilt werden [All91]. Die *cluster* der FX/2800 können dann, ebenso wie eine vollständige FX/8 entweder jeweils einem parallelem Programm zugeordnet werden oder in die einzelnen Prozessoren zerfallen und sequentielle Programme bearbeiten. Die Zuordnung der Prozessoren zu den *clustern* erfolgt durch ein Betriebssystemkommando und ist statisch, d.h. der Scheduler paßt sie nicht im Betrieb an den Prozessorbedarf der Programme an.
Die Computersysteme, die eine dynamische Zuteilung aller Prozessoren auf alle Programme zulassen, müssen einen unter Umständen aufwendigen *task*-Wechsel durchführen, um eine Nutzung leerlaufender Prozessoren für andere Jobs zu ermöglichen. Bei einem *task*-Wechsel muß die Arbeitsumgebung (*task*) des deaktivierten Prozesses gesichert und die *task* des zu aktivierenden Prozesses für den Prozessor verfügbar gemacht werden. Dieser Aufwand (*overhead*) verringert die für die Bearbeitung der Applikationen zur Verfügung stehende Rechenzeit. Weitere Leistungseinbußen können beim *task*-Wechsel dadurch entstehen, daß die zur Nutzung der Datenlokalität benutzten Speicher, insbesondere Cache-Speicher, neu gefüllt werden müssen. Dies betrifft insbesondere auch die *task*-Wechsel, die durch die Nutzung der Prozessoren im konventionellen Mehrprogrammbetrieb sowie im Zeitscheibenbetrieb verursacht werden. Die im Zeitscheibenbetrieb durch *task*-Wechsel verursachten Leistungseinbußen können direkt über die Größe der Zeitscheibe beeinflußt werden.

Sollen Prozessoren, die aufgrund von Synchronisation oder ungünstiger *load balance* nicht für die Bearbeitung eines Programms genutzt werden können, für die Bearbeitung eines anderen Programms eingesetzt werden, so entsteht durch den *task*-Wechsel ein zusätzlicher Aufwand. Eine Steigerung der Durchsatzleistung läßt sich nur dann erreichen, wenn dieser Aufwand geringer ist als die Zeit, die der Prozessor unproduktiv mit *spin waiting* beschäftigt ist. Die Zeit, die ein Prozessor auf den Zugriff auf eine durch eine kritische Sektion geschützte Variable warten muß, wird im allgemeinen zu gering sein, um den aufwendigen *task*-Wechsel eines Vektorprozessors zu rechtfertigen. Im Gegensatz dazu können sequentielle Bereiche genug Prozessorzeit in Anspruch nehmen, um durch einen *task*-Wechsel der wartenden Prozessoren eine Steigerung der Durchsatzleistung zu erreichen. Da das Betriebssystem keine Information darüber hat, ob ein *task*-Wechsel sinnvoll ist, kann eine Unterstützung durch den Benutzer mittels Compiler-Direktiven zu einer effizienteren Arbeitsweise führen.
Weniger problematisch wird die Durchführung eines *task*-Wechsels, wenn der von ihm induzierte Aufwand nur geringfügig über dem einer Synchronisation mittels Semaphoren liegt. Eine hinreichende Reduktion dieses Aufwandes kann eventuell durch eine Hardware-Unterstützung erreicht werden. Neben einer Hardware-Unterstützung durch mehrere Registersätze läßt sich eine Reduzierung des Aufwands dadurch erreichen, daß die Scheduling-Strategie durch die Hardware des Computersystems (z.B. durch *microcode*) implementiert wird. Ein solches Verfahren (ASAP = *automatic self allocating processors* genannt) wurde von Convex zum erstenmal in

der C-2 Serie eingesetzt [vdPvK91] und ist auch in der C-3 Serie implementiert. Die leerlaufenden Prozessoren einer Convex C-2/C-3 durchsuchen die Kommunikationsregister ständig nach ausführbereiten *threads*. Diese werden von einem Prozessor durch eine *fork*- oder eine *spawn*-Anweisung generiert, die zusätzlich die zum Starten der *threads* benötigten Informationen in einen gemeinsamen Registersatz kopiert. Werden keine Prozessoren mehr benötigt, da sich alle *threads* des Programms in Ausführung befinden, so wird eine *join*-Anweisung ausgeführt, durch die die Anforderung von weiteren Prozessoren unterbunden wird. Der letzte der aktivierten Prozessoren, der die *join*-Anweisung ausführt, wird im Gegensatz zu den anderen Prozessoren nicht wieder freigegeben, sondern setzt die Ausführung des Programms fort.

Die Cache-Speicher eines Computersystems werden unabhängig von einer Hardware-Unterstützung der Scheduling-Strategie bei einem *task*-Wechsel in ihrer Wirkung beeinträchtigt, was durch den Begriff der *cache corruption* beschrieben wird. Bei einem *task*-Wechsel überschreibt der neu aktivierte Prozeß A die Daten, die durch den gerade deaktivierten Prozeß B im Cache-Speicher zwischengespeichert wurden, so daß bei einer erneuten Aktivierung des Prozesses A nur noch ein geringer Teil der von ihm benötigten Daten vorhanden ist. Wird der Prozeß B auf einem anderen Prozessor mit einem anderen Cache-Speicher wieder aktiviert (*task migration*), so ist der Inhalt dieses Cache-Speichers für ihn unbrauchbar. Auf Computersystemen mit Cache-Speichern tritt dieses Phänomen insbesondere bei Scheduling-Strategien auf, die das *coscheduling* maximieren, da die einzelnen parallelen *tasks* eines Jobs alle gleichzeitig zur Ausführung kommen und bei einem *task*-Wechsel auf einer großen Anzahl der Prozessoren ein anderer Prozeß zur Ausführung kommt.

Die cache corruption kann durch eine dynamische, lastabhängige Aufteilung der Prozessoren (*dynamic partitioning*) auf die einzelnen parallelen Programme reduziert werden [LV90, TG89]. Im Gegensatz zum *coscheduling*, bei dem einem parallelen Programm für eine kurze Zeitscheibe alle angeforderten Prozessoren zur Verfügung gestellt werden, stellt eine solche Strategie einem parallelen Programm nur wenige Prozessoren für eine längere Zeit zur Verfügung. Werden nur so viele Jobs vom Computersystem bearbeitet wie Prozessoren zur Verfügung stehen, dann werden durch diese Strategie sowohl *task*-Wechsel als auch die cache corruption unterbunden. Die *dynamic-partitioning*-Strategien geben also bei einer hohen Belastung des Computersystems dem Mehrprogrammbetrieb den Vorrang vor der Parallelverarbeitung, erlauben es aber, leerlaufende Prozessoren durch Parallelverarbeitung zu nutzen. Voraussetzung für die Verwendung einer solchen Strategie ist aber, daß die parallele Programme sowohl von einer variablen Anzahl an Prozessoren als auch sequentiell effizient ausführbar sind. Bei der Verwendung von gängigen feingranularen Programmierkonzepten ist dies der Fall, wenn sich die Last gut auf die zur Verfügung stehenden Prozessoren verteilen läßt, so daß eine gute *load balance* erreicht werden kann.

Im Gegensatz zum Scheduling von parallelen Programmen erfordert die Speicher-

verwaltung auf Multiprozessorsystemen mit gemeinsamen Speicher keine neuen Verfahren. Die Speicherverwaltung muß jedoch wegen ihres großen Einflusses auf die Leistung bei der Nutzung von parallelen Programmen berücksichtigt werden. Die Wechselwirkungen zwischen parallelen Programmen und der Speicherverwaltung werden im Abschnitt 2.4 vertieft. Das Scheduling paralleler Programme ist ein aktives Forschungsgebiet, das Bedeutung für die Nutzung von Multiprozessorsystemen besitzt. Die Unterstützung des Schedulers durch den Compiler ist ein Beispiel für die vielfältigen Wechselwirkungen, die bei einer Leistungsbewertung berücksichtigt werden müssen.

2.3 Compiler

Die Programmierung von technisch-wissenschaftlichen Anwendungen für Supercomputer erfolgt zum weitaus größten Teil in der Programmiersprache FORTRAN. Dies hat zum einen historische Gründe und ist zum anderen auf die besondere Eignung von FORTRAN für diese Probleme zurückzuführen: FORTRAN-Programme lassen sich sowohl auf konventionellen Uniprozessor-Systemen als auch auf Multiprozessor-Vektorrechner in effizient ausführbaren Maschinen-Code übersetzen. Ein FORTRAN-Programm kann jedoch nur dann mit hoher Leistung auf einem Supercomputer ausgeführt werden, wenn es an die speziellen Eigenschaften des Systems angepaßt wurde. Die Anpassung von bestehenden FORTRAN-Programmen an einen speziellen Supercomputer sollte, soweit möglich, automatisch durch den FORTRAN-Compiler erfolgen.

Die Leistungsfähigkeit des FORTRAN-Compilers bei der Optimierung und Code-Erzeugung bestimmt, welche Rechenleistung mit einem bestehenden Programm erreicht werden kann. Unter der Optimierung eines Programms versteht man eine Transformation des Programms, von der erwartet wird, daß sie zu einer kürzeren Laufzeit führt. Da der Compiler bei der statischen Analyse des Programms nicht immer entscheiden kann, welcher Code eine höhere Leistung erreicht, kann die Laufzeit eines Programms in ungünstigen Fällen durch Optimierungen des Compilers reduziert werden. Eine optimierende Transformation darf generell nur dann durchgeführt werden, wenn sichergestellt werden kann, daß die Semantik des Programms erhalten bleibt. Wird durch die Optimierung die Ausführungsreihenfolge der arithmetischen Operationen verändert, so kann das optimierte Programm andere numerische Eigenschaften besitzen. Dies ist zwar nicht erwünscht, muß aber häufig akzeptiert werden, wenn das Programm eine höhere Rechenleistung erreichen soll. Optimiert werden kann das Quellprogramm, ein möglicher Zwischen-Code sowie der generierte Maschinen-Code.

Da Supercomputer nur bei der Bearbeitung von Programmen mit einer geeigneten Struktur eine hohe Leistung erreichen, ist die Qualität des FORTRAN-Compilers oft entscheidend für den praktischen Nutzen eines Supercomputers. Besonders

wichtig für die Leistung einer FORTRAN-Anwendung auf einem Vektorrechner ist der erzielte Vektorisierungsgrad des übersetzten Programms. Sollen bestehende FORTRAN-Programme unmittelbar von der Leistung eines Multiprozessor-Vektorrechners profitieren, so muß der Compiler eine automatische Vektorisierung und gleichzeitig eine automatische Parallelisierung durchführen. Diese führt jedoch nicht immer zu der gewünschten Leistungssteigerung. In diesen Fällen kann eine Unterstützung des Compilers durch den Programmierer in Form von Optionen, Annahmen (*Assumptions*) und Direktiven bessere Ergebnisse zur Folge haben. Auf Multiprozessorsystemen mit lokalen Speichern wie z.B. Cache-Speichern hängt die Leistung oft entscheidend von der Nutzung dieser lokalen Speicher ab, so daß die Speicherverwaltung durch den Compiler für den effizienten Einsatz von parallelen Supercomputern an Bedeutung gewinnt.

2.3.1 Skalare Optimierung

Für die Optimierung von skalarem Programm-Code steht eine fundierte Theorie zur Verfügung, die im Regelfall erlaubt, hinreichend effizienten Code zu generieren. Die skalare Optimierung ist auf der Ebene eines maschinenunabhängigen Zwischen-Codes und auf der Ebene der generierten Maschineninstruktionen am wirkungsvollsten. Eine skalare Optimierung auf der Ebene der üblichen, höheren Programmiersprachen (PASCAL, FORTRAN 77) ist nicht hinreichend effektiv, da sich Optimierungen bezüglich der Registernutzung oder Adreßberechnungen nicht in diesen Programmiersprachen darstellen lassen. Eine genaue Beschreibung der im folgenden beschriebenen Optimierungen und der dafür verwendeten Algorithmen findet man in der einführenden Literatur zum Compilerbau [ASU88].

Die zur skalaren Optimierung verwendeten Techniken unterteilt man in algebraische und strukturerhaltende Transformationen. Eine algebraische Transformation ist eine Ersetzung eines algebraischen Ausdrucks durch einen äquivalenten Ausdruck unter Verwendung von algebraischen Gesetzen, wie z.B. $x = x + 0$, $x = x * 1$ oder $x + y = y + x$. Unter einer strukturerhaltenden Transformation versteht man alle übrigen Transformationen des Programm-Codes, die nicht zur Veränderung der Programmsemantik führen. Zu den strukturerhaltenden Transformationen zählen die Umbenennung von Variablen, die Vertauschung der Reihenfolge von Anweisungen und die Einführung von Hilfsvariablen. Die Optimierung wird unabhängig von der zur Transformation verwendeten Technik in maschinenabhängige und maschinenunabhängige Optimierung unterschieden.

Bei der maschinenabhängigen Optimierung werden die speziellen Eigenschaften der Zielmaschine genutzt, für die der Code generiert werden soll, um die Ausführung des Programms zu beschleunigen. Auf konventionellen skalaren Architekturen sind dabei die Nutzung von speziellen Instruktionen des Prozessors und die Nutzung der Prozessorregister von herausragender Bedeutung. Die Nutzung spezieller Prozessorinstruktionen ermöglicht es, mehrere Operationen, wie z.B. Adreßberechnun-

gen und arithmetische Operationen, gleichzeitig durchzuführen oder spezielle, für bestimmte Aufgaben optimierte Befehle (Kopieren von Speicherbereichen, Schleifenbefehle etc.) einzusetzen. Die Vektorinstruktionen eines Vektorprozessors sowie spezielle Chaining-Instruktionen sind Sonderfälle solcher Mehrfachoperationen, die wegen ihrer besonderen Bedeutung für die Leistung eines Vektorrechners gesondert behandelt werden. Bei Prozessoren mit mehreren gleichzeitig arbeitenden Funktionseinheiten kann eine Veränderung der Reihenfolge der Instruktionen zu einer besseren Nutzung der Funktionseinheiten und somit zu einer höheren Rechenleistung führen.

Die Nutzung der Prozessorregister bei der Code-Generierung ist deshalb so bedeutsam für die Leistung von Prozessoren, da bei den meisten heutigen Prozessoren für einen Speicherzugriff das Mehrfache der Zeit eines Registerzugriffs benötigt wird. Häufig werden bei der Register-Optimierung Heuristiken verwendet, da Algorithmen, die unter bestimmten Annahmen eine optimale Registernutzung ermöglichen, oft in der Klasse der NP-vollständigen Probleme liegen und deshalb einen zu hohen Rechenaufwand benötigen.

Die maschinenunabhängige Optimierung soll ein Programm unabhängig von der gewählten Zielmaschine verbessern. Eine effektive maschinenunabhängige Optimierung wird durch die Berechnung von konstanten Ausdrücken, die Wiederverwendung von berechneten Ausdrücken und die Optimierung von Programmschleifen erzielt. Konstante Ausdrücke können schon während der Übersetzung durch den Compiler berechnet und durch den entsprechenden Wert ersetzt werden. Bei der Wiederverwendung von schon einmal berechneten Ausdrücken wird das Ergebnis eines mehrfach auftretenden Ausdrucks nur einmal ausgewertet und in einer Hilfsvariable zwischengespeichert. Anstelle der erneuten Auswertung des Ausdrucks kann das Ergebnis dann direkt aus der Hilfsvariablen geladen werden.

Die Optimierung von Programmschleifen hat sich als besonders wirkungsvoll erwiesen, da der Programm-Code innerhalb von Schleifen in der Regel sehr häufig ausgeführt wird und deshalb einen großen Einfluß auf die Laufzeit des Programms hat. Da der Programm-Code im Schleifenkörper mehrfach durchlaufen wird, ist es günstig, alle Ausdrücke, die innerhalb der Schleifendurchläufe konstant sind, vor dem Eintritt in die Schleife zu berechnen. Diese Ausdrücke, die für jeden Schleifendurchlauf immer ein identisches Resultat ergeben, nennt man schleifeninvariante Ausdrücke. Die schleifeninvarianten Ausdrücke werden bei der Optimerung textuell vor die Schleifen verschoben (siehe Beispiel 2.1).

Um die Korrektheit und Effizienz der Optimierung zu gewährleisten, muß das Programm vor einer Transformation zunächst bezüglich des Datenflusses und des Kontrollflusses analysiert werden. Bei einer nicht hinreichend genauen Analyse können bestimmte Optimierungen nicht erfolgen, da der Compiler eine Transformation nur dann durchführen darf, wenn die Korrektheit der Transformation aus der vorhandenen Information sichergestellt werden kann. Während sich diese Analyse häufig auf

Beispiel 2.1 Verschieben von schleifeninvarianten Ausdrücken.

```
DO I=1,K-1                          T1 = K - 1
   A = A + 2*B*I          ⟹         T2 = 2 * B
END DO                              DO I=1,T1
                                       A = A + T2*I
                                    END DO
```

die Beziehungen innerhalb der Unterprogramme beschränkt, werden bei der interprozeduralen Analyse auch die Beziehungen zwischen den einzelnen Unterprogrammen berücksichtigt.

2.3.2 Vektorisierung

Die hohe Rechenleistung eines Vektorprozessors führt nur dann zu einer Leistungssteigerung, wenn Vektoroperationen ausgeführt werden. Im Gegensatz zu FORTRAN 77 erlaubt FORTRAN 90 eine direkte Nutzung von Vektorprozessoren, indem die arithmetischen Grundfunktionen auch auf Felder bzw. Vektoren anwendbar sind. Da eine solche Nutzung der Vektorverarbeitung in FORTRAN 77 nicht möglich ist, müssen bestehende FORTRAN 77 Anwendungen auf die Nutzung von Vektorinstruktionen hin optimiert (vektorisiert) werden, wenn sie von der Leistung der Vektorprozessoren profitieren sollen. Eine solche Vektorisierung kann automatisch durch den Compiler erfolgen. Dabei werden einfache DO-Schleifen direkt in die entsprechenden Vektorinstruktionen übersetzt (siehe Beispiel 2.2). Die Schleifen werden aufgrund der praktischeren Notation ebenso wie die Vektorinstruktionen in der Syntax von FORTRAN 90 wiedergegeben [SIG89]. Eine Vektorinstruktion führt eine elementweise Verknüpfung der durch die Notation VECTOR(START:STOP) bzw. VECTOR(START:STOP:STRIDE) ausgewählten Teilfelder durch, wobei START den Index des ersten Elementes des Teilfeldes, STOP den Index des letzten Elementes und STRIDE optional einen möglichen Abstand der Elemente des Teilfeldes innerhalb des Feldes VECTOR beschreibt.

Bei einer Vektoroperation werden mehrere Operanden quasi-gleichzeitig in unterschiedlichen Stufen einer *pipeline* bearbeitet, so daß die Ausführungsreihenfolge der Operationen bei einer Vektoroperation gegenüber der bei der sequentiellen DO-Schleife verändert wird. Wird dadurch auch die Semantik des Programms verändert, so darf die Transformation nicht durchgeführt, die Schleife also nicht vektorisiert werden. In der Schleife in Beispiel 2.3 wird bei jeder Berechnung eines `A(I)` das vorherige Ergebnis `A(I-1)` verwendet, was bei einer gleichzeitigen Ausführung mehrerer Anweisungen der Form `A(I)=A(I-1)+B(I)` nicht der Fall ist. Diese Schleife ist also nicht vektorisierbar. Damit nur Schleifen vektorisiert wer-

Beispiel 2.2 Vektorisierbare Schleife und entsprechende Vektorinstruktion.

```
DO I=START,STOP,STRIDE
   A(I) = B(I) + C(I)
END DO
```

$$\Longrightarrow$$

```
A(START:STOP:STRIDE) = B(START:STOP:STRIDE) + C(START:STOP:STRIDE)
```

den, deren Semantik durch die Transformation nicht verändert wird, müssen die Datenabhängigkeiten zwischen den einzelnen Anweisungen analysiert und berücksichtigt werden. Eine solche Analyse der Datenabhängigkeiten zwischen verschiedenen Feldzugriffen wird von der *dependence*-Analyse nach Kuck [PW86, ZC90, Pol88a] geleistet. Bei dieser Analyse werden sowohl echte Datenabhängigkeiten (*true dependences*) als auch die künstlichen Datenabhängigkeiten (*artificial dependences*, [ZC90]), das sind Antiabhängigkeiten (*anti dependences*) und Ausgabeabhängigkeiten (*output dependences*), berücksichtigt, die durch eine mehrfache Verwendung einer Variablen für unterschiedliche Zwischenergebnisse entstehen. Im Gegensatz zu den echten Datenabhängigkeiten können die künstliche Datenabhängigkeiten durch strukturerhaltende Transformationen beseitigt werden, falls dies nicht durch andere Datenabhängigkeiten verhindert wird.

Beispiel 2.3 Nicht-Vektorisierbare Schleife

```
DO I=2,100
   A(I) = A(I-1) + B(I)
END DO
```

Das Ergebnis einer *dependence*-Analyse ist ein gerichteter *dependence*-Graph, dessen Knoten die einzelnen Anweisungen und dessen Kanten die Abhängigkeiten zwischen den Anweisungen repräsentieren. Führt eine Kante von dem Knoten der Anweisung 1 zu dem Knoten der Anweisung 2, so bedeutet dies, daß Anweisung 1 vor Anweisung 2 ausgeführt werden muß, damit die Semantik des Programms erhalten bleibt. In den Fällen, in denen die *dependence*-Analyse nicht entscheiden kann, ob eine Datenabhängigkeit vorliegt, muß der Compiler annehmen, daß eine Datenabhängigkeit vorliegt, damit ausgeschlossen wird, daß die Semantik des Programms verändert wird. Die Genauigkeit, mit der der *dependence*-Graph die Datenabhängigkeiten des Programms beschreibt, wirkt sich also direkt auf den erreichbaren Vektorisierungsgrad aus. In der Literatur werden zahlreiche Verfahren vorgestellt, durch die eine Verfeinerung des *dependence*-Graphen erreicht werden kann.

Eine weitere Möglichkeit, den durch den Compiler erreichbaren Vektorisierungsgrad zu erhöhen, besteht in der Transformation einer nicht vektorisierbaren Schleife in eine vektorisierbare Schleife. In der Literatur [PW86, ZC90] werden unterschiedliche Transformationen dargestellt, z.B. das Auftrennen einer Schleife in eine vektorisierbare und eine nicht vektorisierbare Schleife (*loop distribution*), die Vertauschung von ineinander verschachtelten Schleifen (*loop interchanging*), das Abtrennen von einzelnen Schleifendurchläufen, die eine Vektorisierung verhindern (*loop peeling*) und die Einführung von temporären Variablen und Feldern, durch die künstliche Datenabhängigkeiten beseitigt werden können.

2.3.3 Parallelisierung

Die Parallelverarbeitung ermöglicht auf Multiprozessorsystemen eine erhebliche Reduzierung der Ausführungszeit der dafür geeigneten Programme. Die meisten existierenden Applikationen sind in Programmiersprachen mit sequentieller Semantik wie FORTRAN 77 geschrieben und können deshalb zunächst nicht parallel ausgeführt werden. Mit Hilfe einer automatischen Parallelisierung durch den Compiler können jedoch auch solche Anwendungen ohne zusätzlichen Programmieraufwand von der Parallelverarbeitung profitieren. Die automatische Parallelisierung erfolgt in der Literatur [ZC90] häufig nach dem SPMD-Modell (*SPMD = single-program multiple-data*), bei dem die Prozessoren dieselben Operationen auf unterschiedlichen Daten ausführen. Dies wird durch eine parallele Ausführung von DO-Schleifen erreicht. Eine parallel ausführbare Schleife wird dabei durch eine DOALL-Anweisung gekennzeichnet (siehe Beispiel 2.4).

Beispiel 2.4 Parallele DOALL-Schleife

```
DO I=1,100                          DOALL I=1,100
   A(I) = B(I) + C(I)       ⟹          A(I) = B(I) + C(I)
END DO                              ENDALL
```

Eine DO-Schleife darf nur dann in eine DOALL-Schleife transformiert werden, wenn die Semantik der Schleife durch die parallele Ausführung nicht verändert wird. Eine DO-Schleife ist genau dann parallelisierbar, wenn keine Datenabhängigkeiten zwischen den verschiedenen Schleifendurchläufen (*loop carried dependences*) bestehen. Mit Hilfe der bei der Vektorisierung erwähnten *dependence*-Analyse kann überprüft werden, ob solche Datenabhängigkeiten bestehen. Die Genauigkeit, mit der der durch die *dependence*-Analyse generierte *dependence*-Graph die Datenabhängigkeiten im Programm beschreibt, beeinflußt somit auch den erreichbaren Parallelisierungsgrad. Eine Erhöhung des Parallelisierungsgrades eines Programm kann analog zur Vektorisierung durch strukturerhaltende Transformationen des Programms

erreicht werden [PW86]. So können Datenabhängigkeiten, die zwischen den Schleifendurchläufen bestehen, in einigen Fällen durch eine Index-Transformation (*loop alignment*) in Datenabhängigkeiten innerhalb des Schleifenkörpers transformiert werden. Eine weitere wichtige Technik stellt die Einführung von lokalen Variablen (*privatisation*) dar, durch die künstliche Datenabhängigkeiten aufgelöst werden können [EHJ+91].

Eine Parallelisierung von DO-Schleifen, in denen Abhängigkeiten zwischen Schleifendurchläufen auftreten, wird durch die DOACROSS-Schleife ermöglicht. Damit die Semantik der sequentiellen DO-Schleife bei der parallelen Ausführung als DOACROSS-Schleife erhalten bleibt, wird die korrekte Reihenfolge der Operationen mit Hilfe von expliziten Synchronisationsanweisungen (*send(signal)* und *wait(signal)*) sichergestellt. Die Generierung der zur Synchronisation verwendeten Signale kann dabei automatisch durch den Compiler erfolgen [ZC90]. Da in einer DOACROSS-Schleife durch die explizite Synchronisation alle Datenabhängigkeiten berücksichtigt werden, kann jede sequentielle DO-Schleife in eine DOACROSS-Schleife transformiert werden. Aufgrund der mittels Signalen erzwungenen Synchronisation kann bei einer DOACROSS-Schleife oft nur eine sehr begrenzte Anzahl von Prozessoren genutzt werden. Werden zwischen zwei Synchronisationspunkten nur sehr wenige Operationen ausgeführt, so läuft eine DOACROSS-Schleife auf eine sequentielle Ausführung der Schleifendurchläufe mit unterschiedlichen Prozessoren hinaus. Eine sequentielle Ausführung von Schleifen kann jedoch effizienter auf einem einzigen Prozessor ausgeführt werden. Jede Synchronisationsanweisung schränkt die gleichzeitige Ausführbarkeit von Anweisungen ein und führt zu zusätzlichem Aufwand bei der Bearbeitung der Schleife. Bei der Optimierung von DOACROSS-Schleifen muß zuerst entschieden werden, ob eine Parallelisierung mit Hilfe von Synchronisation schneller ausgeführt werden kann als das sequentielle Programm. Ist dies der Fall, so kann durch eine Minimierung der Synchronisation die Leistung maximiert werden [ZC90].

Bei der Parallelverarbeitung einer DO-Schleife müssen die Schleifendurchläufe auf die parallelen Prozesse bzw. die Prozessoren verteilt werden (*Partitionierung*). Mit steigender Größe der Partitionen wird der Einfluß des Scheduling-Aufwands auf die Leistung geringer. Eine große Anzahl von Partitionen hingegen erleichtert eine gute *load balance* und erzielt häufig eine gute Nutzung der Prozessoren. Die Verteilung der Schleifendurchläufe auf die parallelen Prozesse kann durch den Compiler erfolgen (*statisches Scheduling*). Da dieses Verfahren in vielen Fällen nicht effizient ist, wird die Partitionierung häufig durch einen Library-Scheduler zur Laufzeit durchgeführt (*dynamisches Scheduling*). Neuere Ansätze versuchen das dynamische Scheduling durch vom Compiler generierte Information zu unterstützen (*prescheduling* [Pol88a, Pol88b]).

Bei der Parallelisierung von mehrfach verschachtelten Schleifen wird die Partitionierung durch die Wahl der zu parallelisierenden Schleife bestimmt. Dabei wird häufig die äußerte parallelisierbare Schleife zur Parallelisierung ausgewählt, da dadurch

große Partitionen mit einem geringem Scheduling-Aufwand entstehen. Eine weitere Technik zur Steigerung der Größe der Partitionen stellt die Zusammenfassung von aufeinander folgenden Schleifen zu einer einzigen Schleife dar (*loop fusion*). Die zur Parallelsierung ausgewählte Schleife besitzt jedoch unter Umständen sehr wenig Schleifendurchläufe, so daß eine Nutzung aller zur Verfügung stehenden Prozessoren nicht möglich ist. Eine andere innere Schleife besitzt jedoch unter Umständen erheblich mehr Schleifendurchläufe und damit mehr Potential für eine mögliche Parallelisierung. Lassen sich die Anzahlen der Schleifendurchläufe nicht zur Zeit der Übersetzung bestimmen, so besteht für den Compiler keine Möglichkeit, die optimal zu parallelisierende Schleife auszuwählen. Durch die Reduktion von ineinander verschachtelten Schleifen zu einer Schleife (*loop collapsing* oder *loop coalescing*), entsteht eine Schleife mit einer stark erhöhten Anzahl von Schleifendurchläufen [Pol88a]. Durch dieses Verfahren wird es möglich, mit nur einer parallelen Schleife den Parallelismus mehrerer parallelisierbarer Schleifen zu nutzen.

2.3.4 Sonstige Optimierungen

Da der Zugriff auf den gemeinsamen Speicher meist erheblich zeitaufwendiger ist als der Zugriff auf einen lokalen Speicher, kann durch die Nutzung lokaler Speicher ein erheblicher Geschwindigkeitszuwachs erzielt werden. Die Optimierung der Nutzung von skalaren Registern ist eine solche Optimierung. Die Nutzung von Vektorregistern kann mit den für die Zuteilung von skalaren Registern entwickelten Strategien erfolgen. Übersteigt die Länge der Vektoren die Kapazität der Vektorregister, so müssen die Vektoren in Abschnitten von der Länge der Vektorregister verarbeitet werden. Dies geschieht durch eine Partitionierung der Schleife (*strip mining* oder *loop sectioning*). Die vektorisierbare Schleife wird dabei in eine neue Schleife transformiert, in der Vektorausschnitte von der Länge der Vektorregister verarbeitet werden (siehe Beispiel 2.5). Wenn in jedem Schleifendurchlauf nur Portionen von der Kapazität eines Vektorregisters verarbeitet werden, können Zwischenergebnisse in den Vektorregister gehalten und Zugriffe auf langsamere Ebenen der Speicherhierarchie vermieden werden. Das *strip mining* erhöht zusätzlich die Lokalität der Speicherzugriffe, so daß nicht nur Vektorregister, sondern auch Cache-Speicher und virtuelle Speichersysteme effektiver genutzt werden können.
Während die Verwaltung von Vektorregistern durch den Compiler erfolgt, arbeiten Cache-Speicher traditionell ohne Compiler-Unterstützung. Der Cache-Speicher eines Uniprozessorsystems arbeitet transparent, d.h. der Cache-Speicher braucht nicht bei der Programmierung berücksichtigt zu werden. In Multiprozessorsystemen, insbesondere bei massiv-parallelen Computersystemen, lassen sich Cache-Speicher oft nicht effizient einsetzen, da die Konsistenz einer großen Anzahl von Cache-Speichern nur mit sehr hohem Hardware-Aufwand sichergestellt werden kann. Daher werden Cache-Speicher in kommerziellen Multiprozessorsystemen häufig nur als Ersatz für nicht vorhandene Vektorregister genutzt. Eine bessere Nutzung kann möglicherweise durch programmierbare Cache-Speicher erzielt werden, deren Konsistenz mit

Beispiel 2.5 *strip mining* für einen Vektorcomputer, dessen Vektorregister M Elemente aufnehmen können.

```
DO I=1,N                                  DO I=1,N,M
   A(I) = B(I) + C(I)                        K = MIN(N,I+M-1)
   D(I) = A(I) + B(I)          ⟹            A(I:K) = B(I:K) + C(I:K)
END DO                                       D(I:K) = A(I:K) + B(I:K)
                                          END DO
```

Hilfe von Instruktionen durch die Software aufrecht erhalten wird. Diese Instruktionen, mit deren Hilfe inkonsistent gewordenen Datenbereiche gelöscht oder für ungültig erklärt werden, sollten direkt vom FORTRAN-Compiler generiert werden. Im Rahmen des CEDAR- und des RP3-Projekts wurden solche Software-Strategien zur Herstellung der Cache-Speicher-Konsistenz untersucht [CKM88, CV90]. In kommerziellen Computersystemen kommen solche Strategien jedoch noch nicht zur Anwendung. Die automatische Nutzung von lokalen Hauptspeichern erstreckt sich hauptsächlich auf die Speicherung lokaler Daten. Dabei können Variablen, die nur für Zwischenwerte genutzt werden, in jedem Prozessor lokal gehalten werden (*privatization*, [EHJ+91]). Untersuchungen zur Nutzung von lokalem Speicher sind Gegenstand der Forschung; Resultate und darauf aufbauende Software-Techniken stehen jedoch zur Zeit nicht in kommerziellen Computersystemen zur Verfügung.

Die vielfältigen Optimierungsstrategien, die in modernen FORTRAN-Compilern für Supercomputer eingesetzt werden, erschweren den Vergleich von Supercomputern im Mehrprogramm-Betrieb, da die Ergebnisse, die mit einer Version der Compilers erhalten werden, schon bei der nächsten Version des Compilers völlig verändert werden können. Insbesondere synthetische Benchmark-Programme stellen ein Problem dar, da diese oft keine sinnvollen Berechnungen durchführen und deshalb vom Compiler wegoptimiert werden können. Eine Vergleichbarkeit der Ergebnisse ist nur dann gewährleistet, wenn die Programme auf den verschiedenen Computersystemen vergleichbar optimiert wurden. Dazu können Compiler-Direktiven genutzt werden, durch die ungewünschte Optimierungen verhindert und gewünschte Optimierungen herbeigeführt werden können.

2.4 Arbeitslast

Die bei Leistungsmessungen paralleler Programme im Mehrprogrammbetrieb gemessenen Ergebnisse werden aufgrund der komplexen Architektur heutiger Multiprozessor-Vektorrechner wesentlich durch die Zusammensetzung der Arbeitslast bestimmt. Deshalb kommt der Auswahl der Arbeitslast eine hohe Bedeutung für die

gemessenen Resultate zu. In der Literatur erfolgen die Leistungsmessungen deshalb häufig mit Programmen, von denen angenommen wird, daß sie typisch für eine reale Arbeitslast sind. Insbesondere werden die modifizierten und leichter portierbaren Applikationen aus Benchmark-Sammlungen wie den PERFECT-Benchmarks [BCK+89] oder den SPEC-Benchmarks [Uni89] zu Arbeitslasten kombiniert. So werden in [DI90] Programme des PERFECT-Benchmarks und in [vdPvK91] Programme des *Synthetic Standard Benchmark* zur Leistungsmessung im Mehrprogrammbetrieb eingesetzt. Bieterman verwendet hingegen synthetische Programme mit bestimmten Eigenschaften (z.B. Matrix-Multiplikation), um den Einfluß dieser Eigenschaften auf den Mehrprogrammbetrieb zu untersuchen [Bie87]. Das im Zentralinstitut für Angewandte Mathematik entwickelte und zur Leistungsmessung verwendete System PAR-Bench ermöglicht die Generierung von synthetischen Arbeitslasten nach vorgegebenen Parametern [Nag90b, NL91a, NL91b, LNVD91]. Mit diesem System können beliebige Arbeitslasten modelliert und die Abhängigkeit der gemessenen Leistung von den Parametern der Arbeitslast untersucht werden.

Bei der Charakterisierung von Programmen werden die für die Leistung entscheidenden Eigenschaften von Programmen mit Hilfe von Parametern beschrieben. Werden diese Parameter für reale Anwendungen bzw. reale Arbeitslasten ermittelt, so können Benchmark-Programme an realen Anwendungen studiert werden. So wird in [Wei90] eine Charakterisierung der synthetischen Benchmarks Dhrystone, Whetstone und Linpack durchgeführt, während in [CH91] die modifizierten Anwendungen des SPEC-Benchmarks charakterisiert werden. In [BCG+91] werden die Meßwerte des *hardware performance monitors* einer CRAY X-MP herangezogen, um typische Arbeitslasten eines Rechenzentrums sowohl mit synthetischen Benchmark-Programmen (Livermore kernels, LINPACK) als auch mit modifizierten Anwendungen (PERFECT, SPECmark) zu vergleichen.

In diesem Kapitel werden die Programmparameter diskutiert, die einen Einfluß auf die Leistung bei der Bearbeitung von parallelen Programmen im Mehrprogrammbetrieb haben. Für die Beschreibung von parallelen Programmen werden üblicherweise Parameter verwendet, deren Auswirkungen auf die Leistung im Einprogrammbetrieb von Bedeutung ist. Diese Parameter werden zunächst in Hinblick auf den Einprogrammbetrieb von parallelen Programmen untersucht. Danach wird der Einfluß dieser Parameter auf den Mehrprogrammbetrieb von parallelen Programmen hin überprüft. Unabhängig von diesen Parametern, die nur parallele Programme betreffen, gibt es bestimmte Eigenschaften von Programmen, die auf Multiprozessorsystemen zu Wechselwirkungen zwischen den Jobs führen und den Durchsatz des Rechners beeinflussen. Diese Eigenschaften werden im letzten Abschnitt dargestellt und daraufhin überprüft, ob sie durch die Parallelisierung des Programms verändert werden.

2.4.1 Parallele Programme im Einprogrammbetrieb

Im Einprogrammbetrieb wird die Parallelverarbeitung dazu genutzt, die Ausführungsdauer von einzelnen Programmen zu reduzieren. Die Ausführungsdauer TIME eines Programms bei sequentieller Ausführung dient als Referenz für die mit Hilfe der Parallelverarbeitung erreichbare Beschleunigung (*speedup*). Die Ausführungsdauer $TIME_P$ des Programms bei paralleler Ausführung ist von der Zahl der Prozessoren P abhängig, die zur Ausführung des parallelen Programms zur Verfügung stehen. Der bei der Parallelverarbeitung erreichte *speedup* S_P ist als Quotient aus der Ausführungsdauer TIME des sequentiellen Programms und der Ausführungsdauer $TIME_P$ des parallelen Programms definiert [Gel88]:

$$S_P := \frac{TIME}{TIME_P} \tag{2.1}$$

Bei dieser Definition des *speedups* als Verhältnis realer Ausführungszeiten kann ein *speedup* gemessen werden, der höher als die Anzahl der genutzten Prozessoren P ist. Ein solcher als *superlinear* bezeichneter *speedup* ist für reale Anwendungen jedoch untypisch und wird nur bei der Bearbeitung von speziellen Daten erzielt. Verantwortlich für einen *superlinearen speedup* kann sowohl der FORTRAN-Compiler als auch die durch die Parallelisierung veränderte Ausführungsreihenfolge des Programms sein, die insbesondere bei Suchalgorithmen eine verkürzte Ausführung des parallelen Programms bewirken kann. Abgesehen von diesen Ausnahmen ist der bei der Parallelisierung erreichte *speedup* S_P immer kleiner oder gleich der Anzahl der Prozessoren P. Die Effizienz E gibt das Verhältnis zwischen dem bei einer Parallelisierung erreichten *speedup* und dem bestmöglichen *speedup* von P an:

$$E := \frac{S_P}{P} \tag{2.2}$$

Die Effizienz ist ein Maß für die Nutzung der Prozessoren durch ein paralleles Programm und abgesehen von den oben diskutierten Ausnahmen eine Zahl kleiner eins. Häufig können jedoch nicht alle Programmteile eines Jobs parallel ausgeführt werden. Die Ausführungszeit TIME eines Programms kann in die parallel ausführbare Zeit $TIME_{par}$ und die sequentielle auszuführende Zeit $TIME_{seq}$ zerlegt werden. Innerhalb einer sequentiell auszuführenden Region kann nur ein Prozessor für die Ausführung des Programms genutzt werden, so daß die restlichen $P - 1$ Prozessoren leerlaufen und nicht zur Ausführung des Programms beitragen. Der Parallelisierungsgrad PG eines Programms gibt den Anteil der Ausführungszeit der parallelisierbaren Programmteile an der gesamten Ausführungszeit des Programms an:

$$PG := \frac{TIME_{par}}{TIME} = \frac{TIME_{par}}{TIME_{seq} + TIME_{par}} \tag{2.3}$$

Der bei der Parallelisierung maximal erreichbare *speedup* wird entscheidend von dem Parallelsierungsgrad des Programms beeinflußt. In *Amdahl's law* (Formel 2.4) wird

der maximale erreichbare *speedup* eines Programms in Abhängigkeit vom Parallelisierungsgrad beschrieben:

$$S_P \leq \frac{1}{(1-PG)+\frac{PG}{P}} \leq \frac{1}{1-PG} \tag{2.4}$$

Die Bedeutung von *Amdahl's Law* für die Parallelverarbeitung wird an einem Beispiel deutlich: Bei einem Parallelisierungsgrad von PG = 0.7 kann selbst mit einer beliebig hohen Anzahl Prozessoren kein *speedup* größer als 3.4 erreicht werden. Die Leerlaufzeit $IDLETIME_{PG}$ der Prozessoren in sequentiellen Regionen wird von der Anzahl der zur Verfügung stehenden Prozessoren P bestimmt: $IDLETIME_{PG} = (P-1)*TIME_{seq}$. Der *speedup* nach Amdahl stellt allerdings nur eine obere Schranke für die Beschleunigung dar, die aber nicht erreicht werden muß. Bei der Berechnung des *speedups* nach Amdahl wird der parallele Anteil der Ausführungszeit durch die Anzahl der Prozessoren P geteilt. Dadurch wird eine gleichmäßige Verteilung der Last auf alle Prozessoren, also eine optimale Lastverteilung (*load balance*), implizit vorausgesetzt. In realen Anwendungen wird die Last jedoch häufig nicht gleichmäßig auf alle Prozessoren verteilt. Ist dies der Fall, so laufen einige der Prozessoren nach Beendigung ihrer Last leer, während andere Prozessoren noch mit der Ausführung ihrer Last beschäftigt sind. Die so entstehende Leerlaufzeit $IDLETIME_{LB}$ der Prozessoren innerhalb von parallelen Regionen trägt nicht zur Beschleunigung des parallelen Programms bei und kann somit nicht genutzt werden. Der *load-balance*-Faktor LB stellt das Verhältnis von der zur Ausführung genutzten und der in der parallelen Regionen verbrauchten Zeit (inklusive der Leerlaufzeit) dar:

$$LB := \frac{TIME_{par}}{IDLETIME_{LB} + TIME_{par}} \tag{2.5}$$

Da bei der hier betrachteten Parallelisierung von FORTRAN-Schleifen mindestens ein Prozessor zur Ausführung des Programms beiträgt, kann die Leerlaufzeit maximal $(P-1) * TIME_{par}$ betragen, so daß der *load-balance*-Faktor nach unten durch $\frac{1}{P}$ begrenzt ist. Der *load-balance*-Faktor entspricht damit der Effizienz innerhalb der parallelen Regionen des Programms. Die Abhängigkeit des maximal erreichbaren *speedups* unter Berücksichtigung des Parallelisierungsgrad und des *load-balance*-Faktors wird in Anlehnung an *Amdahl's law* durch die Formel 2.6 wiedergegeben [Lin90]:

$$S_P \leq \frac{1}{(1-PG)+\frac{PG}{LB*P}} \tag{2.6}$$

Ein paralleles Programm benötigt weiterhin noch zusätzliche Prozessorzeit, um den Programm-Code auszuführen, der zum Starten oder zur Synchronisation der kooperierenden Prozesse benötigt wird. Neben einem konstanten Aufwand entsteht bei jeder Synchronisation am Ende einer parallelen Region ein zusätzlicher Aufwand, der den Bedarf des parallelen Programms an Prozessorzeit erhöht. Dieser Aufwand wächst linear mit der Anzahl der bei der Bearbeitung des Programms ausgeführten

globalen Synchronisationsanweisungen [Sto90]. In dem hier betrachteten Programmiermodell ist die Anzahl der globalen Synchronisationsanweisungen mit der Anzahl der ausgeführten parallelen Regionen identisch.

2.4.2 Parallele Programme im Mehrprogrammbetrieb

Im Einprogrammbetrieb steht das gesamte Computersystem für die Ausführung eines einzigen Programms zur Verfügung. Die Ausführungsdauer TIME eines sequentiellen Programms wird im Einprogrammbetrieb einerseits von der benötigten Prozessorzeit und andererseits durch Wartezeiten bestimmt, die bei der Blockierung des Jobs aufgrund von z.B. Ein-/Ausgabeoperationen entstehen. Im Mehrprogrammbetrieb werden die Ressourcen des Computersystems von allen Jobs einer Arbeitslast gemeinsam genutzt. Eine Arbeitslast besteht aus einer Reihe von Jobs, die nach einem bestimmten Zeitplan, der durch die Startzeitpunkte der Jobs festgelegt ist, zur Ausführung gebracht werden. Einem einzelnen Job wird dabei nur ein bestimmter Anteil der zur Verfügung stehenden Prozessorzeit zugeordnet. Der *service*-Faktor wird als das Verhältnis von verbrauchter Prozessorzeit CPUTIME zur Ausführungszeit des Programms im Mehrprogrammbetrieb $\mathrm{TIME}_{\mathrm{MP}}$ definiert:

$$\mathrm{SF} := \frac{\mathrm{CPUTIME}}{\mathrm{TIME}_{\mathrm{MP}}} \tag{2.7}$$

Der *service*-Faktor eines Jobs ist immer durch die Anzahl der Prozessoren, die für die Bearbeitung des Jobs genutzt werden können, nach oben beschränkt. Ist der *service*-Faktor einer Jobs kleiner als eins, so ist es für den Benutzer nicht entscheidbar, ob das Programm parallel oder sequentiell ausgeführt wurde. Der Durchsatz eines Computersystems bei einer gegeben Arbeitslast A wird durch die Ausführungsdauer der gesamten Arbeitslast $\mathrm{TIME}(A)$ beschrieben. Der maximale Durchsatz des Computersystems kann nur dann erreicht werden, wenn zur Ausführung der Arbeitslast alle Prozessoren genutzt werden können. Die Ausführungsdauer einer Arbeitslast bei optimalem Durchsatz $\mathrm{TIME}_{\mathrm{OPT}}(A)$ entspricht dann der Summe der Prozessorzeiten CPUTIME der Jobs, geteilt durch die Anzahl der Prozessoren P des Computersystems:

$$\mathrm{TIME}(A) \geq \mathrm{TIME}_{\mathrm{OPT}}(A) := \frac{1}{\mathrm{P}} \sum_{job \in A} \mathrm{CPUTIME}(job) \tag{2.8}$$

Im Mehrprogrammbetrieb haben Wartezeiten der Jobs auf Ressourcen wie z.B. die Ein/-Ausgabe keinen Einfluß auf den Durchsatz des Computersystems, solange hinreichend viele andere Jobs zur Ausführung bereitstehen. Ist dies nicht der Fall, so laufen die nicht genutzten Prozessoren leer. Die gesamte von der Arbeitslast benötigte Prozessorzeit ist dann um diese Leerlaufzeit erhöht. In einem Multiprozessorsystem mit P Prozessoren müssen mindestens P sequentielle Jobs zur Ausführung

bereit stehen, damit alle Prozessoren genutzt werden können. Im Gegensatz dazu reicht zumeist ein paralleler Job aus, um alle Prozessoren des Computersystems zu nutzen. Daher kann die Leerlaufzeit einer Arbeitslast durch Parallelisierung der Programme reduziert werden.

Die Anzahl der Jobs, die im Mehrprogrammbetrieb gleichzeitig zur Ausführung bereitstehen, wird durch den Speicherplatzbedarf der einzelnen Jobs im Verhältnis zur Hauptspeicherkapazität des Computersystems bestimmt. Dabei muß berücksichtigt werden, daß ein Teil des Hauptspeichers vom Betriebssystem benötigt wird und nicht von den Programmen der Arbeitslast genutzt werden kann. In einem Computersystem mit Realspeicherkonzept muß die Summe des angeforderten Speicherplatzes der Jobs geringer sein als die freie Kapazität des Hauptspeichers, damit alle Jobs gleichzeitig im Hauptspeicher bereitstehen können. Ist dies nicht der Fall, so entscheidet die *job-scheduling*-Strategie des Betriebssystems, welche der Jobs sich gleichzeitig im Hauptspeicher befinden. Im Zeitscheibenbetrieb müssen die Jobs dann abwechselnd auf den Sekundärspeicher ausgelagert werden (*swapping*), wodurch die Ein-/Ausgabe des Computersystems erheblich belastet wird und nur wenige Jobs zur Ausführung bereitstehen. Die Steigerung der Durchsatzleistung, die bei der Parallelisierung von Programmen mit sehr großem Speicherplatzbedarf erreicht werden kann, wurde für CRAY-Rechner mit realem Hauptspeicherkonzept in [Bie87] und [Lin90] nachgewiesen. Für ein Computersystem mit virtuellem Speicherkonzept ist dagegen nicht der von den Jobs angeforderte Speicherplatz entscheidend, sondern die Anzahl der Seitenrahmen des physikalischen Speichers, die für den aktiven Arbeitsbereich (*working set*) der Jobs benötigt werden.

Die Anzahl der ausführbereiten Prozesse wird im Mehrprogrammbetrieb nicht nur durch die begrenzte Hauptspeicherkapazität eingeschränkt. Nicht ausführbereit sind alle Prozesse, die auf die Verfügbarkeit einer zur Ausführung benötigten Ressource warten. Die im Hauptspeicher befindlichen Jobs können insbesondere dann nicht ausgeführt werden, wenn sie auf die Beendigung von Ein-/Ausgabeoperationen warten [Bie87]. Die Ein-/Ausgabewartezeit eines Jobs ist die Zeitspanne, die ein Job aufgrund von Ein-/Ausgabeoperationen blockiert ist. Der Zusammenhang zwischen den von einem Job durchgeführten Ein-/Ausgabeoperationen der transferierten Datenmenge und der Ein-/Ausgabewartezeit ist jedoch in hohem Maß von dem spezifischen Ein-/Ausgabesystem der Computersysteme abhängig und wird hier nicht weiter betrachtet.

Parallele Programme können jedoch aufgrund der Lastverteilung und des Parallelisierungsgrades nicht immer alle Prozessoren für die Ausführung nutzen. Bei einem statischen Scheduling wird jedem parallelen Programm eine feste Prozessorzahl zur Verfügung gestellt, die während der Ausführungsdauer des Jobs konstant bleibt. Die Leerlaufzeiten $\mathrm{IDLETIME}_{\mathrm{LB}}$ und $\mathrm{IDLETIME}_{\mathrm{PG}}$ aufgrund von sequentiellen Regionen und der *load balance* führen dabei zu einem erhöhten Bedarf an Prozessorzeit CPUTIME und zu einem Anstieg der Ausführungsdauer der Arbeitslast. Im Gegensatz dazu werden die von einem parallelen Programm nicht genutzten

Prozessoren beim dynamischen Scheduling automatisch für die Bearbeitung anderer Programme genutzt. Der Parallelisierungsgrad PG und die *load balance* LB wirken sich dann nur noch auf die Ausführungszeit des parallelen Programms, nicht jedoch auf die Ausführungsdauer der Arbeitslast aus. Zur Nutzung der leerlaufenden Prozessoren muß jedoch ein *task*-Wechsel erfolgen. Dadurch entsteht bei der Ausführung einer parallelen Region ein zusätzlicher Aufwand, der erheblich höher als der beim statischen Scheduling entstehende Aufwand sein kann. Die dafür zusätzlich benötigte Prozessorzeit ist linear von der Anzahl der von dem Programm ausgeführten parallelen Regionen abhängig. Der durch einen *task*-Wechsel verursachte Aufwand hängt bei Computersystemen mit lokalen Cache-Speichern von den Eigenschaften der Programmen ab. Der zusätzliche Aufwand, der durch das Neuladen und eventuell das Sichern der Cache-Speicher (*cache corruption*) nach einem *task*-Wechsel erfolgt, wird von der vom Programm genutzten Cache-Speicher-Kapazität bestimmt [Sto90]. Der Bedarf an Cache-Speicher-Kapazität von verschiedenen sequentiellen Programmen des SPEC-Benchmarks wird in [CH91] untersucht. In Computersystemen mit Cache-Speicher kann ein sequentielles Programm aufgrund der oben genannten Gründe häufig effizienter ausgeführt werden. Stehen in einem solchen Rechner hinreichend viele Jobs zur Ausführung bereit, um alle Prozessoren zu nutzen, so kann das Betriebssystem durch die sequentielle Ausführung von parallelen Programmen den Durchsatz steigern [Pol89]. Ist dies der Fall, so entsteht bei der Ausführung eines parallelen Programms nur ein minimaler Aufwand. Der durch zusätzliche *task*-Wechsel und *cache corruption* verursachte Aufwand entfällt dann.

2.4.3 Wechselwirkungen von Programmen im Mehrprogrammbetrieb

Die Ausführungszeit einer Arbeitslast wird auf einem Multiprozessorsystem von der gemeinsamen Nutzung der Ressourcen durch die Jobs beeinflußt. Dabei kommt es immer dann zu Konkurrenzsituationen, in denen Verzögerungen auftreten, wenn der Bedarf der Jobs die zur Verfügung stehenden Ressourcen übersteigt. Die Parameter einer Arbeitslast werden deshalb im folgenden relativ zu einem bestimmten System angegeben. So wird die Ausführungsdauer eines Jobs einerseits von den auszuführenden Operationen und andererseits von der Leistung des Computersystems bestimmt. Die Belastung durch einen Job wie z.B. eine Menge von auszuführenden Rechenoperationen wird daher mittels einer Leistung (z.B. MFLOPS) und der Dauer der Belastung (z.B. der Prozessorzeit) beschrieben. Die für die Vollendung eines Jobs benötigten Operationen lassen sich in Rechen-, Speicher- und Ein-/Ausgabeoperationen unterteilen.

Die von einem Job auf einem Computer erreichte Rechenleistung wird mit Hilfe der Einheit MFLOPS (*million floating point operations per second*) angegeben. Die Rechenoperationen benötigen eine bestimmte Ausführungsdauer, über die der Job weitere Ressourcen des Computersystems benötigt. Die Ausführungsdauer, die für

die Rechenoperationen auf einem Computersystem benötigt werden, hängt jedoch nicht nur von der Anzahl der Rechenoperationen, sondern auch von dem Typ der Operationen (Addition, Multiplikation, Division), der Art der Ausführung (skalar oder vektoriell) und beim *chaining* von deren Reihenfolge ab. Die hier betrachteten Computersystemen verwenden nur privat genutzte Rechenwerke, bei denen keine Konkurrenzsituationen auftreten können. Dies jedoch immer der Fall; so werden die arithmetischen *pipelines* der FUJITSU VP-S von bis zu zwei Prozessoren gemeinsam genutzt.

Im Gegensatz zu den Rechenoperationen führen die Speicheroperationen eines Jobs auf den hier betrachteten Multiprozessorsystemen mit gemeinsamem Hauptspeicher zu einer Wechselwirkung zwischen den Jobs der Arbeitslast. Die Konkurrenz der gleichzeitig auf verschiedenen Prozessoren ausgeführten Jobs um den Zugriff auf den gemeinsamen Hauptspeicher führt zu Zugriffskonflikten und damit zu Verzögerungen bei der Ausführung. Die von einem Job benötigte Speicherleistung wird durch die Einheit MMREFS (*million memory references per second*) beschrieben. Je mehr Speicheroperationen von den gleichzeitig ausgeführten Jobs durchgeführt werden, desto größer ist die Wahrscheinlichkeit, daß es aufgrund von Zugriffskonflikten zu Verzögerungen kommt. Der Einfluß von Zugriffsverzögerungen auf die Ausführungsdauer von Programmen im Mehrprogrammbetrieb wird z.B. in [Hof90] durch Messungen auf der CRAY X-MP und CRAY Y-MP bestätigt. Da diese Konflikte nur bei gleichzeitigem Zugriff der Prozessoren auf den gemeinsamen Hauptspeicher auftreten können, ist neben der Häufigkeit von Speicherzugriffen jedes einzelnen Jobs auch die Korrelation der Speicherzugriffe in der Arbeitslast bedeutsam. Die Korrelation der Speicherzugriffe zwischen unabhängigen sequentiellen Jobs ist von der Arbeitslast abhängig. Im Gegensatz dazu sind die Speicherzugriffe der kooperierenden Prozesse eines Programms, das nach dem SPMD-Modell parallelisiert wurde, hoch korreliert. Die von den Jobs durchgeführten Speicherzugriffe können skalar oder vektoriell erfolgen, wobei nur durch vektorielle Speicheroperationen die maximale Speicherbelastung erreicht wird. Typische Vektoroperationen besitzen eine Schrittweite (*stride*) von eins, die den verschränkten Speicher eines Vektorrechner in optimaler Weise nutzen. Vektorspeicherzugriffe können jedoch mit anderen Schrittweiten erfolgen, wobei insbesondere Zugriffe mit einer Zweierpotenz als Schrittweite nur einen Teil der Speicherbänke nutzen und deshalb zu einer hohen, ungleichmäßigen Belastung des Speichersystems führen. Durch solche Speicherzugriffe kann nicht nur die Ausführungsdauer der Jobs mit ungünstigem Speicherzugriffverhalten, sondern auch die Ausführungsdauer der übrigen Jobs der Arbeitslast ansteigen. Das durch die Parallelverarbeitung veränderte Zugriffsverhalten kann auch in diesen Fällen Auswirkungen auf die Ausführungsdauer einer Arbeitslast haben.

Der Zugriff auf globale Cache-Speicher unterliegt denselben Einflüssen wie der auf einen gemeinsamer Hauptspeicher. Dabei muß berücksichtigt werden, daß die Effizienz der globalen Cache-Speicher ebenfalls durch die gemeinsame Nutzung beeinträchtigt werden kann. Entscheidend dafür ist die Anzahl der von einem Job modifizierten Cache-Zeilen (*footsteps*, [Sto90]). Bei der gleichzeitigen Ausführung

von unterschiedlichen Jobs auf verschiedenen Prozessoren konkurrieren die Jobs um den Platz im gemeinsamen Cache-Speicher. Je größer die Anzahl der modifizierten Cache-Zeilen ist, desto größer ist die Wahrscheinlichkeit, daß die Jobs sich die Zeilen im Cache-Speicher wechselseitig überschreiben. Während die Nutzung lokaler Cache-Speicher für die Bearbeitung sequentieller Programme häufig sehr gut funktioniert, bereitet die Nutzung von lokalen Cache-Speichern für die globalen Daten paralleler Programme Schwierigkeiten (siehe Abschnitt 2.3.4). Der Leistungsverlust, der durch Parallelverarbeitung auf Computersystemen mit nicht-konsistenten Cache-Speichern eintritt, wird durch die Cache-Fehlerrate der Jobs in Bezug auf globale Daten bestimmt.

Die Ein-/Ausgabe wird ebenso wie der gemeinsame Hauptspeicher von allen Jobs der Arbeitslast gemeinsam genutzt. Daher entstehen auch bei der Ausführung von Ein-/Ausgabeoperationen Wechselwirkungen zwischen den Jobs der Arbeitslast. Die Ein-/Ausgabeleistung eines Jobs wird durch die Einheit MIOS (*million array elements transferred to or from an I/O device per second*) wiedergegeben. Die Untersuchung des Einflusses von Ein-/Ausgabeoperationen auf die Ausführung einer Arbeitslast ist jedoch zu umfangreich, um in dieser Arbeit berücksichtigt zu werden. Eine Beeinflussung der Messungen durch die Ein-/Ausgabe kann jedoch nicht ausgeschloßen werden, da zum Laden und zur Ausführung der Jobs Ein-/Ausgabeoperationen benötigt werden.

Die speziellen Eigenarten paralleler Programme treten insbesondere beim Scheduling auf und legen eine Untersuchung der Effizienz und des Aufwands verschiedener Scheduling-Strategien und deren Auswirkungen auf den Durchsatz nahe. Die Wechselwirkungen von parallelen Programmen mit der Arbeitslast sind insbesondere in Bezug auf die Relevanz von Speicherzugriffen für die Leistung von Multiprozessor-Vektorrechnern mit gemeinsamem Hauptspeicher beachtlich und müssen bei Messungen berücksichtigt werden.

2.5 Meßmethoden

Eine Leistungsmessung ermöglicht die Bestimmung der Leistung des gesamten Computersystems für eine bestimmte Aufgabe und damit die Untersuchung und Quantifizierung von Effekten, die durch die Wechselwirkung der leistungsrelevanten Komponenten untereinander hervorgerufen werden. Zur Messung der Leistung ist es notwendig, die für die Bearbeitung eines Jobs benötigte Zeit hinreichend exakt zu bestimmen. Die Genauigkeit der Messung wird zunächst durch die zeitliche Auflösung der zur Messung verwendeten Uhr eingeschränkt. Dabei muß berücksichtigt werden, daß einige der Meßwerkzeuge Ressourcen des Computersystems benötigen und somit einen Aufwand (*Meß-overhead, artifact*) verursachen, der die Ausführungsgeschwindigkeit des zu messenden Programms verringert. Die so gemessene Leistung beschreibt das Computersystem nur in der Meßsituation und unterscheidet

sich durch einen Meßfehler von der tatsächlichen Leistung des Computersystems. Dieser Meßfehler muß genau untersucht und quantifiziert werden, da eine korrekte Leistungsanalyse durch falsche Annahmen über die Genauigkeit der Meßergebnisse erschwert wird.

In einer Mehrprogrammumgebung erfordert die Zeitmessung eines Jobs Unterstützung durch das Betriebssystem oder spezielle Meßwerkzeuge, da es ohne solche Hilfsmittel beispielsweise nicht möglich ist, die einem Job zugeteilte Prozessorzeit zu bestimmen. Die Analyse der gemessenen Leistung wird vereinfacht, wenn die Meßwerkzeuge neben der abgelaufenen Zeit auch noch Informationen über den Ablauf des gemessenen Programms zur Verfügung stellen.
Ein Monitor ist ein Werkzeug, das ein Computersystem überwacht und dem Benutzer dadurch Informationen über die Abläufe im Computersystem zur Verfügung stellt. Die von einem Monitor beobachteten Ereignisse, nennt man *events*, und sie signalisieren Zustandsübergänge im Computersystem, die mit dem Anfang oder dem Ende von leistungsrelevanten Vorgängen verknüpft sind. Solche *events* können in der Durchführung einer Fließkommmaoperation, im Auftreten eines Speicherzugriffkonflikts oder im Auslagern eines Jobs bestehen. Die Überwachung solcher *events* durch einen Monitor erfolgt entweder in festen Zeitabständen (*sampling*) oder kontinuierlich. Eine kontinuierliche Überwachung wird bei der Generierung von Ablaufprotokollen (*tracing*) verwendet.
Monitore unterteilt man in Hinblick auf die Realisierung in Software- und Hardware-Monitore. Ein Software-Monitor ist ein Meßprogramm, das auf dem Computersystem selbst abläuft und den Zustand eines zu messenden Programms überwacht. Dazu benötigt es aber Ressourcen des Computersystems und erzeugt dadurch einen Meß-*overhead*, der einen signifikanten Einfluß auf die gemessene Leistung haben kann. Ein Hardware-Monitor ist ein durch zusätzliche Hardware bereitgestelltes Meßwerkzeug, daß mit eigenen Ressourcen ausgestattet ist und deshalb von den Ressourcen des Computersystems unabhängig sein kann. Daher kann ein Hardware-Monitor bei einer Messung verwendet werden, ohne daß ein Meß-*overhead* und der damit verbundene Meßfehler auftreten kann. Die durch Hardware-Monitore meßbaren *events* umfassen aber nicht alle durch Software-Monitore meßbaren *events*, so daß auch kombinierte Monitore (Hybrid-Monitore) eingesetzt werden.

2.5.1 Zeitmessung

Für eine exakte Zeitmessung, die die Grundlage der Leistungsmessung mit oder ohne Unterstützung durch Monitore darstellt, wird eine Echtzeituhr im Computersystem benötigt. Eine Echtzeituhr ist eine durch Hardware realisierte Uhr, die sich im direkten Zugriff der Prozessoren befindet und ihrerseits keinen Einfluß auf den Ablauf der Arbeitslast hat. Eine oft verwendete Realisierung besteht aus einem Zähler, der einen Takt zählt, dessen Periode die zeitliche Auflösung der Messung bestimmt. Um sehr genaue Messungen zu ermöglichen, sollte die Auflösung des Zählers we-

nigstens im Bereich der Zeit liegen, die für die Ausführung einer Instruktion oder eines Speicherzugriffs benötigt wird. Da der Systemtakt die kleinste Zeiteinheit des Computersystems darstellt, in der Zustandsübergänge erfolgen, ist er als Zeitbasis der Echtzeituhr besonders geeignet. Um auch längere Zeitintervalle mit hoher Genauigkeit messen zu können, muß das Zählregister eine entsprechende Bitbreite besitzen; so stellen die in der CRAY X-MP/Y-MP verwendeten 46-Bit Zähler eine ausreichende Dimensionierung dar [Nel89].

Die mit einer Echtzeituhr erzielbare Meßgenauigkeit hängt nicht nur von ihrer zeitlichen Auflösung, sondern auch von einer möglichen Zugriffsverzögerung ab. Erfolgt der Zugriff aller Prozessoren auf die Echtzeituhr über ein gemeinsam genutztes Register, so können Verzögerungen durch Zugriffskonflikte entstehen. Der mögliche Meßfehler ist dann mindestens so groß wie die maximale Verzögerung beim gleichzeitigen Zugriff aller Prozessoren auf die Uhr. Dieser Fehler kann durch einen lokalen Uhr-Zugriff für jeden Prozessor vermieden werden. Bei massiv-parallelen Multiprozessorsystemen mit gemeinsamem Speicher wird der Aufwand geringer, wenn jeder Prozessor eine lokale Uhr erhält [Car89]. Um die Vergleichbarkeit der Uhrzeiten einzelner Prozessoren sowie die Rekonstruierbarkeit der Reihenfolge von Ereignissen auf verschiedenen Prozessoren zu gewährleisten, müssen die lokalen Uhren synchronisiert werden.
Ein besonders schneller Zugriff auf eine lokale Echtzeituhr wird ermöglicht, wenn sich diese als Register im Prozessor befindet, da der Zugriff auf die Echtzeituhr über einen Prozessorport einen größeren Aufwand verursacht. Die Zugriffsgeschwindigkeit auf die Echtzeituhr kann zusätzlich durch die Verwaltung des Zugriffsrechts verringert werden. Die geringste Verzögerung wird erreicht, wenn eine Applikation die Uhr direkt auslesen darf. Ist der Uhr-Zugriff geschützt, so daß ein Systemaufruf für das Auslesen der Uhr benötigt wird, entstehen weitere Zugriffsverzögerungen, die den Meß-*overhead* erhöhen und die Meßgenauigkeit begrenzen.

Eine Zeitmessung zur Leistungsbestimmung muß im Rahmen der angegebenen Meßgenauigkeit reproduzierbar sein, um eine Leistungsanalyse zu unterstützen. Dazu muß die Meßumgebung genau definiert und festgelegt werden, was im Mehrprogrammbetrieb insbesondere die Hintergrundlast betrifft. Eine Möglichkeit zur Beschreibung der Arbeitslast des Supercomputers während der Messung besteht in ihrer Charakterisierung mit Hilfe von Parametern [Wei90, CH91, BCG+91]. Dazu müssen allerdings alle Einflußgrößen der Arbeitslast bekannt sein, was durch die komplexe Architektur heutiger Supercomputer und die zeitlichen Varianz der Parameter einer Arbeitslast erschwert wird. Die Reproduzierbarkeit der Meßergebnisse wird wesentlich erhöht, wenn die Messung auf einer dedizierten Maschine mit einer genau festgelegten Hintergrundlast erfolgt [Bie87, Lin90, Nag90b]. Die Hintergrundlast muß sowohl in Bezug auf die Applikationen als auch in Bezug auf die Startzeit der Jobs festgelegt sein.

Die Messung der für ein Programm benötigten CPU-Zeit ist im Mehrprogrammbetrieb nicht durch die Messung der verstrichenen Realzeit (*Real-time*) möglich,

da in ihr neben der Prozessorzeit auch noch die Zeit enthalten ist, Programm auf die Bearbeitung wartet *(Wait-time)*. Das Betriebssystem muß also bei einem *task*-Wechsel die Echtzeituhr abfragen und die gemessenen Zeiten für jeden Job getrennt akkumulieren. Die von einem Job verbrauchte Prozessorzeit wird, da sie für die Betriebsmittelabrechnung (*accounting*) benötigt wird, von den meisten Betriebssystemen zur Verfügung gestellt. Für die Leistungsanalyse ist jedoch eine Aufschlüsselung der verstrichenen Zeit auf die unterschiedlichen Aktivitäten eines Programms wünschenswert. Die von einem Job benötige Zeit für die Ausführung von Benutzer-Code (*CPU-time*), von System-Code für den Job (*System-time*), die Wartezeit (*Wait-time*) und die Ein-/Ausgabezeit (*IO-time*) sollten getrennt akkumuliert und ausgewiesen werden. Auf Multiprozessorsystemen ist bei der Messung von parallelen Programmen eine Aufschlüsselung der Prozessorzeit auf die einzelnen Prozessoren nützlich. Ergänzend kann für jede Prozessoranzahl n die Zeit angegeben werden, in der das Programm gleichzeitig von genau n Prozessoren bearbeitet wird (*Overlap-time*). Eine durch das Betriebssystem möglichst fein untergliederte Aufteilung der *Real-time* auf die Aktivitäten des Jobs stellt eine erste Informationsquelle für die Leistungsanalyse dar.
Für das CEDAR-Projekt [KTV+91] wurde wegen der mangelnden Unterstützung der Zeitmessung durch das Betriebssystem HRTIME entwickelt, ein Hilfsprogramm zur Zeitmessung [Mal89]. Die Auflösung der dabei verwendeten Echtzeituhr von 10 μ-Sekunden erlaubt keine hochgenauen Messungen und ist hauptsächlich für das *profiling* zur Unterstützung der Leistungssteigerung von Programmen (*performance debugging*) gedacht. Durch HRTIME wird die Elapsed-Time in Ausführungszeit und Wartezeit zerlegt, die dann noch weiter aufgeschlüsselt werden. Für die parallele Ausführung wird die Benutzer- und Systemzeit für jeden Prozessor getrennt ausgegeben.

2.5.2 Software-Monitore

Durch einen Software-Monitor können Informationen über den zeitlichen Ablauf bei der Ausführung der zu messenden Programme gesammelt und zur Verfügung gestellt werden. Ein Software-Monitor ist ein Meßprogramm und kann deshalb oft mit geringem Aufwand implementiert und in das zu messende Programm integriert werden. Ein Meßprogramm ist modifizierbar und kann deshalb flexibel eingesetzt und an unterschiedliche Meßaufgaben adaptiert werden. Da ein Software-Monitor von dem Computersystem selbst ausgeführt wird, ist er in seinen Möglichkeiten durch die des Computersystems beschränkt. Er kann deshalb zum einen nur solche Ereignisse erfassen, die vom Computersystem für eine Abfrage durch Software zur Verfügung gestellt werden. Zum anderen benötigt der Software-Monitor zur Verarbeitung (Zählen, Protokollieren) der gemessenen Daten eine bestimmte Anzahl von Instruktionen und damit eine Zeit, in der keine neu eintreffenden Ereignisse erfaßt werden können. Der zeitliche Abstand, in dem zwei *events* eintreffen dürfen, muß also relativ groß sein (Ausführungszeit einiger bis zu einigen hundert Instruktionen),

damit eine Überwachung durch einen Software-Monitor effektiv möglich ist.

Die vom Software-Monitor benötigten Ressourcen des Computersystems stehen nicht für die Ausführung des zu messenden Programms zur Verfügung. Ein Software-Monitor erzeugt somit prinzipbedingt einen Meß-*overhead*, der bei einer vollständigen Überwachung proportional mit der Häufigkeit der zu messenden *events* steigt. Der durch den Meß-*overhead* eines Software-Monitors induzierte Meßfehler kann in einigen Fällen durch eine Umverteilung (*trade off*) des Aufwands zwischen den Ressourcen Prozessorzeit, Hauptspeicherplatz und Ein-/Ausgabeoperationen reduziert werden. So werden zur Speicherung von Ablaufprotokollen Pufferspeicher im Hauptspeicher verwendet, die entweder das ganze Protokoll aufnehmen können oder auf den Sekundärspeicher gerettet werden müssen, bevor sie überlaufen. Der für das Protokoll benötigte Speicherbedarf kann mittels Vorverarbeitung bzw. Reduktion der Daten durch die Prozessoren gesenkt werden, wodurch aber die benötigte Prozessorzeit steigt. Die Kapazität des für das Protokoll zur Verfügung stehenden Hauptspeichers bestimmt, ob und wie oft dieser auf den Sekundärspeicher übertragen werden muß, und damit auch, wie viele Ein-/Ausgabeoperationen vom Monitor durchgeführt werden müssen.
Der Meß-*overhead* eines Software-Monitors muß durch vergleichende Messungen mit und ohne Monitor bzw. ein- und ausgeschaltetem Monitor quantifiziert werden, um die Größenordnung des Meßfehlers bestimmen und seinen Einfluß abschätzen zu können. Die zur Verfügung stehenden Meßmethoden *sampling* und *tracing* unterscheiden sich durch den entstehenden Meß-*overhead* und die erreichbare Meßgenauigkeit.

Sampling Beim *sampling* wird das Computersystem in konstanten Zeitabständen, also periodisch, überwacht, und die dabei gewonnenen Informationen werden gegebenenfalls protokolliert. Der durch das *sampling* eingeführte Meß-*overhead* steigt proportional mit der *sampling*-Frequenz, da die Routinen des Software-Monitors mit kleiner werdender *sampling*-Periode häufiger aufgerufen werden und mehr Leistung des Computersystems benötigen. Der Meß-*overhead* eines *sampling-software*-Monitors läßt sich also durch das Erhöhen der *sampling*-Periode reduzieren und so an eine gewünschte Meßfehlergrenze anpassen. Da aber die erreichbare zeitliche Auflösung beim *sampling* durch die Dauer der *sampling*-Periode beschränkt ist, wird die Genauigkeit der Messung durch Erhöhung der *sampling*-Periode verringert. Bei der Wahl der *sampling*-Periode muß also der Zielkonflikt zwischen dem Wunsch nach maximaler zeitlicher Auflösung und minimalem Meß-*overhead* berücksichtigt und ein entsprechender Kompromiß gefunden werden.
Eine weitere Fehlerquelle stellen beim *sampling* Ereignisse dar, deren Dauer kürzer als die *sampling*-Periode ist und die deshalb auftreten können, ohne vom Software-Monitor erkannt zu werden. Dieser Effekt ist in der Nachrichtentechnik als *aliasing* bekannt und verhindert eine vollständige Rekonstruktion des Ablaufs der Ereignisse im Computersystem. Die Ergebnisse von Messungen, bei denen *aliasing* auftritt, können aber statistisch ausgewertet werden, wenn nachgewiesen werden kann, daß

der Erwartungswert des Meßfehlers gleich Null ist, die Messung also nicht durch systematische Fehler verfälscht ist.
In [DI90] wurde *sampling* zur Bestimmung des Mehrprogramm-*overheads* benutzt. Die Störung der Meßwerte durch *aliasing* wurde dabei durch ein weißes Rauschen modelliert. Um ein Maß für die Störung zu erhalten, wurde dann die Standardabweichung der Meßwerte experimentell bestimmt. Schließlich zeigten die gemessenen Werte, daß der Erwartungswert des Meßfehlers im statistischen Sinne, d.h. im Rahmen der Fehlerwahrscheinlichkeit, gleich Null ist. Das *sampling* wird in Uniprozessorsystemen zur Erzeugung von Laufzeitstatistiken (*profiling*) benutzt, indem der Programmzähler des Prozessors abgefragt und einem Unterprogramm zugeordnet wird. Das UNIX-Kommando *prof* ist ein Beispiel für ein solches *profiling tool*.

Event Tracing Das *event tracing* wird in Software-Monitoren immer dann verwendet, wenn vollständige Ablaufprotokolle oder eine hohe zeitliche Auflösung benötigt werden. Es unterscheidet sich vom *sampling* dadurch, daß der Monitor passiv wartet, bis er durch ein *event* aktiviert wird. Die zu überwachende Software muß den Monitor also selbständig aufrufen, was durch eine Instrumentierung der zu messenden Software erreicht wird. Der Software-Monitor wird dazu entweder durch zusätzlichen Code (*macros*) in das zu messende Programm integriert oder durch in das Programm eingefügte Monitoraufrufe aktiviert. Da das Instrumentieren der Software mit einem großen Arbeitsaufwand verbunden sein kann, gibt es sowohl Compiler als auch spezielle Werkzeuge, die nach Vorgabe von Optionen eine Instrumentierung für diverse Standardaufgaben automatisch durchführen. Eine automatische Instrumentierung ist aber nicht in allen Fällen möglich oder sinnvoll. Sollen bei der Messung Aktivitäten des Betriebssystems oder des Laufzeitsystems überwacht werden, so müssen auch diese dafür vorbereitet bzw. instrumentiert sein. Ein *event tracing software*-Monitor, der zur Leistungsmessung verwendet werden soll, muß neben der Information über das Auftreten und die Reihenfolge von *events* auch noch Informationen über die verstrichene Zeit zwischen den *events* bereitstellen. Dazu wird jedes *event* im Ablaufprotokoll um einen Eintrag mit dem aktuellen Wert der Echtzeituhr (*time stamp*) ergänzt. Ein *event*-Eintrag im Protokoll sollte neben dem *time stamp* auch noch Zusatzinformationen aufnehmen können, die zur Laufzeit an den Monitor übergeben werden und ein *event* genauer spezifizieren.
Die durch einen Software-Monitor überwachten *events* lassen sich in zwei Gruppen unterteilen; in die Standard-*events* und die Benutzer-*events* (*user events*). Die Standard-*events* sind alle durch den Software-Monitor und die Instrumentierungswerkzeuge vordefinierten *events*, zu denen auch die *events* des Betriebs- und des Laufzeitsystems gehören, die auch als System-*events* bezeichnet werden. Im Gegensatz dazu sind die Benutzer-*events* für den Benutzer frei definierbar und erlauben es, mit einem Software-Monitor Ereignisse zu untersuchen, die speziell für eine Messung benötigt werden oder nicht automatisch von Software-Werkzeugen erkannt werden können Im folgenden werden die Eigenschaften eines im Rahmen der Forschung entwickelten (CTRACE) und zweier kommerzieller *event tracing*-Werkzeuge (CRAY MHTB und ATEXPERT) erläutert.

- Im Cedar-Projekt wurde CTRACE, ein *event-tracing-software*-Monitor entwickelt, der sowohl durch den FORTRAN-Compiler als auch durch die Bibliotheken und das Betriebssystem unterstützt wird. Der Compiler unterstützt die Instrumentierung von Standard-*events* bezüglich Unterprogrammen, Basisblöcken, Schleifenparallelität, Synchronisationsanweisungen etc. mit Hilfe des Präprozessors. Das Betriebssystem erlaubt die Protokollierung von *task*-Wechseln sowie weiteren *task*-spezifischen *events*, und die Bibliotheken stehen in einer instrumentierten Version zur Verfügung. Für die Benutzer-*events* werden von CTRACE drei verschiedene Klassen zur Verfügung gestellt: mark, entry und exit. Die *events* der Klasse mark sind für einfache Benutzer-*events* vorgesehen. Speziell für *events*, die paarweise auftreten und den Anfang (*entry*) und das Ende (*exit*) eines Zustands kennzeichnen, werden die anderen beiden Klassen bereitgestellt, deren korrekte Anwendung durch die Werkzeuge überprüft wird [Mal89].

- Der *multitasking history trace buffer* (MHTB) ist ein kommerziell verfügbarer *event-tracing*-Monitor der Firma CRAY für die X-MP/Y-MP Rechnerfamilie [Cra89]. Der MHTB unterstützt bis zu 63 Benutzer-*events* und 39 System-*events* aus dem Bereich des *macrotasking* wie z.B. *locks*, *barrier* etc. Der MHTB ist fest in die Laufzeitumgebung integriert und protokolliert in einem zirkulären Pufferspeicher die letzten 1000 *events* eines *multitasking*-Jobs. Daher entfällt beim MHTB der Unterschied zwischen Meßumgebung und Normalbetrieb. Sollen aber mehr als 1000 *events* protokolliert werden, so muß das Retten des Pufferspeichers auf den Sekundärspeicher durch eine Anweisung veranlaßt werden, was einen zusätzlichen Meß-*overhead* zur Folge hat. Zur Auswertung der erzeugten Ablaufprotokolle auf X-Window-Systemen steht neuerdings das graphische Analysepaket *GMAT* zur Verfügung.

- Der Software-Monitor ATEXPERT ist ein spezialisiertes Werkzeug zur Bestimmung der durch Parallelisierung mittels *autotasking* erreichbaren Leistungssteigerung. Die dafür benötigten Informationen werden mit Hilfe von *event tracing* beim Ablauf des durch ATEXPERT automatisch instrumentierten Programms gesammelt. Die Auswertung der so gewonnenen Information wird durch eine graphische Darstellung auf der Basis von X-Window erleichtert. Dieses Werkzeug dient dazu Engpässe bei der Parallelisierung von Programmen zu erkennen und unterstützt damit, ebenso wie ein *profiler*, die Leistungssteigerung von Programmen [Cra91].

Der beim *event tracing* entstehende Meß-*overhead* läßt sich nicht über einen Parameter steuern wie das *sampling* über die *sampling*-Periode. Beim *event tracing* muß der Monitor so angelegt sein, daß er die Ressourcen am wenigsten nutzt, deren Einfluß auf die Messung am größten ist. In einigen Fällen läßt sich der für das *event tracing* benötigte Aufwand in Bezug auf Prozessorzeit, Speicherplatzbedarf und Ein-/Ausgabeoperationen so umverteilen, daß ein geringer Störung der Messung erreicht wird. So läßt sich das Protokoll von komplexen Ereignissen, deren

Erkennung zuviel Prozessorzeit benötigen würde, manchmal aus dem einer größeren Anzahl einfacher *events*, das mehr Speicherplatz benötigt, nach der Messung rekonstruieren. Im Cedar-Projekt wurde so aus dem Protokoll der Zeitpunkte der Task-Wechsel aller Prozessoren die Anzahl der Prozessoren bestimmt, die gleichzeitig an einem parallelen Job arbeiten.

In [Gai89] wird eine Möglichkeit der Hardware-Unterstützung vorgestellt, mit der der Einfluß des Software-Monitors auf die Messung reduziert oder sogar vermieden werden kann. Die Ausführung des zu messenden Programms wird dabei durch einen parallelen Trap oder Interrupt so unterbrochen, daß sämtliche Prozessoren und alle anderen Vorgänge im Computersystem gleichzeitig angehalten und so konserviert werden, daß nach der Bearbeitung der Trap-Routine (z.B. des Software-Monitors) das gesamte Computersystem wieder gleichzeitig gestartet werden kann, ohne daß eine Beeinflussung des Ablaufs stattgefunden hat. Man nennt eine solche Vorrichtung *parallel trap* [Gai89] oder *stop on a dime* [Tel89]. Die Realisierung einer solchen Hardware ist jedoch bei Multiprozessor-Vektorrechnern aufwendig, da bei der Zustandssicherung auch der Zustand aller internen Pipelines sowie der Stand der eingeleiteten Speicherzugriffe durch Vektorladeoperationen rekonstruierbar sein müßte.

2.5.3 Hardware-Monitore

In allen Bereichen, in denen Software-Monitore nicht auf die zu überwachenden *events* zugreifen, diese nicht schnell genug erfassen oder bei der Messung einen zu hohen Meß-*overhead* erzeugen, werden Hardware-Monitore eingesetzt. Ein Hardware-Monitor ist ein Meßwerkzeug, das durch unabhängige, eigene Meßressourcen realisiert ist und die Signale der Computerhardware überwacht. Es gibt eine Vielzahl von Signalen der Prozessoren (Datenbus, Adressbus, Ausführung einer Instruktion oder Gleitkommaoperation), der Cache-Speicher (Cache-Treffer/Fehler), des Hauptspeichers (Zugriffskonflikt, Zugriff auf eine Bank) etc., die durch einen Hardware-Monitor überwacht werden können.
Zu dieser Überwachung stehen spezielle Meßressourcen in Form von Zählern und Komparatoren zur Verfügung; neuere Monitore besitzen auch eigene Prozessoren, Haupt- und Sekundärspeicher. Diese Ressourcen können selbständig das Computersystems überwachen und die dabei gemessenen Daten weiter verarbeiten. Aufgrund der Spezialisierung dieser Meßressourcen kann die zur Verarbeitung der gemessenen Daten benötigte Zeit gegenüber Software-Monitoren stark reduziert werden, so daß sich auch zeitlich erheblich kürzer aufeinanderfolgende *events* messen lassen. Zur Ableitung der *events* aus den am Hardware-Monitor anliegenden Signalen stehen mehrere Techniken zur Verfügung, die sich einerseits durch den Aufwand und andererseits durch ihre Mächtigkeit voneinander unterscheiden [Car89].

- Es werden nur *events* erkannt, die Hardware-Signalen direkt entsprechen.

- Ein *event* wird erkannt, wenn eine durch eine logische Gleichung oder ein Signalmuster spezifizierte Kombination von Signalen eintrifft.
- Ein *event* wird mit einer Folge von am Monitor anliegenden Signalmustern identifiziert.

Die vom Hardware-Monitor erkannten Ereignisse müssen ausgewertet (gezählt) oder mit Hilfe von *sampling* oder *event tracing* für eine Auswertung nach der Messung protokolliert werden. Dem Zählen kommt eine Sonderstellung bei der Verarbeitung von Ereignissen zu; einerseits wegen der leichten und effizienten Realisierbarkeit, andererseits wegen der Bedeutung der Anzahl der *events*, die oft mit der Menge der geleisteten Arbeit identifiziert werden kann. Die Anzahl der gleichzeitig zählbaren *events* ist durch die Anzahl der Zähler im Monitor beschränkt. Da diese jedoch nicht beliebig erhöht werden kann, werden oft mehrere Signale von jedem Zähler alternativ gezählt. Die gleichzeitig zählbaren Ereignisse bilden dann eine Gruppe. Um alle Gruppen von Ereignissen für eine Arbeitslast zählen zu können, muß die Messung dann entsprechend der Anzahl der Gruppen wiederholt werden.
Beim Zählen stellen Überschreitungen der Bereichsgrenzen, sogenannte Zählerüberläufe, eine mögliche Fehlerquelle dar. Durch Zähler großer Bitbreiten (HPM der CRAY X-MP: 46 Bit, [Cra86]) läßt sich dieses Problem für Messungen in realistischen Zeitintervallen beseitigen. Das Auftreten eines Zählerüberlaufs muß dem Benutzer mitgeteilt werden, da die gemessenen Werte dadurch wertlos werden können. Möchte man die Relationen zwischen den Zählern wahren, so stellt ein Anhalten aller Zähler beim Überlauf eines Zählers eine Lösung dar [BMN89]. Das Überlaufen der Zähler kann durch ein *sampling* der Zähler ausgeschlossen werden, wenn die Zähler in so kurzen Abständen periodisch ausgelesen, protokolliert und zurückgesetzt werden, daß ein Zählerüberlauf innerhalb einer *sampling*-Periode nicht möglich ist. Das so gewonnene Protokoll der Zählerstände beschreibt die geleistete Arbeit des Computer-Systems außerdem mit größerer zeitlicher Auflösung.

Das *sampling* kann in Hardware-Monitoren ebenso wie in Software-Monitoren zur Überwachung und statistischen Auswertung von *events* eingesetzt werden. Der Hardware-Monitor muß dazu allerdings über einen Haupt- und eventuell einen Sekundärspeicher verfügen. Verwendet der Hardware-Monitor dazu die Speicher des Computersystems, so kann das gemessenen System beeinflußt werden, und es kommt aufgrund des Meß-*overheads* zu Meßfehlern. Um eine störungsfreie Messung zu ermöglichen, sollte der Hardware-Monitor daher mit einem eigenen Haupt- und Sekundärspeicher ausgestattet werden. Das *event tracing* durch einen Hardware-Monitor kann die Zeitinformation durch einen *time stamp*) bereitstellen. Ist ein Hardware-Monitor ein integrativer Bestandteil eines Computersystems, so kann er zusätzlich durch Meßprogramme unterstützt werden. Einen solche Kombination von Software- und Hardware-Monitor nennt man Hybrid-Monitor. Ein solcher Hybrid-Monitor ist insbesondere zur Unterstützung des *event tracings* von *software events* geeignet und ermöglicht Messungen mit geringeren Meßfehlern. Die Software-Seite des Monitors muß dabei nur noch für eine Instrumentierung

sorgen, die den Hardware-Monitor aktiviert; die Hardware-Seite kann dann die Ergänzung um einen *time stamp* und das Abspeichern in dem Speicher des Hardware-Monitors durchführen. Im folgenden werden zwei in ein Computersystem integrierte Hardware-Monitore vorgestellt.

- In der CRAY X-MP/Y-MP Serie befindet sich der *hardware performance monitor* (HPM). Er enthält für jeden Prozessor neun Zähler, von denen einer den Systemtakt zählt und somit als Echtzeituhr dient. Die übrigen acht Zähler können jeweils eine von vier Gruppen mit acht *events* gleichzeitig zählen. Gezählt werden können alle Fließkommaoperationen, diverse Befehlsgruppen, Speicheraktivitäten, und verschiedene Verzögerungen durch belegte Register und Funktionseinheiten etc. [Nel89].
- Im RP3-Projekt enthält jeder der Prozessoren einen *performance monitor chip* (PMC). Der PMC wird entweder vom Prozessor oder dem Ein-/Ausgabe-System gesteuert. Er enthält mehrere Zähler, die einige in Gruppen eingeteilte Ereignisse zählen können. Die *events* betreffen fast ausschließlich die Speicherstruktur, also den Cache-Speicher, den lokalen Speicher, den virtuellen Speicher und das Speicher-Netzwerk-Interface. Der PMC kann absolute und virtuelle Adressen sowie einige andere Signale des Speicherzugriffs mittels *sampling* protokollieren.

Die Anwendung von Hardware-Monitoren, die nur eigene Ressourcen verwenden, ist bei einer Messung unproblematisch. Werden vom Hardware-Monitor zusätzlich Ressourcen des Computersystems wie z.B. Hauptspeicherkapazität benötigt, so muß der entstehende Meß-*overhead* wie bei einem Software-Monitor quantifiziert und gegebenenfalls minimiert werden. Da sich die Fähigkeiten von Software- und Hardware-Monitoren ergänzen, hängt die Wahl des zur Messung verwendeten Werkzeuges von der Art des zu untersuchenden Effektes ab.

Kapitel 3

Multiprozessor-Vektorrechner mit gemeinsamem Speicher

Die heute verfügbaren Multiprozessor-Vektorrechner mit gemeinsamem Speicher sind aufgrund der Erfahrung, die viele der Hersteller schon mit Vorgängermodellen gewonnen haben, weitgehend ausgereift und mit stabilen Betriebssystemen und leistungsfähigen FORTRAN-Compilern ausgestattet. Sie werden häufig für technisch-wissenschaftliche Probleme eingesetzt und nach ihrer Leistung in Minisupercomputer bzw. Supercomputer unterschieden. Die in dieser Arbeit analysierten Computersysteme unterscheiden sich sowohl in der Architektur als auch in der Leistung erheblich, obwohl die *peak performance* der Spitzenmodelle Alliant FX/2828, CONVEX C3880 und CRAY X-MP/416 eine direkte Vergleichbarkeit nahe legt (siehe Tabelle 3.1).

Neben unterschiedlichen Architekturen werden in diesen Computersystemen auch unterschiedliche Technologien verwendet. In der Tabelle 3.2 werden die Leistungs-

Computersystem	FX/2828	C3880	X-MP/416	Y-MP8/8128
Prozessoren	28	8	4	8
Rechenleistung [MFLOPS] 64 Bit	1120	960	940	2666
Rechenleistung [MFLOPS] 32 Bit	2240	1920	-	-
Speicherbandbreite [MWorte/s]	53/160	1100	1880	8530
Speicherverschränkung	8	256	64	256
Maximale Speicherzykluszeit [ns]	600	233	34	30
Globaler Cache-Speicher [KWorte]	2048	-	-	-
Hauptspeicher [MWorte]	512	512	16	128
SSD [MWorte]	-	-	32	512

Tabelle 3.1: Spitzenmodelle der Computerserien

daten der in den Systemen verwendeten Prozessoren gegenübergestellt. Die Alliant FX/2828 verwendet sowohl für die Prozessoren als auch für den Hauptspeicher preisgünstige Standardbauteile. Um dennoch eine einem Minisupercomputer entsprechende Rechenleistung zu erzielen, wird eine relativ hohe Anzahl von bis zu 28 Prozessoren eingesetzt, die durch eine extensive Nutzung von Cache-Speichern ihre *peak performance* erreichen sollen. Bei der CONVEX C3880 kommt nur noch beim Hauptspeicher Standard-Technologie zum Einsatz, wodurch die Realisierung großer Hauptspeicherkapazitäten erleichtert wird. Die CONVEX C3880 zeichnet sich gegenüber den C3200 und den C3400 Modellen von CONVEX bei den Vektorprozessoren und dem Verbindungsnetzwerk durch eine erheblich leistungsfähigere Technologie (Gallium-Arsenid) aus. Im Gegensatz zu den beiden gerade betrachteten Computersystemen zählen die Systeme der Firma CRAY Research Inc. eindeutig zu den Supercomputern. Die CRAY X-MP/416 ist von der CRAY Y-MP8/832 abgelöst worden, die als Nachfolgermodell die doppelte Anzahl von Prozessoren und eine um den Faktor 1.41 erhöhte Taktrate realisiert. Die Architektur der Systeme unterscheidet sich aus der Sicht des Anwenders jedoch nur geringfügig. Bei beiden Modellen wurden sowohl für die Prozessoren als auch für den Hauptspeicher keine Standardbausteine verwendet. Als Nachfolger der YMP8/832 wurde inzwischen die mit 16 Prozessoren ausgestattete CRAY C-90 angekündigt, bei der auch die Taktrate noch einmal wesentlich gesteigert werden konnte.

Prozessor der	FX/2828	C3880	X-MP/416	Y-MP8/832
Taktperiode [ns]	25	16.67	8.5	6
Rechenleistung [MFLOPS] 64 Bit	40	120	235	333
Rechenleistung [MFLOPS] 32 Bit	80	240	-	-
Speicherbandbreite [MWorte/s]	20	60	350	500
Vektorregister	-	8	8	8
Daten Cache-Speicher [KWorte]	1	2	-	-

Tabelle 3.2: Leistung eines einzelen Prozessors im Überblick

Um eine Vergleichbarkeit der technischen Daten zu erleichtern, beziehen sich die Daten, soweit keine anderen Angaben erfolgen, auf Gleitkommazahlen mit einer Wortbreite von 64 Bit, die als Gleitkommaoperationen doppelter Genauigkeit bezeichnet werden. Dieser Sprachgebrauch unterscheidet sich von dem der Firma CRAY, die unter einfacher Genauigkeit Gleitkommazahlen mit einer Länge von 64 Bit und unter doppelter Genauigkeit Gleitkommazahlen mit einer Länge von 128 Bit versteht. Soweit nicht anders vermerkt, beziehen sich Daten bezüglich der Speicherkapazität und der Speicherbandbreite ebenfalls auf 64 Bit-Worte. Dadurch entspricht ein Datenelement für den Prozessor genau einem Wort, so daß die Vergleichbarkeit der Rechenleistung mit der Speicherbandbreite vereinfacht wird.

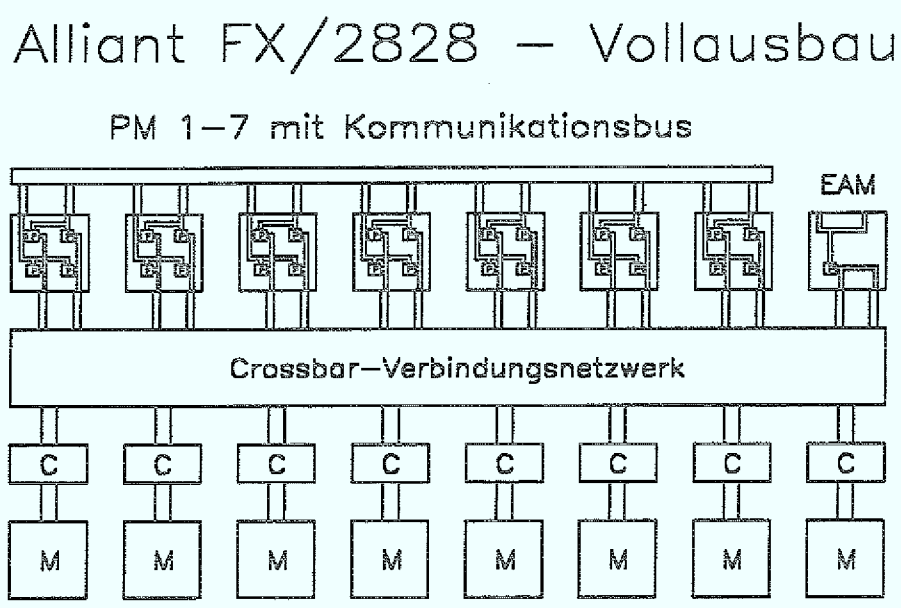


Abbildung 3.1: Die Alliant FX/2828 ist mit sieben Prozessormodulen Modell 400 mit jeweils vier Prozessoren (PM), einem Ein-/Ausgabemodul (EAM), acht globalen Cache-Speichern (C) und acht Speicherbänken (M) ausgestattet

3.1 Die ALLIANT FX/2800-Serie

Die Alliant FX/2800-Serie stellt nach der FX/8 und FX/80-Serie die dritte Generation von Multiprozessor-Vektorrechnern der Alliant Computer System Corporation dar. In der Maximalausstattung, dem Modell Alliant FX/2828, sind 29 Standardmikroprozessoren des Typs Intel i860 installiert, die auf einen bis zu 500 MWorte (4 GByte) großen gemeinsamen Hauptspeicher zugreifen. Von den 29 Prozessoren dienen 28 Prozessoren als Rechenelemente, und ein Prozessor wird ausschließlich für die Ein-/Ausgabe genutzt. Mit diesen 28 Rechenelementen erreicht die Alliant FX/2828 eine *peak performance* von 1120 MFLOPS bei doppelter Genauigkeit (IEEE 754, 64 Bit) und 2240 MFLOPS bei einfacher Genauigkeit (IEEE 754, 32 Bit). Jede Alliant FX/2800 besitzt acht Steckplätze für Prozessor- oder Ein-/Ausgabemodule, von denen mindestens einer mit einem Ein-/Ausgabemodul belegt werden muß. Die maximal sieben Steckplätze für Prozessormodule können für rechenintensive Anwendungen mit Prozessormodulen Modell 400 und für speicherintensive Anwendungen mit Prozessormodulen Modell 200 ausgestattet werden.

Ein Prozessormodul Modell 400 ist mit vier Prozessoren bestückt, die über zwei gemeinsame Speicherzugänge verfügen. Im Gegensatz dazu enthält ein Prozessormodul Modell 200 nur zwei Prozessoren, die jeweils einen eigenen Speicherzugang

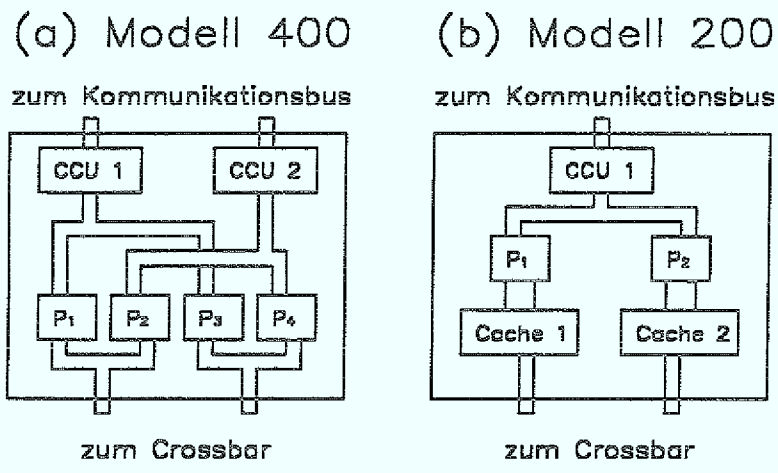


Abbildung 3.2: (a) Prozessormodul Modell 400 mit vier Prozessoren (P), zwei Speicherzugängen und zwei Zugängen zum Kommunikationsbus (CCU)
(b) Prozessormodul Modell 200 mit zwei Prozessoren, (P), zwei zusätzlichen Cache-Speichern, zwei Speicherzugängen und einer CCU

und einen zusätzlichen Cache-Speicher von 32 KWorten enthalten (siehe Abbildung 3.2). Die Konsistenz dieser beiden Cache-Speicher wird durch die Hardware sichergestellt. Ein Ein-/Ausgabemodul besitzt einen Prozessor und einen VME-Bus-Zugang, über den Ein-/Ausgabe-Operationen mit einer Bandbreite von 3.125 MWorte/s (25 Mbyte/s, übertragen nicht in 64-Bit, sondern in 32 Bit-Worten) durchgeführt werden können. Die hier genannten Daten sind [All91] entnommen.

Die Alliant FX/2800-Serie zeichnet sich insbesondere durch ihre mehrstufige Speicherhierarchie aus (siehe Abbildung 3.1). Die i860-Prozessoren besitzen auf dem Chip einen Instruktions-Cache-Speicher mit einer Kapazität von 0.5 KWorten und einen Daten-Cache-Speicher mit einer Kapazität von 1 KWorten, der nach der *write-in*-Strategie aktualisiert wird. Die Konsistenz der Cache-Speicher verschiedener Prozessoren wird jedoch nicht von der Hardware sichergestellt. Die Prozessoren sind über einen der Speicherzugänge des Prozessormoduls mit einem Crossbar-Verbindungsnetzwerk verbunden, das die Kommunikation mit den acht globalen Cache-Speichern ermöglicht. Die globalen Cache-Speicher sind mit privaten Speicherbänken verbunden und enthalten deshalb nur Daten aus disjunkten Speicherbereichen, so daß ihre Konsistenz sichergestellt ist. Zur Unterstützung der Parallelverarbeitung besitzt die Alliant FX/2800-Serie einen Kommunikationsbus für die Prozessorkommunikation, der von den Prozessoren über die sogenannte *concurrency control unit* (CCU) angesprochen wird. Je zwei Prozessoren müssen eine CCU gemeinsam nutzen und können deshalb keine unterschiedlichen parallelen Programme bearbeiten. Die FX/2800-Serie bildet die Basis der von Alliant

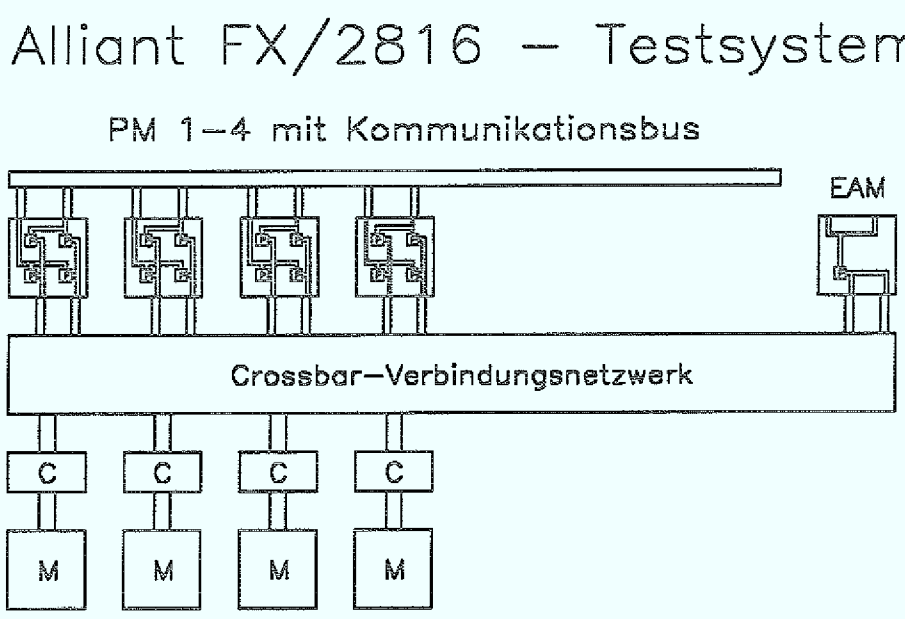


Abbildung 3.3: Die Alliant FX/2816 ist mit vier Prozessormodulen Modell 400 mit jeweils vier Prozessoren (`PM`), einem Ein-/Ausgabemodul (`EAM`), vier globalen Cache-Speichern (`C`) und vier Speicherbänken (`M`) ausgestattet

angekündigten massiv-parallelen CAMPUS/800 MPP mit bis zu 800 Prozessoren. Dieses Computersystem besteht aus 32 *clustern* mit jeweils 24 i860-Prozessoren auf einem gemeinsamen Hauptspeicher. Die *cluster* bilden ein Computersystem mit verteiltem Hauptspeicher und kommunizieren miteinander über ein Verbindungsnetzwerk.

Die Alliant FX/2800-Serie verwendet als Betriebssystem Concentrix-FX/2800 Version 2.1, welches ein Derivat des BSD-UNIX 4.3 ist. Das Betriebssystem unterstützt sowohl übliche UNIX-Prozesse als auch *threads*. Die Ausführung von *threads* ist in der aktuellen Version des Betriebssystems, im Gegensatz zu der Ausführung von UNIX-Prozessen, noch nicht optimiert. Bereitgestellt werden der Ada-Compiler FX/Ada-2800 sowie die automatisch vektorisierenden und parallelisierenden Compiler FX/Fortran-2800 und FX/C-2800. Der Fortran-Compiler FX/2800 bietet eine Unterstützung von DOACROSS-Schleifen und generiert die dafür benötigten Synchronisationsanweisungen automatisch. Der Compiler nutzt den nicht kohärenten *on-chip*-Cache-Speicher bei der Parallelverarbeitung als Vektorregister. Die Vektorisierung besteht beim FX/Fortran-2800 in der Nutzung des *dual instruction mode* und der *pipelined arithmetic instructions* [Int90].

Als Testkonfiguration stand eine Alliant FX/2816 zur Verfügung, die mit vier Prozessormodulen des Typs Modell 400, also 16 Prozessoren, vier globalen Cache-Speichern

(Gesamtkapazität: 256 KWorte) und vier Hauptspeicherbänken (Gesamtkapazität: 16 MWorte) ausgerüstet ist. Das System ist mit einem Ein-/Ausgabe-Modul ausgestattet (siehe Abbildung 3.3). Die Tabelle 3.3 gibt die maximalen Bandbreiten wieder, mit der Daten zwischen den verschiedenen Stufen der Speicherhierarchie der FX/2816 ausgetauscht werden können.

Stufe	Bandbreite [MWorte/s]	Anzahl	Produkt [MWorte/s]	Performance/Bandbreite [MFLOPS / (MWorte/s)]
i860-Cache	80	16	1280	0.5
i860-Port	20	16	320	2
Crossbar-Port	10	8	80	8
Global-Cache	20	4	80	8
Hauptspeicher	7	4	27	24

Tabelle 3.3: Bandbreiten auf den verschiedenen Stufen der Speicherhierarchie der FX/2816 nach [Int90, All91]. Die Spalte Performance/Bandbreite gibt an wieviele Rechenoperationen während eines Speichertransfers durchgeführt werden müssen, um die *peak performance* zu erreichen

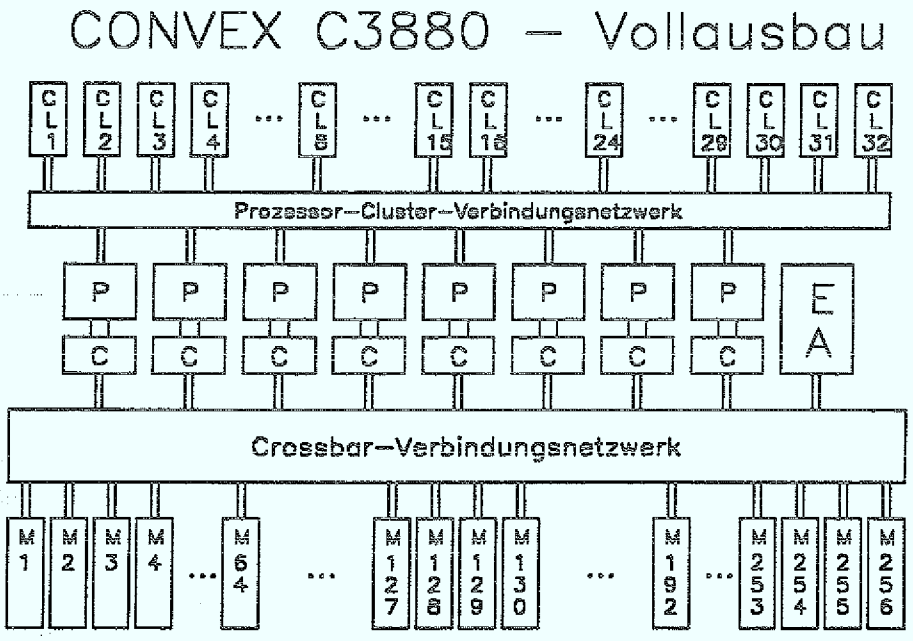


Abbildung 3.4: Die CONVEX C3880 mit acht Prozessoren (P), acht Cache-Speichern(C), 256 Speicherbänken (M) und 32 gemeinsamen Registersätzen (CL)

3.2 Die CONVEX C3-Serie

Die CONVEX Computer Corporation bietet mit der C3-Serie nach der C1- und der C2-Serie ebenso wie Alliant ihre dritte Generation von Multiprozessor-Vektorrechner an. Im Gegensatz zu Alliant entwickelte CONVEX zunächst Uniprozessor-Vektorrechner, die als Minisupercomputer vermarktet wurde. Die Leistung des Systems wurde später durch die Verwendung mehrerer Prozessoren gesteigert. Während die CONVEX Computersysteme bisher mehr im Bereich der Minisupercomputer lagen, erreicht das Spitzenmodell C3880 (siehe Abbildung 3.4) sowohl bei den Rechenoperationen als auch bei der Speicherbandbreite eine Leistung, die dem Bereich der Supercomputer zugeordnet werden kann. Die Systeme der C3800-Serie können mit bis zu acht Vektorprozessoren ausgestattet werden, mit denen sie eine Spitzenleistung von 960 MFLOPS bei doppelter Genauigkeit (IEEE 754, 64 Bit) und 1920 MFLOPS bei einfacher Genauigkeit (IEEE 754, 32 Bit) erreichen können. Neben den C3800 Modellen umfaßt die C3-Serie auch noch die weniger leistungsstarken Modelle C3200 und C3400. Die Vektorverarbeitung erfolgt registerorientiert unter Verwendung von acht Vektorregistern, die je 128 Elemente aufnehmen können. Die einzelnen Prozessoren mit einer *peak performance* von 120 MFLOPS können ihre Vektorregister mit einer Speicherbandbreite von 60 MWorten/s mit dem Hauptspeicher austauschen. Die acht Prozessorzugänge sind über ein Crossbar-Verbindungsnetzwerk mit den bis zu 256 Bänken des Hauptspeichers verbunden, die zusammen eine Bandbreite von 1100 MWorten/s und eine maximale Kapazität von 500 MWorten (4 GByte) zur Verfügung stellen. Die Prozessoren verfügen sowohl über einen Instruktions-Cache (2 KWorte) als auch über einen Daten-Cache (2 KWorte), der zur Speicheraktualisierung die *write-through*-Strategie verwendet. Der Daten-Cache-Speicher wird in der C3-Serie nur zur Speicherung von Skalaren und nicht für Vektoren genutzt, seine Konsistenz wird von der Hardware sichergestellt. Alle Modelle der C3-Serie unterstützen eine virtuelle Speicherverwaltung nach dem Verfahren *demand paging*. Die Prozessorkommunikation wird in den C3800 Modellen durch 32 Sätze mit jeweils 128 gemeinsamen Registern unterstützt, so daß im Multiprogramming-Betrieb mit bis zu 32 parallelen Programmen das Sichern des gemeinsamen Registersatzes beim *task*-Wechsel entfallen kann.

Das Betriebssystem der CONVEX C3-Serie ist CONVEX OS in der Version 10.0, das ebenso wie CONCENTRIX von Alliant auf dem BSD-UNIX 4.3 aufbaut und den POSIX Standard 1003.1 erfüllt. Das Betriebssystem CONVEX OS wird durch einen in der Hardware implementierten Thread-Scheduler für parallele Programme unterstützt. Dieser Hardware-Thread-Scheduler setzt leerlaufende Prozessoren automatisch für die Parallelverarbeitung ein und wird ASAP (*automatic self allocating processors / as soon as possible*) genannt. Für die mit der C2-Serie Software-kompatible C3-Serie existieren automatisch vektorisierende und parallelisierende Compiler für FORTRAN, C und Ada. Der FORTRAN-Compiler CONVEX FORTRAN unterstützt auch eine Generierung von Synchronisationsanweisungen für DOACROSS-Schleifen [CON90].

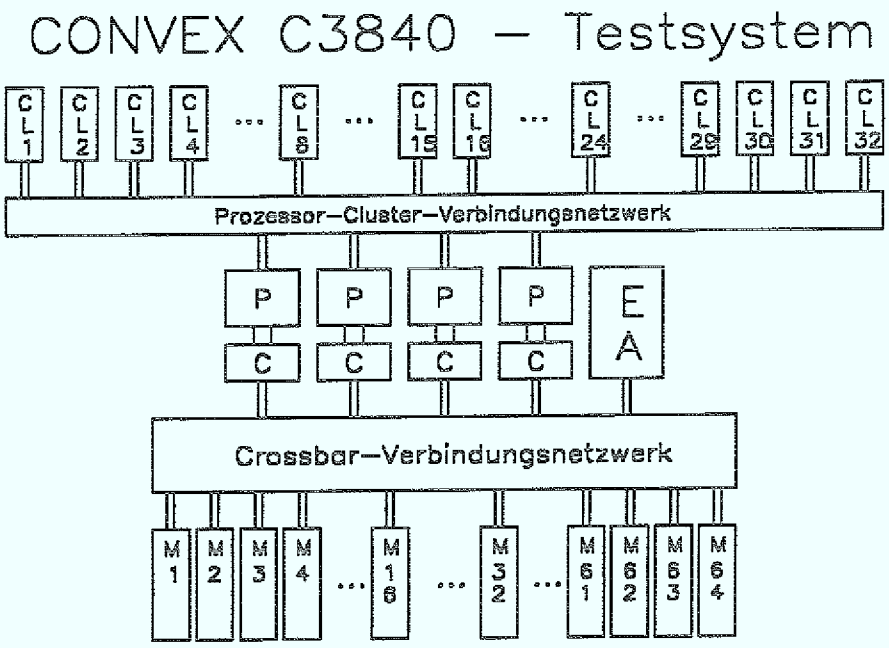


Abbildung 3.5: Die CONVEX C3840 mit vier Prozessoren (P), vier Cache-Speichern(C), 64 Speicherbänken (M) und 32 gemeinsamen Registersätzen (CL)

Das Testsystem ist eine CONVEX C3840, die mit vier Prozessoren ausgestattet ist und demnach eine *peak performance* von 480 MFLOPS besitzt. Der Hauptspeicher besitzt eine Kapazität von 128 MWorten und besteht aus 64 Speicherbänken. Da CONVEX eine maximale Speicherzykluszeit von 233.4 ns angibt, läßt sich eine Speicherbandbreite von 270 MWorten/s für das Testsystem berechnen.

3.3 Die CRAY X-MP/Y-MP Rechnerfamilien

Die Firma CRAY Research Inc. ist bekannt für hochleistungsfähige Vektorsupercomputer. Mit der CRAY X-MP-Familie wurde 1983 der erste Multiprozessor-Vektorrechner von CRAY vorgestellt, der zunächst mit zwei (X-MP/2x) und dann 1985 mit vier Prozessoren (X-MP/4x) ausgeliefert wurde. Die CRAY Y-MP-Familie ist seit 1988 der Nachfolger der CRAY X-MP-Serie und kann mit bis zu acht Prozessoren ausgestattet werden. Die Verarbeitung von Vektoren erfolgt registerorientiert mit Hilfe von acht Vektorregistern, die jeweils 64 Elemente aufnehmen können. Die Vektorprozessoren dieser Rechner-Familien besitzen jeweils zwei Speicherzugänge zum Laden von Daten, einen zum Speichern von Daten und einen Speicherzugang, der ausschließlich für Ein-/Ausgabe-Operationen genutzt wird. Die vier Speicherzugänge der einzelnen Vektorprozessoren sind über ein mehrstufiges Verbindungsnetzwerk mit einer hohen Zahl (16-256) von sehr schnellen Speicherbänken

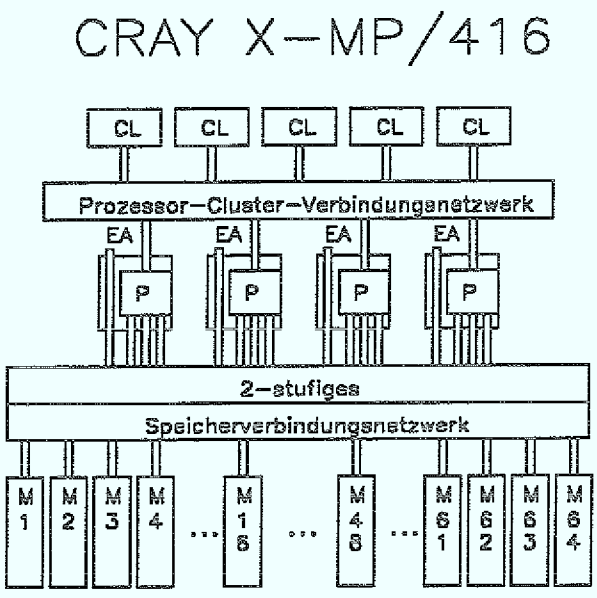


Abbildung 3.6: Die CRAY X-MP/416 besitzt vier Prozessoren (P) mit je drei Speicherzugängen zu den Vektorregistern und einem Speicherzugang für die Ein-/Ausgabe, 64 Speicherbänke (M) und fünf gemeinsamen Registersätze (CL)

(Zugriffszeit: 34 ns bzw. 30 ns) verbunden. Die Systeme verwenden bis auf Instruktionspuffer mit einer Kapazität von 128 Worten anstelle von Cache-Speicher zusätzliche 64 Adress- und 64 Datenregister (B- und T-Register) in denen Skalare zwischengespeichert werden können. Die Speicherverwaltung wird durch Segmentregister für Daten- und Code-Segment unterstützt; die CRAY-Rechner besitzen jedoch keine virtuelle Speicherverwaltung. Es besteht jedoch die Möglichkeit, die Systeme mit einem großen zusätzlichen Halbleiterspeicher (SSD) auszurüsten und somit zwischen dem Hauptspeicher und dem Sekundärspeicher eine weitere Stufe in die Speicherhierarchie einzufügen. Dieser Speicher kommuniziert über zwei programmierbare Kanäle mit einer Bandbreite von jeweils ca. 155 MWorten/s (1250 MByte/s in 128 Bit-Wörten) direkt mit dem Hauptspeicher. Die Prozessorkommunikation wird durch gemeinsame Registersätze (*cluster*) unterstützt, die jeweils 32 Semaphoren 16 gemeinsame Register enthalten. Die 16 gemeinsamen Register entsprechen den jeweils acht Adress- und Skalarregistern in den Prozessoren.

Mit dem Betriebssystem UNICOS steht für die CRAY-Supercomputer ein UNIX-kompatibles Betriebssystem zur Verfügung, das das frühere Betriebssystem COS abgelößt hat. Es unterstützt seit der Version 6.0 eine dynamische Vergabe der Prozessoren an parallele Programme mit einem „cooperative parallel interface" (CPI) genannten Verfahren. Mit dem Konzept des Autotasking und dem Übersetzungssystem CF77 stellt CRAY einen ausgereiften automatisch vektorisierenden und paral-

lelisierenden FORTRAN-Compiler zur Verfügung.

3.3.1 Die CRAY X-MP/416

Die CRAY X-MP/416 besitzt vier Vektorprozessoren mit einer Taktperiode von 8.5 ns (ältere Modelle: 9.5 ns) und einer *peak performance* von 940 MFLOPS (235 MFLOPS pro Vektorprozessor bei 64-Bit Gleitkommazahlen), die auf einen gemeinsamen Hauptspeicher mit einer Kapazität von 16 MWorten zugreifen können (siehe Abbildung 3.6). Jeder der vier Prozessoren besitzt für den Austausch der Register mit dem Speicher drei *ports* mit einer Bandbreite von jeweils 117 MWorten/s. Zur vollen Nutzung dieser zwölf *ports* wird eine Speicherbandbreite von 1411 MWorten/s benötigt. Bei einer gleichzeitigen Nutzung der vier zusätzlichen Ein-/Ausgabe-*ports* erhöht sich dieser Wert auf 1882 MWorte/s. Diese Speicherbandbreite ist die auch die maximale Speicherbandbreite, die von den 64 Speicherbänken über ein zweistufiges Speicherverbindungsnetzwerk geleistet werden kann. Das Speichersystem stellt also genau die Bandbreite zur Verfügung, die benötigt wird, um alle 16 Speicherzugänge der Prozessoren gleichzeitig zu nutzen. Der optionale SSD besitzt bei dem Testsystem X-MP/416 eine Kapazität von 32 MWorten. Zur Prozessorkommunikation steht den vier Prozessoren der X-MP/416 mit fünf *clustern* ein *cluster* mehr zur Verfügung als Prozessoren vorhanden sind [Cra86].

3.3.2 Die CRAY Y-MP8/832

Die CRAY Y-MP8/832 besitzt acht Vektorprozessoren mit einer gegenüber der X-MP/416 reduzierten Taktperiode von 6 ns. Dadurch wird eine *peak performance* von 2666 GFLOPS (333 MFLOPS pro Vektorprozessor) erreicht. Die acht Vektorprozessoren sind über ein dreistufiges Speicherverbindungsnetzwerk mit 256 Speicherbänken verbunden, die zusammen eine Kapazität von 32 MWorten besitzen (siehe Abbildung 3.7). Im Verhältnis zu der Bandbreite der jeweils vier Speicherzugänge (166 MWorte/s) der acht Prozessoren mit einer Gesamtbandbreite von 5333MWorten/s ist die vom Hauptspeicher bereitgestellte Bandbreite von 8533 MWorten/s (33 MWorte/s pro Speicherbank) überproportional angestiegen. Der SSD gehört bei der Y-MP zur Standardausstattung und besitzt in dem Testsystem eine Kapazität von 128 MWorten. Für die Prozessorkommunikation stehen analog zur X-MP für acht Prozessoren neun *cluster* zur Verfügung.

3.4 Vergleichende Darstellung

Die wichtigsten Kenngrößen der in dieser Arbeit analysierten Computersysteme werden in der Tabelle 3.4 noch einmal wiedergegeben. Die CRAY Y-MP ist sowohl in

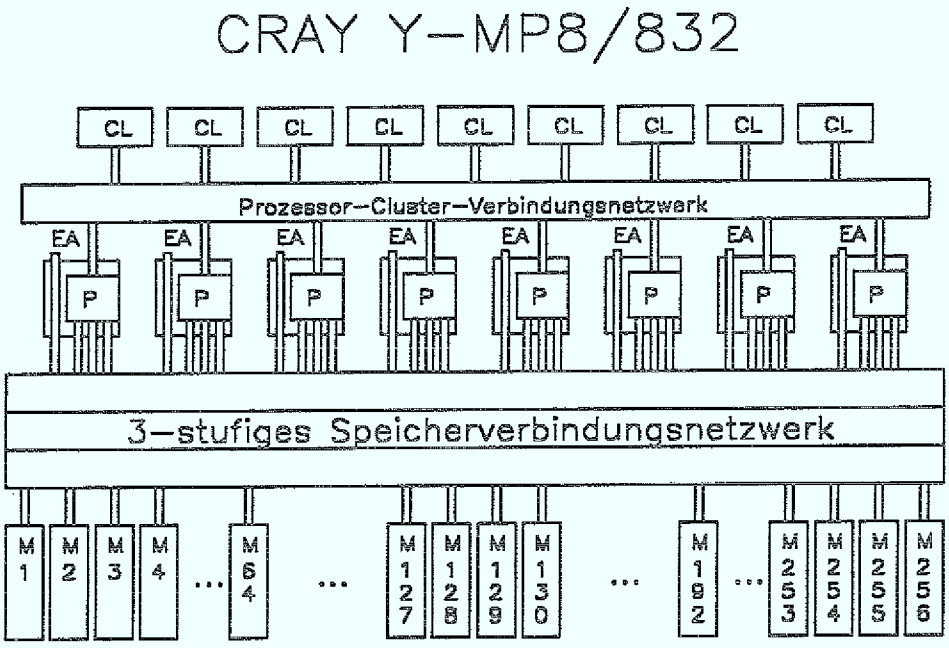


Abbildung 3.7: Die CRAY Y-MP8/832 besitzt acht Prozessoren (P) mit je drei Speicherzugängen zu den Vektorregistern und einem Speicherzugang für die Ein-/Ausgabe, 256 Speicherbänke (M) und neun gemeinsame Registersätze (CL)

Bezug auf die Rechenleistung als auch in Bezug auf die maximale Bandbreite des gemeinsamen Speichers erheblich leistungsfähiger als die übrigen Computersysteme. In Bezug auf die Rechenleistung sind die Systeme Alliant FX/2816, CONVEX C3840 und CRAY X-MP/416 vergleichbar, während die CRAY Y-MP sowohl in der Rechenleistung als auch der Bandbreite des Speichersystems erheblich leistungsfähiger ist. Die Testsysteme unterscheiden sich deutlich in der Balancierung von Rechenleistung und Speicherleistung. Die Hauptspeicherleistung der Alliant FX/2816 von 27 MWorten/s ist im Vergleich zu der Leistung der übrigen Systeme sehr gering. Der Hauptspeicher wird zwar durch einen extrem großen globalen Cache-Speicher unterstützt, dessen Leistung von 80 MWorten/s jedoch auch geringer als die Hauptspeicherleistung der übrigen Testsysteme ist. Das Verhältnis von Rechenleistung zur maximalen Speicherbandbreite beträgt 8 zu 1 bei der Alliant FX/2816 und 2 zu 1 bei der CONVEX C3840. In beiden Fällen müssen also mehr Rechenoperationen als Speicheroperationen durchgeführt werden, um die *peak performance* zu erreichen. Im Gegensatz dazu beträgt das Verhältnis 2 zu 3 bei den CRAY-Rechner, daß heißt es können mehr Speicheroperationen als Rechenoperationen durchgeführt werden. Während bei der CONVEX C3840 und der CRAY X-MP die Leistung des Hauptspeichers der maximalen Leistung der Prozessor-*ports* entspricht, ist die Hauptspeicherleistung der CRAY Y-MP ungefähr doppelt so hoch wie die Leistung der Prozessor-*ports*.

Computersystem	FX/2816	C3840	X-MP/416	Y-MP/832
Prozessoren	16	4	4	8
peak performance [MFLOPS]	640	480	940	2666
Speicherbandbreite [MWorte/s]	27/80	270	1882	8533
Globaler Cache-Speicher [MWorte]	0.256	-	-	-
Hauptspeicher [MWorte]	16	128	16	32
SSD [MWorte]	-	-	32	128
Gemeinsame Registersätze	-	32	5	9
Virtuelle Speicherverwaltung	ja	ja	nein	nein
Dynamisches Scheduling	nein	ja	ja	ja

Tabelle 3.4: Vergleichende Übersicht der Kenngrößen der zur Leistungsmessung verfügbaren Computersysteme

Die Alliant FX/2816 und die CONVEX C3840 bieten ein virtuelles Hauptspeicherkonzept, während die beiden CRAY-Systeme ein reales Hauptspeicherkonzept besitzen. Dafür besitzen die beiden CRAY Rechner einen zusätzlichen schnellen Halbleiterspeicher, den SSD. Die Hauptspeicherkapazität der CONVEX C3840 ist erheblich größer als die der übrigen Testsysteme und entspricht von der Kapazität her dem SSD der CRAY Y-MP. Der Hauptspeicher liegt jedoch im Gegensatz zum SSD im direkten Zugriff der Prozessoren. Die Hauptspeicherkapazität der Alliant FX/2816 war bei der Testkonfiguration erheblich geringer als die der CONVEX, obwohl beide Systeme prizipiell mit der gleichen Hauptspeicherkapazität ausgestattet werden können.

Die Ausführung von parallelen Programmen wird von den Testsystemen in unterschiedlichen Maß unterstützt. Die Alliant FX/2816 wird in statische *cluster* konfiguriert, die dann zur Parallelverarbeitung von Programmen genutzt werden können. Die CONVEX C3840 unterstützt eine dynamische Zuteilung der Prozessoren mit Hilfe von ASAP, einem Hardware-Scheduler. Im Gegensatz dazu ist die Scheduling-Strategie auf den CRAY-Systemen ausschließlich durch Software realisiert. Während das Betriebssystem UNICOS vor der Version 6.0 ausschließlich ein statisches Scheduling unterstützt, bietet die Version 6.0 zusätzlich ein dynamisches Scheduling. Die Kommunikation der parallelen Programme wird auf der Alliant FX/2816 durch einen gemeinsamen Bus unterstützt. Auf den CONVEX und den CRAY-Systemen werden gemeinsame Registersätze zur Kommunikation eingesetzt. Dabei bietet die CRAY X-MP/416 fünf, die CRAY Y-MP8/832 neun und die CONVEX C3840 32 gemeinsame Registersätze bzw. *cluster*. Die Parallelverarbeitung wird also von allen Testsystemen durch zusätzlich *hardware* unterstützt.

Kapitel 4

Meßmöglichkeiten des PAR-Bench-Systems

Das PAR-Bench-System wurde entwickelt, um reproduzierbare Leistungsmessungen von Multiprozessor-Vektorrechnern im Mehrprogrammbetrieb durchzuführen und graphisch auszuwerten [Lin90]. Es ermöglicht insbesondere eine flexible Leistungsmessung von Arbeitslasten, die parallele Programme enthalten. Dazu werden im PAR-Bench-System im Gegensatz zu dem zur Zeit bevorzugten Ansatz keine realen Anwendungen bzw. leichter portierbare modifizierte Anwendungen (SPECmark [Uni89], PERFECT [BCK+89]), sondern synthetische Benchmark-Programme verwendet. Dies erlaubt eine erheblich flexiblere Parametrisierung der zu untersuchenden Arbeitslast und vereinfacht somit die Analyse des Systemverhaltens, da die einzelnen Parameter der Arbeitslast in bestimmten Grenzen unabhängig voneinander verändert werden können. Die Arbeitslast wird dabei durch eine Reihe von Parametern charakterisiert. Diese Parameter können auf der CRAY X-MP/Y-MP Supercomputer-Familie mit Hilfe des Hardware-Performance-Monitors für bestehende, reale Arbeitslasten ermittelt werden, die dann mit Hilfe des PAR-Bench-Systems durch eine synthetische Arbeitslast modelliert werden können. Mit Hilfe des PAR-Bench-Systems ist es möglich, eine reale Arbeitslast mit einem erheblich geringerem Verbrauch an Prozessorzeit zu simulieren und trotzdem alle charakteristischen Eigenschaften Eigenschaften beizubehalten. Somit können Leistungsmessungen schneller und effizienter durchgeführt werden.

Zur Leistungsmessung wird die synthetische Arbeitslast des PAR-Bench-Systems dann auf einem dedizierten Computersystem ausgeführt. Da das PAR-Bench-System, in FORTRAN implementiert wurde, ist eine Portierung auf andere Computersysteme in den meisten Fällen erheblich einfacher als die Portierung einer realen Arbeitslast. Die einmalige Portierung von PAR-Bench auf ein Computersystem ermöglicht es, mit diesem System beliebige Arbeitslasten zu modellieren und zu messen. Der Job-Output der synthetischen Arbeitslast kann mit Hilfe des graphischen Auswertungswerkzeuges GRANSYS in Form von Benchmark-Plots ausgewertet wer-

den. Das PAR-Bench-System wurde von W. E. Nagel entwickelt [Nag90a, Nag90b] und im Rahmen einer Diplomarbeit von M. A. Linn um zusätzliche Parameter signifikant erweitert. [Lin90] enthält eine genaue Beschreibung der Funktionsweise und der Parameter von PAR-Bench.

4.1 Generierung der synthetischen Arbeitslast

Die synthetische Arbeitslast des PAR-Bench-Systems besteht aus einer Reihe von Jobs, die ein gemeinsames Programm mit eventuell unterschiedlichen Parametern ausführen. Dieses Programm enthält eine Reihe von Kernen, die das Computersystem in unterschiedlichem Maße mit der Arithmetik bzw. dem Transfer von Vektoren zwischen Prozessoren, Hauptspeicher und der Ein-/Ausgabe belasten. Beim Generierungslauf werden die Jobs in dem durch eine Parameter-Datei festgelegten Zeitpunkt gestartet und modellieren die gewünschte Arbeitslast, in dem sie dynamisch den Kern zur Bearbeitung auswählen, durch den die gewünschten Endwerte der Parameter am besten approximiert werden. Die Ausführungsreihenfolge der Kerne wird dabei in der Job-Datei protokolliert und steht nach dem Generierungslauf für beliebige weitere Ausführungsläufe zur Verfügung. Die dynamisch approximierten Parameter des PAR-Bench-Systems sind:

- die erreichte Speichertransferleistung in MMREFS (million memory references per second), also in Millionen Speicherzugriffe pro Sekunde;
- die erreichte Rechenleistung in MFLOPS (million floating point operations per second), also in Millionen Gleitkommaoperationen pro Sekunde;
- die erreichte Ein-/Ausgabetransferleistung in MIOS (million array elements transferred to or from an I/O-Device per second), also der Anzahl der Feldelemente (Worte), die zwischen dem peripheren Speicher und dem Hauptspeicher ausgetauscht werden;
- Die von dem Job verbrauchte Prozessorzeit in Sekunden.

Diese Parameter der Programmkerne sind im Mehrprogrammbetrieb von der gesamten Arbeitslast abhängig, da sie durch die Konkurrenz der Jobs bzw. der Prozessoren um die Ressourcen wie z.B. die Hauptspeicherbandbreite beeinflußt werden. Die Auswahl der auszuführenden Programmkerne erfolgt deshalb dynamisch in Abhängigkeit von den bisher gemessenen Leistungswerten. Die Auswahl der Kerne erfolgt im Generierungslauf auf der CRAY Y-MP unter Nutzung des HPM (*hardware performance monitor*), wodurch eine hohe Meßgenauigkeit erzielt werden kann. Andere Parameter haben zwar einen Einfluß auf die genannten dynamischen Parameter, werden aber selbst nicht durch die Arbeitslast beeinflußt und können deshalb statisch voreingestellt werden. Das Par-Bench-System unterstützt die folgenden „statischen“ Parameter:

- Die Größe des vom Programm angeforderten Hauptspeicherplatzes in Worten. Auf Systemen mit virtueller Speicherverwaltung hat dieser Parameter keine große Bedeutung, da auf den zusätzlich angeforderten Speicherplatz nicht zugegriffen wird.
- Der *stride* mit dem die Vektoroperationen auf den Daten durchgeführt werden. Ein Stride größer als eins erhöht die Belastung auf das Speichersystem, indem einerseits die Anzahl der genutzten Speicherbänke reduziert und andererseits die Lokalität der Speicherzugriffe verringert wird.
- Die Priorität der Jobs. Niedrige Werte für die Priorität bedeuten hier eine bevorzugte Ausführung des Jobs.

Da für den Generierung- und den Ausführungslauf von PAR-Bench eine einheitliche Parameter-Datei verwendet wird, werden in der Parameter-Datei einige zusätzliche Parameter aufgeführt, die im Generierungslauf nicht berücksichtigt werden. Diese Parameter bestimmen die speziellen Eigenschaften von Jobs mit parallelen Programmen. Da der Generierungslauf immer sequentiell ausgeführt wird, werden Parameter, die die Parallelisierung betreffen, im Generierungslauf ignoriert. Die nur im Ausführungslauf berücksichtigten Parameter zur Parallelisierung von Jobs sind:

- Die Anzahl der angeforderten Prozessoren, also die maximale Anzahl der Prozessoren, die an der Ausführung der Jobs beteiligt werden sollen. Auf Computersystemen mit einer statischen Einteilung der Prozessoren in *cluster* wie die Alliant FX/2816 wird unabhängig von diesem Parameter jeweils ein ganzer *cluster* für das parallele Programm angefordert.
- Der Parallelisierungsgrad nach Amdahl, der das Verhältnis von paralleler Ausführungszeit zur sequentieller Ausführungszeit kennzeichnet und die mögliche Beschleunigung durch Parallelverarbeitung begrenzt (siehe Abschnitt 2.4.1).
- Die *load balance* gibt die Verteilung der zur Verfügung stehenden Arbeit auf die Prozessoren in Prozent an. Definiert wird der Wert hier durch das Verhältnis von der durchschnittlich nutzbaren Prozessorzahl zu der Zahl der angeforderten Prozessoren (siehe Abschnitt 2.4.1).

Die durch PAR-Bench zu modellierende Arbeitslast wird durch eine Parameter-Datei beschrieben, in der alle Parameter (statische, dynamische und Parallelisierungsparameter) enthalten sind. In dieser Arbeit wird nur ein Ausschnitt der Parameterdatei zur Interpretation benötigt, der im folgenden beschrieben wird.

Jede Zeile der Parameter-Datei steht für eine Anzahl von Jobs mit gleichen Parametern, die zum gleichen Zeitpunkt gestartet werden. Die erste Spalte (`#`) legt die Anzahl der Jobs fest, die mit den in der Zeile angegebenen Parametern zur Arbeitslast beitragen. Die zweite Spalte (`DEL`) bestimmt den Zeitraum in Sekunden, der nach dem Start der Arbeitslast verstreichen soll, bis die Jobs gestartet werden. In der Spalte 10 (`PRNAME`) steht der Programmname des Programms, das von den Jobs ausgeführt wird. Die in dieser Arbeit verwendeten Arbeitslasten bestehen ausschließlich aus mit Hilfe von PAR-Bench generierten synthetischen Benchmark-Jobs,

```
#,DEL,PR,SEC,  MM, FL, IO, SIZE, MODE ,PRNAME, ST,PG, LB, N
 8,00,10, 30, 050,200,000,  000,'EXE' ,'bench', 1,  0,  0,1
 8,01,10, 30, 010,120,000,  000,'EXE' ,'bench', 1,  0,  0,1
 8,02,10, 20, 040,120,000,  000,'EXE' ,'bench', 1, 70,100,8
 4,03,10, 20, 080,200,000,  000,'EXE' ,'bench', 1,100, 70,8
```

Tabelle 4.1: Eine Parameter-Datei für PAR-Bench.

die durch den Programmnamen 'bench' gekennzeichnet sind. Alternativ können jedoch auch artfremde Programme wie z.B. reale Anwendungen innerhalb einer synthetischen Arbeitslast gestartet werden. Die Spalte 9 (MODE) entscheidet, ob ein Generierungs- ('GEN') oder ein Ausführungslauf ('EXE') durchgeführt werden soll. Die Spalten 4-7 beschreiben die anzustrebenden dynamischen Parameter: Spalte 4 (SEC) die verbrauchte Prozessorzeit in Sekunden, Spalte 5 (MM) die Speichertransferleistung in MMREFS, Spalte 6 (FL) die Rechenleistung in MFLOPS und Spalte 7 (IO) die Ein-/Ausgabetransferleistung in MIOS. Die statischen Parameter sind in den Spalten 3, 8 und 11 aufgeführt: Spalte 3 (PR) enthält die Priorität, Spalte 8 legt den (SIZE) der Speicherbedarf in KWorten und Spalte 11 (ST) der Stride für die Vektoroperationen fest. Die Parameter für parallele Programme werden schließlich in den Spalten 12-14 angegeben: Spalte 14 (PG) der gewünschte Parallelisierungsgrad in Prozent, Spalte 15 (LB) die *load balance* in Prozent und Spalte 16 (N) die maximale Prozessorzahl.

4.2 Messung und Auswertung

Der Ausführungslauf kann im Gegensatz zum Generierungslauf auf allen Computersystemen ausgeführt werden, auf die PAR-Bench portiert wurde. Soll eine Messung nach einem Generierungslauf auf einem anderen Computersystem durchgeführt werden, so müssen zunächst die Job-Dateien, in denen die Kernfolgen protokolliert wurden, auf das zu messende System übertragen werden. Diese Dateien liegen als unformatierte FORTRAN-Dateien (binär codiert) vor, da der Aufwand, der durch das Lesen der Kernfolgen im Ausführungslauf entsteht, gering gehalten werden soll. Zur Übertragung werden die Job-Dateien in formatierte FORTRAN-Dateien transformiert, da die Codierung von unformatierten FORTRAN-Dateien auf verschiedenen Computersystemen nicht standardisiert ist. Die Dateien werden dann auf das zu messende System übertragen und dort wieder in unformatierte FORTRAN-Dateien zurücktransformiert.

Zusätzlich muß die Parameterdatei für einen Ausführungslauf modifiziert werden, indem in der Spalte 9 (MODE) jeweils ein 'EXE' als Kennzeichnung für einen Ausführungslauf eingetragen wird. Da die dynamischen Parameter Rechenleistung (FL), Speichertransferleistung (MM), Ein-/Ausgabe-Transferleistung (IO) und Pro-

zessorzeit (SEC) durch die im Generierungslauf erstellten Job-Dateien festgelegt werden, lassen sich diese im Ausführungslauf nicht mehr verändern. Alle übrigen Parameter sind variabel und können für einen Ausführungslauf neue Werte annehmen. Dazu gehören insbesondere die Parallelisierungsparameter, durch die der Einfluß von Parallel-Programmen auf die Leistung im Mehrprogramm-Betrieb untersucht werden kann. Die Parallelisierungsparameter werden im Ausführungslauf in Abhängigkeit von der Ausführungsreihenfolge der Programmkerne dynamisch approximiert.

Beim Ausführungslauf generiert jeder Job nach dem Prinzip des *event tracing* eine Ausgabe-Datei, die Daten über das Ausführungsverhalten des Jobs enthält. Ausgegeben werden z.B. die Zeitpunkte, an denen die Ausführung der Jobs beginnt und beendet wird, die verbrauchte Prozessor- und Systemzeit sowie die bei der Parallelverarbeitung erreichten Werte (Parallelisierungsgrad und *load balance*). Die in den Ausgabe-Dateien enthaltenen Informationen können entweder direkt oder mit Hilfe des graphischen Auswertungssystems GRANSYS [Lin90, NL91a] ausgewertet werden. Durch dieses Auswertungssystem werden zum einen Daten berechnet, die das globale Verhalten eines Benchmarks betreffen, d.h. das Verhalten der Arbeitslast als Ganzes, und zum anderen Benchmark-Plots generiert, in denen der zeitliche Ablauf bei der Bearbeitung der einzelnen Jobs sichtbar wird. Von GRANSYS werden die folgenden Meßwerte aus den Ausgabe-Dateien eines Benchmark-Laufes bestimmt und mit dem Benchmark-Plot ausgegeben.

- Die *Bench-time* soll die Bearbeitungsdauer der Arbeitslast wiedergeben. Sie ist die Differenz zwischen Start- und Endzeitpunkt der Bearbeitung der Arbeitslast. Der Startzeitpunkt wird auf den Eintrittszeitpunkt des ersten Jobs in das Computersystem normiert. Der Endzeitpunkt wird als Durchschnitt zwischen den Zeitpunkten, bis zu denen die Prozessoren des Computersystems ausgelastet sind, definiert. Bei sequentiellen Programmen entspricht dies den Endzeitpunkten der letzten N Jobs, wobei N der Anzahl der Prozessoren im Computersystem entspricht.
- Die *Total-time* ist die von der Arbeitslast verbrauchte Prozessorzeit. Sie ist das Produkt aus der *Bench-time* und der Anzahl der Prozessoren des Computersystem.
- Die *CPU-time* ist die Summe der von den Jobs der Arbeitslast verbrauchten Prozessorzeit.
- Die *System-time* ist die Summe der von den Jobs bei der Bearbeitung von Systemroutinen des Betriebssystems verbrauchten Prozessorzeit.
- Die *Idle-time* ist die Zeit, in der Prozessorzeit weder für die Bearbeitung von Benutzer- noch für die Bearbeitung von System-Code verwendet wird. Sie wird als Differenz der *Total-time* und der für Benutzer- oder System-Code verbrauchten Prozessorzeit (*CPU-time* + *System-time*) gebildet.
- Die *Wait-time* wird aus der Summe der Ausführungszeiten der Jobs abzüglich der bei der Bearbeitung von Benutzer- oder System-Code verbrauchten Prozessorzeit gebildet. Die Wait-time ermöglicht es, Aussagen über die Antwortzeit

des Computersystem bei der Bearbeitung der Jobs zu treffen.

- Unter der *Real-time* wird die Ausführungszeit eines Jobs verstanden, also die Differenz zwischen End- und Startzeitpunkt.

Der Bereich *Parallel-Jobs* gibt Informationen über die parallelen Programme der Arbeitslast wieder. Alternativ können in diesem Bereich auch Informationen über sequentielle Programme ausgegeben werden, die mit den parallelen Programmen aus anderen Messungen verglichen werden sollen. Die *CPU-time*, *System-time* und *Real-time* geben hier die Summen der entsprechenden Prozessor- bzw. Ausführungszeiten der parallelen Jobs wieder. Der unter *Service* angegebene *service*-Faktor setzt die Prozessorzeiten ins Verhältnis zur Ausführungszeit der Jobs. Der bei der System-time angegebene *service*-Faktor setzt die Summe aus CPU-time und System-time ins Verhältnis zur Ausführungszeit.

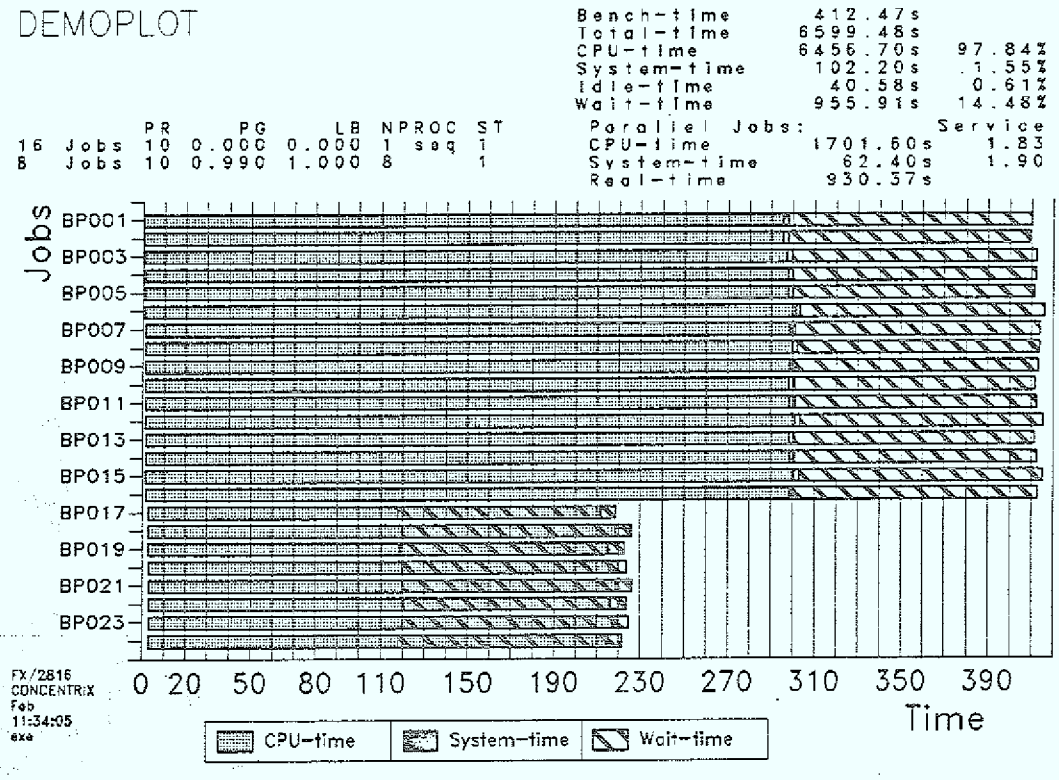


Abbildung 4.1: Beispiel für einen Benchmark-Plot.

Die Abbildung 4.1 zeigt einen mit Hilfe des GRANSYS-Systems generierten Benchmark-Plot. Jeder Balken repäsentiert einen Job der Arbeitslast, dessen Nummer BPxxx an der Y-Achse abgelesen werden kann. An der X-Achse ist die Realzeit seit dem Start des ersten Jobs der Arbeitslast angetragen. Bei der Interpretation der Balken müssen zwei zwischen zwei Arten von Balken unterschieden werden:

- Im einen Fall ist die Ausführungszeit eines Jobs höher als seine Prozessor-

zeit. Dies trifft für alle sequentiellen Jobs einer Arbeitslast zu. In diesem Fall gibt der linke Rand des Balkens den Startzeitpunkt und der rechte Rand den Endzeitpunkt des Jobs an. Das Muster innerhalb des Balkens gibt dann zusätzliche Information über die Anteile der CPU-time, der System-time und der Wait-time an der Ausführungsdauer des Jobs. Die Muster innerhalb der Balken geben jedoch keine Information darüber, zu welchem Zeitpunkt ein Job CPU-time oder System-time benötigt hat.

- Der zweite Fall tritt dann ein, wenn die Ausführungszeit eines parallelen Jobs geringer ist als sein Bedarf an Prozessorzeit. Diese Balken können daran erkannt werden, daß das Muster für die Wait-time das Muster für die CPU-time und die System-time überlagert. Die Länge des Balken gibt in diesem Fall die Summe aus CPU-time und System-time des Jobs an. In diesem Fall tritt eine negative Wait-time auf und das linke Ende des Musters für die Wait-time gibt den Endzeitpunkt des Jobs an.

Damit ist die Beschreibung des Systems PAR-Bench in seiner aktuellen Version abgeschloßen. Im folgenden werden die Probleme und Lösungen diskutiert, die bei der Portierung des Systems PAR-Bench entstanden.

4.3 Portierung des Systems PAR-Bench

Das PAR-Bench-System soll vergleichbare Leistungsmessungen von Multiprozessor-Vektorrechnern mit gemeinsamen Speicher im Mehrprogramm-Betrieb ermöglichen. Daher wurde das Ziel einer leichten Portierbarkeit des PAR-Bench-System bei der Entwicklung berücksichtigt. Da aber moderne Meßwerkzeuge wie *hardware*-Monitore und hochgenaue Echtzeituhren in den aktuellen Versionen der Betriebssysteme und FORTRAN-Compiler bisher nicht standardisiert sind, müssen die entsprechenden Routinen bei einer Portierung des System modifiziert werden. Der Aufruf von maschinenabhängigen Systemroutinen wurde jedoch in einige wenige Unterprogramme verkapselt. Das ursprüngliche Ziel, ein einziges Quellprogramm für alle zu messenden Computersysteme zu verwenden, wurde jedoch aufgegeben, da dies zu Schwierigkeiten mit den Bindern der eingesetzten Computersysteme führte. Stattdessen wird in der vorliegenden Version des PAR-Bench-Systems mit Hilfe eines Präprozessors aus den entsprechend angepaßten Unterprogrammen und den portablen Unterprogrammen das passende Quellprogramm zu einem Computersystem generiert. Dabei werden die Programmroutinen, die auf der CRAY X-MP bzw. Y-MP für den Generierungslauf eingesetzt werden durch Programme mit leerem Programmkörpern ersetzt. Diese Programmroutinen sind auf den Zielrechnern nicht erforderlich, da sie nur für den Generierungslauf benötigt werden, der auf einer CRAY X-MP bzw. Y-MP durchgeführt wird.

Bei der Anpassung der für den Ausführungslauf benötigten maschinenabhängi-

gen Programmteile wurde versucht, auf Standard-UNIX-Systemaufrufe zurückzugreifen, um eine Basis für die Portierung auf weitere UNIX-Systeme zu schaffen. Die Messung der verbrauchten Prozessor- und Systemzeit wurde mit dem UNIX-Systemaufruf *dtime* durchgeführt. Das Datum und die Uhrzeit konnten dem Standard konform über die Funktion *fdate* ermittelt werden. Die Auflösung der so erhaltenen Uhrzeit beträgt jedoch nur eine Sekunde und ist deshalb für genaue Zeitmessungen nicht geeignet. Eine genauere Angabe der Uhrzeit ist jedoch nicht mit Hilfe von UNIX-Systemaufrufen erreichbar, da solche in den verwendeten UNIX-Derivaten noch nicht standardisiert sind. Das Betriebssystem Convex OS stellt einen Systemaufruf (*gettimeofday*) zur Verfügung, mit dem die aktuelle Uhrzeit bis auf Mikrosekunden genau wiedergegeben wird, der aber nicht von dem FORTRAN-Compiler unterstützt wird. Solche Systemaufrufe werden von einem C-Programm (*sysif.c: System-Interface*) aus angesprochen, das wiederum von dem FORTRAN-Programm aufgerufen wird. Auf dieselbe Art kann darüber hinaus noch ein Systemaufruf (*getrusage*) zur Bestimmung von Informationen bezüglich *paging*, *swapping* und *task*-Wechsel aufgerufen und genutzt werden.

Auf der Alliant FX/2816 steht unter CONCENTRIX kein Systemaufruf zur Verfügung, der die aktuelle Uhrzeit mit einer höheren Genauigkeit als eine Sekunde wiedergibt. Daher wurde zusätzlich der Zähler der Echtzeituhr mit einer zeitlichen Auflösung von 10 μ-Sekunden durch einen Systemaufruf ausgelesen und protokolliert. Das Auswertungsprogramm GRANSYS mußte für die Verwendung dieser Information modifiziert werden. Der Aufruf der Echtzeituhr ist jedoch in den verwendeten UNIX-Derivaten nicht standardisiert, so daß auf einen maschinenabhängigen Systemaufruf (*hrcget*) zurückgegriffen werden mußte. Bei einer Portierung auf ein weiteres UNIX-System mit ähnlichen Voraussetzungen muß dieser Systemaufruf durch den systemspezifischen Systemaufruf des neuen Systems ersetzt werden.

Auch in den CRAY-Routinen wurden Modifikationen erforderlich, da mit der neuen UNICOS-Version 6.0 ein von PAR-Bench benutzter Systemaufruf nicht mehr zur Verfügung steht. Zur Ermittlung des von dem Programm benötigten Speicherplatzes steht der vorher genutzte Systemaufruf (*memory*) nicht mehr zur Verfügung. Der dazu in Vorgänger-Versionen verfügbare Systemaufruf konnte jedoch als C-Programm unter Nutzung des Systemaufrufs (*sbreak*) emuliert und in das PAR-Bench-System eingebunden werden. Eine Ermittlung des aktuell belegten Speicherplatzes wird von keinem der FORTRAN-Compiler bzw. der UNIX-Derivate unterstützt. Eine Anforderung von zusätzlichem Speicherplatz ist im Prinzip möglich, wird jedoch aufgrund der statischen Speicherverwaltung von FORTRAN 77 nicht von den FORTRAN-Compilern bereitgestellt. Da die Computersysteme CONVEX C3840 und Alliant FX/2816 mit einem virtuellem Speicher ausgestattet sind, ist die Anforderung von zusätzlichem, nicht genutztem Speicher wenig sinnvoll und daher nicht erforderlich.

Eine weitere Anpassung erforderten die zur Parallelisierung verwendeten Compiler-

Direktiven. Diese werden benötigt, um die Parallelisierung der Kerne so zu beeinflußen, daß eine Parametrisierung des Parallelisierungsgrades und der *load balance* ermöglicht wird. Sie erlauben es weiterhin die Parallelisierung auf den Computersystemen so zu beeinflußen, daß die von den Compilern generierten Programme einen vergleichbaren Parallelisierungsgrad haben. Die Compiler-Direktiven werden in Form von Kommentaren in das Quellprogramm eingebracht, so daß sie von den Compilern anderer Hersteller ignoriert werden und somit im weiteren Sinne portabel sind. Somit ist für die parallelisierbaren Kerne nur ein Quellprogramm erforderlich, das mit den unterschiedlichen Compiler-Direktiven für alle FORTRAN-Compiler ausgestattet ist, auf die das PAR-Bench-System portiert wurde. Jeder der Hersteller benutzt herstellerspezifische und damit nicht standardisierte Compiler-Direktiven.

Die verschiedenen Compiler der zu messenden Computersysteme führen unterschiedliche Optimierungen durch und besitzen nicht für jede optimierende Transformation eine Compiler-Direktive, über die Einfluß auf die Optimierung genommen werden kann. So wurden sowohl vom Alliant FORTRAN-Compiler als auch von dem von CONVEX mit Hilfe eines *strip mining* über mehrfach verschachtelte Schleifen der größte Teil der Speicherzugriffe bei der Optimierung entfernt. Da der Alliant FORTRAN-Compiler das *strip mining* über mehrere verschachtelte Schleifen nur bei der Übersetzung des Programms für die parallele Ausführung durchführt, ergaben sich bei der Messung der Effizienz bei der Parallelisierung keine sinnvollen Ergebnisse (siehe Abbildung 4.2, Effizienz A). Da sich das Verhalten des Code-Generators und des Optimierers in der aktuellen Version des Compilers nicht hinreichend beeinflussen läßt, um diese Effekte zu vermeiden, wurden die Kerne so modifiziert, daß eine sinnvolle Übersetzung des Codes auf allen Computersystemen erreicht wurde (siehe Abbildung 4.2, Effizienz B). Die für die neuen Kerne gemessene Effizienz überschreitet nicht mehr 100%, erscheint jedoch sehr gering. Dies läßt sich durch die verhältnismäßig geringe Speicherbandbreite des globalen Cache-Speicher und des Hauptspeicher erklären, da die Kerne mit einer hohen Hauptspeicherbelastung die geringste Effizienz bei der Parallelisierung erreichen. Um ungewünschte Effekte durch die Optimierungen der Compiler auszuschließen, wurde der von den Compilern generierte Maschinen-Code untersucht und so die Belastung des Hauptspeichers durch die Kerne verifiziert. Der für die Alliant modifizierte Code lief auch auf der CONVEX wie gewünscht, so daß keine neuen Modifikationen nötig wurden.

Ein weiteres Problem trat bei den Grenzen (*limits*) auf, die den Benutzern durch die Konfiguration des Betriebssystem vorgegeben sind. Bei den Messungen auf der Alliant konnte das NQS nicht genutzt werden, da für jeden Benutzer nur zehn Jobs gleichzeitig ausgeführt werden. Daher wurde der Benchmark-Generator so modifiziert, daß die Jobs auf der Alliant nicht mittels *qsub* ans NQS geschickt, sondern mittels & als Hintergrundjob gestartet wurden. Auf der CONVEX war dies auf der Testmachine ebenfalls nachteilig, so daß die Jobs auf der CONVEX ebenfalls mittels & in den Hintergrund geschickt werden. Auf der CRAY X-MP bzw. Y-MP führte dies aufgrund des realen Speicherkonzepts zu Problemen. Informationen, die das NQS auf der CRAY mit den Ausgaben der Jobs lieferte, stehen bei einem Hinter-

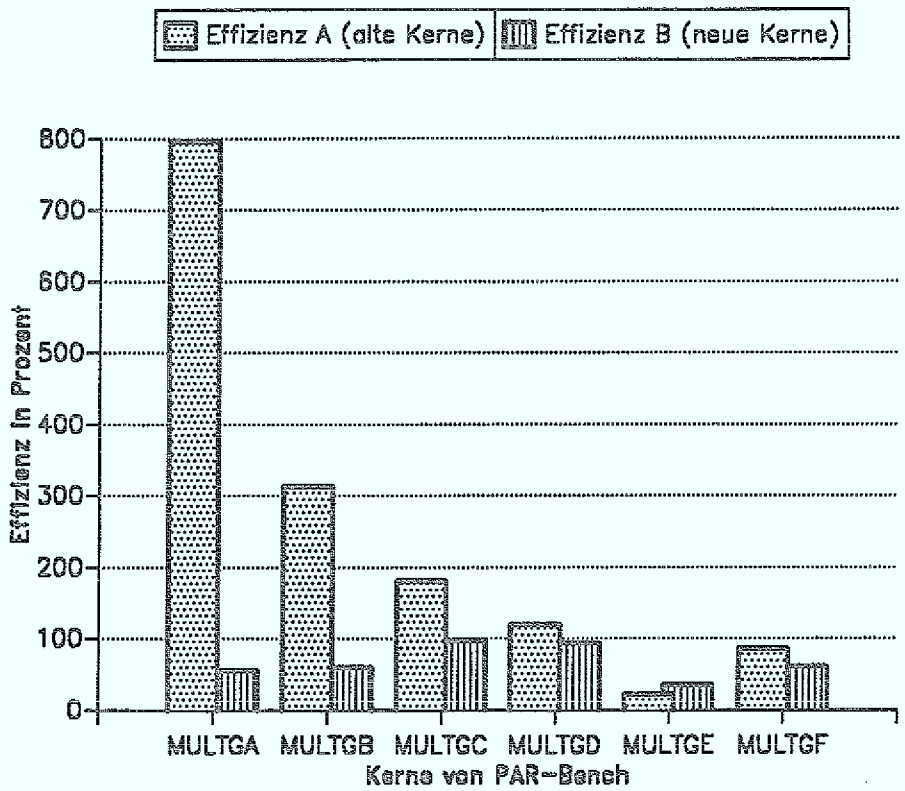


Abbildung 4.2: Die Effizienz der für die Parallelverarbeitung übersetzten Kerne auf einem 8-Prozessor-Cluster gegenüber den sequentiellen Kernen für die alten KERNE (Effizienz A) und die neuen Kerne (Effizienz B) im dedizierten Betrieb

grundjob nicht mehr zur Verfügung und müssen über UNIX-Kommandos ermittelt werden. Diese Kommandos werden jeweils in einer neuen *shell* gestartet, wobei auf den CRAY-Rechnern mit realem Hauptspeicher eine vollständige Kopie der alten *shell* erzeugt werden muß. Dadurch entsteht ein zusätzlicher Speicherplatzbedarf, der bei einigen Arbeitslasten zu Speicherengpässen auf der CRAY X-MP/416 führte. Deshalb werden die Jobs auf der CRAY X-MP/416 weiterhin über das NQS gestartet, während die Jobs auf den übrigen Rechnern als Hintergrundjobs gestartet werden. Da jedoch kein absoluter Vergleich der Meßergebnisse erfolgen soll, ist dieser Unterschied nicht wesentlich für den Vergleich der Meßergebnisse.

Kapitel 5

Leistungsmessung

Das System PAR-Bench ermöglicht die Generierung von Arbeitslasten in Abhängigkeit von einer Vielzahl von Parametern. Die Parameter-Datei des Systems PAR-Bench charakterisiert eine Arbeitslast nur in bezug auf die vorgegebenen Eigenschaften. Eine vollständige Modellierung beliebiger Arbeitslasten kann und soll auch nicht von PAR-Bench geleistet werden. Das System PAR-Bench dient deshalb nicht zur Bestimmung einer absoluten Leistungsangabe für eine vorgegebene Arbeitslast. Statt dessen ermöglicht das System PAR-Bench eine Untersuchung der Abhängigkeit der Leistung von bestimmten Parametern. In den Leistungsmessungen wird deshalb eine Arbeitslast vorgegeben, die als Referenz für vergleichende Messungen dient. In den folgenden Messungen wird dann jeweils ein Parameter variiert und der Einfluß auf die Leistung in bezug auf die Referenzmessung gemessen. Die Arbeitslasten werden auf der CRAY X-MP/416 generiert, so daß sich die Parameter auf die Leistung der CRAY X-MP/416 beziehen.

Im Mehrprogrammbetrieb von parallelen Programmen gibt es vielfältige Wechselwirkungen, die einen Einfluß auf die gemessene Leistung haben. Diese Wechselwirkungen entstehen bei der Konkurrenz der Jobs um gemeinsame Ressourcen und besitzen deshalb eine relativ hohe Varianz. Der Einfluß dieser Wechselwirkungen kann bei der Leistungsmessung bedeutsamer sein als der des untersuchten Parameters. In der Referenzmessung werden deshalb alle Parameter so gewählt, daß die Ausführungsdauer der Arbeitslast nur wenig durch diese Wechselwirkungen beeinflußt wird. Diese Parameter betreffen die Speicherverwaltung, die Ein-/Ausgabe und den Datentransfer zwischen den Prozessoren und dem gemeinsamen Speicher.

- Der Einfluß der Speicherverwaltung auf die Ausführung der Arbeitslast soll minimiert werden. Dies ist insbesondere deshalb von Bedeutung, weil eine Untersuchung der Leistung virtueller Speicherkonzepte nicht in PAR-Bench vorgesehen ist. Es wird deshalb kein zusätzlicher Speicher angefordert, indem der Parameter SIZE jeweils auf 0 gesetzt wird. Zusätzlich werden die Arbeitslasten so gewählt, daß der freie Arbeitsspeicher der Computersysteme für die

gesamte Arbeitslast ausreichend ist.

- Die gemeinsame Nutzung der Ein-/Ausgabe wird in dieser Arbeit nicht untersucht. Die Jobs der gemessenen Arbeitslasten führen daher nur ein Minimum an Ein-/Ausgabeoperationen durch, d.h. der Parameter IO wird auf 0 MIOOPS gesetzt.
- Die Wechselwirkungen, die durch die Nutzung des gemeinsamen Hauptspeichers auf den Testsystemen entstehen, werden gezielt untersucht. Für die Referenzmessung wird eine Arbeitslast ausgewählt, die das Speichersystem niedrig belastet. Da sich die Speicherarchitektur der Alliant jedoch wesentlich von der der übrigen Testsysteme unterscheidet, wurde für die Alliant eine besonders niedriger Wert für die Belastung des gemeinsamen Hauptspeichers gewählt.

In den ersten Messungen wird das grundsätzliche Verhalten der Computersysteme untersucht, wenn ein Teil der Arbeitslast parallel ausgeführt wird. Die nächsten Messungen bestimmen die Leistung eines Computersystems bei der Ausführung von Arbeitslasten mit parallelisierten Jobs, in Abhängigkeit von dem Parameter Parallelisierungsgrad. Der Einfluß des Speichersystems auf die Parallelisierung von Programmen bildet den letzten Punkt der Untersuchung.

5.1 Messungen auf der Alliant FX/2816

Die Architektur der Alliant FX/2816 zeichnet sich durch die Verwendung einer Cache-Speicherhierarchie aus, in der sowohl lokale als auch globale Cache-Speicher zum Einsatz kommen. Bei einer Leistungsmessung ist die genaue Kenntnis über die Nutzung der Cache-Speicher durch die Jobs der Arbeitslast erforderlich, um eine realitätsnahe Beurteilung der Meßergebnisse zu ermöglichen. Die *on-chip*-Cache-Speicher werden vom FORTRAN-Compiler verwaltet, da ihre Konsistenz nicht von der Hardware sichergestellt wird. Bei der Übersetzung von parallelen Programmen wird der *on-chip*-Cache-Speicher deshalb vom FORTRAN-Compiler als Vektorregister genutzt, indem mehrfach genutzte Daten explizit zwischen dem *on-chip*-Cache-Speicher und den niedrigeren Ebenen der Speicherhierarchie transferiert werden. Im Gegensatz dazu arbeiten die *on-chip*-Cache-Speicher bei der Ausführung von sequentiellen Programme wie auf einem Einprozessorsystem transparent und werden nur in geringem Maß vom Compiler berücksichtigt. Die Ausführungsdauer der PAR-Bench-Kerne wird von der unterschiedlichen Nutzung des *on-chip*-Cache-Speichers in sequentiellen und parallelen Programmen nur geringfügig beeinflußt. Der *on-chip*-Cache-Speicher kann deshalb in beiden Fällen mit den Vektorregistern der anderen Testsysteme gleichgesetzt werden.

Der globale Cache-Speicher der Alliant FX/2816 entspricht jedoch keinem Architekturmerkmal der anderen Testsysteme. Seine Zykluszeit ist jedoch mit der Zykluszeit

der Speicherbänke der anderen Computersysteme vergleichbar. Die Kapazität des globalen Cache-Speichers ist groß genug, um alle Daten, die im Zugriff der gleichzeitig ausgeführten Jobs stehen, aufzunehmen. Der Speicherplatz, der von sechzehn Jobs im globalen Cache-Speicher genutzt wird, beträgt ca. 96 KWorte, also ungefähr 38% der Kapazität des globalen Cache-Speichers. Daher ist davon auszugehen, daß alle Daten während einer Zeitscheibe aus dem globalen Cache-Speicher bedient werden, selbst wenn die aufgrund seiner Organisation nutzbare Kapazität geringer ist als seine tatsächliche Kapazität ist (siehe Abschnitt 2.1.3). Während im globale Cache-Speicher nur die häufig benutzten Daten Platz finden müssen, muß der Hauptspeicher das gesamte Programm und alle Daten aufnehmen können, wenn das Auslagern von Seiten verhindert werden soll. Da die zu messenden Arbeitslasten das Computersystem auch bei sequentieller Ausführung vollständig nutzen sollen, müssen sich in jeder Arbeitslast mindestens 16 Jobs gleichzeitig im Computersystem befinden. Um Effekte, die aufgrund der Speicherverwaltung entstehen, zu vermeiden sollen alle Jobs einer Arbeitslast gleichzeitig in den Hauptspeicher der Testkonfiguration passen. Bei einer Begrenzung des *strides* auf vier ist der statische Speicherplatzbedarf der Datenfelder klein genug, um hinreichend große Arbeitslasten zu ermöglichen. Da die Testkonfiguration der Alliant FX/2816 mit vier Speicherbänken ausgestattet ist, ist eine Begrenzung des maximalen *strides* auf vier auch für andere Messungen vertretbar.

Bei der Alliant FX/2816 müssen die Prozessoren des Computersystems statisch auf die für die Parallelverarbeitung verwendeten *cluster* eingeteilt werden. Die Konfiguration der FX/2816 als ein *cluster* mit 16 Prozessoren wird von Alliant nicht empfohlen, da sich die Prozessoren dann zu sehr beim Zugriff auf den *crossbar*-Zugang behindern. Zudem wurde die Konfiguration einer Alliant als 16-Prozessor-*cluster* zunächst nicht unterstützt; daher wurde die Alliant für die Messungen als zwei *cluster* mit jeweils acht Prozessoren konfiguriert.

5.1.1 Arbeitslasten mit günstigen Eigenschaften

Die in diesem Abschnitt durchgeführten Messungen dienen zur Bestimmung einer unteren Schranke für den Aufwand, der bei der Parallelverarbeitung auf der Alliant FX/2816 entsteht. Dazu werden die Job-Parameter so gewählt werden, daß keine bzw. nur geringe Wechselwirkungen zwischen den Jobs oder den kooperierenden Prozessen eines parallelen Jobs entstehen. Die parallelen Jobs sollen dabei in Bezug auf die parallele Ausführbarkeit optimale Eigenschaften besitzen. In späteren Messungen wird der zusätzliche Aufwand, der bei der Parallelverarbeitung von Arbeitslasten mit ungünstigeren Parametern entsteht, dann mit diesen Messungen verglichen.

- Die dynamischen Parameter der Jobs werden so gewählt, daß eine niedrige Speicherbelastung entsteht. Eine niedrige Speicherbelastung kann durch eine

Arbeitslast erreicht werden, deren Jobs rechenintensive Programme ausführen. Die Rechenleistung der Alliant FX/2816 beträgt 640 MFLOPS, während die Speicherleistung durch die maximale Speicherbandbreite der globalen Cache-Speicher auf 80 MMREFS beschränkt ist. Daher können bei einer vollständigen Überlappung acht Rechenoperationen und ein Speicherzugriff gleichzeitig durchgeführt werden. Bei einer Wahl von zwölf Rechenoperationen auf einen Speicherzugriff wird daher maximal $\frac{8}{12} = 66.6\%$ der Speicherbandbreite des globalen Cache-Speichers beansprucht. Diese Wahl wird durch eine Kernfolge realisiert, die auf der CRAY X-MP mit den Parametern 120 MFLOPS und 10 MMREFS generiert wurde.

- Die Parallelisierungsparameter werden für die parallelen Jobs so gewählt, daß eine möglichst balancierte parallele Ausführung der Jobs erreicht wird. Die Anzahl der genutzten Prozessoren beträgt deshalb N=8, wodurch alle Prozessoren eines *clusters* genutzt werden. Der Parallelisierungsgrad und die *load balance* werden mit PG=100% und LB=100% jeweils optimal vorgegeben.

Einprogrammbetrieb Zunächst soll untersucht werden, wie effizient parallele Programme auf einer Alliant FX/2816 ohne Hintergrundlast ausgeführt werden können. Dazu wird ein Job mit den oben angegebenen Parametern in der Messung FXM1 sequentiell und in der Messung FXM2 parallel ausgeführt. Dabei muß jedoch berücksichtigt werden, daß dieses parallele Programm nur auf einem *cluster* mit acht Prozessoren ausgeführt wird und acht Prozessoren leerlaufen. Das hat zur Folge, daß nur die halbe Rechenleistung genutzt werden kann und die Speicherbelastung nur die Hälfte der Speicherbelastung beträgt, die bei der Nutzung aller Prozessoren entsteht. Die bei dieser Messung erhaltenen Ergebnisse werden in der Tabelle 5.1 wiedergegeben. Der *speedup* beträgt bei der parallelen Ausführung mit acht Prozessoren 7.75, was einer Effizienz von 96.98% entspricht. Diese Meßwerte zeigen, daß auf der Alliant FX/2816 mit günstigen parallelen Programmen eine hohe Effizienz und damit eine hohe Beschleunigung erzielbar ist.

Messung	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
FXM1 sequentiell	192.3	100.0	193.3	1.0	100.0
FXM2 parallel	196.0	101.9	24.9	7.75	97.0

Tabelle 5.1: Effizienz und *speedup* eines rechenintensiven Jobs auf einem 8-Prozessor-*cluster* der Alliant FX/2816

Mehrprogrammbetrieb Die Untersuchung solcher besonders günstiger Jobs soll nun im Mehrprogrammbetrieb erfolgen. Die zu messende Arbeitslast besteht aus der Hintergrundlast mit den 16 Jobs BP001-016 und den 8 Testjobs BP017-024.

Mit Hilfe der 16 sequentiellen Jobs der Hintergrundlast wird erreicht, daß während der gesamten Ausführungsdauer der Testjobs hinreichend viele Jobs bereitstehen, um alle 16 Prozessoren der FX/2816 zu nutzen. Nach den 16 sequentiellen Jobs der Hintergrundlast werden weitere acht Testjobs gestartet, die dann sequentiell oder parallel mit unterschiedlichen variablen Parametern ausgeführt werden. Die Messung FXM3, in der neben den 16 Jobs der Hintergrundlast auch alle 8 Testjobs sequentiell ausgeführt werden, dient dabei als Referenz für den Vergleich mit den folgenden Messungen. Die für diese Messung auf der CRAY X-MP generierten Kernfolgen wurden auch für alle weiteren Messungen genutzt; eine Ausnahme bilden lediglich die Messungen in Abschnitt 5.1.3. Die zur Generierung der Arbeitslast verwendete Parameter-Datei wird in Tabelle 5.2 wiedergegeben (Beschreibung siehe Abschnitt 4.1).

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
16,	00,	10,	30,	010,	120,	000,	...	1,	000,	000,	1
08,	02,	10,	20,	010,	120,	000,	...	1,	000,	000,	1

Tabelle 5.2: Parameter für die Referenzmessung FXM3 mit sequentiellen Testjobs

In der zweiten Messung FXM4 werden alle acht Testjobs (BP017-BP024) parallel ausgeführt. Wie auch bei der Messung ohne Hintergrundlast wurde ein optimaler Parallelisierungsgrad PG=100% und eine optimale *load balance* LB=100% vorgegeben (siehe Tabelle 5.3).

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
16,	00,	10,	30,	010,	120,	000,	...	1,	000,	000,	1
08,	02,	10,	20,	010,	120,	000,	...	1,	**100**,	**100**,	**8**

Tabelle 5.3: Parameter für die Messung FXM4 mit acht parallelen Testjobs

Meßergebnisse Der Benchmark-Plot der sequentiellen Referenzmessung FXM3 wird in Abbildung 5.1 a) und der Benchmark-Plot der Vergleichsmessung mit acht parallelen Jobs wird in Abbildung 5.1 b) wiedergegeben. Anhand dieser Benchmark-Plots kann die Ausführungsdauer und der zusätzliche Aufwand von parallelen Programmen im Mehrprogrammbetrieb analysiert werden. Die Analyse erbrachte die im folgenden beschriebenen Ergebnisse:

- Bei der sequentiellen Referenzmessung FXM3 ist die Summe aus CPU- und System-time zu hoch, da sie die zur Verfügung stehende Prozessorzeit, die Total-time, um 1.97% übersteigt. Die Idle-time, die als Differenz von Total-time und der Summe aus CPU- und System-time gebildet wird, ist daher

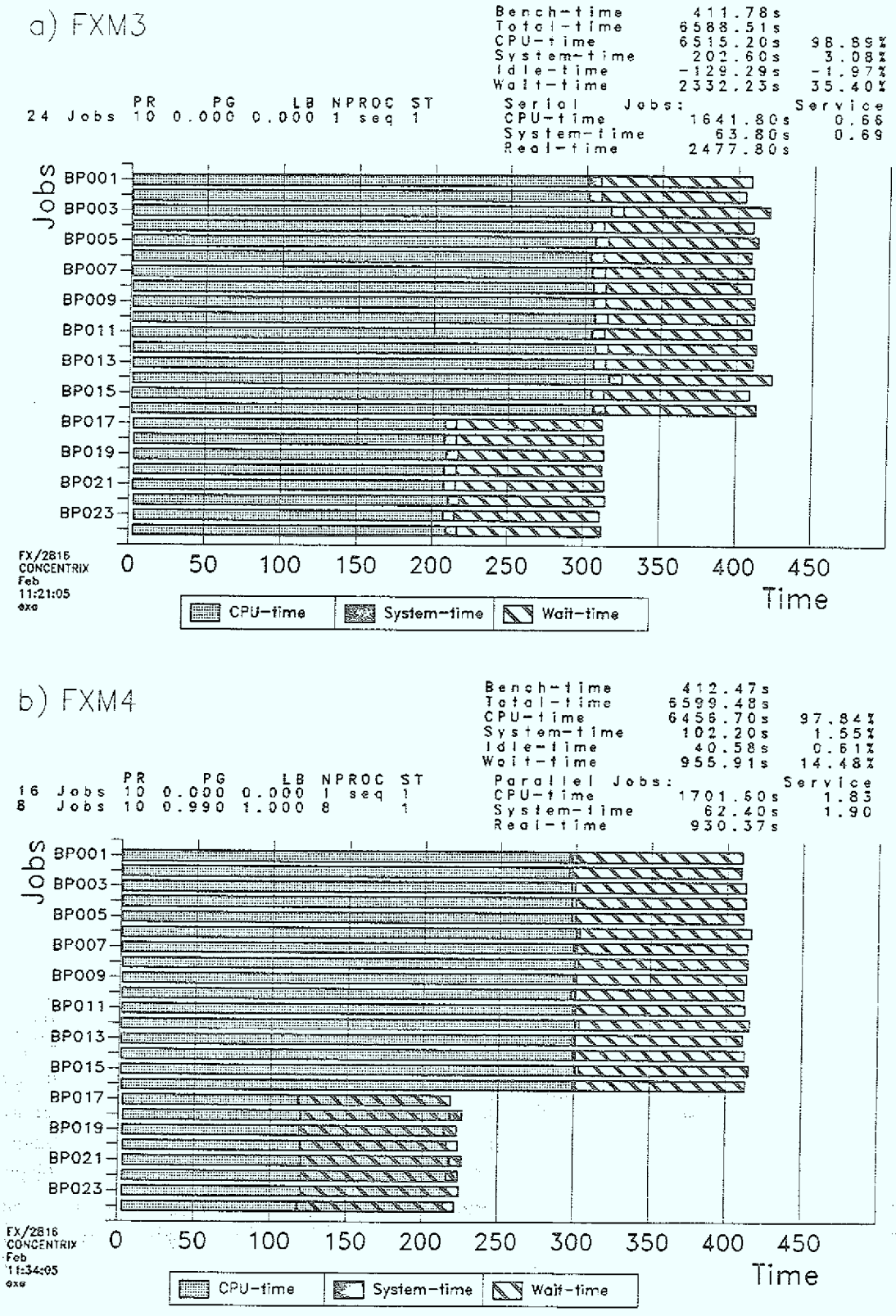


Abbildung 5.1: Benchmark-Plot der Messung FXM3 (a) und FXM4 (b)

negativ. Diese Werte lassen sich reproduzieren und zeigen, daß die von dem Systemaufruf *time* gelieferten Werte für die CPU-time und die System-time hohe Meßfehler aufweisen.

- Beim Vergleich der Bench-time und der Total-time der Referenzmessung FXM3 mit der Messung FXM4 fällt auf, daß diese bei der Messung FXM4 nahezu unverändert sind. Der durch die Parallelisierung der Testjobs verursachte Anstieg der Total-time muß auch in Relation zur CPU-time der Testjobs bei sequentieller Ausführung von 1641.8 Sekunden gesetzt werden. Der Anstieg der Total-time von 10.97 Sekunden entspricht 0.67% dieser CPU-time und ist damit so gering, daß er praktisch keine Bedeutung hat. Der bei der parallelen Ausführung der Jobs entstehende Aufwand ist hier vernachlässigbar.

- Die Real-time der parallelen Testjobs BP017-024 ist in Messung FXM4 gegenüber der Real-time der Testjobs in Messung FXM3 deutlich reduziert. Eine genauere Untersuchung dieser Beschleunigung erfolgt anhand des *service*-Faktors. Der *service*-Faktor gibt das Verhältnis von der CPU-time zur Real-time der Testjobs an. Bei der Referenzmessung FXM3 befinden sich während der Ausführungsdauer der Testjobs 24 Jobs im Computersystem. Bei einer gleichmäßigen Verteilung der Prozessorzeit auf alle Jobs ergibt sich ein *service*-Faktor von $\frac{16}{24} = 0.\bar{6}$. Der in der Referenzmessung FXM3 für die sequentiellen Testjobs gemessene *service*-Faktor stimmt mit diesem Wert überein (siehe Abbildung 5.2 a)). Im Gegensatz dazu ist der *service*-Faktor der Testjobs von 1.83 in der Messung FXM4 deutlich größer als eins. Da ein solchen *speedup* nur durch die parallele Ausführung eines Programms erreicht werden kann, belegt dieser Wert, daß die Testjobs in der Messung FXM4 auch tatsächlich parallel ausgeführt werden. Der *service*-Faktor eines Jobs zeigt weiterhin an, ob ein Job gegenüber der Hintergrundlast bevorzugt ausgeführt wird. Bei einer Akkumulation der *service*-Faktoren der acht parallelen Jobs zeigt sich, daß den acht parallelen Jobs im Durchschnitt 14.64 der 16 Prozessoren zur Verfügung gestellt werden. Auf den beiden *clustern* werden die parallelen Jobs also bevorzugt ausgeführt.

Eine genauere Untersuchung der Ausführungsdauer von parallelen Jobs wird im folgenden am Job BP024 der Arbeitslasten in den Messungen FXM3 und FXM4 vorgenommen. Aus den Benchmark-Plots in Abbildung 5.1 ist ersichtlich, daß die Ausführungszeiten der Testjobs nur in geringem Umfang variieren und der Job BP024 die Testjobs BP017-024 hinreichend gut repräsentiert. Neben der Messung mit acht parallelen Jobs wurden auch Messungen mit einem, zwei, vier und 24 parallelen Jobs in der Arbeitslast durchgeführt. Die Tabelle 5.4 gibt die bei diesen Messungen für den Testjob BP024 erhaltenen Resultate im Vergleich zu den im Einprogrammbetrieb und im Mehrprogrammbetrieb mit ausschließlich sequentiellen Jobs gemessenen Werten wieder.

Zu a) Die Tabelle 5.4 a) gibt die im Einprogrammbetrieb gemessenen Werte wieder. Der in den Messungen FXM1 und FXM2 verwendete Jobs ist mit dem Job BP024 der

Messungen FXM3-FX10 identisch, so daß ein direkter Vergleich der Zeiten möglich ist. Die CPU-time des Jobs bei sequentieller Ausführung in der Messung FXM1 wird mit 100% gleichgesetzt und gilt als Maßstab für den zusätzlichen Aufwand in den folgenden Messungen.

Zu b) In der Tabelle 5.4 b) werden die bei der Referenzmessung FXM3 gemessenen Werte des Jobs BP024 wiedergegeben, die eine Bestimmung des Mehrprogrammaufwands bei der Ausführung sequentieller Jobs ermöglichen. Die CPU-time des Jobs BP024 beträgt mit der Hintergrundlast 107% der CPU-time bei der Ausführung ohne Hintergrundlast. Der Mehrprogrammaufwand bewirkt also bei der Arbeitslast mit geringer Speicherbelastung der Referenzmessung FXM3 gegenüber der Messung FXM1 ohne Hintergrundlast einen Anstieg der CPU-time um 7%.

Zu c) Die Tabelle 5.4 c) enthält die Meßwerte der Arbeitslasten, die sich von der Referenzmessung FXM3 nur in der Anzahl der parallelen Jobs innerhalb der Arbeitslast unterscheiden.

a) Einprogrammbetrieb

Messung	par. Jobs	Real-time [Sek.]	CPU-time [Sek.]	CPU-time [Prozent]	*service* -Faktor	*speedup*	*speedup* Summe
FXM1	0	193.3	192.3	100	1.00	0.99	-
FXM2	1	24.9	196.0	102	7.87	7.75	-

b) Mehrprogrammbetrieb

Messung	par. Jobs	Real-time [Sek.]	CPU-time [Sek.]	CPU-time [Prozent]	*service* -Faktor	*speedup*	*speedup* Summe
FXM3	0	308.9	205.6	107	0.62	0.66	-

c) Mehrprogrammbetrieb mit parallelen Programmen

Messung	par. Jobs	Real-time [Sek.]	CPU-time [Sek.]	CPU-time [Prozent]	*service* -Faktor	*speedup*	*speedup* Summe
FXM4a	1	28.1	208.0	108	7.55	6.88	6.88
FXM4b	2	29.4	218.4	113	7.49	6.75	13.50
FXM4c	4	60.8	210.4	109	3.65	3.18	12.72
FXM4	8	115.0	211.2	110	1.90	1.67	13.36
FXM4d	24	355.6	212.8	111	0.66	0.54	12.96

Tabelle 5.4: Meßergebnisse des Jobs BP024 bei optimalen Voraussetzungen und einer variierenden Zahl von parallelen Jobs in der Arbeitslast.

- In der Messung FXM4a ist der Job BP024 der einzige parallele Job der Arbeitslast. Die CPU-time des Jobs ist bei der parallelen Ausführung gegenüber der sequentiellen Ausführung nur in geringen Maße erhöht und ist gegenüber dem Mehrprogrammaufwand von 7% vernachlässigbar. Der *speedup* des Jobs BP024 beträgt im Mehrprogrammbetrieb 6.88 im Vergleich zum *speedup* von 7.75 im Einprogrammbetrieb. Der geringere *speedup* ist zum größten Teil auf den Anstieg der CPU-time aufgrund des Mehrprogammbetriebs (Anstieg der Zugriffsverzögerungen) und den leicht reduzierten *service*-Faktor zurückzuführen.

- Die zwei parallelen Jobs in Messung FXM4b können gleichzeitig auf den beiden *clustern* der Alliant ausgeführt werden. Daher besteht nur ein geringer Unterschied zwischen dem in der Messung FXM4a und dem in Messung FXM4b erreichten *speedup*. Das gleiche gilt für den *service*-Faktor.

- In den Messungen mit mehr als zwei parallelen Jobs wird die Rechenleistung der beiden *cluster* auf den parallelen Jobs gemeinsame genutzt. Dadurch erreichen die Jobs nur noch ein niedrigeren *service*-Faktor, was zur Folge hat, daß auch der erzielte *speed up* geringer wird. Dies kann deutlich in den Messungen FXM4c und FXM4 beobachtet werden. Die parallelen Jobs werden dabei stark bevorzugt ausgeführt und bekommen fast die gesamte Rechenzeit der Alliant zugeteilt. In der Messung FXM4d werden alle Jobs parallel ausgeführt, was eine gleichberechtigte Ausführung der Jobs zur Folge hat. Da bei dieser Messung mehr parallele Jobs zur Ausführung bereitstehen als Prozessoren in der Alliant installiert sind, ist eine Beschleunigung durch Parallelverarbeitung nicht mehr möglich. Der *service*-Faktor der parallelen Jobs entspricht deshalb dem der sequentiellen Jobs in Messung FXM3.

Auf der Alliant FX/2816 werden parallele Programme also auch dann parallel auf einem *cluster* ausgeführt, wenn mehr Jobs zur Ausführung bereitstehen als Prozessoren. Der zusätzliche Aufwand der dadurch entsteht, ist für Programme mit günstigen Eigenschaften moderat. Dabei wird die Ausführung der Hintergrundlast wird durch die parallelen Jobs nicht wesentlich beeinträchtigt. Im folgenden werden Messungen für Arbeitslasten mit weniger idealen Eigenschaften durchgeführt.

5.1.2 Arbeitslasten mit niedriger Parallelität

Die Parallelisierung von realen Anwendungen ist selten so erfolgreich, daß eine vollständige Parallelisierung und eine optimale *load balance* erreicht wird. In derartigen Programmen gibt es häufig Abschnitte, die nicht oder nur geringfügig von der Parallelverarbeitung profitieren können. Im Mehrprogrammbetrieb stehen Prozessoren, die aufgrund von sequentiell auszuführenden Bereichen oder einer ungleichmäßigen Lastverteilung nicht genutzt werden können, für die Bearbeitung

anderer Programme zur Verfügung. Auf der Alliant FX/2816 ist dies jedoch aufgrund der statischen Einteilung der Prozessoren in *cluster* nur bedingt möglich. Ein Programm kann in sequentiellen Bereichen auf einem einzelnen Prozessor und in parallelen Bereichen auf einem *cluster* ausgeführt werden. Ein Wechsel zwischen der Ausführung auf einem *cluster* und einem Prozessor kann durch einen Systemaufruf erreicht werden, erfolgt jedoch nicht automatisch durch den FORTRAN-Compiler.

Parallelisierungsgrad In der Messung FXM10 wird untersucht, ob sich die Ausführungsdauer einer Arbeitslast mit parallelen Jobs, die einen geringeren Parallelisierungsgrad besitzen, entsprechend den Erwartungen erhöht. Dazu werden die Testjobs BP017-024 mit einem Parallelisierungsgrad von PG=70% ausgeführt. Die *load balance* bleibt bei dieser Messung mit LB=100% unverändert. Die Parameterdatei der Messung FXM10 wird in der Tabelle 5.5 wiedergegeben.

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
16,	00,	10,	30,	010,	120,	000,	...	1,	000,	000,	1
08,	02,	10,	20,	010,	120,	000,	...	1,	**70**,	**100**,	**8**

Tabelle 5.5: Parameter für die Messung FXM10 - Parallele Testjobs mit einem Parallelisierungsgrad von PG=70%.

Nach *Amdahl's Law* kann bei einem Parallelisierungsgrad von 70% mit acht Prozessoren maximal ein *speedup* von 2.58 erreicht werden. Bei der Ausführung eines Programms mit einem derartigen Parallelisierungsgrad auf einem 8-Prozessor-*cluster* kann also nur die 2.58-fache Beschleunigung erreicht werden, obwohl acht Prozessoren zur Ausführung bereitgestellt werden. Dadurch erhöht sich die CPU-time in der Theorie auf das 3.1-fache der CPU-time des Jobs bei sequentieller Ausführung.

Meßergebnisse In der Abbildung 5.2 b) wird der Benchmark-Plot der Messung FXM10 dem der sequentiellen Referenzmessung FXM3 gegenübergestellt. Die CPU-time der Testjobs beträgt bei paralleler Ausführung das 11.4-fache der CPU-time bei sequentieller Ausführung in Messung FXM3. Ein so hoher Anstieg läßt sich jedoch nicht allein durch Wartezeiten aufgrund des statischen Schedulings erklären. Der zusätzliche Anstieg der CPU-time wird durch den Compiler bei Nutzung des *on-chip*-Cache-Speichers in den sequentiellen Bereichen eines parallelen Programms verursacht. Der Compiler wird in PAR-Bench mit Hilfe der Compiler-Direktive *cvdS noconcur* dazu veranlaßt, den Programm-Code für die sequentiellen Bereiche nicht zu parallelisieren. Der so für einen sequentiellen Bereich in einem parallelen Programm generierte Maschinen-Code ist identisch mit dem Maschinen-Code, der für ein sequentielles Programme generiert wird. Während der *on-chip*-Cache-Speicher bei der Ausführung eines sequentiellen Programms ständig genutzt wird, ist er bei der Ausführung eines parallelen Programms nur als "Vektorregister" verfügbar und

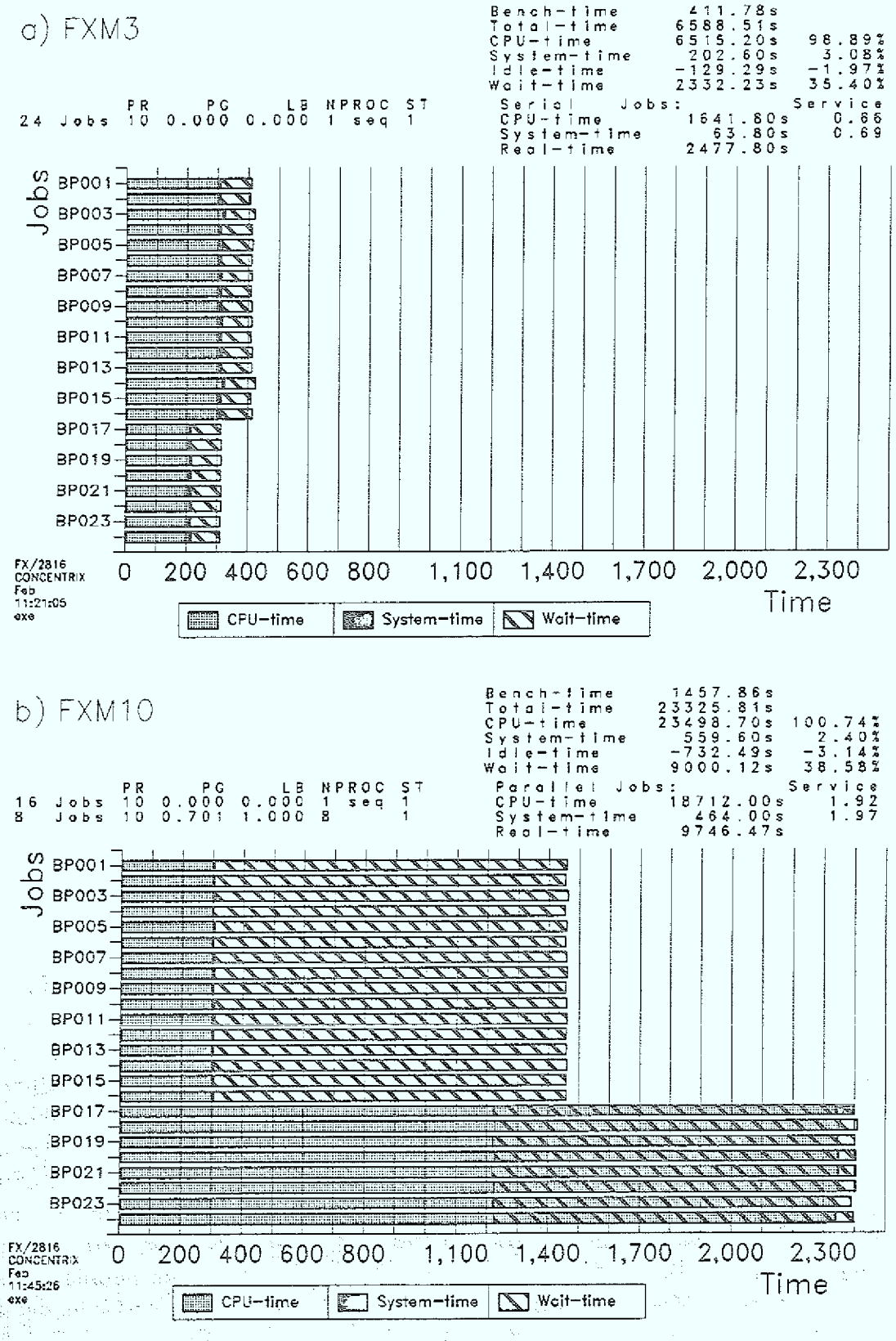


Abbildung 5.2: Benchmark-Plot der Messung FXM3 (a) und FXM10 (b)

für andere Daten gesperrt. Der in den sequentiellen Bereichen eines parallelen Programms generierte Maschinen-Code nutzt diese „Vektorregister" jedoch nicht, da er eine automatische Nutzung des Cache-Speichers wie in einem sequentiellen Programm erwartet.

Innerhalb der sequentiellen Bereiche erfolgt der Zugriff auf Zwischenergebnisse deshalb nicht aus dem *on-chip*-Cache-Speicher, sondern aus dem erheblich langsameren globalen Cache-Speichern bzw. aus dem Hauptspeicher. Die Verzögerungen durch diese Speicherzugriffe führen zu dem Anstieg der CPU-time innerhalb der sequentiellen Bereiche und somit zu der erhöhten Ausführungsdauer. Dieser Effekt wirkt sich auf Programme mit sequentiellen Bereichen so stark aus, daß die durch das statische Scheduling verursachten Leerlaufzeiten nicht eindeutig belegt werden können.

Load Balance Die *load balance* ist der zweite Einflußfaktor, der die Nutzung von Prozessoren innerhalb eines parallelen Programms bestimmt. Nur bei einer gleichmäßigen Verteilung der Last auf alle zur Verfügung stehenden Prozessoren wird ein maximaler *speedup* erreicht. Eine gleichmäßige Verteilung der Arbeitslast ist aber nicht möglich, wenn sich der Zeitbedarf eines Schleifendurchlaufs erst zur Laufzeit entscheidet. Die *load balance* ist auch dann nicht optimal, wenn ein Programm unter Umständen nur eine bestimmte Anzahl von Prozessoren für die Ausführung des Programms nutzen kann. Insbesondere in DOACROSS-Schleifen kann häufig nur ein kleiner Teil der Prozessoren genutzt werden; diese Schleifen werden jedoch hier nicht untersucht. Aufgrund der vorgegebenen und durch Synchronisationsanweisungen sichergestellten Ausführungsreihenfolge entstehen bei einer ungleichmäßigen Verteilung der Last Leerlaufzeiten, in denen einige Prozessoren erst wieder genutzt werden können, wenn andere ihre Arbeit beendet haben. Bei einem statischen Scheduling wie bei der Alliant FX/2816 können solche Leerlaufzeiten nicht für die Bearbeitung anderer Jobs genutzt werden. Der Einfluß des statischen Schedulings auf die Ausführungsdauer einer Arbeitslast wird in der Messung FXM11 untersucht. Die Parameterdatei der Messung FXM11 wird in Tabelle 5.6 wiedergegeben. Die *load balance* der parallelen Testjobs wird auf LB=70% reduziert, während ein optimaler Parallelsierungsgrad von PG=100% wie in Messung FXM4 vorgegeben wird.

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
16,	00,	10,	30,	010,	120,	000,	...	1,	000,	000,	1
08,	02,	10,	20,	010,	120,	000,	...	1,	**100**,	**070**,	**8**

Tabelle 5.6: Parameter für die Messung FXM11 - Parallele Testjobs mit einer *load balance* von LB=70%.

Die Definition des Parameters *load balance* wurde so gewählt, daß der Parameter LB direkt die erreichbare Effizienz und damit den erreichbaren *speedup* beschreibt (siehe Abschnitt 2.4.1). Damit kann bei der parallelen Ausführung der Testjobs

bestenfalls ein *speedup* von 5.6 erreicht werden, obwohl acht Prozessoren für die Ausführung zur Verfügung stehen. Im zeitlichen Mittel werden somit 2.4 der acht Prozessoren nicht genutzt und laufen leer. In der Theorie steigt dadurch die CPU-time auf das $\frac{8}{5.6} = 1.43$-fache der CPU-time bei sequentieller Ausführung. Der erwartete Anstieg der CPU-time gegenüber der sequentiellen Ausführung beträgt demnach 43%.

Meßergebnisse Die Abbildung 5.3 b) gibt den bei der Messung FXM11 erhaltenen Benchmark-Plot mit dem der sequentiellen Referenzmessung FXM3 wieder. Der gemessene Anstieg der CPU-time gegenüber der sequentiellen Ausführung der Testjobs beträgt 42.19%. Dieser Wert stimmt mit dem erwarteten Wert im Rahmen der Messgenauigkeit überein. Die Ausführungszeit der gesamten Arbeitslast erhöht sich relativ zu der CPU-time der parallelen Jobs um 36% und bestätigt damit die Größenordnung des Anstiegs der gemessenen Prozessorzeit. Der Unterschied zwischen den beiden Werten läßt sich auf eine geringere Belastung des Speichersystems durch die leerlaufenden Prozessoren und damit auf eine reduzierte Anzahl von Zugriffskonflikten zurückführen. Die gemessenen Werte bestätigen den erwarteten Anstieg der Ausführungsdauer von parallelen Jobs mit ungünstiger Lastverteilung aufgrund des statischen Schedulings der Alliant FX/2816.

5.1.3 Arbeitslasten mit hoher Speicherbelastung

Die Speicherbelastung der rechenintensiven Jobs für die Referenzmessung FXM3 wurde so gewählt, daß sie maximal 66% der Speicherbandbreite des globalen Cache-Speichers betragen konnte. Die effektive Speicherbandbreite, die diese Jobs im Einprogrammbetrieb auf einer Alliant FX/2816 erreichen, beträgt jedoch nur ca. 1.22 MMREFS. Bei einer Nutzung aller 16 Prozessoren beträgt die Speicherbelastung somit ca. 20 MMREFS. Diese niedrige Speicherbelastung ist zum einen darauf zurückzuführen, daß die Rechen- und Speicheroperationen nicht gleichzeitig ausgeführt werden. Zum anderen wird die Speicherbelastung durch den zusätzlichen Aufwand verringert, der durch die Instruktionen zur Ablaufsteuerung entsteht. Im Mehrprogrammbetrieb wurde in der Messung FXM3 aufgrund von Zugriffsverzögerungen eine Speicherbelastung von ca. 16 MMREFS erreicht. Diese Speicherbelastung beträgt ungefähr 20% der maximalen Speicherbandbreite und entspricht damit der Forderung nach einer geringen Belastung des Speichersystems. Die Speicherbelastung einer Arbeitslast kann als dynamischer Parameter von PAR-bench nur über eine neue Kernfolge und damit über einen neuen Generierungslauf auf der CRAY X-MP modifiziert werden. Im folgenden werden die Parameter beschrieben, in denen sich die speicherintensive von der rechenintensiven Arbeitslast unterscheidet.

- Die Speicherbelastung durch die neue Arbeitslast soll im Bereich der maximalen Speicherbandbreite des globalen Cache-Speichers von 80 MMREFS liegen.

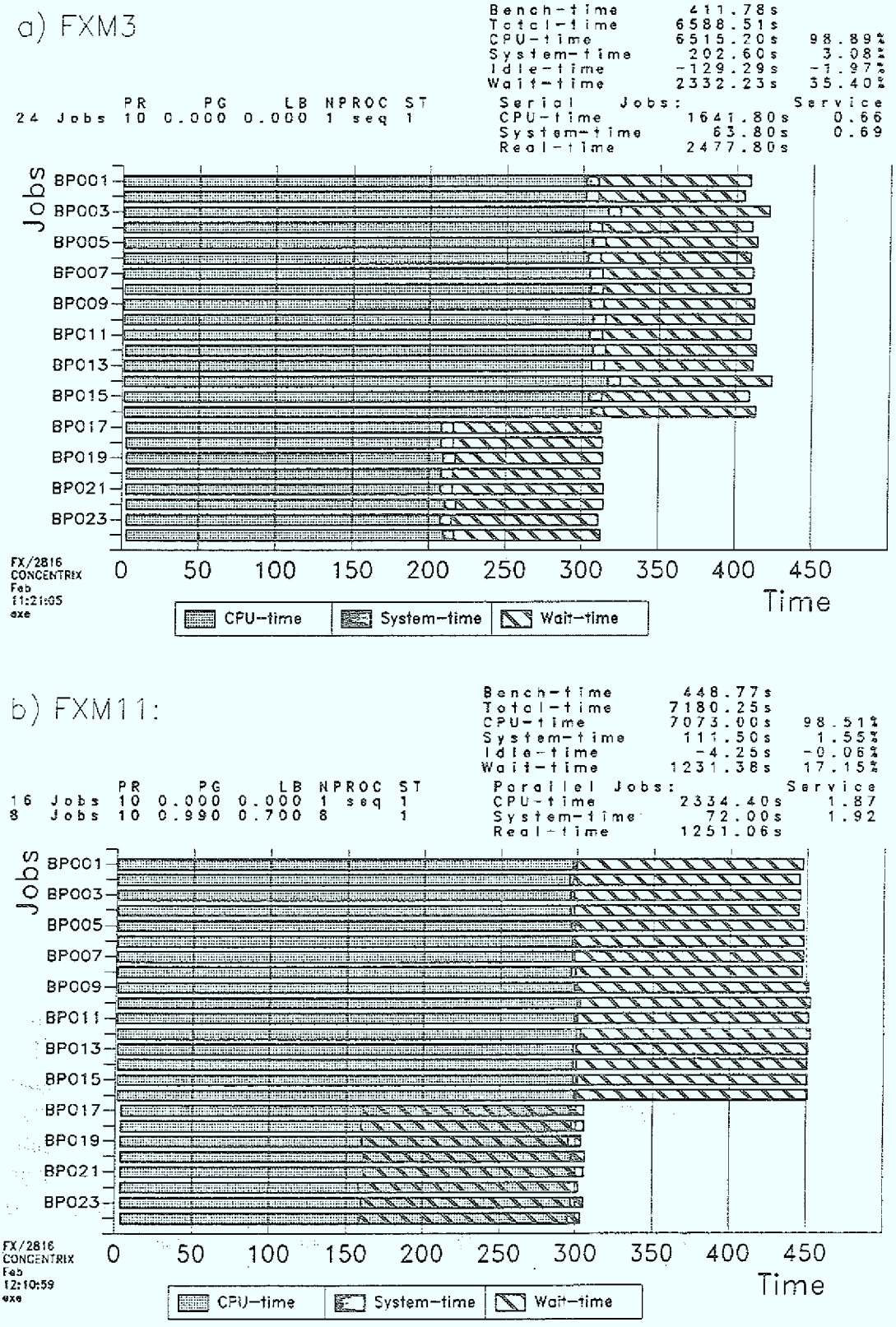


Abbildung 5.3: Benchmark-Plot der Messung FXM3 (a) und FXM11 (b)

Da die Speicherbelastung auf der Alliant FX/2816 in der Messung FXM3 ohne Zugriffsverzögerungen ungefähr 20 MMREFS beträgt, wird ungefähr das vierfache der Speicherbelastung benötigt. Für die Generierung der Arbeitslast auf der CRAY X-MP wurde der Parameter MM von 10 MMREFS auf 40 MMREFS erhöht, während der Parameter FL mit 120 MFLOPS beibehalten wurde.

- Die Ausführungsdauer der Testjobs mit hoher Speicherbelastung wird aufgrund von der erhöhten Zugriffsverzögerungen ansteigen. Da die Testjobs aber mit einer Hintergrundlast mit den Parametern aus Messung FXM3 ausgeführt werden sollen, muß die Ausführungsdauer der Testjobs kleiner als die Ausführungsdauer dieser Hintergrundlast sein. Deshalb wird auf der CRAY X-MP die Prozessorzeit mit dem Parameter SEC von 20 Sekunden für die rechenintensiven Testjobs auf 10 Sekunden für die speicherintensiven Testjobs reduziert.

Einprogrammbetrieb Der Einfluß der erhöhten Speicherbelastung wird zunächst wieder im Einprogrammbetrieb untersucht. Ein Job mit den oben angegebenen Parametern wird dazu in der Messung FXM20 sequentiell und in der Messung FXM21 auf einem *cluster* parallel ausgeführt. Die Ergebnisse dieser Messungen werden in Tabelle 5.7 wiedergegeben.

Messung	Real-time [Sek.]	CPU-time [Sek.]	CPU-time [Prozent]	*speedup*	Effizienz [Prozent]
FXM20	119.3	118.5	100	1	100
FXM21	17.8	142.4	120	6.70	83.8

Tabelle 5.7: Effizienz und *speedup* eines speicherintensiven Jobs auf einem 8-Prozessor-*cluster* der Alliant FX/2816.

Obwohl bei der parallelen Ausführung des Programms nur ein *cluster* der Alliant und damit nur die Hälfte der verfügbaren Prozessoren genutzt werden, fällt die Effizienz der Parallelverabeitung der speicherintensiven Jobs gegenüber den rechenintensiven Jobs von 96.9% auf 83.8% deutlich ab. Im Mehrprogrammbetrieb bei der Nutzung aller 16 Prozessoren ist deshalb mit einem weiteren Anstieg der Ausführungsdauer zu rechnen. Auf der Alliant FX/2816 wird für den speicherintensiven Job in der Messung FXM20 eine Speicherbelastung von 3.4 MMREFS gemessen. Die Speicherbelastung der speicherintensiven Jobs beträgt also nicht das 4-fache sondern nur das 2.8-fache der Speicherbelastung durch die rechenintensiven Jobs. Der dadurch entstehende Aufwand ist dennoch erheblich.

Während im Einprogrammbetrieb jeweils nur ein Prozessor für die Ausführung eines sequentiellen Programms genutzt werden, können im Mehrprogrammbetrieb auch se-

quentielle Programme alle Prozessoren eines Rechners nutzen. Dadurch entstehen im Mehrprogrammbetrieb von sequentiellen Programmen ähnliche Verzögerungen durch Zugriffskonflikte wie bei der Ausführung eines parallelen Programms. Bei der parallelen Ausführung von speicherintensiven Programmen erfolgt jedoch durch alle Prozessoren eines *clusters* eine hohe Speicherbelastung, während im Mehrprogrammbetrieb von sequentiellen Programmen ein Ausgleich zwischen rechenintensiven und speicherintensiven Programmen erfolgen kann. Dieser Effekt wird in den Messungen FXM22 und FXM23 untersucht.

Mehrprogrammbetrieb Die Hintergrundlast BP001-016 besteht bei der Messung FXM22 ebenso wie bei der Referenzmessung FXM3 aus rechenintensiven Jobs mit geringer Speicherbelastung. Im Gegensatz dazu sind die Testjobs BP017-BP024 nun speicherintensiv und belasten den Speicher entsprechend den oben für den Einprogrammbetrieb gemessenen Werten. In der Messung FXM22 werden die speicherintensiven Testjobs sequentiell ausgeführt. Während der Ausführungsdauer der 8 Testjobs mit einer Speicherbelastung von ca. 3.4 MMREFS werden dann noch die 16 Jobs der Hintergrundlast mit einer Speicherbelastung von 1.2 MMREFS ausgeführt. Von diesen 24 Jobs können nur 16 Jobs gleichzeitig bearbeitet bearbeitet werden. Die durchschnittliche Gesamtspeicherbelastung beträgt bei dieser Messung also ca. 31 MMREFS bzw. 38% der Speicherbandbreite der globalen Cache-Speicher. Der Generierungslauf für die Messung FXM22 wurde auf der CRAY X-MP mit den in Tabelle 5.8 wiedergegebenen Parametern durchgeführt.

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
16,	00,	10,	30,	010,	120,	000,	...	1,	000,	000,	1
08,	02,	10,	**10**,	**040**,	120,	000,	...	1,	000,	000,	1

Tabelle 5.8: Parameter für die Referenzmessung FXM22 - Sequentielle Testjobs mit hoher Speicherbelastung.

In der Messung FXM23 werden die 8 speicherintensiven Testjobs parallel ausgeführt, wobei sowohl der Parallelisierungsgrad mit PG=100% und auch die *load balance* mit LB=100% optimal vorgegeben. Die Parameter-Datei für die Messung FXM23 wird in der Tabelle 5.9 wiedergegeben.

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
16,	00,	10,	30,	010,	120,	000,	...	1,	000,	000,	1
08,	02,	10,	**10**,	**040**,	120,	000,	...	1,	**100**,	**100**,	**8**

Tabelle 5.9: Parameter für die Messung FXM23 - Parallele Testjobs mit hoher Speicherbelastung.

Während der Bearbeitung der parallelen Jobs werden nahezu keine sequentiellen Jobs ausgeführt. Die 16 Prozessoren belasten dabei das Speichersystem mit ca. 54 MMREFS, während nach der Ausführung der parallelen Testjobs die Speicherbelastung durch die Hintergrundlast nur noch ca. 20 MMREFS beträgt. Die Speicherbelastung von 54 MMREFS beträgt ca. 68% der maximal Bandbreite der globalen Cache-Speicher und führt zu einer deutlich höheren Belastung des Speichersystems. Dies führt sowohl an den vier globalen Cache-Speichern als auch an den acht gemeinsamen Speicherzugängen der Prozessoren zu Zugriffskonflikten (siehe Abschnitt 3.1). Bei einem gleichzeitigen Zugriff über alle Speicherzugänge können nur die Hälfte der Zugriffe direkt bedient werden. Bei der anderen Hälfte der Zugriffe steigt die Zugriffszeit um 50% an. Zusätzlich wird durch einen blockierten Prozessor der gemeinsame Speicherzugang und damit ein zweiter Prozessor blockiert. Diese beiden Engpässe führen bei der parallelen Ausführung der speicherintensiven Testjobs zu einem Anstieg der Ausführungszeit. Da jedoch auch bei der Messung FXM22 eine relativ hohe Speicherbelastung von ca. 31 MMREFS erreicht wird, ist auch bei dieser Messung die Ausführungsdauer erhöht und der Unterschied zwischen beiden Messungen geringer. Die bevorzugte Verarbeitung der parallelen Testjobs mit hoher Speicherbelastung führt auch dazu, daß diese schneller beendet werden und dann nicht mehr die rechenintensive Hintergrundlast stören.

Meßergebnisse In der Abbildung 5.4 werden die Benchmark-Plots der Messungen FXM22 und FXM23 wiedergegeben. Die CPU-time der Testjobs ist bei der Messung FXM23 um 32% angestiegen. Der Anstieg der Total-time um 338.84 Sekunden entspricht einem Anstieg von 27.4% relativ zur CPU-time der sequentiell ausgeführten Testjobs. Der unterschiedliche Anstieg von Total-time und CPU-time stimmt mit der Vorhersage überein, daß die Hintergrundlast in der Messung FXM23 nach Beendigung der parallelen Testjobs schneller ausgeführt werden kann. Sowohl der Anstieg der CPU-time als auch der Anstieg der Total-time dokumentieren, daß der durch die Parallelisierung bewirkte Anstieg von Zugriffsverzögerungen erheblich sein kann und auf der Alliant FX/2816 berücksichtigt werden muß. Die Ergebnisse dieser Messungen zeigen, daß nicht nur die Nutzung der Alliant FX/2816 als ein *cluster* mit 16 Prozessoren durch das Speichersystem beeinträchtigt wird. Auch im Mehrprogrammbetrieb paralleler Programme wird der Durchsatz des Computersystems durch parallele Programme in hohem Maß beeinträchtigt.

5.1.4 Zusammenfassung der Meßergebnisse

Die Messungen zeigen, daß die Leistung der Alliant FX/2816 bei der Bearbeitung von parallelen Programmen im Mehrprogrammbetrieb in hohem Maße von den Eigenschaften der Arbeitslast abhängig ist. Die Beurteilung der Meßergebnisse wurde durch die ungenaue Abrechnung der CPU-time und der System-time durch den Systemaufruf *time* erschwert. Die Leistungsmessungen auf die FX/2816 mit dem

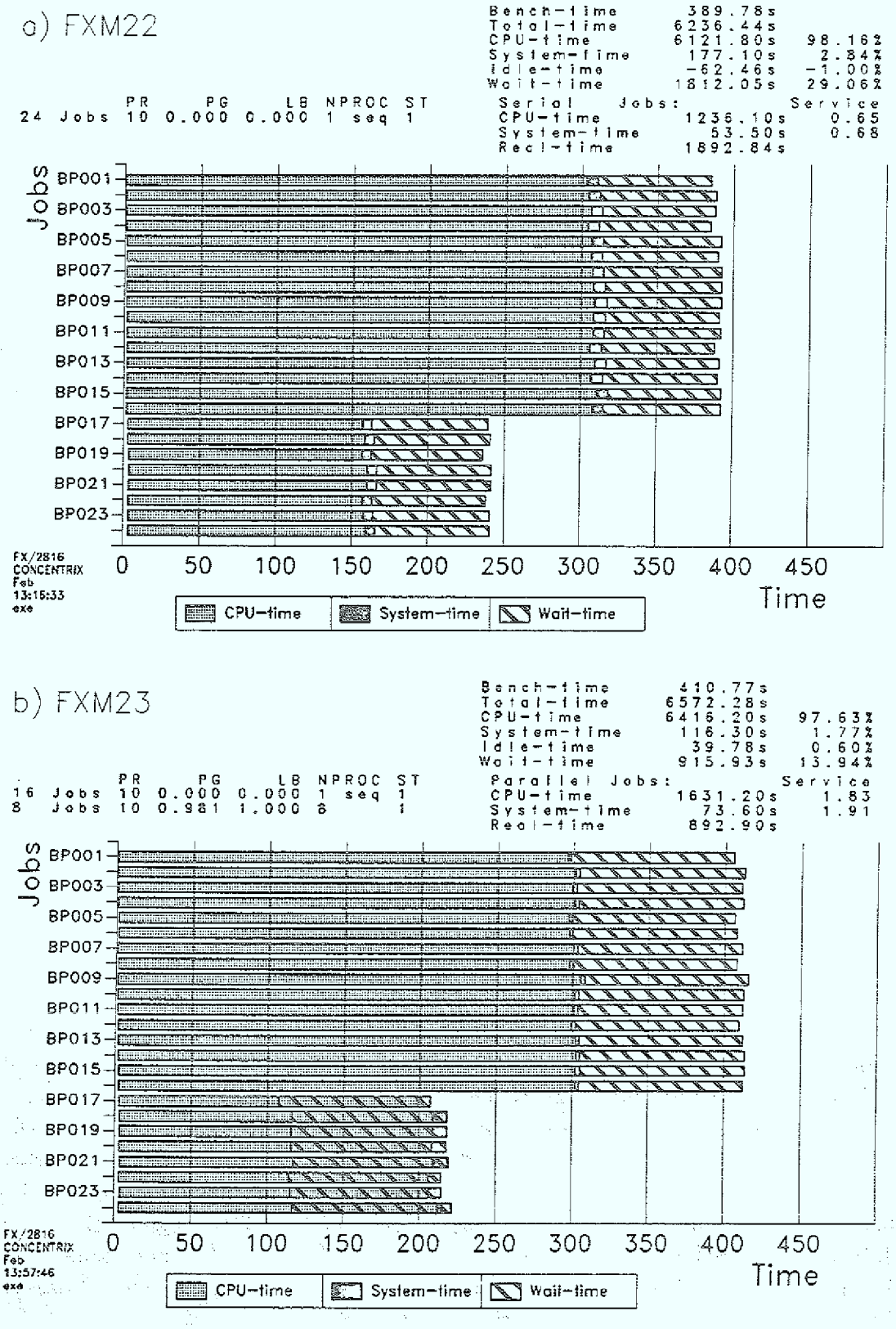


Abbildung 5.4: Benchmark-Plot der Messung FXM22 (a) und FXM23 (b)

Betriebssystem CONCENTRIX führen zu den folgenden Aussagen:

- Im Einprogrammbetrieb erreicht die Alliant FX/2816 bei der parallelen Ausführung von sehr rechenintensiven Programmen mit acht Prozessoren eine hohe Effizienz von 96.9%.
- Im Mehrprogrammbetrieb werden parallele Programme auf den als *cluster* konfigurierten Prozessoren bevorzugt ausgeführt. Dies führt zu einer deutlichen Beschleunigung der parallelisierten Programme. Bei der parallelen Ausführung von sehr rechenintensiven Programmen entsteht im Mehrprogrammbetrieb nur ein geringer zusätzlicher Aufwand, der kleiner als 1% ist.
- Aufgrund der statischen Einteilung der Prozessoren in *cluster* entstehen bei der Ausführung von parallelen Programmen mit niedrigem Parallelisierungsgrad bzw. ungünstiger *load balance* Leerlaufzeiten, die den Durchsatz der FX/2816 massiv reduzieren. Eine Compiler-Inkonsistenz führt bei den mit *cvd$ nocon-cur* gekennzeichneten sequentiellen Bereichen zu einer schlechten Nutzung der lokalen Cache-Speicher.
- Die Parallelisierung von speicherintensiven Programmen verursacht auf der Alliant FX/2816 auch im Mehrprogrammbetrieb Leistungseinbußen von bis zu 27%.

Die effizient Ausführung paralleler Programme ist auf der Alliant FX/2816 im Mehrprogrammbetrieb also nur für eine eingeschränkte Klasse von Programmen möglich. Diese Programme müssen einerseits nahezu vollständig parallel gut lastbalanciert sein. Andererseits dürfen diese Programme nicht wesentlich speicherintensiver als die Hintergrundlast sein.

5.2 Messungen auf der Convex C3840

Die Architektur der CONVEX C3840 ist von der Konzeption her stärker an traditionelle Vektorrechner angelehnt als die Alliant FX/2816. Auf der CONVEX C3840 wird die Vektorverarbeitung ausschließlich durch den verschränkten Speicher und die Vektorregister beschleunigt. Der Cache-Speicher wird nur für skalare Daten und Instruktionen genutzt und hat deshalb keinen Einfluß auf die Vektorverarbeitung. Da die Kerne von PAR-Bench vektorisiert sind, hat der Cache-Speicher keinen wesentlichen Einfluß auf die Ausführungsdauer der Arbeitslasten. Die von den Jobs durchgeführten Speicheroperationen werden also direkt aus dem gemeinsamen Hauptspeicher bedient. Da die C3840 im Gegensatz zur Alliant vier Prozessoren besitzt, reichen auch schon vier sequentielle Jobs, um die Maschine vollständig zu nutzen, so daß Arbeitslasten mit einer geringeren Anzahl von Jobs gemessen werden können. Die minimale Speicherkapazität, mit der eine CONVEX C3840

ausgeliefert wird, beträgt 64 MWorte und ist damit größer als die aller übrigen Testsysteme. Der Speicherbedarf der Arbeitslasten war deshalb erheblich kleiner als die Hauptspeicherkapazität der C3840 und brauchte daher nicht bei den Messungen berücksichtigt werden. Das PAR-Bench-System wurde daher in seiner Standardkonfiguration mit einer maximalen Schrittweite von 32 belassen. Die CONVEX C3840 unterstützt ein dynamisches Scheduling, daß eine feste Einteilung der Prozessoren wie auf der Alliant unnötig macht.

Zum Test standen zwei verschiedene CONVEX-Systeme zur Verfügung. Die Taktperiode betrug zur Zeit der Messung bei einem der Systeme noch 18 ns statt den angestrebten 16.67 ns, da das System noch mit einer Vorabversion der Prozessoren ausgestattet war. Die Messungen auf der C3840 mit der leicht erhöhten Taktperiode von 18 ns sind mit dem Symbol $\star$ gekennzeichnet.

5.2.1 Arbeitslasten mit günstigen Eigenschaften

In diesem Abschnitt wird zunächst der Aufwand bestimmt, der unter günstigen Voraussetzungen bei der Parallelverarbeitung auf der CONVEX C3840 entsteht. Da die Parameter der Arbeitslast für die Alliant FX/2816 aufgrund der geringen Hauptspeicherleistung sehr extrem gewählt werden mußten, werden für die Messungen auf der CONVEX C3840 neue Arbeitslasten verwendet.

- Das Verhältnis zwischen der Spitzenrechenleistung und der maximalen Speicherbandbreite ist auf der CONVEX C3840 erheblich besser balanciert als bei der Alliant FX/2816. Die Peak-Performance der CONVEX C3840 beträgt 480 MFLOPS und der gemeinsame Hauptspeicher stellt eine Bandbreite von 274 MMREFS zur Verfügung. Bei der vollständigen Nutzung dieser Leistungen werden auf eine Speicheroperation ungefähr zwei Rechenoperationen durchgeführt. Für eine geringe Speicherbelastung wurde auf eine Rechenoperation vier Speicheroperationen vorgegeben, wodurch die Speicherbelastung durch die Arbeitslast höchstens 50% der maximal möglichen Speicherbandbeite beträgt. Diese Bedingung wird durch die Jobs, die auf der CRAY X-MP mit FL = 200 MFLOPS und MM = 50 MMREFS generiert werden, erfüllt.

- Der Parallelsierungsgrad und die *load balance* der parallelen Jobs werden mit PG=100% und LB=100% und damit optimal vorgegeben. Dabei sollen die parallelen Jobs alle vier Prozessoren nutzen, was durch den Parameter N=4 vorgegeben wird.

Einprogrammbetrieb Die Effizienz der Parallelverarbeitung wird zunächst im Einprogrammbetrieb, also ohne Hintergrundlast, untersucht. In der Messung C38M1a* wird ein Job mit den oben angegebenen Parametern sequentiell ausgeführt.

Die Verzögerungen, die durch Zugriffskonflikte um den gemeinsamen Hauptspeicher entstehen, werden in den Messungen C38M1b-d* untersucht. In diesen Messungen werden zwei, drei und vier identische Jobs gleichzeitig ausgeführt (siehe Tabelle 5.10).

Messung	Jobs	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	Real-time [Prozent]	Effizienz [Prozent]
C38M1a*	1	74.5	100.0	75.4	100.0	100.0
C38M1b*	2	74.7	100.2	75.7	100.4	100.0
C38M1c*	3	75.3	101.0	77.1	102.3	97.7
C38M1d*	4	76.0	102.0	77.6	102.9	97.2

Tabelle 5.10: Ausführungsdauer eines rechenintensiven Jobs auf der C3840 mit einer Taktperiode von 18 ns in Abhängigkeit von der Anzahl der Jobs bei sequentieller Ausführung

Der Mehrprogrammaufwand für die vier Jobs in Messung C38M1d* beträgt ungefähr 2.9% und der Anstieg der CPU-time beträgt ungefähr 2.0%. Der Anstieg der CPU-time ist auf Zugriffskonflikte um den gemeinsamen Hauptspeicher zurückzuführen, während der Anstieg der Real-time zusätzlich Verzögerungen aufgrund von Konflikten um die Ein-/Ausgabe enthalten kann.

In den nächsten Messungen wird der Job aus Messung C38M1a parallel ausgeführt. Der parallele Job besitzt entsprechend den Vorgaben mit einem Parallelsierungsgrad von PG=100% und einer *load balance* von LB=100% optimale Parallelität. In der Messung C38M2d* wird der Job mit N=4 Prozessoren parallel ausgeführt, während in den Messungen C38M2a-c* der Parameter N von 1 bis 3 variiert wird (siehe Tabelle 5.11). Beim Vergleich der CPU-time der sequentiellen Jobs in den Messungen

Messung	N	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
C38M2*	1	74.5	100.0	75.4	1.00	100.0%
C38M2*	2	74.7	100.2	38.0	1.98	99.2%
C38M2*	3	75.3	101.0	25.7	2.93	97.8%
C38M2*	4	76.4	102.6	19.9	3.79	94.7%

Tabelle 5.11: Effizienz und *speedup* eines rechenintensiven Jobs auf der CONVEX C3840 mit einer Taktperiode von 18 ns in Abhängigkeit von der Zahl der genutzten Prozessoren N

C38M1a-c* mit der des parallelen Jobs in den Messungen C38M2a-c* fällt auf, daß diese paarweise gleich sind. Nur in der Messung C38M2d* ist die CPU-time des parallelen Jobs um 0.6 Sekunden geringfügig erhöht. Aus den Messungen C38M1d*

geht hervor, daß die Zugriffskonflikte einen Anstieg um ungefähr 1.5 Sekunden (2% der CPU-time) verursachen. Der bei der Messung C38M2d* mit vier Prozessoren erreichte *speedup* beträgt 3.79 und die Effizienz beträgt somit 94.7%. Die Effizienz von 94.7% entspricht einem Anstieg der genutzten Prozessorzeit von 75.4 Sekunden in der Messung C38M1a* um 4.2 Sekunden auf 4 Prozessoren * 19.9 Sekunden = 79.6 Sekunden in der Messung C38M2d*. Zieht man die 1.5 Sekunden, die auch in der Messung C38M1d* aufgrund von Zugriffskonflikten entstehen, von diesen 4.2 Sekunden ab, so bleibt ein Aufwand von 2.7 Sekunden bzw. 3.6% der Real-time in Messung C38M1a* der durch die Parallelisierung verursacht wird.

Mehrprogrammbetrieb Die Untersuchung des Parallelisierungsaufwands von derartigen rechenintensiven Jobs erfolgt nun im Mehrprogrammbetrieb. An die zu messenden Arbeitslasten wurden eine Reihe von Forderungen gestellt. Zum einen soll die Arbeitslast aus einer ähnlichen Anzahl von Jobs bestehen wie die Arbeitslasten auf der Alliant FX/2816. Zum anderen sollen die für die CONVEX C3840 verwendeten Arbeitslasten auch für die Messungen auf der CRAY X-MP/416 und Y-MP8/832 übernommen werden, woraus sich zwei weitere Anforderungen an die Arbeitslast ergeben. Die Speicherkapazität der X-MP/416 beschränkt die Anzahl der Benchmark-Jobs auf ca. 16 gleichzeitig im Speicher, ohne das Jobs ausgelagert werden müssen. Die CRAY Y-MP8/832 besitzt acht Prozessoren, so daß die Hintergrundlast auch aus mindestens acht Jobs bestehen muß. Die zu messende Arbeitslast besteht deshalb aus 16 Jobs, von denen die acht Jobs BP001-008 Hintergrundlast und die acht Jobs BP009-016 Testjobs sind. Die Arbeitslast wird in vier Gruppen mit je vier Jobs im Abstand von jeweils einer Sekunde gestartet, um das System nicht durch einen gleichzeitigen Start aller 16 Jobs auf zu ungewöhnliche Weise zu belasten. Für die folgenden Messungen wurden auf der CRAY X-MP mit der Parameterdatei in Tabelle 5.12 die Kernfolgen für die Arbeitslast generiert.

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
04	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	01,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	02,	10,	30,	050,	200,	000,	...	1,	000,	000,	1
04	03,	10,	30,	050,	200,	000,	...	1,	000,	000,	1

Tabelle 5.12: Parameter für die Referenzmessung C38M3 mit rechenintensiver Arbeitslast und sequentieller Ausführung der Testjobs

In der Messung C38M3 werden die acht Testjobs BP009-016 sequentiell ausgeführt, während die Testjobs in der Messung C38M4 parallel ausgeführt werden. Die Parallelität der Testjobs ist, wie oben angegeben, optimal, d.h. der Parallelisierungsgrad beträgt PG=100% und die *load balance* beträgt LB=100%; die parallelen Jobs können N=4 Prozessoren für die Ausführung nutzen (siehe Tabelle 5.13).

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
04	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	01,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	02,	10,	30,	050,	200,	000,	...	1,	**100**,	**100**,	4
04	03,	10,	30,	050,	200,	000,	...	1,	**100**,	**100**,	4

Tabelle 5.13: Parameter für die Messung C38M4 mit acht parallelen Testjobs

Meßergebnisse Der Benchmark-Plot der sequentiellen Referenzmessung C38M3 wird in Abbildung 5.5 a) und der der Messung C38M4 mit parallelen Testjobs in der Abbildung 5.5 b) wiedergegeben. Die Analyse dieser Meßwerte liefert wie auf der Alliant nicht nur Ergebnisse über die Verarbeitung von parallelen Programmen im Mehrprogrammbetrieb, sondern auch über die Abrechnung der CPU-time und System-Time:

- Durch die Parallelisierung der Testjobs steigt die Bench-time in der Messung C38M3 um 5.72 Sekunden an, was einem Anstieg der Total-time um 22.96 Sekunden entspricht. Im Verhältnis zur CPU-time der Testjobs bei sequentieller Ausführung von 560.6 Sekunden beträgt der Anstieg der Total-time 4%. Dieser Anstieg entspricht im Rahmen der Meßgenauigkeit ungefähr dem Anstieg von 3.6%, der im Einprogrammbetrieb gemessen wurde.
- Die Real-time der parallelen Testjobs von 2269.32 Sekunden wurde gegenüber der Ausführungsdauer der sequentiellen Testjobs von 2285.33 Sekunden praktisch nicht reduziert. In einer ausreichend großen Arbeitslast, in denen für jeden Prozessor ein Programm zur Ausführung bereit steht, werden parallele Programme nicht schneller ausgeführt als sequentielle Programme. Die parallelen Programme bekommen nur zusätzliche Prozessoren zugeteilt, wenn die Prozessoren anderenfalls leerlaufen würden. Diese Strategie entspricht dem *dynamic partitioning* in Abschnitt 2.2.2.
- Der Anstieg der Bench-time bzw. der Total-time, der durch die Parallelsierung der Testjobs entsteht, wird weder in der CPU-time noch in der System-time abgerechnet.

Eine Untersuchung des Parallelsierungsaufwands ist auf der CONVEX C3840 also nicht an Hand der durch den Systemaufruf *time* zurückgegebenen CPU-time bzw. System-time möglich, da dieser den entstehenden Aufwand nicht abrechnet. Der Anstieg der Bench-time ist jedoch reproduzierbar und wird im folgenden in Abhängigkeit von der Anzahl der parallelen Programme in der Arbeitslast untersucht.

Variable Anzahl von parallelen Jobs Als Nachweis für den Aufwand, den die Ausführung paralleler Programme im Mehrprogrammbetrieb verursacht, wird in

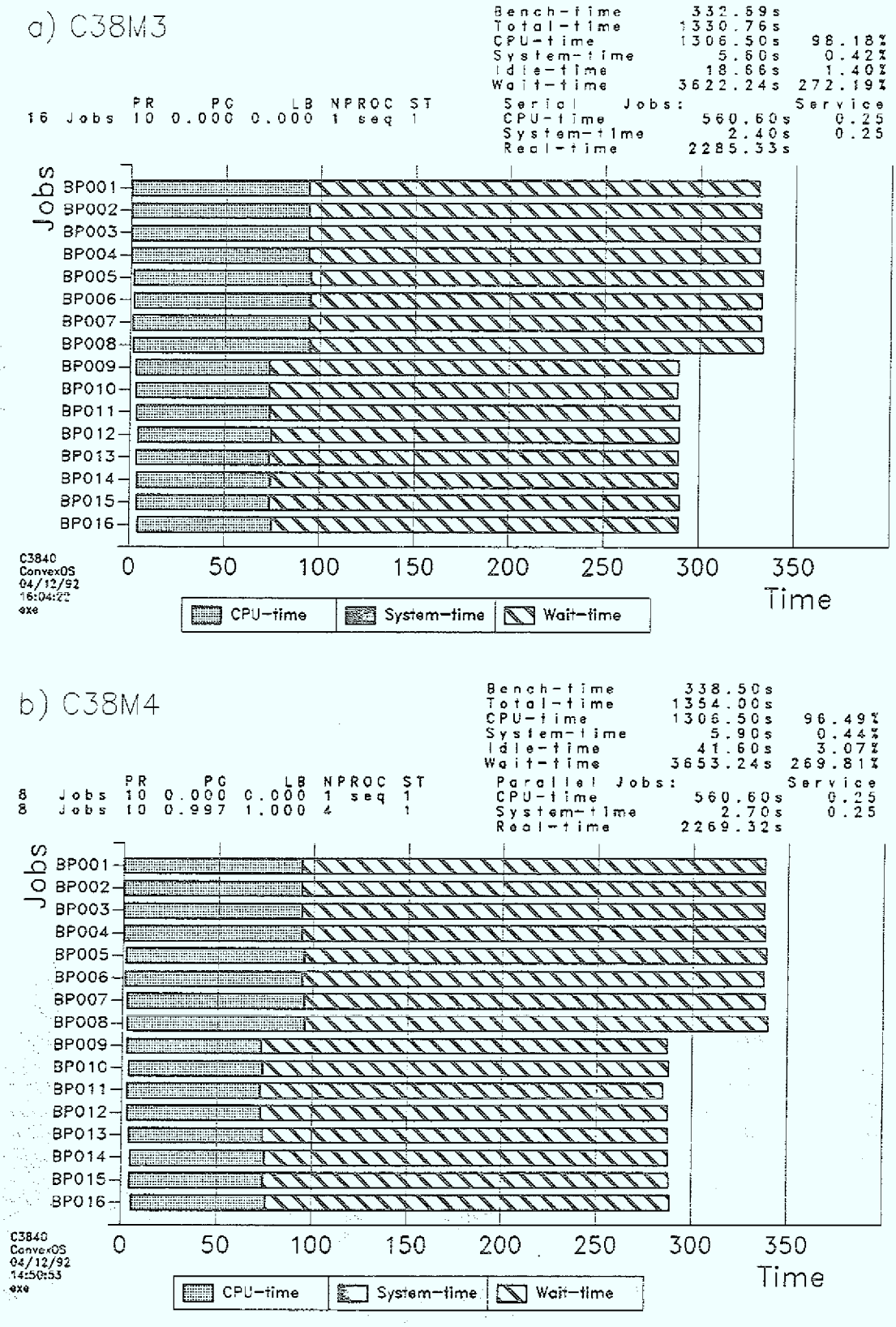


Abbildung 5.5: Benchmark-Plot der Messung C38M3 (a) und C38M4 (b)

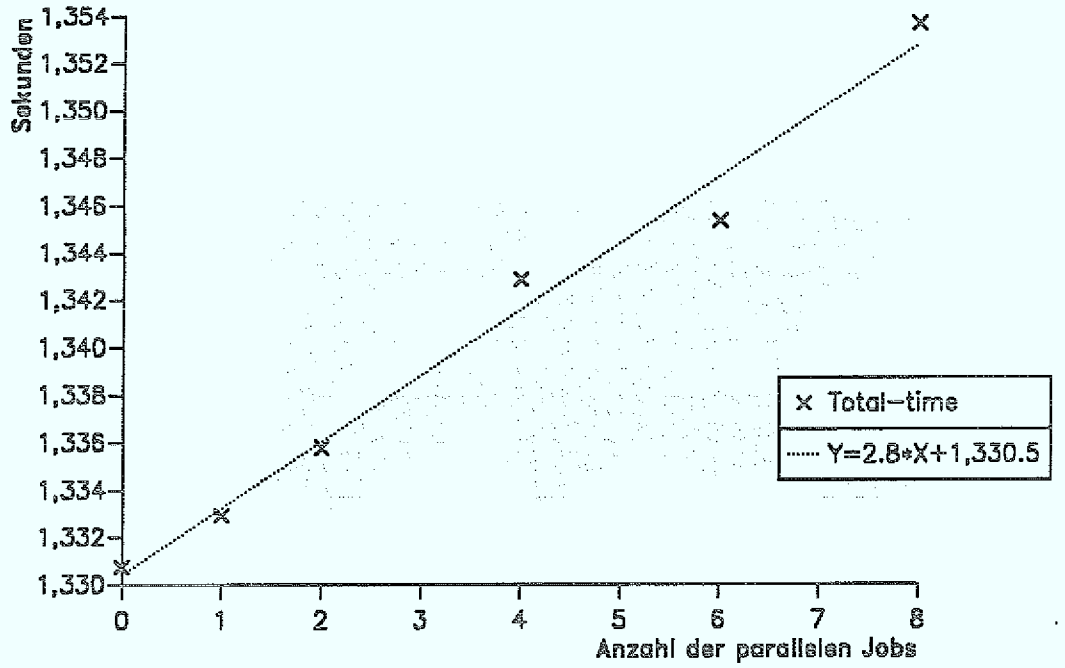


Abbildung 5.6: Anstieg der Total-time in Abhängigkeit von der Anzahl der parallelisierten Testjobs

den folgenden Messungen der Anteil der parallelen Programme an der Arbeitslast variiert. In den Messungen C38M4a-d werden nicht acht parallele Testjobs wie in der Messung C38M4 sondern nur einer, zwei, vier oder sechs der Testjobs parallelisiert. Die parallelen Jobs besitzen dabei einen Parallelisierungsgrad von PG=100% und eine *load balance* von LB=100%. Die bei diesen Messungen erhaltenen Ergebnisse werden in der Tabelle 5.14 wiedergegeben und in Abbildung 5.6 veranschaulicht.

Messung	par. Jobs	Bench-time [Sek.]	Total-time [Sek.]	CPU-time [Sek.]	System-time [Sek.]	Idle-time [Sek.]
C38M3	0	332.7	1330.8	1306.5	5.6	18.7
C38M4a	1	333.2	1332.9	1306.5	5.6	20.8
C38M4b	2	333.9	1335.8	1306.5	5.6	23.7
C38M4c	4	335.7	1342.9	1306.7	5.6	30.1
C38M4d	6	336.3	1345.4	1306.3	5.6	33.5
C38M4	8	338.4	1353.7	1306.5	5.6	33.5

Tabelle 5.14: Meßergebnisse der Messungen C38M4a-d mit einer variablen Anzahl von parallelisierten Jobs

Meßergebnisse Bei den Messungen wurde in unregelmäßigen Abständen eine erhöhte Idle-time registriert. Diese Störungen waren jedoch nicht systematisch und konnten auch nicht reproduziert werden. Es wurden deshalb mehrere Meßreihen durchgeführt und der jeweils beste Meßwert als Meßergebnis verwendet. Der Korrelationskoeffizient für die Abhängigkeit der Total-time von der Anzahl der parallelen Jobs in der Arbeitslast beträgt für diese Meßwerte $r = 0.992$. Die graphische Darstellung in Abbildung 5.6 legt die Vermutung nahe, daß zwischen Total-time und der Anzahl der parallelen Jobs ein linearer Zusammenhang gegeben ist. Eine lineare Regression ergibt die Geradengleichung:

$$\text{Total-time} = 2.8\text{s} * \text{Anzahl der parallelisierten Jobs} + 1330.5\text{s} \qquad (5.1)$$

Der Aufwand von 2.8 Sekunden, der durch die Parallelisierung eines Jobs entsteht, stimmt mit dem für den Einprogrammbetrieb ermittelten Wert von 2.7 Sekunden im Rahmen der Meßgenauigkeit überein. Der Aufwand, der im Mehrprogammbetrieb durch die sequentielle Ausführung eines parallelisierten PAR-Bench-Jobs verursacht wird, entspricht also dem Aufwand, der bei der parallelen Ausführung des Jobs im Einprogrammbetrieb entsteht.

5.2.2 Arbeitslasten mit niedriger Parallelität

Aufgrund des dynamischen Schedulings können die Leerlaufzeiten in parallelen Programmen mit niedrigem Parallelisierungsgrades bzw. schlechter *load balance* auf der CONVEX C3840 für die Bearbeitung von anderen Jobs genutzt werden. Das Scheduling der CONVEX nach dem Prinzip des *dynamic partitioning* verhindert die parallele Ausführung von Programmen in Arbeitslasten mit einer hohen Anzahl von ausführbereiten Jobs. Dadurch werden die bei Parallelverarbeitung entstehenden Leerlaufzeiten vermieden. In der folgenden Messungen wird überprüft, wie gut die Umsetzung dieser Konzepte auf der CONVEX C3840 gelungen ist.

In der Messung C38M10c wird der Scheduler des Betriebssystems ConvexOS anhand einer Arbeitslast untersucht, in der die parallelen Testjobs einen niedrigen Parallelisierungsgrad von PG=70% besitzen. Die *load balance* beträgt bei dieser Messung LB=100% ebenso wie in der Messung C38M4 (siehe Tabelle 5.15).

Meßergebnisse In Abbildung 5.7 b) wird der Benchmark-Plot der Messung C38M10c dem der Messung C38M4 mit vollständig parallelen Jobs in Abbildung 5.7 a) gegenübergestellt. Im Gegensatz zur Alliant ist auf der CONVEX kein Anstieg der Bench-time bzw. der Total-time, sondern ein geringfügiger Rückgang der Total-time erkennbar. Dieser Rückgang kann auf die geringere Anzahl von parallelen Regionen zurückgeführt werden, die ein paralleler PAR-Bench-Job mit niedrigem Parallelisierungsgrad ausführt. Da parallele Programme mit niedriger Parallelität auf der CONVEX im Mehrprogrammbetrieb keine Leistungseinbußen verursachen,

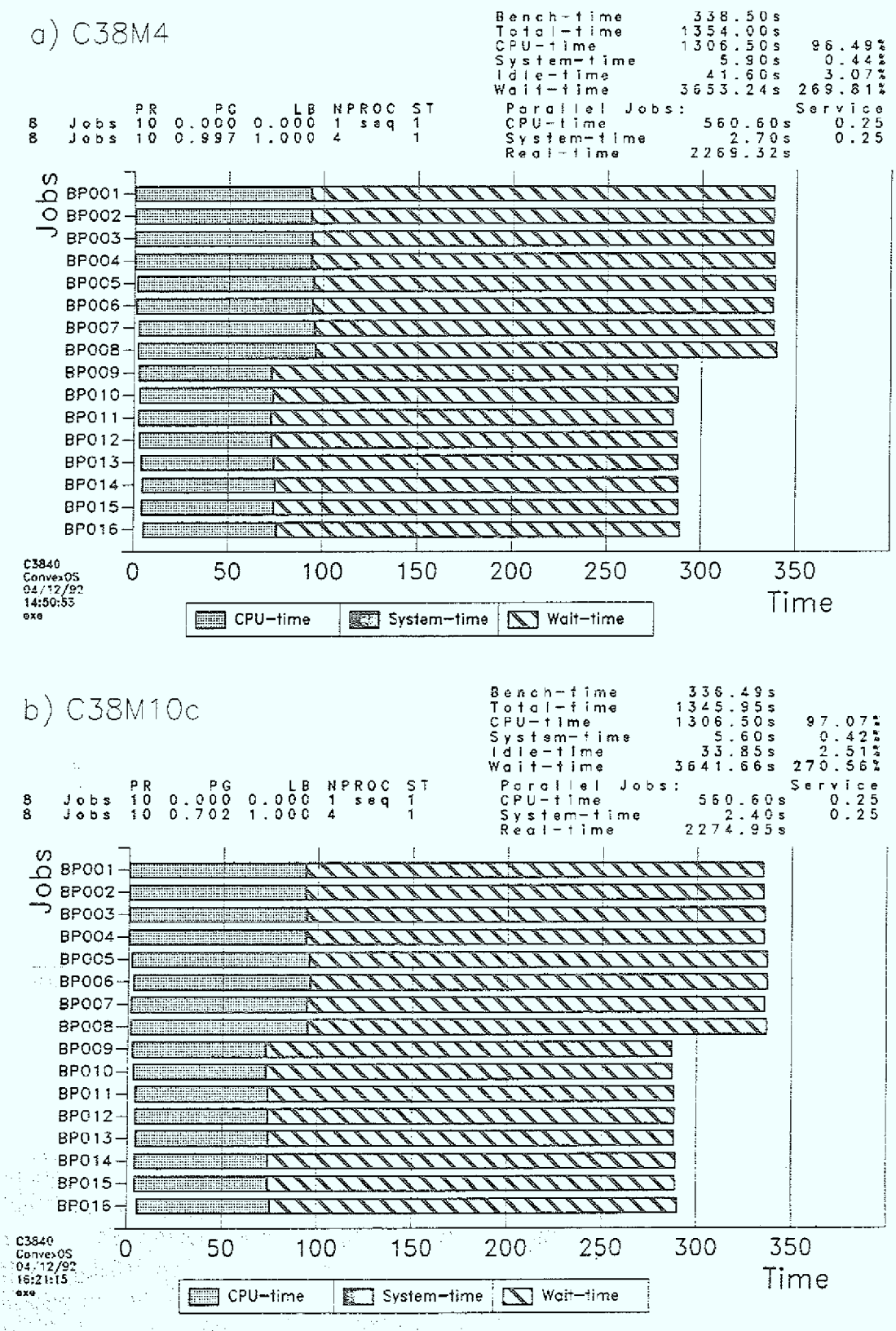


Abbildung 5.7: Benchmark-Plot der Messung C38M4 (a) und C38M10c (b)

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
04	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	01,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	02,	10,	30,	050,	200,	000,	...	1,	**070**,	**100**,	4
04	03,	10,	30,	050,	200,	000,	...	1,	**070**,	**100**,	4

Tabelle 5.15: Parameter für die Messung C38M10c - Parallele Testjobs mit einem Parallelisierungsgrad von PG=70%

wird im folgenden der zusätzliche Aufwand bestimmt, der durch das dynamische Scheduling verursacht wird.

Scheduling-Aufwand Der Aufwand, der im Mehrprogrammbetrieb bei der Ausführung von parallelen Programmen entsteht, kann in zwei Komponenten zerlegt werden. Zum einen kann für einen parallelen Job ein konstanter Aufwand (*startup*) anfallen, der benötigt wird, um die Parallelverarbeitung vorzubereiten. Zum anderen entsteht bei der Ausführung jeder parallelen Region ein Aufwand, der durch die Anforderung von zusätzlichen Prozessoren verursacht wird. Die Anzahl der parallelen Regionen, die von einem PAR-Bench-Job ausgeführt werden, kann für eine feste Arbeitslast nur über den Parameter Parallelisierungsgrad variiert werden. Der Parallelisierungsgrad eines Jobs wird von PAR-Bench dadurch approximiert, daß der nächste auszuführende Kern in Abhängigkeit vom bisher erreichten Parallelisierungsgrad sequentiell oder parallel ausgeführt wird. Je höher der Parallelisierungsgrad eines Jobs ist, desto mehr parallele Regionen führt dieser aus. Jeder der parallele ausgeführten Kerne enthält genau eine parallele Schleife, also genau eine parallele Region. Die Kernfolgen von Jobs mit gleichen dynamischen Parametern wie z.B. die Testjobs BP009-017 in Messung C38M10c sind jedoch nicht identisch. Deshalb ist die Anzahl der parallel ausgeführten Regionen in Abhängigkeit vom Parallelisierungsgrad gewissen Schwankungen unterworfen, die jedoch weniger als 10% vom Durchschnittswert abweichen. Die Anzahl der parallelen Regionen, die ein Job ausführt, werden während der Messung in der Protokoll-Datei des Jobs festgehalten. So wurden von einem Testjob in den Messungen C38M2d* und C38M4 mit einem Parallelisierungsgrad von PG=100% durchschnittlich 305 $\pm$ 0 parallele Regionen und von einem Testjob in Messung C38M10c mit PG=70% durchschnittlich 194 $\pm$ 2 parallele Regionen ausgeführt.

Im folgenden wird der Parallelisierungsaufwand in Abhängigkeit von der Anzahl der parallelen Regionen bestimmt. Dazu wird in den Messungen C38M10a-d die Anzahl der parallelen Regionen mit Hilfe des Parameters Parallelisierungsgrad variiert. Die Messungen werden mit acht parallelen Testjobs durchgeführt, um eine Verstärkung des Effekts zu erreichen. Das ist notwendig, da die Genauigkeit der Messungen nicht hoch genug ist, um den Unterschied in der Ausführungsdauer zu bestimmen, der durch die Variation des Parallelisierungsgrades eines einzigen Jobs verursacht

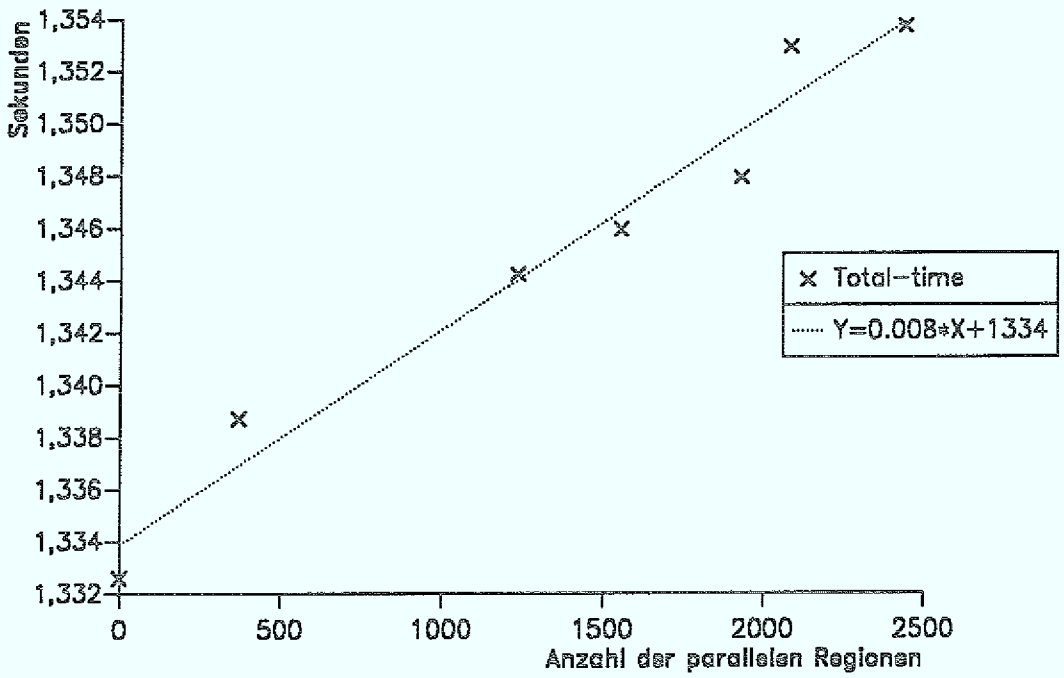


Abbildung 5.8: Anstieg der Total-time in Abhängigkeit von der Anzahl der parallelen Regionen

wird. Für alle acht Testjobs einer Arbeitslast wird derselbe Parallelisierungsgrad vorgegeben. Der Parameter *load balance* wird auf LB=100% beibehalten. Die Meßergebnisse, die bei den Messungen mit einer unterschiedlichen Anzahl von parallelen Regionen erhalten wurden, werden in der Tabelle 5.16 wiedergegeben und in der Abbildung 5.8 graphisch dargestellt.

Messung	par. Reg.	Bench-time [Sek.]	Total-time [Sek.]	CPU-time [Sek.]	System-time [Sek.]	Idle-time [Sek.]
C38M3	0	332.7	1330.8	1306.5	5.6	18.7
C38M10a	1236	336.0	1344.2	1306.5	5.6	32.0
C38M10b	1555	336.5	1345.9	1306.5	5.6	33.8
C38M10c	1930	336.9	1347.9	1306.5	5.6	35.8
C38M10d	2084	338.2	1352.9	1306.5	5.6	40.8
C38M4	2440	338.4	1353.7	1306.5	5.6	41.7

Tabelle 5.16: Meßergebnisse der Messungen C38M10a-d mit einer variablen Anzahl von parallelen Regionen

Meßergebnisse Die Total-time steigt in den Messungen monoton mit der Anzahl der parallelen Regionen und zeigt ein annähernd lineares Verhalten. Die Linearität ist jedoch nicht so deutlich erkennbar wie in Abbildung 5.6. Eine lineare Regression ergibt eine Korrelation von $r = 0.95$ und die folgenden Gleichung der Regressionsgeraden:

$$\text{Total-time} = 8.2\text{ms} * \text{Anzahl der parallelen Regionen} + 1333.2\text{s} \qquad (5.2)$$

Damit ergibt sich im Mehrprogrammbetrieb eine Aufwand von ca. 8 ms für jede parallele Region. Die Genauigkeit der Messung ist jedoch aufgrund der festen Granularität der Par-Bench-Jobs und der Varianz bei der Durchführung der notwendigen Ein-/Ausgabeoperationen nicht beliebig hoch. Der konstante Aufwand der parallelisierten Jobs ergibt sich aus der Differenz des für acht sequentielle Testjobs gemessen Werts von 1330.76 s und dem Wert der Regressionsgeraden für $X = 0$ von 1333.2 s. Die Differenz von 2.44 s für acht Jobs ergibt einen Aufwand von 0.3 s für einen parallelen Job. Dieser Wert stellt jedoch nur eine obere Grenze für den tatsächlichen Aufwand dar. Im Initialisierungs- und Auswertungsteil befinden sich einige weitere Schleifen. Da diese nicht explizit als nicht parallelisierbar gekennzeichnet sind, können einige der Schleifen durch den FORTRAN-Compiler parallelisiert werden. Dadurch kommt für jeden parallelisierten Job von PAR-Bench ein konstanter Aufwand hinzu.

5.2.3 Arbeitslasten mit hoher Speicherbelastung

In den vorhergehenden Untersuchungen wurden nur Arbeitslasten mit rechenintensiven Jobs untersucht. Bei der Alliant FX/2816 zeigte sich, daß die Belastung des gemeinsamen Hauptspeichers so bedeutend für die Durchsatzleistung ist, daß sie bei der Parallelisierung berücksichtigt werden muß. Die rechenintensiven Jobs aus der Messung C38M1 erreichten nur eine Speicherbelastung von ca. 20 MMREFS pro sequentiellen Job. Bei einer Nutzung aller vier Prozessoren durch eine solche Arbeitslast beträgt die Belastung des Speicher durch die rechenintensiven Jobs 80 MMREFS, das sind ca. 32% der maximalen Speicherbandbreite der CONVEX C3840 mit 18ns. Die Belastung des gemeinsamen Speichers durch die rechenintensiven Jobs ist also vergleichsweise niedrig. Für die folgenden Messungen werden daher neue Arbeitslasten auf der CRAY X-MP/416 generiert, die eine erheblich höhere Belastung des gemeinsamen Hauptspeichers erreichen. Diese speicherintensiven Jobs sollen auch für die CRAY X-MP und die CRAY Y-MP eine akzeptable Belastung des gemeinsamen Hauptspeichers erreichen. Die neuen speicherintensiven Arbeitslasten wurden auf der CRAY X-MP mit den Parametern MM = 180 MMREFS und FL = 130 MFLOPS generiert. Das Verhältnis von Speicherzugriffen zu Rechenoperationen beträgt damit ungefähr 3 zu 2 und stellt auch für die CRAY-Rechner eine hohe Belastung des gemeinsamen Speichers dar.

Einprogrammbetrieb Zur Bestimmung des durch Zugriffsverzögerungen verursachten Aufwands werden die ersten Messungen im Einprogrammbetrieb durchgeführt. In der Messung C38M20a* wird dazu ein speicherintensiver Testjob mit den oben angegebenen Parametern im Einprogrammbetrieb ausgeführt. Diese Messung wird ergänzt durch die Messungen C38M20b-d* in der zwei, drei bzw. vier identische speicherintensive Jobs ausgeführt werden. Die dabei erzielten Ergebnisse werden in der Tabelle 5.17 wiedergegeben.

Messung	Jobs	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	Real-time [Prozent]	Effizienz [Prozent]
C38M21a*	1	108.2	100.0	109.4	100.0	100.0
C38M21b*	2	109.0	100.1	110.4	100.9	99.1
C38M21c*	3	112.4	103.9	114.0	104.2	96.0
C38M21d*	4	125.0	115.5	128.4	117.4	85.2

Tabelle 5.17: Ausführungsdauer eines speicherintensiven Jobs auf der C3840 mit einer Taktperiode von 18 ns in Abhängigkeit von der Anzahl der Jobs bei sequentieller Ausführung

Aus der Tabelle 5.17 ist ersichtlich, daß die hohe Speicherbelastung durch die Testjobs zu einem erheblich höheren Anstieg der Real-time und der CPU-time führen. Der Anstieg der Real-time eines Jobs beträgt in der Messung C38M21d* 19.0 Sekunden bzw. 17.4% der Real-time. Der Anstieg durch Zugriffsverzögerungen kann jedoch mit größerer Meßgenauigkeit am Anstieg der CPU-time gemessen werden, da die abgerechnete Prozessorzeit den Zeitraum umfaßt in der die Speicherzugriffe erfolgen. Die Zugriffsverzögerungen im Mehrprogrammbetrieb mit vier parallelen Testjobs verursachen einen Anstieg der CPU-time um 16.8 Sekunden, was 15.5% der CPU-time eines Testjobs entspricht. Ein sequentieller Job erreichte dabei eine Speicherleistung von ca. 50 MMREFS, was bei Nutzung von 4 Prozessoren 200 MMREFS und somit 80% der maximalen Speicherbandbreite der CONVEX C3840 mit 18ns entspricht. In den Messungen C38M22a-d* wird der speicherintensive Testjob aus Messung C38M20a* parallelisiert und mit einem, zwei, drei und vier Prozessoren parallel ausgeführt, indem der Parameter N von eins bis vier variiert wird (siehe Tabelle 5.18).

Der Anstieg der CPU-time der parallelen Jobs verhält sich in Abhängigkeit von der Anzahl der parallelen Testjobs genauso wie der Anstieg der CPU-time in Abhängigkeit von der Anzahl der sequentiellen Jobs. Dabei ist die CPU-time der parallelen Testjobs durchgehend um ca. 0.5 Sekunden niedriger als bei den sequentiellen Testjobs. Da der Parallelisierungsaufwands jedoch nicht in der CPU-time abgerechnet wird, kann diese Differenz auch ausschließlich durch die Abrechnung verursacht werden. In [vdPvK91] wird die Nutzung der gemeinsamen Register in parallelen Programmen als mögliche Ursache für diesen Effekt genannt. In beiden Fällen ist ein Anstieg der CPU-time um ca. 15% aufgrund von Zugriffsverzögerungen zu beobachten.

Messung	N	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
C38M22a*	1	107.7	99.6	109.0	1.00	100.4%
C38M22b*	2	108.7	100.5	55.2	1.98	99.1%
C38M22c*	3	111.8	103.3	38.0	2.88	96.0%
C38M22d*	4	124.6	115.2	32.1	3.41	85.2%

Tabelle 5.18: Effizienz und *speedup* eines speicherintensiven Jobs auf der CONVEX C3840 mit einer Taktperiode von 18 ns in Abhängigkeit von der Zahl der genutzten Prozessoren N

Mehrprogrammbetrieb Im Mehrprogrammbetrieb kann durch die parallele Ausführung von speicherintensiven Programmen eine höhere Belastung des Hauptspeichers und somit ein Anstieg der Ausführungsdauer verursacht werden. Da das Betriebssystem ConvexOS parallele Jobs bei einer ausreichend hohen Hintergrundlast nur sequentiell ausführt, sollte im Mehrprogrammbetrieb durch die Parallelisierung eines Jobs kein Anstieg der CPU-time verursacht werden. Für diese Messungen wird eine neue Arbeitslast verwendet, die im folgenden beschrieben wird.

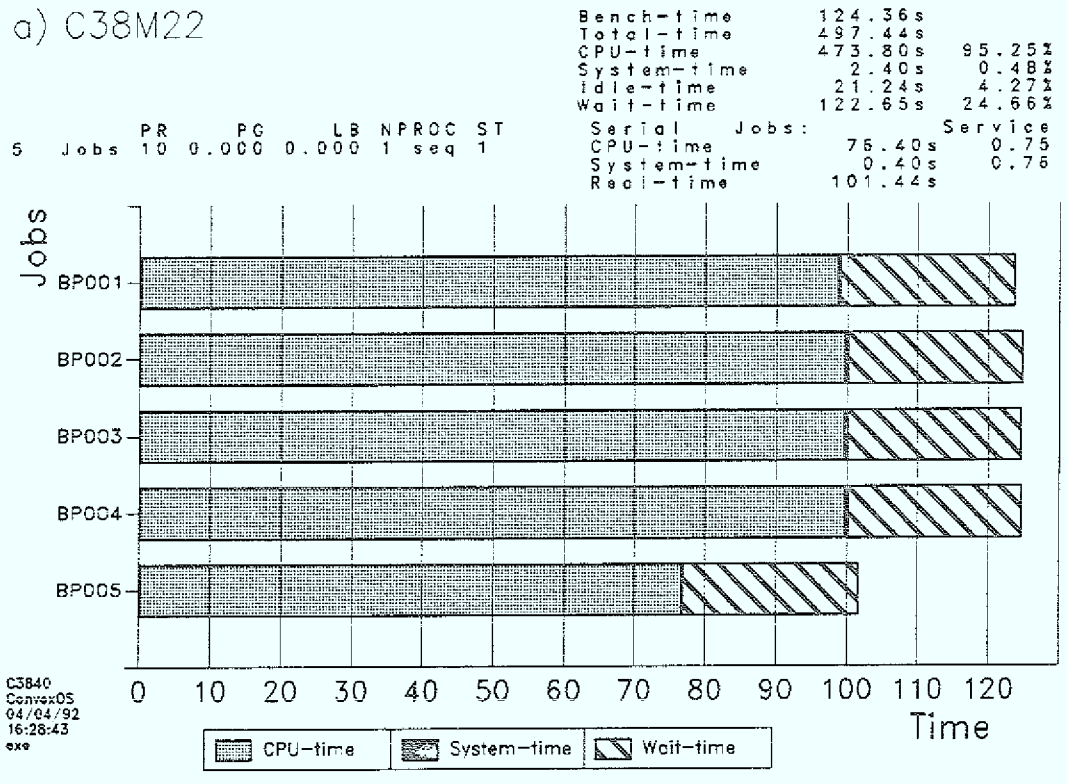
- Die Arbeitslast besteht aus vier rechenintensiven Hintergrundjobs BP001-004, die sequentiell ausgeführt werden. Die dynamischen Parameter dieser Hintergrundjobs entsprechen den rechenintensiven Jobs aus den vorangegangen Abschnitten, d.h. FL = 200 MFLOPS und MM = 50 MMREFS.

- Es wird ein speicherintensiver Testjobs BP005 in der Arbeitslast ausgeführt, dessen dynamische Parameter denen der Messung C38M20a-d* und C38M21a-d* entsprechen, d.h. FL = 130 MFLOPS und MM = 180 MMREFS. Die Prozessorzeit der Jobs wird auf 20 Sekunden der CRAY X-MP angesetzt, da speicherintensive Jobs auf der C3840 im Verhältnis zur X-MP/416 langsamer ausgeführt werden als rechenintensive Jobs.

- Der Testjobs wird bei der Parallelisierung optimal parallel ausgeführt, wobei eine Parallelisierungsgrad von PG=100%, eine *load balance* von LB=100% und N=4 Prozessoren vorgegeben werden.

In der Messung C38M22* wird der speicherintensive Testjob BP005 als sequentielles Programm ausgeführt. Die Parameter der Messung C38M22* werden in der Tabelle 5.19 wiedergegeben.

In der Messung C38M23* wird der speicherintensive Testjob BP005 gemäß den oben angegebenen Vorgaben parallelisiert. Die Parameter der Messung C38M23* werden in der Tabelle 5.20 wiedergegeben.



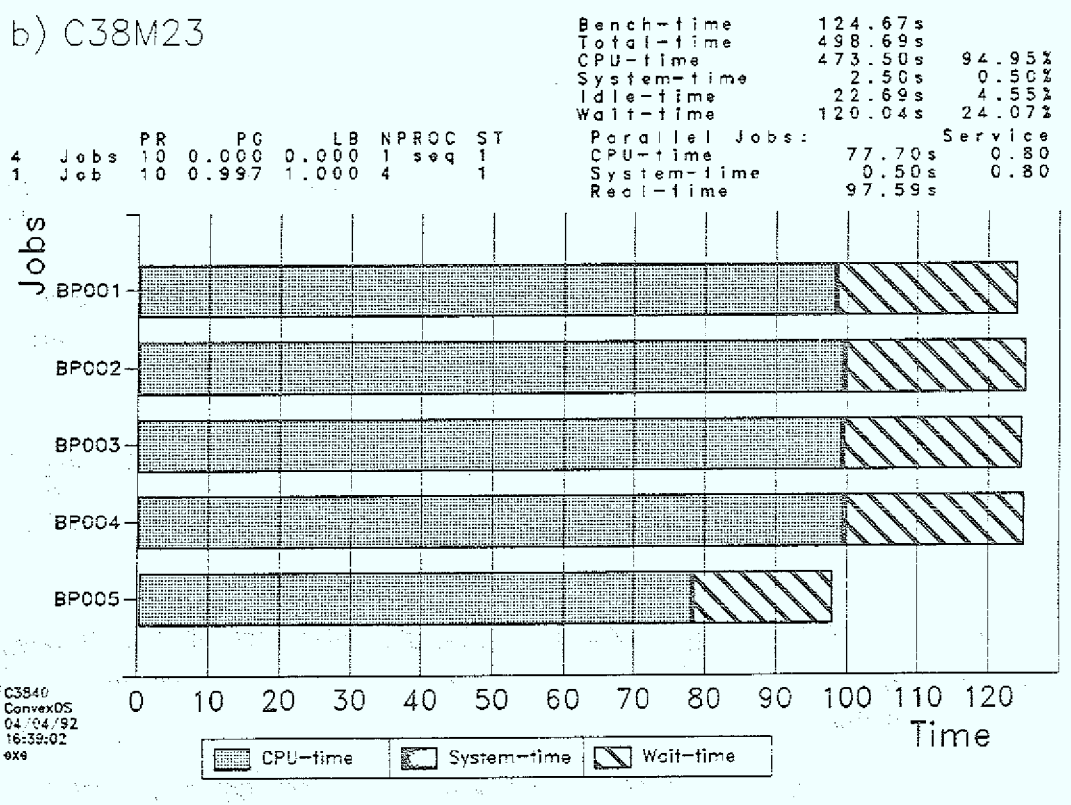


Abbildung 5.9: Benchmark-Plot der Messung C38M22* (a) und C38M23* (b)

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
4	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
1,	01,	10,	**20**,	180,	130,	000,	...	1,	000,	000,	1

Tabelle 5.19: Parameter für die Referenzmessung C38M22* mit rechenintensiver Arbeitslast und sequentieller Ausführung des speicherintensiven Testjobs

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
4	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
1,	01,	10,	20,	180,	130,	000,	...	1,	**100**,	**100**,	**4**

Tabelle 5.20: Parameter für die Referenzmessung C38M22* mit rechenintensiver Arbeitslast und paralleler Ausführung des speicherintensiven Testjobs

Meßergebnisse Die Abbildung 5.9 zeigt den Benchmark-Plot der Messung C38M22* (a) über dem Benchmark-Plot der Messung C38M23* (b). Die Real-time des Testjobs BP005 ist in der parallelen Version um 3.85 Sekunden bzw. 3.8% gegenüber der sequentiellen Version verkürzt. Der dadurch verursachte Anstieg des *service*-Faktors von 0.75 auf 0.8 ist jedoch gering und ist kein hinreichendes Indiz für Parallelverarbeitung. Die CPU-time des parallelisierten Testjobs ist jedoch um 1.3 Sekunden bzw. 1.7% gegenüber der CPU-time des sequentiellen Testjobs erhöht. Da in den bisherigen Messungen kein Anstieg der CPU-time durch Parallelisierung nachweisbar war, ist dieser Anstieg nur auf eine Zunahme von Zugriffskonflikten bei der Ausführung des Testjobs zu erklären. Dies ist jedoch nur möglich, wenn Testjobs trotz ausreichender Arbeitslast zum Teil parallel ausgeführt wurde. Diese Annahme wird dadurch bestätigt, daß die CPU-time der Hintergrundlast durch die Parallelisierung des Testjobs reduziert wird, so daß die CPU-time der Arbeitslast in der Messung C38M23* sogar minimal unter der CPU-time der Arbeitslast in der Messung C38M22* liegt. Ein zusätzlicher Aufwand bei der Parallelisierung von speicherintensiven Jobs läßt sich also für die CONVEX C3840 nicht nachweisen.

5.2.4 Zusammenfassung der Meßergebnisse

Die Messungen auf der CONVEX C3840 geben Aufschluß über einige wesentliche Aspekte der Parallelverarbeitung auf der CONVEX C3840 mit dem Betriebssystem ConvexOs. Diese Ergebnisse zeigen, daß der Mehrprogrammbetrieb paralleler Programme bei der Architektur der C3840 und in ConvexOs berücksichtigt wurden. Die Ergebnisse lassen sich wie folgt zusammenfassen:

- Bei der Ausführung von parallelen rechenintensiven PAR-Bench-Jobs wird im Einprogrammbetrieb eine Effizienz von 94.7% erreicht. Der durch die Parallelisierung entstehende Aufwand wird jedoch weder in der CPU-time noch in der System-time abgerechnet. Dadurch entsteht der Eindruck, daß die Ausführung von parallelen Programmen keine zusätzlichen Kosten verursacht, was aber nicht stimmt.

- Parallele Jobs werden im Mehrprogrammbetrieb nicht bevorzugt ausgeführt und üblicherweise wie sequentielle Jobs behandelt. In der Messung C38M23* ist für den Job BP005 ein geringes Maß an Parallelverarbeitung nachweisbar. Der zusätzliche Aufwand, der durch ein paralleles Programm im Mehrprogrammbetrieb entsteht, entspricht dem Aufwand, der im Einprogrammbetrieb gemessen wird.

- Das dynamische Scheduling bzw. das *dynamic partitioning* der CONVEX führt zu keinen Leerlaufzeiten bei Programmen mit geringer Parallelität. Der zusätzliche Aufwand, der im Mehrprogrammbetrieb durch den Scheduler verursacht wird, ist gering und beträgt für einen vollständig parallelen Benchmark-Job durchschnittlich ca. 5%. Der Aufwand für die Ausführung einer parallelen Region konnte auf 8 ms quantifiziert werden.

- Im Mehrprogrammbetrieb mit ausreichender Hintergrundlast entsteht durch die Parallelisierung von speicherintensiven Testjobs kein zusätzlicher Aufwand, der über den Aufwand hinaus geht, der durch die Parallelisierung rechenintensiver Jobs entsteht.

Der Ausführung paralleler Programme also führt auf der CONVEX C3840 zu keiner wesentlichen Reduzierung des Durchsatzes. Die parallele Ausführung von Jobs führt zu einem geringen, aber nachweisbaren zusätzlichen Aufwand. Dabei werden die parallele Programme im Mehrprogrammbetrieb mit hoher Arbeitslast nicht gegenüber den sequentiellen Programmen bevorzugt.

5.3 Messungen auf der CRAY X-MP/416

Die CRAY X-MP/416 zeichnet sich durch ein im Verhältnis zur Rechenleistung außergewöhnlich leistungsfähiges Speichersystem aus. Die Vektorregister eines Prozessors werden über die drei *ports* eines jeden Prozessors ausschließlich mit dem verschränkten Hauptspeicher ausgetauscht. In der CRAY X-MP Architektur werden im Gegensatz zur CONVEX C3840 für Skalare und Vektoren ausschließlich Register und kein Cache-Speicher verwendet. Der Speichertransfer erfolgt also sowohl bei den Vektor- als auch bei den Skalaroperationen immer direkt zwischen den Registern und dem gemeinsamem Hauptspeicher.

Die Hauptspeicherkapazität der CRAY X-MP/416 von 16 MWorten ist für heutige Supercomputer gering und kann bei Arbeitslasten mit hohem Speicherplatzbedarf zu Engpässen führen. Dieses Problem wird dadurch verschärft, daß die Rechner der X-MP-Serie einen realen Hauptspeicher in Verbindung mit einem UNIX-kompatiblen Betriebssystem verwenden. Bei der Ausführung eines neuen Kommandos wird mit Hilfe einer *fork*-Anweisung eine Kopie der aktuellen *shell* erzeugt. Die Ausführung eines UNIX-Kommando führt deshalb zu einer beachtlichen Verringerung des freien Hauptspeicherplatzes. Bei einem Start der Jobs im Hintergrund werden jedoch zusätzliche UNIX-Kommandos zur Zeitmessung benötigt. Deshalb kann auf der CRAY X-MP/416 keine Arbeitslast mit sechzehn von PAR-Bench generierten Jobs im Hintergrund ausgeführt werden, ohne daß es zu einem Auslagern von Jobs kommt. Das Auslagern der Jobs auf den SSD erfolgt zwar außerordentlich schnell, führt aber zu einem zusätzlichen Aufwand, der die Meßergebnisse stark beeinflußt. Die Jobs werden aus diesem Grund mittels *qsub* durch das NQS und nicht, wie auf der Alliant FX/2816 oder der CONVEX C3840, im Hintergrund gestartet (siehe Abschnitt 4.3). Um die Meßgenauigkeit der Messungen zu erhöhen, wurde der SSD genutzt um sämtliche Dateien zwischenzuspeichern und den Einfluß der Ein-/Ausgabeoperationen auf die Meßergebnisse zu verringern. Dadurch wird die Reproduzierbarkeit der Messungen wesentlich erhöht, ohne daß die Ausführungsdauer der Arbeitslasten signifikant (maximal 0.5 Sekunden) reduziert wird.

Das Betriebssystem UNICOS 6.1 unterstützt zwei verschiedene Scheduler, die für unterschiedliche Betriebsarten optimiert sind. Der Benutzer kann mit der *environment*-Variable MP_DEDICATED den geeigneteren Scheduler auswählen. Der Scheduler 5.X entspricht dem Scheduler von UNICOS 5.1 und ist für den Einprogrammbetrieb von parallelen Programme optimiert (MP_DEDICATED=1). Im Gegensatz dazu ist der Scheduler 6.X für den Mehrprogrammbetrieb konzipiert und gibt nicht benötigte Prozessoren freiwillig wieder ab (MPDEDICATED=0). Im Mehrprogrammbetrieb wird als Voreinstellung der neue Scheduler 6.X genutzt. Deshalb werden Messungen, in denen der alte Scheduler 5.X verwendet wird, im folgenden mit dem Symbol σ explizit gekennzeichnet. Eine Untersuchung des Schedulers von UNICOS 6.0 im Vergleich zum Scheduler von UNICOS 5.1 wurde bereits mit PAR-Bench durchgeführt [LNVD91].

5.3.1 Arbeitslasten mit günstigen Eigenschaften

Für die Messungen von Arbeitslasten mit möglichst günstigen Eigenschaften werden die Lasten genutzt, die auf der CRAY X-MP/416 für die Messungen auf der CONVEX C3840 generiert wurden. Die Speicherbelastung dieser rechenintensiven Jobs mit den dynamischen Parametern FL = 200 MFLOPS und MM = 50 MMREFS ist für die CRAY X-MP mit einer im Verhältnis zur Rechenleistung erheblich höheren Speicherleistung als die CONVEX C3840 natürlich ebenfalls gering. Die Parallelisierungsparameter der parallelen Jobs betragen PG=100%, LB=100% und

N=4.

Einprogrammbetrieb Der durch die Parallelisierung eines Programms entstehende Aufwand wird zuerst wieder im Einprogrammbetrieb untersucht. Dazu wird in der Messung XM1a der rechenintensive Job aus Messung C38M1a sequentiell ausgeführt. In der Messung XM1d werden vier dieser Job gleichzeitig ausgeführt, wodurch eine Quantifizierung des Aufwand ermöglicht wird, der durch Zugriffsverzögerungen verursacht wird (siehe Tabelle 5.21).

Messung	Jobs	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	Real-time [Prozent]	Effizienz [Prozent]
XM1a	1	30.29	100.0	30.36	100.0	100.0%
XM1d	4	30.33	100.1	30.79	101.4	98.6%

Tabelle 5.21: Ausführungsdauer von rechenintensiven Jobs auf der CRAY X-MP/416 in Abhängigkeit von der Anzahl der Jobs bei sequentieller Ausführung

Durch den Mehrprogrammbetrieb wird die durchschnittliche Real-time der Jobs um ca. 1.4% erhöht, während der Anstieg der CPU-time nur ungefähr 0.1% beträgt, also faktisch gleich Null ist. Der Anstieg der Ausführungsdauer aufgrund von Zugriffskonflikten entspricht dem Anstieg der CPU-time und ist für die rechenintensiven Jobs zu vernachlässigen. In den Messungen XM2 und XM2$^{\sigma}$ wird der Job aus der Messung XM1 parallel ausgeführt. Die Parallelisierungsparameter betragen PG=100%, LB=100% und N=4. Während in Messung XM2 der für den Mehrprogrammbetrieb konzipierte Scheduler 6.X verwendet wird, kommt in Messung XM2$^{\sigma}$ der für den Einprogrammbetrieb optimierte Scheduler 5.X zum Einsatz. Die *slave hold time* des Schedulers 6.X ist in der Messung XM2 für den PAR-Bench-Job optimiert (MP_HOLDTIME=150000). Die so erhaltenen Meßergebnisse werden in Tabelle 5.22 aufgeführt.

Messung	MP_DED	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
XM2$^{\sigma}$	1	31.10	102.7	7.87	3.86	96.5%
XM2	0	30.95	102.2	7.80	3.89	97.3%

Tabelle 5.22: Effizienz und *speedup* eines rechenintensiven Jobs auf der CRAY X-MP/416

Bei Verwendung des Schedulers 5.X in Messung XM2$^{\sigma}$ wird eine Effizienz von 96.5% erreicht, während für den Scheduler 6.X eine Effizienz von 97.3% gemessen wird. Aufgrund der erheblich kürzeren Ausführungsdauer der Jobs auf den

Meßergebnisse In Abbildung 5.10 werden die Benchmark-Plots der Messung XM3 (a) mit sequentieller Ausführung der Testjobs und der Messung XM4 (b) mit parallelisierten Testjobs wiedergegeben. Eine Analyse der Meßergebnisse liefert die Resultate:

- Die Startzeitpunkte der Jobs sind in beiden Messungen gegenüber den Vorgaben in der Parameterdatei verspätet. Nach den Vorgaben der Parameterdatei sollen jeweils vier Gruppen mit jeweils 4 Jobs mit einer Verzögerung von jeweils 1 Sekunde nach jeder Gruppe gestartet werden. In den Benchmark-Plots der Messungen XM3 und XM4 in Abbildung 5.10 ist jedoch deutlich zu erkennen, daß sich die Startzeitpunkte der Jobs auf einen Zeitraum von ca. 16 Sekunden statt den vorgegebenen 3 Sekunden erstrecken. Diese Verzögerung wird durch das NQS der CRAY-Systeme verursacht und zeigt sich nicht in späteren Messungen auf der CRAY Y-MP8/832, die ohne NQS durchgeführt werden.

- Die Total-time ist in der Messung XM4 mit den parallelen Testjobs um 9.56 Sekunden höher als in der Messung XM3. Die Meßgenauigkeit der Total-time ist jedoch erheblich geringer als die Meßgenauigkeit der CPU-time. Die Varianz der Total-time beträgt ungefähr 4 Sekunden, während die Varianz der CPU-time unter 1 Sekunde liegt. Als mögliche Ursache für die hohe Varianz der Total-time kann das NQS in Betracht gezogen werden.

- Die CPU-time der Arbeitslast steigt bei der parallelen Ausführung der Testjobs um 8.57 Sekunden an. Da dieser Wert im Rahmen der Meßgenauigkeit dem Anstieg der CPU-time der parallelen Testjobs von 8.56 Sekunden entspricht, kann er eindeutig auf die Parallelisierung zurückgeführt werden. Der Anstieg der CPU-time beträgt 3.5% der CPU-time der Testjobs und ist damit geringfügig höher als der Anstieg von 2.2%, der im Einprogrammbetrieb gemessen wird. Der Anstieg der System-time um 1.05 Sekunden indiziert einen leicht erhöhten Systemaufwand. Die Summe von beiden ergibt einen Aufwand von 9.61 Sekunden, das sind 3.9% des Aufwands der acht parallelen Testjobs.

- Der *service*-Faktor von 0.41 der parallelen der Testjobs ist gegenüber dem Wert von 0.25 bei den sequentiellen Testjobs deutlich erhöht. Die parallelen Testjobs werden also bevorzugt ausgeführt. Da den acht parallelen Testjobs der Arbeitslast nur vier Prozessoren der CRAY X-MP/416 gegenüberstehen, kann bei einer gleichmäßigen Verteilung der Prozessorzeit auf alle parallelen Testjobs maximal ein *service*-Faktor von 0.5 erzielt werden. Ein solcher *service*-Faktor kleiner als eins läßt sich sowohl durch die parallele als auch durch die sequentielle Ausführung der Jobs erreichen. Aus der Protokolldatei der parallelen Jobs geht hervor, daß die Jobs in geringem Maß parallel ausgeführt wurden. Im Mittel werden jedem parallelen Testjob durchschnittlich 1.16 Prozessoren gleichzeitig zugeordnet. Der Scheduler von UNICOS 6.0 führt die acht parallelen Jobs also praktisch ohne *coscheduling* und damit beinahe sequentiell aus.

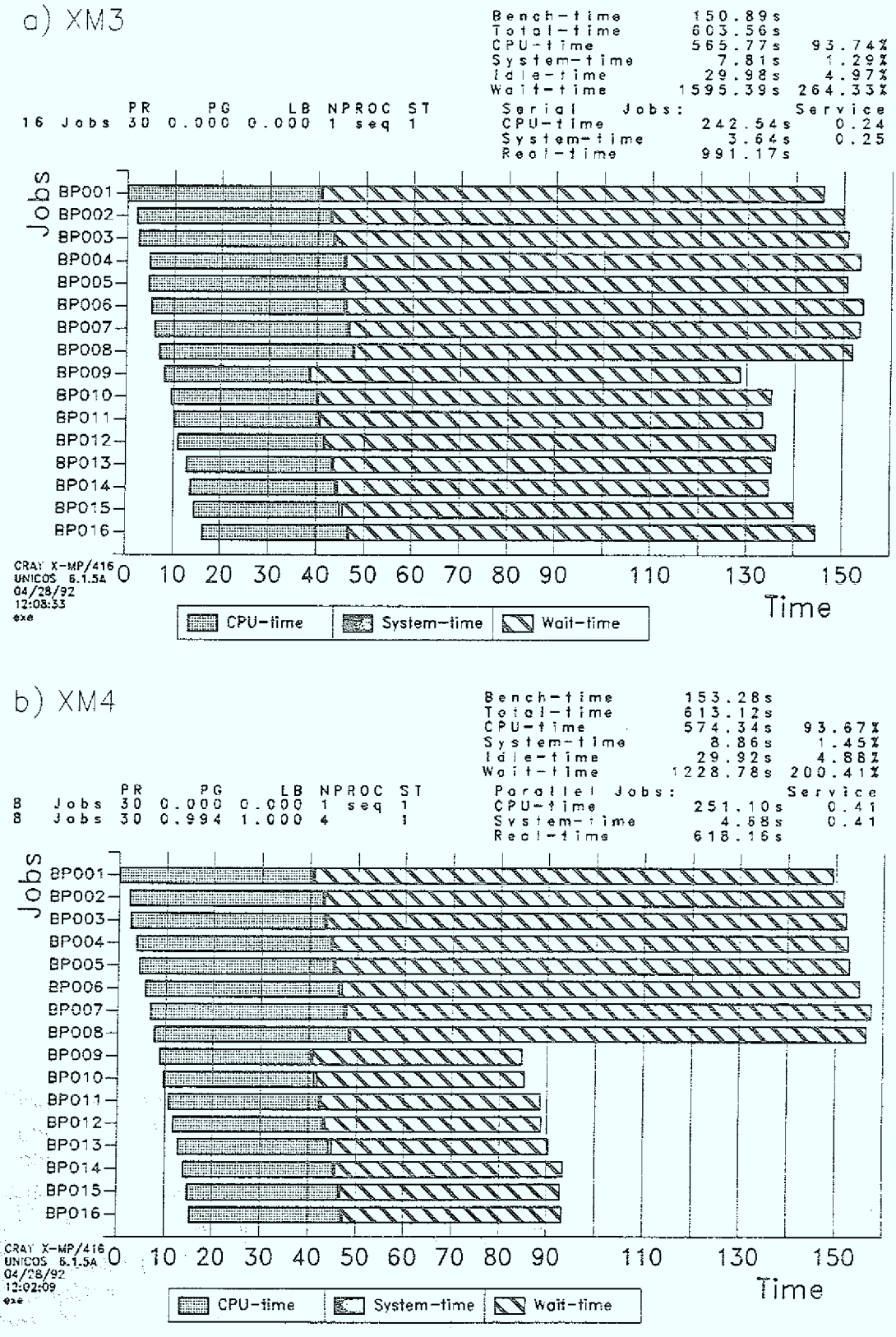


Abbildung 5.10: Benchmark-Plot der Messung XM3 (a) und XM4 (b)

Variable Anzahl paralleler Jobs In der Messung XM3 mit acht parallelisierten Testjobs wurde nur ein geringes Maß an Parallelverarbeitung erreicht. Dies ist insbesondere deshalb der Fall, weil mehr parallele Testjobs ausführbereit sind als Prozessoren in der CRAY X-MP/416 installiert sind. Deshalb werden in den Messungen XM4a-d nur einer, zwei, vier oder sechs der Testjobs parallelisiert. Die Meßergebnisse werden in der Tabelle 5.25 wiedergegeben, während in der Abbildung 5.11 die akkumulierte CPU-time aller Jobs in Abhängigkeit von der Anzahl der parallelisierten Testjobs wiedergegeben wird.

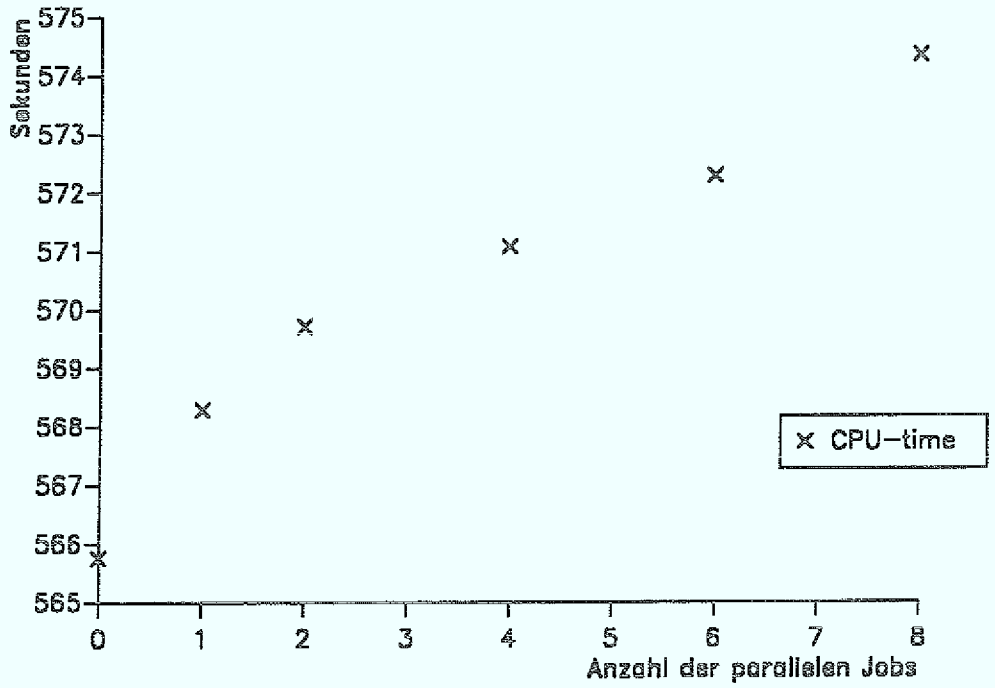


Abbildung 5.11: Akkumulierte CPU-time der Arbeitslast in Abhängigkeit von der Anzahl der parallelen Testjobs auf der CRAY X-MP/416

Meßergebnisse Die CPU-time steigt in Abhängigkeit von der Anzahl der parallelen Jobs monoton, jedoch nicht linear an (siehe Abbildung 5.11). Die CPU-time ist in Messung XM4a mit einem parallelen Job um 2.5 Sekunden gegenüber der Referenzmessung XM3 erhöht. Im Vergleich dazu beträgt der Anstieg bei acht parallelen Testjobs nur 8.5 Sekunden, das sind ungefähr 1 Sekunde für jeden parallelen Job. Der Anstieg der CPU-time wird also mit zunehmender Anzahl von parallelen Jobs geringer. Jeder der Testjobs führt 305 parallele Regionen aus und benötigt dafür 3.3% mehr CPU-time bei einer Arbeitslast mit acht parallelen Jobs und 8.2% mehr CPU-time, wenn er der einzige parallele Job der Arbeitslast ist. Die System-time nimmt für zwei parallele Jobs ein Minimum an, daß sogar 0.5 Sekunden unter der

Messung	par. Jobs	Arbeitslast				Testjobs	
		Bench-time	Total-time	CPU-time	System-time	*service*-Faktor	zugeteilte Proz.
XM3	0	150.9	603.6	565.8	7.8	0.25	1.00
XM4a	1	151.3	605.4	568.3	7.6	0.86	1.49
XM4b	2	151.0	604.1	569.7	7.3	0.74	1.29
XM4c	4	151.7	606.9	571.1	7.9	0.57	1.24
XM4d	6	153.1	612.5	572.3	8.4	0.48	1.19
XM4	8	153.3	613.1	574.3	8.9	0.41	1.16

Tabelle 5.25: Meßergebnisse der Messungen XM4a-d mit einer variablen Anzahl von parallelisierten Jobs

System-time in der Referenzmessung XM3 ohne parallele Jobs liegt. Danach steigt die System-time um durchschnittlich 0.265 Sekunden für jeden parallelem Job an.

Im folgenden soll untersucht werden, wie der gemessene Aufwand mit dem Scheduling von parallelen Programmen zusammenhängt. In der Tabelle 5.25 werden die Meßwerte der Messungen XM4a-d wiedergegeben, die das Scheduling der parallelen Jobs betreffen. Je weniger parallele Jobs in der Arbeitslast sind, desto höher ist der *service*-Faktor der parallelen Jobs. In den hier durchgeführten Messungen steigt der *service*-Faktor jedoch nicht über eins, was darauf zurückzuführen ist, das in einer Arbeitslast 16 Jobs gleichzeitig ausgeführt werden. Der *service*-Faktor von 0.86 eines parallelen Testjobs in Messung XM4a beträgt zwar das 3.4-fache des *service-Faktors* eines sequentiellen Testjobs, ist aber immer noch kleiner als eins. Nach der Information aus der Protokolldatei werden die kooperierenden Prozesse eines parallelen Programms nur zum geringen Teil gleichzeitig ausgeführt (siehe Tabelle 5.25, Spalte „zugeteilte Prozessoren"). Demnach erfolgt praktisch kein *coscheduling* der kooperierenden Prozesse eines parallelen Programms (siehe Abschnitt 2.2.2). Der Grad an *coscheduling* nimmt in den Messungen von 1.49 bei einem parallelem Programm auf 1.16 bei acht parallelen Jobs ab.

5.3.2 Arbeitslasten mit niedriger Parallelität

Der in den früheren UNICOS-Versionen benutzte Scheduler 5.X parkt die von einem parallelen Programm nicht genutzten Prozessoren auf einer *semaphore* nach dem Prinzip des *spin waiting*. Die dabei entstehenden Leerlaufzeiten führen bei parallelen Programmen mit niedrigem Parallelisierungsgrad bzw. schlechter *load balance* zu einer Verringerung des Durchsatzes. Der neue Scheduler 6.X, der seit UNICOS 6.0 für die CRAY-Rechner zur Verfügung steht, erlaubt es, Programme mit einer niedrigen Parallelität bzw. schlechter *load balance* auf der CRAY X-MP so auszuführen, daß Leerlaufzeiten von anderen Programmen genutzt werden können.

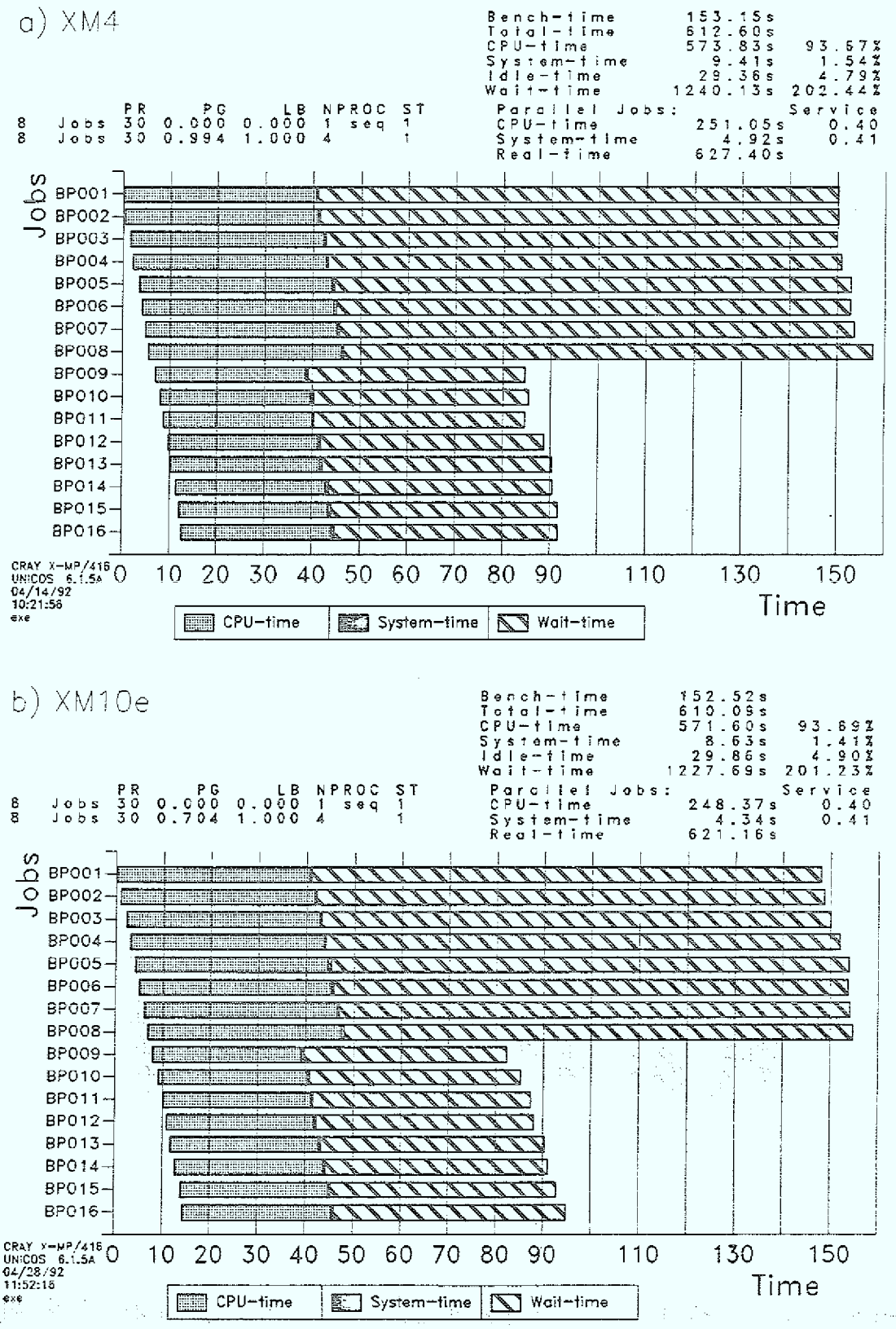


Abbildung 5.12: Benchmark-Plot der Messung XM4 (a) und XM10e (b)

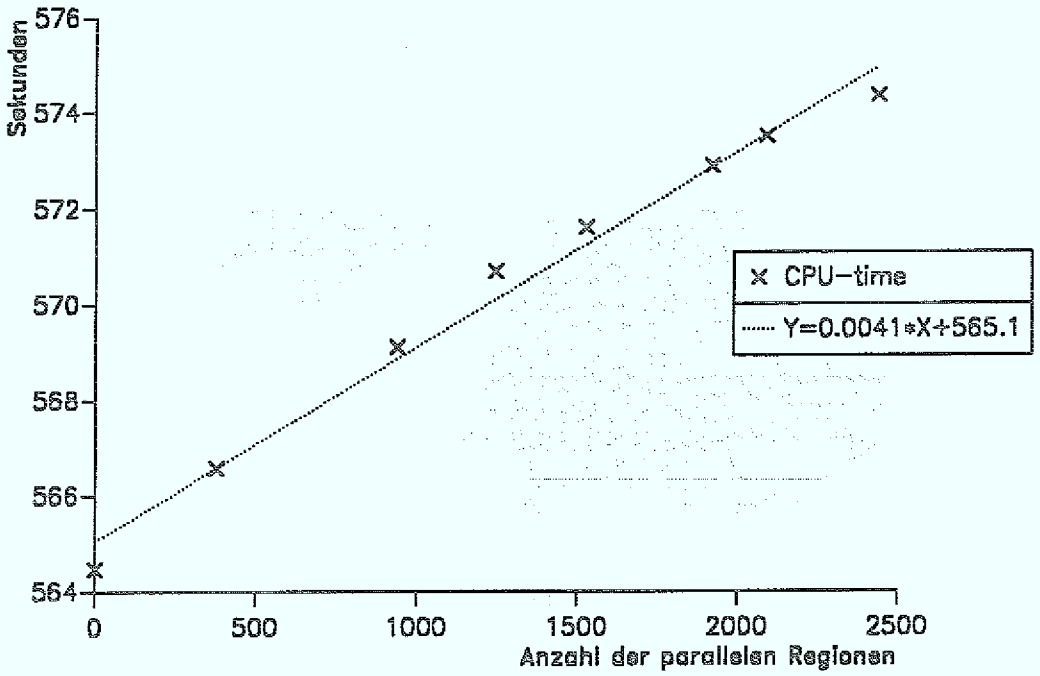


Abbildung 5.13: Anstieg der CPU-time in Abhängigkeit von der Anzahl der parallelen Regionen auf der CRAY X-MP/416

Messung	par. Reg.	Arbeitslast				Testjobs
		Bench-time	Total-time	CPU-time	System-time	*service*-Faktor
XM10a	0	150.5	602.0	564.5	7.7	0.26
XM10b	377	151.6	606.3	566.6	8.3	0.34
XM10c	939	152.0	608.2	569.1	8.9	0.41
XM10d	1244	152.4	609.6	570.7	9.0	0.41
XM10e	1527	152.5	610.1	571.6	8.6	0.41
XM10f	1920	153.3	613.2	572.9	8.9	0.41
XM10g	2092	153.3	613.3	573.5	9.2	0.41
XM4	2440	153.3	613.1	574.3	8.9	0.41

Tabelle 5.27: Meßergebnisse der Messungen XM10a-g mit einer variablen Anzahl von parallelen Regionen

mit hoher Genauigkeit durch eine lineare Funktion approximiert werden (siehe Abbildung 5.13). Der Korrelationskoeffizient der CPU-time in Abhängigkeit von der Anzahl der parallelen Regionen beträgt für die Meßwerte $r = 0.992$. Die Gleichung

der so ermittelten Regressionsgeraden lautet:

$$\text{CPU-time} = 0.0041\text{s} * \text{Anzahl der parallelen Regionen} + 565.1\text{s} \quad (5.3)$$

Der durchschnittliche Anstieg für eine parallele Region beträgt in den Messungen mit acht parallelen Testjobs also ungefähr 4.1 ms. Es muß berücksichtigt werden, daß parallele Testjobs in PAR-Bench zur Einstellung der *load balance* einen geringfügig anderen Code durchlaufen. Eine Testmessung ergab, daß dieser Aufwand für einen Testjobs mit PG=100% bzw. 305 parallelen Regionen ungefähr 0.14 Sekunden an CPU-time verursacht. Damit müssen von den 4.1 ms für eine parallele Region noch ungefähr 0.46 ms abgezogen werden, die durch parallele PAR-Bench Jobs zusätzlich verursacht werden. Der Aufwand, der durch die parallele Ausführung der Programme verursacht wird, beträgt demnach ca. 3.6 ms für eine parallele Region. Bei der Betrachtung des *service*-Faktors in Tabelle 5.25 fällt auf, das der maximale *service*-Faktor schon bei einem Parallelisierungsgrad von 30% bzw. 939 parallelen Regionen erreicht wird. Danach stagniert der Anstieg des *service*-Faktors. Die System-time steigt mit dem *service*-Faktor um 1.2 Sekunden und nimmt dann einen durchschnittlichen Wert von ca. 8.9 Sekunden. Das läßt darauf schließen, daß in diesem Bereich eine Sättigung eintritt und den parallelen Programmen keine zusätzlichen Prozessoren mehr zugeteilt werden.

5.3.3 Arbeitslasten mit hoher Speicherbelastung

In den bisherigen Messungen wurden rechenintensive Jobs mit geringer Speicherbelastung verwendet, die nur in geringem Maße von der Leistung des Speichersystems beeinflußt werden. Im Einprogrammbetrieb erreichte ein derartiger rechenintensiver Job eine Speicherbelastung von ca. 50 MMREFS (Messung XM1). Im Mehrprogrammbetrieb multipliziert sich diese Belastung mit den vier Prozessoren zu einer Speicherbelastung von 200 MMREFS, das sind nur 11% der maximalen Speicherbandbreite der CRAY X-MP/416. Ein relevanter Anstieg der Ausführungszeit aufgrund von Zugriffsverzögerungen wurde für diese rechenintensiven Jobs in der Messung XM1d nicht beobachtet. Die speicherintensiven Jobs, die für die Messungen auf der CONVEX C3840 verwendet werden, erreichen auf der CRAY X-MP/416 eine Speicherleistung von 180 MMREFS und eine Rechenleistung von 130 MFLOPS. Im Mehrprogrammbetrieb wird dadurch bei Nutzung aller vier Prozessoren eine Speicherbelastung von 720 MMREFS, also ca. 40% der maximalen Hauptspeicher-Bandbreite der CRAY X-MP/416 erreicht.

Einprogrammbetrieb Der durch Zugriffsverzögerungen entstehende Aufwand wird durch einen Vergleich von einer Messung im Einprogrammbetrieb mit Messungen im Mehrprogrammbetrieb ermittelt. In der Messung YM20a wird ein speicherintensiven Testjobs im Einprogrammbetrieb ausgeführt. Zum Vergleich werden in der Messungen YM20b-d zwei, drei bzw. vier dieser Testjobs gleichzeitig ausgeführt.

Alternativ dazu wird der speicherintensive Testjob in der Messung XM21b-d$^{\sigma}$ mit zwei, drei und vier Prozessoren unter Nutzung des statischen Schedulers parallel ausgeführt (siehe Tabelle 5.28).

Sequentielle Jobs						
Messung	Jobs	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	Real-time [Prozent]	Effizienz [Prozent]
XM20a	1	26.46	100.0	26.56	100.0	100.0%
XM20b	2	26.95	108.3	27.02	101.7	98.3%
XM20c	3	28.66	100.0	28.75	108.3	92.4%
XM20d	4	30.64	115.8	30.81	116.0	86.2%
Parallele Jobs						
Messung	N	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
XM21b$^{\sigma}$	2	26.86	101.5	13.49	2.00	100.5%
XM21c$^{\sigma}$	3	28.73	108.6	9.62	2.76	99.6%
XM21d$^{\sigma}$	4	30.68	115.9	7.70	3.44	86.2%

Tabelle 5.28: Ausführungsdauer von speicherintensiven Jobs auf der CRAY X-MP/416
Messung XM20x: sequentielle Job mit Hintergrundlast
Messung XM21x^{σ}: parallele Ausführung mit N Prozessoren

In der Messung XM20d mit vier speicherintensiven Testjobs kommt es zu einem Anstieg der CPU-time um 4.18 Sekunden bzw. 15.8%. Dieser Anstieg wird durch Verzögerungen aufgrund von Zugriffskonflikten verursacht. Dieselben Zugriffskonflikte zeigen sich natürlich auch bei der parallelen Ausführung eines Programms auf einem dedizierten Computersystem im Einprogrammbetrieb. Daher ist die CPU-time bei der parallelen Ausführung in Messung XM21d$^{\sigma}$ ebenfalls um 4.22 Sekunden bzw. 15.5% gegenüber Messung XM20a erhöht. Die Differenz zwischen dem Anstieg der CPU-time bei vier sequentiellen Jobs und einem parallelen Job von 0.04 Sekunden entspricht 0.15% und ist damit nicht nachweisbar. Die Varianz der Ausführungsdauern ist bei speicherintensiven Jobs also so hoch, daß Scheduling-Aufwand für diese Jobs nicht nachweisbar ist.

Mehrprogammbetrieb Im Gegensatz zur parallelen Ausführung eines speicherintensiven Programm im Einprogrammbetrieb ist bei der parallelen Ausführung eines Programms im Mehrprogrammbetrieb nicht mit einer Reduzierung des Durchsatzes zu rechnen. Im Mehrprogrammbetrieb auf der CRAY X-MP/416 werden einem parallelem Programm in der Messung XM4a durchschnittlich 1.49 Prozessoren gleichzeitig zugeordnet. Der Einfluß, den die höhere Speicherbelastung dabei auf die Ausführungsdauer der parallelen Jobs hat ist, marginal. Dieser Sachverhalt wurde zusätzlich durch eine Meßreihe analog zu den Messungen C38M22 und

C38M23 auf der CONVEX überprüft. Dabei zeigt sich nicht nur, daß der Anstieg der CPU-time der parallelen Jobs gering ist, sondern auch, daß die CPU-time der Hintergrundlast im gleichen Maß reduziert wird. Die parallele Ausführung von speicherintensiven Programmen hat somit auf einer CRAY X-MP/416 keine negativen Auswirkungen auf den Durchsatz.

5.3.4 Zusammenfassung der Meßergebnisse

Die CRAY X-MP/416 in Verbindung mit dem Betriebssystem UNICOS 6.0 unterstützt die effiziente Ausführung von parallelen Programmen im Mehrprogrammbetrieb. Die durchgeführten Messungen führen zu folgenden Resultaten:

- Im Einprogrammbetrieb von hochparallelen PAR-Bench-Jobs erreicht der neue Scheduler von UNICOS 6.0 eine Effizienz von 97.3%, die der Effizienz des für den Einprogrammbetrieb optimierten Schedulers von UNICOS 5.1 entspricht.

- Im Mehrprogrammbetrieb führt die Ausführung von parallelen Programmen zu einem geringen zusätzlichen Aufwand von ca. 4% in einer Arbeitslast mit acht parallelen Testjobs und zu einem etwas höheren Aufwand von ca. 8% in einer Arbeitslast mit nur einem parallelen Job. Dabei werden parallelen Jobs bevorzugt ausgeführt, die erreichte Beschleunigung ist jedoch gering.

- Der neue Scheduler von UNICOS 6.0 gibt leerlaufende Prozessoren freiwillig ab und erreicht deshalb auch bei der Ausführung von parallelen Programmen mit niedrigem Parallelisierungsgrad einen hohen Durchsatz. Ein Leerlaufen von Prozessoren ist in den Messungen nicht nachweisbar. In einer Arbeitslast mit acht parallelen Jobs verursacht die Ausführung einer parallelen Region ca. 3.5 ms Prozessorzeit.

- Bei der parallelen Ausführung von Programmen im Mehrprogrammbetrieb braucht das Speichersystem nicht berücksichtigt zu werden. Der Durchsatz der CRAY X-MP/416 wird auch durch die Parallelisierung von speicherintensiven Jobs nicht negativ beeinflußt.

Im Mehrprogrammbetrieb wird der Durchsatz der CRAY X-MP/416 durch die Ausführung von parallelen Programmen nur in geringem Maß reduziert. Der durch das Scheduling paralleler Programme verursachte Aufwand wird dabei von der Anzahl der parallelen Programme in der Arbeitslast bestimmt, ist jedoch gering.

5.4 Messungen auf der CRAY Y-MP

Die CRAY Y-MP ist der logische Nachfolger der CRAY X-MP und erweitert die Architektur der X-MP-Serie auf bis zu acht Prozessoren mit einer gesteigerten Taktrate. Die Architektur der CRAY Y-MP8/832 basiert auf einer Register-Register-Machine mit einem verschränkten Hauptspeicher, der durch 256 schnelle Speicherbänke realisiert wird. Durch eine Veränderung der Hauptspeicherorganisation konnte die Speicherleistung im Verhältnis zur Rechenleistung überproportional erhöht werden. Die Hauptspeicherkapazität der CRAY Y-MP8/832 beträgt mit 32 MWorten das doppelte der Hauptspeicherkapazität der CRAY X-MP/416 und ist daher für alle hier durchgeführten Leistungsmessungen ausreichend. Daher können die Leistungsmessungen auf der CRAY Y-MP8/832 sowohl mit als auch ohne NQS durchgeführt werden. Aufgrund der höheren Prozessorzahl kann die CRAY Y-MP8/832 bei der Parallelverarbeitung nicht nur vier, sondern acht Prozessoren verwenden. Da der Benutzer eines Rechners daran interessiert ist, alle Prozessoren des Systems zu nutzen, werden die Leistungsmessungen auf der CRAY Y-MP8/832 mit parallelen Programmen durchgeführt, die alle acht Prozessoren anfordern. Zusätzlich werden jedoch auch vergleichende Messungen durchgeführt, in denen nur vier Prozessoren genutzt werden. Für die CRAY Y-MP steht mit UNICOS 6.1 das gleiche Betriebssystem zur Verfügung. Für die Messungen können deshalb auch hier die zwei Scheduler verwendet werden, die über die *environment*-Varible MP_DEDICATED ausgewählt werden: entweder der alte Scheduler 5.X (MP_DEDICATED=1) oder der neue Scheduler 6.X, der nicht benötigte Prozessoren freiwillig wieder abgibt (MP_DEDICATED=0). Die Messungen, in denen die parallelen Jobs den alten Schedulers 5.X nutzen, sind weiterhin durch das Symbol σ gekennzeichnet.

5.4.1 Arbeitslasten mit günstigen Eigenschaften

Für die Leistungsmessungen auf der CRAY Y-MP8/832 werden wie auf der CRAY X-MP/416 und der CONVEX C3840 wiederum identische Arbeitslasten verwendet. Die Parameterdateien werden jedoch in der Regel so modifiziert, daß parallele Jobs alle N=8 Prozessoren der CRAY Y-MP8/832 nutzen. Für die nun folgenden Messungen werden also rechenintensive Jobs verwendet, die auf der CRAY X-MP/416 mit den dynamischen Parametern FL = 200 MFLOPS und MM = 50 MM-REFS generiert wurden. Für die parallele Ausführung der Testjobs werden die Parameter Parallelisierungsgrad mit PG=100%, die *load balance* mit LB=100% und die Anzahl der Prozessoren mit N=8 vorgegeben.

Einprogrammbetrieb Die Untersuchung paralleler Jobs wird zunächst im Einprogrammbetrieb durchgeführt. Als Referenz dient der sequentielle Lauf eines rechenintensiven Jobs in Messung YM1a. Zur Bestimmung der durch Speicherzugriffskonflikte verursachten Verzögerungen werden in den Messungen YM1b, YM1d

und YM1f zwei, vier bzw. acht identische Jobs gleichzeitig gestartet. Die für diese Messungen erhaltenen Ergebnisse werden in Tabelle 5.29 wiedergegeben. Die Jobs werden in dieser Messung nicht über das NQS, sondern als Hintergrundjobs gestartet. Die Wartezeit auf die Ein-/Ausgabe führt bei der CRAY Y-MP8/832 aufgrund der kurzen Ausführungsdauer der parallelen Jobs zu einer erheblichen Beeinträchtigung der Messungen. Um eine Vergleichbarkeit der Messungen zu gewährleisten, wurde die Arbeitslast diesmal nicht skaliert; dies muß bei der Bewertung der Ergebnisse berücksichtigt werden. Die durch die Ein-/Ausgabeoperationen verursachten Effekte konnten jedoch durch die Nutzung des SSD als Disk-Cache reduziert werden.

Messung	Jobs	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	Real-time [Prozent]	Effizienz [Prozent]
YM1a	1	21.65	100.0	21.65	100.0	100.0%
YM1b	2	21.65	100.0	21.66	100.0	100.0%
YM1d	4	21.66	100.0	21.77	100.6	99.4%
YM1f	8	21.67	100.1	21.79	100.6	99.4%

Tabelle 5.29: Ausführungsdauer von rechenintensiven Jobs auf der CRAY Y-MP8/832 in Abhängigkeit von der Anzahl der Jobs bei sequentieller Ausführung

Die Real-time ist bei der gleichzeitigen Ausführung von acht Jobs in Messung YM1f nahezu unverändert, der Anstieg der CPU-time beträgt nur 0.1% und ist vernachlässigbar. Der durch Zugriffsverzögerungen verursachte Aufwand entspricht dem Anstieg der CPU-time und ist damit für die rechenintensiven Jobs bedeutungslos. In den Messungen YM2b, YM2d und YM2f wird der rechenintensive Job aus der Messung YM1a mit zwei, vier bzw. acht Prozessoren parallel ausgeführt. Die parallelen Jobs benutzen dabei den neuen Scheduler 6.X, der die effiziente Ausführung von parallelen Programmen im Mehrprogrammbetrieb unterstützt. Die *slave hold time* des Schedulers 6.X ist in den Messungen wie auf der CRAY X-MP/416 für den PAR-Bench-Job optimiert (MP_HOLDTIME=150000). Die so erhaltenen Meßergebnisse werden in der Tabelle 5.30 aufgeführt.

Messung	N	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
YM2b	2	22.08	102.0	11.11	1.95	97.4%
YM2d	4	22.20	102.5	5.59	3.87	96.8%
YM2f	8	22.44	103.6	2.91	7.44	93.0%

Tabelle 5.30: Effizienz und *speedup* eines rechenintensiven Jobs mit dem neuen Scheduler 6.X (MP_DEDICATED=0)

Die Effizienz fällt in den Messungen mit der Anzahl der genutzten Prozessoren auf 93% bei acht Prozessoren ab. Bei Nutzung von vier Prozessoren entspricht die Effizi-

enz den auf der CRAY X-MP/416 gemessenen Werten. Die CPU-time steigt bei dem mit acht Prozessoren ausgeführten Job um 3.6% an, wobei der Aufwand, der durch Zugriffsverzögerungen verursacht wird, vernachlässigbar ist. Ein großer Teil des Anstiegs der CPU-time wird stattdessen beim *spin waiting* der zusätzlichen Prozessoren in den kurzen sequentiellen Regionen zwischen den parallelen Schleifen verbraucht. Zum Vergleich werden diese Messungen mit den Bezeichnungen YM2b$^{\sigma}$ YMP2d$^{\sigma}$ und YMP2f$^{\sigma}$ unter Verwendung des alten Schedulers 5.X wiederholt (siehe Tabelle 5.31). Der *speed up* bzw. die Effizienz der parallelen Jobs ist bei der Ausführung mit

Messung	N	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
YM2b$^{\sigma}$	2	22.40	103.5	11.22	1.93	96.5%
YM2d$^{\sigma}$	4	22.18	102.4	5.70	3.79	95.0%
YM2f$^{\sigma}$	8	22.44	103.6	2.82	7.68	96.0%

Tabelle 5.31: Effizienz und *speedup* eines rechenintensiven Jobs mit dem alten Scheduler 5.X (MP_DEDICATED=1) auf der CRAY Y-MP8/832

dem alten Scheduler 5.X mit den Resultaten beim neuen Schedulers 6.X vergleichbar. Während die Effizienz für zwei bzw. vier Prozessoren beim neuen Scheduler 6.X geringfügig besser sind, ist der Wert für acht Prozessoren geringfügig schlechter. Die Leistung des Schedulers 5.X und des neuen 6.X Schedulers entsprechen sich im Einprogrammbetrieb also bis auf geringe Unterschiede, die bei den Messungen nicht mehr als 0.1 Sekunden Ausführungszeit ausmachen. Die CPU-time ist in den Messungen mit vier bzw. acht Prozessoren bei beiden Schedulern im Rahmen der Meßgenauigkeit gleich. Das deutet darauf hin, daß der Scheduling-Aufwand bei den beiden Schedulern für die Verarbeitung von vollständig parallelen Programmen im Einprogrammbetrieb gleich ist.

Mehrprogrammbetrieb Die Leistung des neuen Schedulers 6.X wird jetzt im Mehrprogrammbetrieb untersucht. Die für die CONVEX C3840 und die CRAY X-MP/416 verwendeten Arbeitslasten werden auch für die Leistungsmessungen auf der CRAY Y-MP8/832 eingesetzt. Die Parameterdateien der Messungen werden jedoch in einigen Punkten modifiziert. Zum einen wird wie schon im Einprogrammbetrieb die Anzahl der angeforderten Prozessoren eines parallelen Programms auf N=8 erhöht. Zum anderen werden alle acht sequentiellen Jobs BP001-BP008 der Hintergrundlast gleichzeitig gestartet. Dadurch sollen alle acht Prozessoren der Y-MP8/832 sofort genutzt und unnötige Leerlaufzeiten vermieden werden. Die in der Messung YM3 verwendete Arbeitslast entspricht in den übrigen Parametern den Arbeitslasten der Messungen XM3 und C38M3 (siehe Tabelle 5.32). In den zwei folgenden Leistungsmessungen werden die Jobs wie auf der CRAY X-MP mit dem NQS gestartet.

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
04	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	**00**,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	02,	10,	30,	050,	200,	000,	...	1,	000,	000,	1
04	03,	10,	30,	050,	200,	000,	...	1,	000,	000,	1

Tabelle 5.32: Parameter für die Referenzmessung YM3 und YM3x mit rechenintensiver Arbeitslast und sequentieller Ausführung der Testjobs

In der Messung YM4 werden die Testjobs, analog zu den Messungen XM4 und C38M4, parallel ausgeführt. Während der Parallelisierungsgrad PG=100% und die *load balance* LB=100% beibehalten werden, wird die Anzahl der genutzten Prozessoren auf N=8 erhöht (siehe Tabelle 5.33).

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
04	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	**00**,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	02,	10,	30,	050,	200,	000,	...	1,	**100**,	**100**,	**8**
04	03,	10,	30,	050,	200,	000,	...	1,	**100**,	**100**,	**8**

Tabelle 5.33: Parameter für die Messung YM4 und YM4x mit einer rechenintensiven Arbeitslast und parallelisierten Testjobs

Meßergebnisse Der Benchmark-Plot der Messung YM3 wird in Abbildung 5.14 (a) im Vergleich zum Benchmark-Plot der Messung YM4 in Abbildung 5.15 (b) wiedergegeben. Die daraus ableitbaren Ergebnisse entsprechen im wesentlichen den Resultaten auf der CRAY X-MP/416:

- Die Jobs der Arbeitslast starten gegenüber den in der Parameterdatei vorgegebenen Startzeitpunkten verzögert. Die Verzögerung des Startzeitpunktes des letzten Jobs beträgt ca. 8 statt 3 Sekunden. Dieser Wert ist zwar absolut betrachtet niedriger als auf der X-MP/416, ist jedoch in Relation zur Ausführungsdauer der Arbeitslast ungefähr gleich.
- Die Total-time der Arbeitslast ist in Messung YM4 mit den parallelen Testjobs um 17.36 Sekunden erhöht. Die Varianz der Total-time, d.h. der Meßfehler, ist jedoch auch auf der CRAY Y-MP8/832 bei den Messungen mit NQS zu hoch, um Abweichungen in dieser Größenordnung sinnvoll zu interpretieren.
- Die CPU-time der Testjobs wird durch die Parallelisierung um 8.28 Sekunden erhöht. Dieser Anstieg entspricht auch dem Anstieg der CPU-time der gesamten Arbeitslast um 8.29 Sekunden. Dieser Parallelisierungsaufwand beträgt

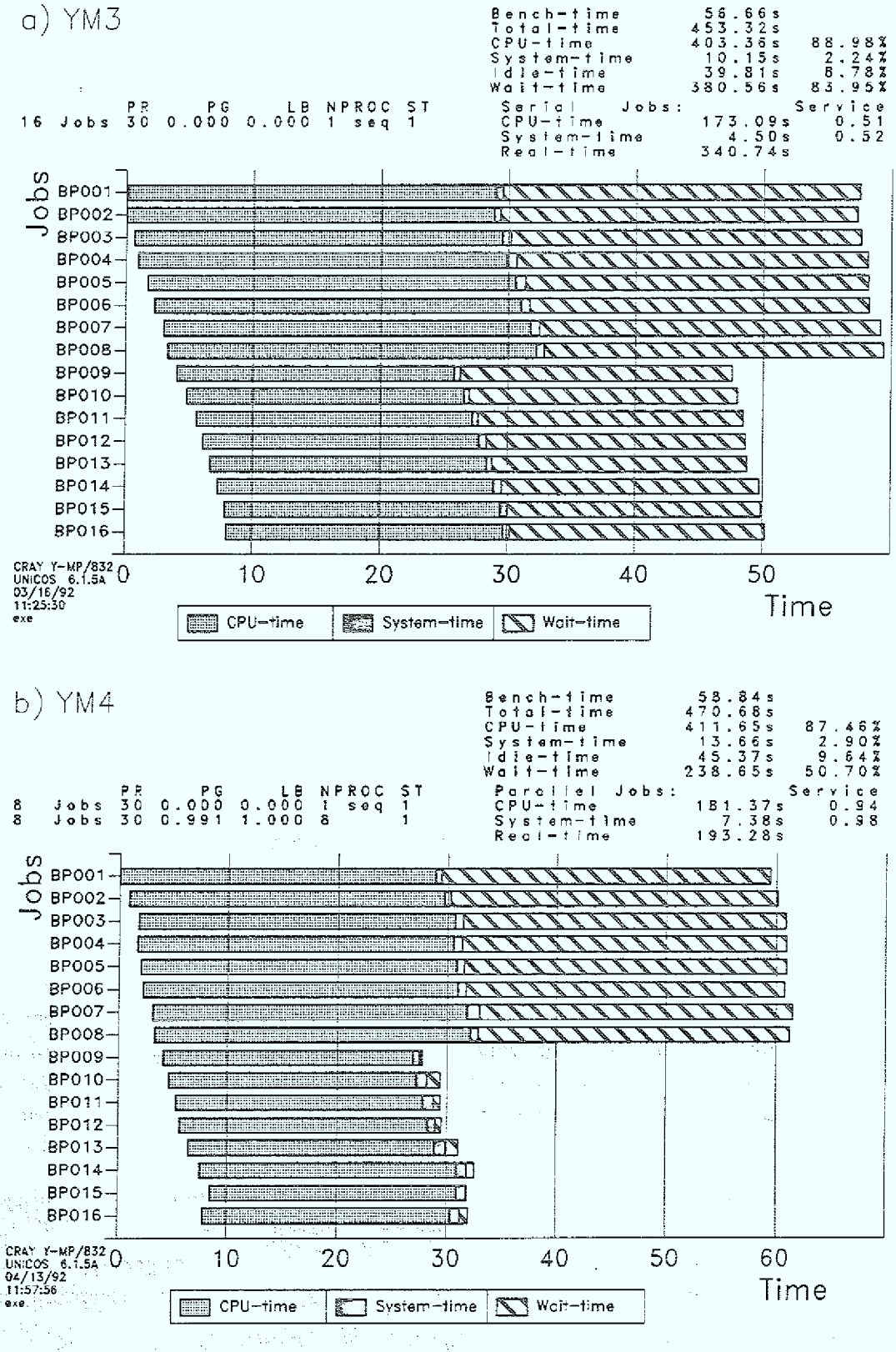


Abbildung 5.14: Benchmark-Plot der Messung YM3 (a) und YM4 (b)

4.78% der CPU-time der Testjobs und ist damit gegenüber dem Anstieg der CPU-time von 3.6% im Einprogrammbetrieb leicht erhöht. Die System-time steigt bei der Messung mit den parallelen Testjobs um 3.51 Sekunden an.

- Der *service*-Faktor der Testjobs wird durch die parallele Ausführung von 0.51 auf 0.94 erhöht. Wie schon auf der CRAY X-MP/416 werden parallele Jobs deutlich bevorzugt ausgeführt. Da der für die parallelisierten Testjobs gemessene *service*-Faktor jedoch kleiner als eins ist, können die parallelisierten Testjobs in Messung YM4 sowohl parallel als auch sequentiell ausgeführt worden sein. Nach den Informationen aus der Protokolldatei werden jedem parallelen Job durchschnittlich 1.6 Prozessoren gleichzeitig zugeordnet. Die parallelen Testjobs werden also nur in einem geringen Maß parallel ausgeführt. Dies ist einerseits darauf zurückzuführen, daß acht parallele Jobs um die acht Prozessoren der CRAY Y-MP8/832 konkurrieren, andererseits führt der Scheduler kein reines *coscheduling* durch.

Variable Anzahl von parallelen Jobs In den Messungen YM3 und YM4 werden die Jobs der Arbeitslast mit dem Kommando *qsub* des NQS gestartet. Da die Startzeitpunkte der Messung durch das NQS beeinflußt wurden und die Varianz der Total-time bei den Messungen erheblich ist, werden die Jobs in allen folgenden Messungen im Hintergrund und nicht über das NQS gestartet. Zusätzlich wird, wie auch schon bei den Messungen XM4a-d auf der CRAY X-MP/416, der SSD als Disk-Cache für die Ein-/Ausgabeoperationen genutzt, um eine erhöhte Meßgenauigkeit zu erreichen. Die Messungen YM3 und YM4 werden in den Messungen YM3x und YM4x unter diesen neuen Bedingungen wiederholt. Diese Untersuchung wird durch Messungen ergänzt, in denen nicht acht, sondern eine variable Anzahl der Testjobs parallel ausgeführt werden. In den Messungen YM4a-d werden nur einer, zwei, vier bzw. sechs der Testjobs parallelisiert. Die so erhaltenen Meßergebnisse werden in der Tabelle 5.34 wiedergegeben.

Messung	par. Jobs	Arbeitslast				Testjobs
		Bench-time	Total-time	CPU-time	System-time	*service*-Faktor
YM3x	0	50.7	405.3	403.2	1.2	0.51
YM4a	1	50.9	407.0	404.7	1.8	2.40
YM4b	2	51.1	408.9	406.2	2.1	2.33
YM4c	4	51.4	411.4	408.0	2.6	1.62
YM4d	6	51.6	412.2	409.1	3.2	1.18
YM4x	8	51.8	415.1	410.3	4.0	0.94

Tabelle 5.34: Ergebnisse der Messungen YM4a-d mit einer variablen Anzahl von parallelisierten Jobs

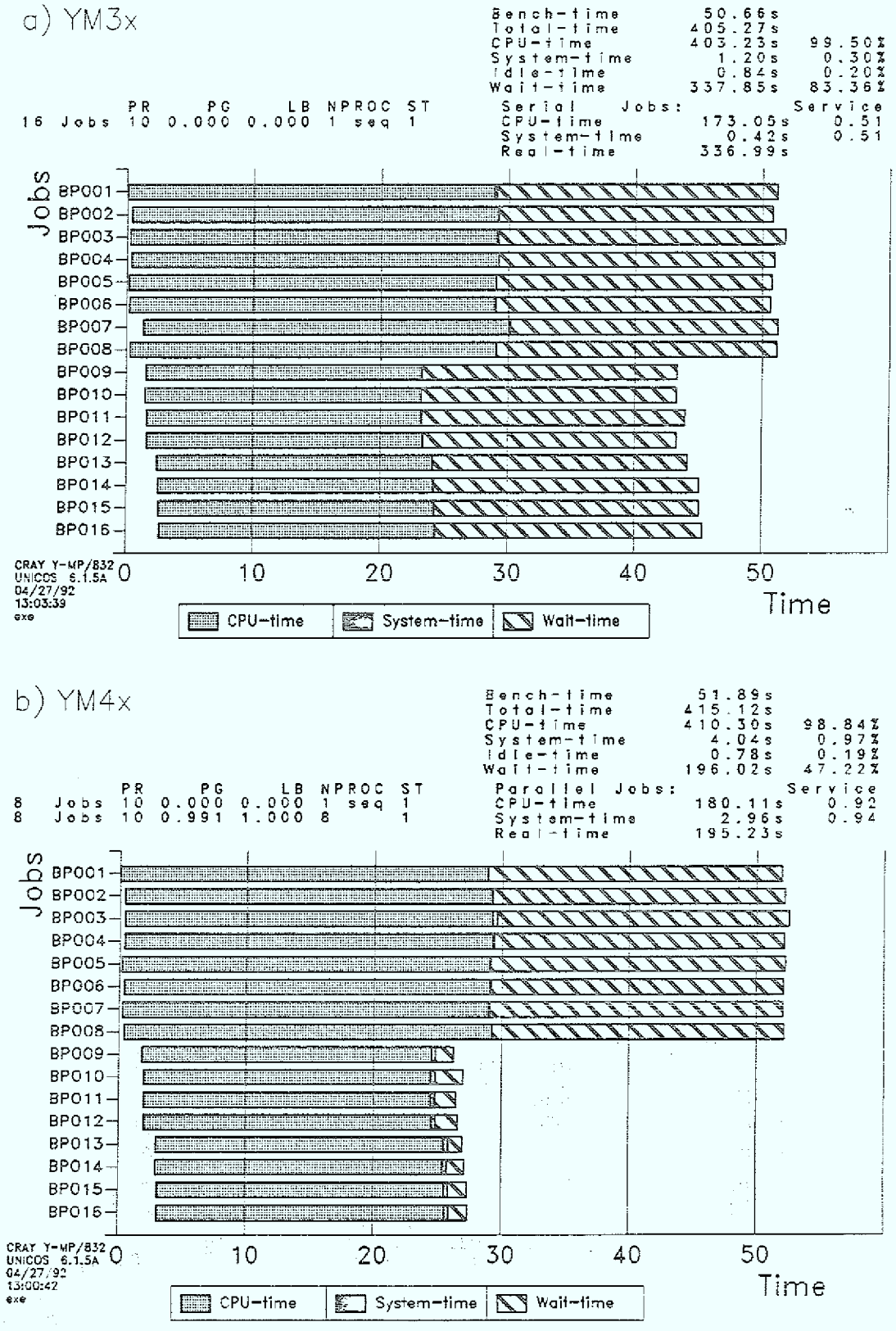


Abbildung 5.15: Benchmark-Plot der Messung YM3x (a) und YM4x (b)

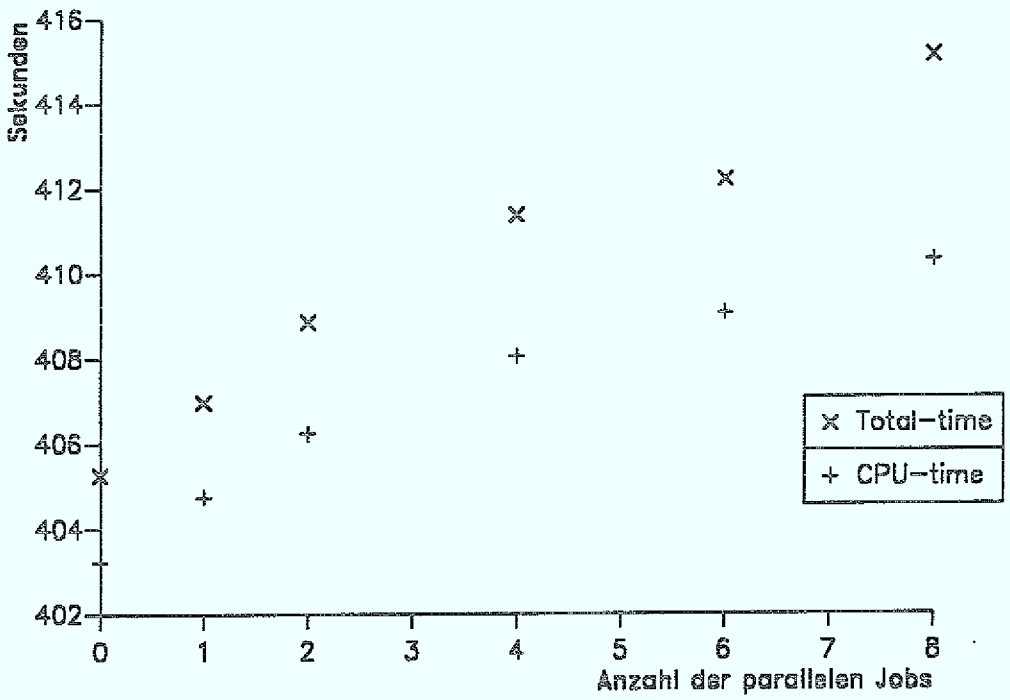


Abbildung 5.16: Anstieg der CPU-time in Abhängigkeit von der Anzahl der parallelen Jobs auf der CRAY Y-MP8/832

Meßergebnisse Die Benchmark-Plots der Messungen YM3x und YM4x werden in Abbildung 5.15 dargestellt. Die Startzeitpunkte der Jobs entsprechen den Vorgaben der Parameterdatei bei diesen Messungen erheblich besser als in den Messungen, in denen die Jobs mit dem NQS gestartet wurden. Gegenüber den Messungen YM3 und YM4 in Abbildung 5.14 ist sowohl die System-time als auch die Idle-time erheblich verkürzt, während die CPU-time im Rahmen der Meßgenauigkeit gleich ist. Die Nutzung des SSD für die Ein-/Ausgabe und das Ausschalten des NQS führen zu einer erheblichen Steigerung der Meßgenauigkeit, die es ermöglicht, auch die gemessene Total-time für die Analyse zu nutzen. Der Anstieg von CPU-time und System-time summiert sich in Messung YM4x mit acht parallelen Testjobs zu 9.9 Sekunden. Dieser Anstieg entspricht 5.7% der Summe aus CPU-time und System-time der acht parallelen Jobs.

In Abbildung 5.16 werden die Total-time und die CPU-time der Arbeitslast in Abhängigkeit von der Anzahl der parallelen Jobs graphisch dargestellt. Es ist deutlich zu erkennen, daß die CPU-time monoton mit der Anzahl der parallelen Jobs steigt, dieser Anstieg jedoch nicht linear ist. Je mehr Jobs in der Arbeitslast parallel ausgeführt werden, desto geringer wird der Anstieg der CPU-time pro par-

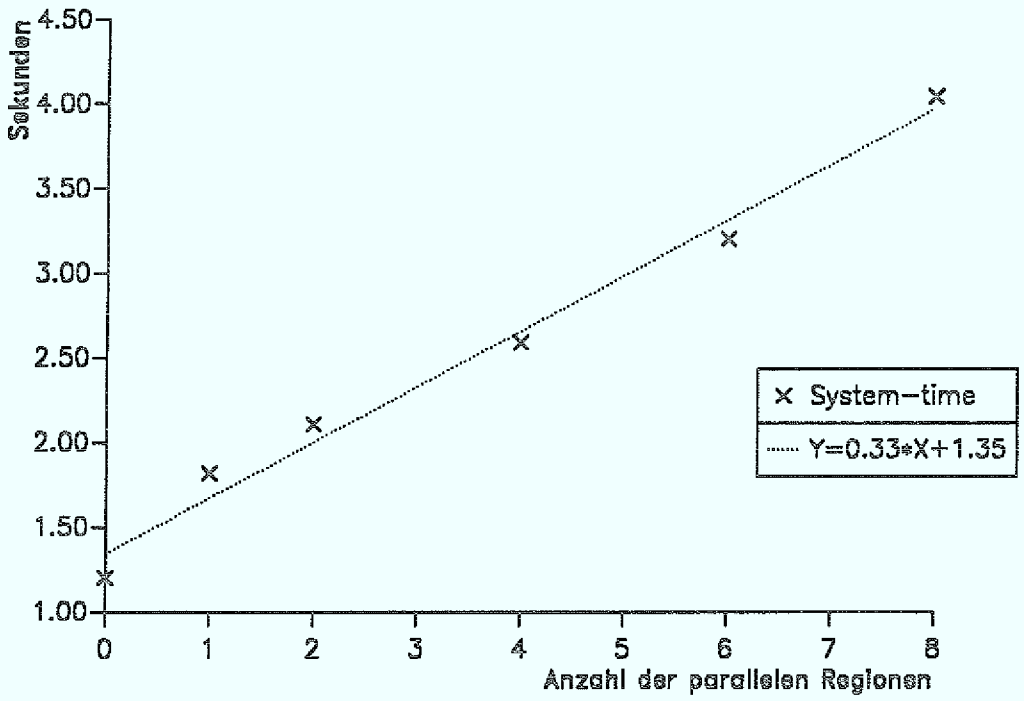


Abbildung 5.17: Anstieg der System-time in Abhängigkeit von der Anzahl der parallelen Jobs auf der CRAY Y-MP8/832

allelen Job. Während der erste parallele Job einen Anstieg der CPU-time um 1.52 Sekunden verursacht, beträgt der Anstieg bei acht parallelen Jobs nur 0.88 Sekunden pro Job. Der Anstieg der Total-time ist geringfügig steiler als der Anstieg der CPU-time. In der Total-time ist neben der CPU-time auch die System-time enthalten. In Abbildung 5.17 wird die System-time in Abhängigkeit von der Anzahl der parallelen Jobs graphisch veranschaulicht. Die Abhängigkeit der System-time von der Anzahl der parallelen Jobs kann im Bereich der Messung durch eine lineare Funktion approximiert werden. Der Korrelationskoeffizient von $r = 0.993$ für diese Abhängigkeit deutet auf einen solchen Zusammenhang hin. Eine lineare Regression für die System-time in Abhängigkeit von der Anzahl der parallelen Jobs liefert die Regressionsgerade:

$$\text{System-time} = 0.33\text{s} * \text{Anzahl der parallelisierten Jobs} + 1.35\text{s} \qquad (5.4)$$

Bei der Beurteilung des entstehenden Aufwands muß berücksichtigt werden, ob und in welchem Maß die parallelisierten Testjobs in den Messungen auch parallel ausgeführt werden. Für diese Untersuchung können die in der Tabelle 5.34 wiedergegebenen *service*-Faktoren der parallelen Testjobs herangezogen werden. Die auf der CRAY Y-MP8/832 gemessenen *service*-Faktoren sind erheblich höher als die auf

der CRAY X-MP/416 gemessenen Werte und zeigen eindeutig, daß die parallelen Testjobs in den Messungen YM4a-d auch parallel ausgeführt wurden. In Messung YM4a mit nur einem parallelen Testjob wird sogar ein *service*-Faktor von 2.40 erreicht. Eine geringe Anzahl von Jobs kann also auch im Mehrprogrammbetrieb noch deutlich durch Parallelverarbeitung beschleunigt werden. Bei einem Start der Jobs als Hintergrundjobs steht jedoch keine Information darüber zur Verfügung, wieviele Prozessoren einem parallelen Job gleichzeitig zugeordnet werden, so daß keine Meßwerte über das *coscheduling* zur Verfügung stehen.

5.4.2 Arbeitslasten mit niedriger Parallelität

Der neue Scheduler 6.X wurde entwickelt, um im Mehrprogrammbetrieb Leerlaufzeiten, die bei der der Bearbeitung von parallelen Programmen entstehen, zu vermeiden. Wie in den vorangegangenen Messungen gezeigt wurde, erreichen die Programme auf CRAY Y-MP8/832 aufgrund der höheren Prozessorzahl auch einen höheren Grad an Parallelverarbeitung als auf der CRAY X-MP/416. Damit steigt auf der CRAY Y-MP8/832 auch der Einfluß des Schedulings bzw. der Leerlaufzeiten auf den Durchsatz des Rechners. Ein Vergleich des alten Schedulers 5.X mit dem neuen Scheduler 6.X von UNICOS wurde bereits in [LNVD91] durchgeführt. Die Untersuchung der Leerlaufzeiten in Programmen mit signifikantem sequentiellen Anteil erfolgt in der Messung YM10e. Die acht parallelisierten Testjobs besitzen dabei einen Parallelisierungsgrad von PG=70%. Die Parameter der Messung YM10e unterscheiden sich von denen in den Messungen XM10d und C28M10c lediglich in der Anzahl N der angeforderten Prozessoren (siehe Tabelle 5.35).

#,	DEL,	PR,	SEC,	MM,	FL,	IO,	...	ST,	PG,	LB,	N
04	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	00,	10,	40,	050,	200,	000,	...	1,	000,	000,	1
04	02,	10,	30,	050,	200,	000,	...	1,	070,	100,	8
04	03,	10,	30,	050,	200,	000,	...	1,	070,	100,	8

Tabelle 5.35: Parameter für die Messung YM10e - Parallele Testjobs mit einem Parallelisierungsgrad von PG=70%

Meßergebnisse Abbildung 5.18 zeigt den Benchmark-Plot der Messung YM10e (b) mit zum Teil parallelen Testjobs unter dem der Messung YM4x (a) mit vollständig parallelen Testjobs. Die Total-time ist bei Messung YM10e um 0.4 Sekunden erhöht, was vernachlässigbar gering ist. Der Durchsatz der CRAY Y-MP8/832 wird von dem niedrigen Parallelisierungsgrad der Testjobs praktisch überhaupt nicht beeinflußt. Insbesondere gilt dies auch für den *service*-Faktor, der bei den Testjobs mit einem Parallelisierungsgrad von PG=70% den selben Wert erreicht

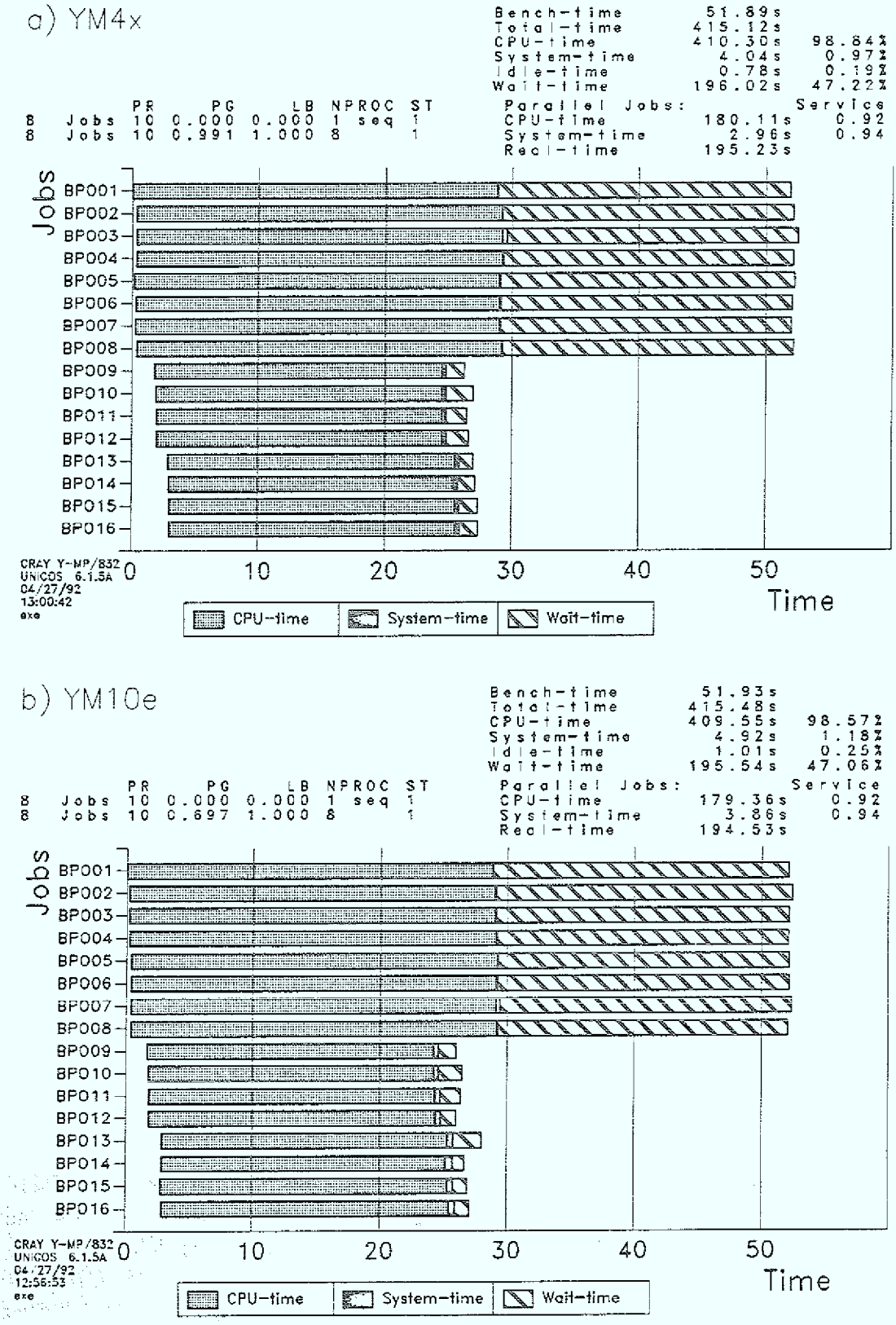


Abbildung 5.18: Benchmark-Plot der Messung YM3x (a) und YM10f (b)

wie für die vollständig parallelen Testjobs mit PG=100%. Im Anschluß erfolgt deshalb eine Untersuchung der CPU-time und der System-time in Abhängigkeit von der Zahl der parallelen Regionen eines Jobs.

Scheduling-Aufwand Die Anzahl der parallelen Regionen, die ein paralleler PAR-Bench-Job durchläuft, wird wie in vorangegangenen Messungen auf der CONVEX C3840 und der CRAY X-MP/416 über den Parallelisierungsgrad der Jobs variiert. Je höher der Parallelisierungsgrad eines Jobs ist, desto mehr parallele Programmkerne und damit parallele Regionen werden von dem Job ausgeführt. In den Messungen YM10a-h werden jeweils alle acht Testjobs BP009-016 parallelisiert. Alle acht Testjobs einer Messung bekommen dabei den gleichen Parallelisierungsgrad und eine *load balance* von LB=100% als Vorgabe. Der Parallelisierungsgrad der Testjobs wird bei den verschiedenen Messungen verändert, wodurch eine jeweils andere Zahl von parallelen Regionen durchlaufen wird. Die so erhaltenen Meßergebnisse sind in Tabelle 5.36 aufgeführt.

Messung	par. Reg.	Arbeitslast				Testjobs
		Bench-time	Total-time	CPU-time	System-time	*service*-Faktor
YM10a	0	50.6	404.7	402.2	2.1	0.51
YM10b	370	51.1	408.7	404.2	3.0	0.68
YM10c	938	51.7	413.6	407.8	5.2	0.87
YM10d	1238	51.9	415.4	409.3	5.5	0.92
YM10e	1514	51.9	415.5	409.6	4.9	0.94
YM10f	1927	51.9	415.4	410.0	4.6	0.94
YM10g	2092	51.8	414.6	409.7	4.2	0.94
YM4x	2440	51.9	415.1	410.3	4.0	0.94

Tabelle 5.36: Ergebnisse der Messungen YM10a-g mit einer variablen Anzahl von parallelen Regionen

Meßergebnisse In Abbildung 5.19 werden CPU-time und Total-time in Abhängigkeit von der Anzahl der parallelen Regionen graphisch dargestellt. Es ist deutlich erkennbar, daß die Abhängigkeit der CPU-time von der Anzahl der parallelen Regionen nicht linear ist. Ein lineares Verhalten kann nur im Bereich zwischen 0 und 1200 parallelen Regionen bzw. bei einem Parallelisierungsgrad zwischen 0% und 50% beobachtet werden. In diesem Bereich wird der Anstieg der CPU-time durch zwei Effekte bestimmt:

- Die Jobs mit niedrigem Parallelisierungsgrad führen nur relativ selten eine parallele Region aus und fordern deshalb nur selten zusätzliche Prozessoren

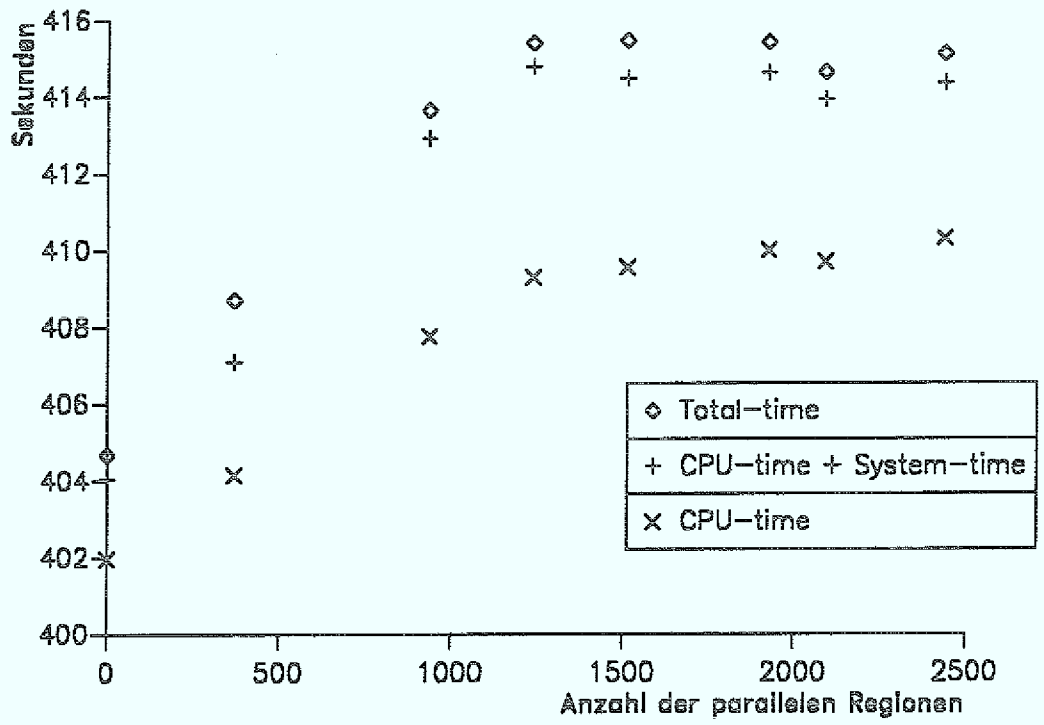


Abbildung 5.19: Anstieg der CPU-time in Abhängigkeit von der Anzahl der parallelen Regionen auf der CRAY Y-MP8/832

an. Dadurch müssen die Prozessoren auf weniger parallele Jobs verteilt werden, die dann mehr zusätzliche Prozessorzeit zugeteilt bekommen. Als Folge davon steigt der *service*-Faktor in den Messungen YM10a-d mit der Anzahl der parallelen Region bzw. dem Parallelisierungsgrad.

- Die zusätzlich angeforderten Prozessoren führen am Ende der parallelen Regionen für die Dauer der *slave hold time* ein *spin waiting* durch, bis sie wieder abgegeben werden.

Im unteren Bereich der Kurve kann die Abhängigkeit der CPU-time, der System-time und der Total-time durch eine lineare Funktion angenähert werden. Die Gleichung der Regressionsgeraden lauten:

$$\text{System-time} = 2.9\text{ms} * \text{Anzahl der parallelen Regionen} + 2.1\text{s} \quad (5.5)$$

$$\text{CPU-time} = 6.0\text{ms} * \text{Anzahl der parallelen Regionen} + 402.0\text{s} \quad (5.6)$$

$$\text{Total-time} = 8.7\text{ms} * \text{Anzahl der parallelen Regionen} + 405.1\text{s} \quad (5.7)$$

Der Anstieg der Total-time entspricht dabei ungefähr der Summe des Anstieg von CPU-time und System-time. Für die PAR-Bench-Jobs mit einem niedrigen Paral-

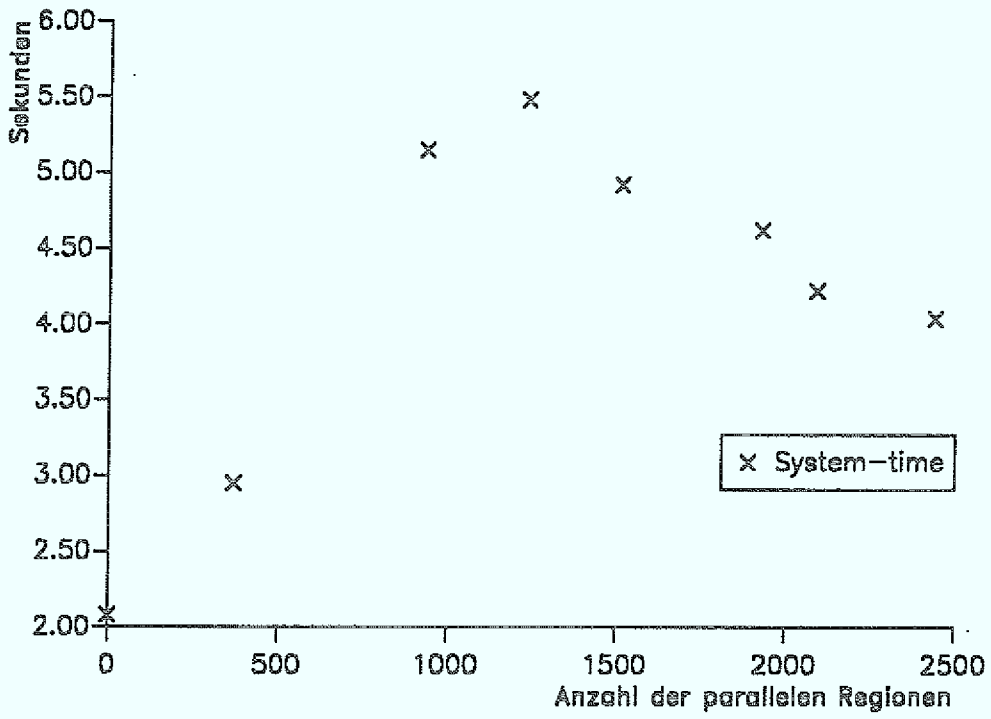


Abbildung 5.20: Anstieg der System-time in Abhängigkeit von der Anzahl der parallelen Regionen auf der CRAY Y-MP8/832

lelisierungsgrad kann demnach ein Aufwand von 8.9 ms für jede parallele Region berechnet werden. Wie eine vergleichende Messung zeigt, liegt der Aufwand, der bei der parallelen Ausführung eines Kerns von PAR-Bench verursacht wird, bei der CRAY Y-MP8/832 bei 0.5 ms. Dieser Aufwand muß von den 8.9 ms abgezogen werden und ergibt einen Aufwand von ca. 8.4 ms für jede zusätzliche parallele Region. Es konnte nachgewiesen werden, daß der Anstieg der CPU-time zum Teil durch die *slave hold time* von 0.3 ms verursacht wird (siehe Abschnitt 2.2.2). Für sieben zusätzlich angeforderte Prozessoren, die je 0.3 ms leerlaufen (*busy waiting*) ergibt sich damit ein Aufwand von 2.1 ms. Dieser Aufwand kann durch Modifikation der *slave hold time* reduziert werden, was in Abhängigkeit von den Eigenschaften des Programms auch nachteilig sein kann.

Für Jobs mit einem Parallelisierungsgrad über 50% verlaufen die Meßwerte in Abbildung 5.19 jedoch völlig anders. Während die CPU-time nur noch geringfügig ansteigt, stagniert die Total-time. Die Stagnation der Total-time wird durch einen Rückgang der System-time bei Programmen mit einem höheren Parallelisierungsgrad als 50% bewirkt. In Abbildung 5.20 ist deutlich erkennbar, daß die System-time bei ungefähr 1220 parallelen Regionen das Maximum annimmt und

dann wieder abfällt. Eine Erklärung dieses Verhaltens wird bei der Untersuchung des *service*-Faktors in Tabelle 5.36 ersichtlich. Der *service*-Faktor steigt bei den Jobs YM10e-h trotz erhöhter Anzahl von parallelen Regionen nicht weiter. Die parallelen Jobs bekommen demnach keine zusätzlichen Prozessoren mehr zugeteilt, und es entsteht nur noch ein geringer zusätzlicher Aufwand. Die hohe Anzahl von gleichzeitig angeforderten Prozessoren scheint vielmehr die Wirkung zu haben, daß parallele Regionen beendet werden, bevor zusätzliche Prozessoren bereitgestellt werden. Dadurch kommt es dann zu einem geringeren Scheduling-Aufwand, der zu einer Reduzierung der System-time führt. Der Scheduling-Aufwand für eine parallele Region ist deshalb für einen vollständig parallelen Job in Messung YM4x geringer. Der Scheduling-Aufwand läßt sich durch den Vergleich der Werte aus Messung YM4x mit denen aus Messung YM10a berechnen. In der Messung YM10a wird das parallel übersetzte Programm mit einem Parallelsierungsgrad von 0% ausgeführt. Für eine parallele Region beträgt der Aufwand in Messung YM4x demnach 3.3 ms CPU-time, 0.8 ms System-time und 4.3 ms Sekunden Total-time. Diese Werte wurde auch für die Meßreihe mit erhöhter *slave hold time* gemessen. Abzüglich des durch PAR-Bench verursachten Aufwands von 0.5 ms beträgt der Aufwand der bei der Ausführung einer parallelen Region in einer Arbeitslast mit hochparallelen Jobs durchschnittlich ca. 3.6 ms.

5.4.3 Arbeitslasten mit hoher Speicherbelastung

Die bisherigen Leistungsmessungen wurden nur mit rechenintensiven Jobs durchgeführt, deren Ausführung nur marginal vom Speichersystem beeinflußt wurde. So erreicht ein rechenintensiver Job auf der CRAY Y-MP8/832 eine Speicherbelastung von ca. 70 MMREFS. Die Speicherbelastung summiert sich bei Nutzung aller acht Prozessoren auf ca. 560 MMREFS, was nur 6.5% der maximalen Speicherleistung der CRAY Y-MP8/832 entspricht. Ein Anstieg der CPU-time aufgrund von Zugriffsverzögerungen konnte dabei im Bereich der Meßgenauigkeit nicht festgestellt werden. Das Speichersystem der CRAY Y-MP8/832 wird wie schon auf der CRAY X-MP/416 und der CONVEX C3840 mit Hilfe der speicherintensiven Jobs untersucht, die auf der CRAY X-MP/416 mit FL = 130 MFLOPS und MM = 180 MMREFS generiert wurden. Aufgrund des erhöhten Prozessortaktes der CRAY Y-MP8/832 läßt sich für einen speicherintensiven Job eine Speicherbelastung von 255 MMREFS erwarten. Bei Ausführung dieser speicherintensiven Jobs auf allen acht Prozessoren beträgt die Speicherbelastung demnach 2040 MMREFS, das sind ungefähr 23% der maximalen Speicherbandbreite der CRAY Y-MP8/832.

Einprogrammbetrieb Als Referenz für die Bestimmung des Aufwands, der auf der CRAY Y-MP durch Zugriffsverzögerungen entsteht, dient eine Messung im Einprogrammbetrieb. In der Messung YM20 wird ein speicherintensiver Job im Einprogrammbetrieb ausgeführt. Der Aufwand, der durch Zugriffsverzögerungen entsteht,

wird dann in den Messungen YM20d und YM20f mit vier bzw. acht gleichzeitig ausgeführten speicherintensiven Jobs untersucht. In der Messung YM21d$^{\sigma}$ und YM21f$^{\sigma}$ wird der Job dann unter Nutzung von N=4 bzw. N=8 Prozessoren parallel ausgeführt. Die dabei erhaltenen Meßergebnisse werden in der Tabelle 5.37 wiedergegeben. Die parallelen Jobs werden unter Verwendung des Schedulers 5.X ausgeführt. Wegen der höheren Meßgenauigkeit wurden die Jobs als Hintergrundjobs gestartet und der SSD als Disk-Cache für die Ein-/Ausgabeoperationen verwendet.

Sequentielle Jobs						
Messung	Jobs	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	Real-time [Prozent]	Effizienz [Prozent]
YM20a	1	19.58	100.0	19.80	100.0	100.0%
YM20d	4	19.61	100.2	19.62	99.1	100.9%
YM20f	8	20.41	104.2	20.52	103.6	96.5%
Parallele Jobs						
Messung	N	CPU-time [Sek.]	CPU-time [Prozent]	Real-time [Sek.]	*speedup*	Effizienz [Prozent]
YM21d$^{\sigma}$	4	19.63	100.3	4.9	4.00	100.0%
YM21f$^{\sigma}$	8	20.93	106.9	2.6	7.61	95.7%

Tabelle 5.37: Effizienz und *speedup* für speicherintensive Jobs auf der CRAY Y-MP8/832
Messung YM20x: Sequentielle Ausführung in einer Arbeitslast aus N Jobs
Messung YM21x: Parallele Ausführung mit N Prozessoren

Aus der Tabelle 5.37 ist ersichtlich, daß die speicherintensiven Jobs auf der CRAY Y-MP8/832 nur in geringem Maß durch Zugriffskonflikte verzögert werden. Die Zugriffsverzögerungen führen in Messung YM20f mit acht speicherintensiven Jobs zu einem Anstieg der CPU-time von nur 0.83 Sekunden bzw. 4.2%. Bei der parallelen Ausführung eines speicherintensiven Jobs steigt die CPU-time des parallelen Jobs um 1.35 Sekunden bzw. 6.9% an. Zieht man von diesen 6.9% die 4.2% ab, die durch Zugriffsverzögerungen verursacht werden, so bleibt ein Scheduling-Aufwand von 2.7% der CPU-time übrig. Dieser Scheduling-Aufwand liegt sogar geringfügig unterhalb des Scheduling-Aufwands für die rechenintensiven Jobs. Als Erklärung dafür kann in Betracht gezogen werden, daß die Varianz der Messungen mit speicherintensiven Jobs höher ist und die geringer Meßgenauigkeit zu dem Unterschied führt.

Mehrprogrammbetrieb Bei der parallelen Ausführung eines speicherintensiven mit acht Prozessoren steigt die Ausführungsdauer im Einprogrammbetrieb geringfügig an. Bei den Messungen mit einer Hintergrundlast erreicht der parallele Job in Messung YM4a jedoch nur einen *service*-Faktor von 2.4, wobei diesem Job

weniger als vier Prozessoren gleichzeitig zugeteilt werden. Die Messungen YM21d und YM22d$^{\sigma}$ zeigen jedoch, daß die CPU-time eines mit vier Prozessoren parallel ausgeführten speicherintensiven Jobs auf der CRAY Y-MP um weniger als 0.2% ansteigt. Dieser zusätzliche Anstieg aufgrund von Zugriffsverzögerungen ist für den Durchsatz von Arbeitslasten bedeutungslos. Diese Vorüberlegungen werden durch eine Meßreihe bestätigt, die analog zu den Messungen C38M22 und C38M23 in Abschnitt 5.2.3 konstruiert wurden. Auf eine ausführliche Beschreibung der Messungen wird an dieser Stelle verzichtet. Die parallele Ausführung eines speicherintensiven Jobs mit einer Hintergrundlast führt damit zu keinem nachweisbar höheren Aufwand als die parallele Ausführung eines rechenintensiven Jobs.

5.4.4 Zusammenfassung der Meßergebnisse

Trotz der doppelten Prozessorzahl der CRAY Y-MP8/832 ist der Aufwand für die Ausführung von parallelen Programmen gegenüber der CRAY X-MP/416 nur geringfügig erhöht. Die Messungen auf der CRAY Y-MP8/832 führten zu folgenden Resultaten:

- Bei der Ausführung eines parallelen PAR-Bench-Jobs mit acht Prozessoren wird im Einprogrammbetrieb eine Effizienz von 96% erreicht. Die für den Scheduler 6.X gemessene Leistung liegt nur minimal unter der für den Scheduler 5.X gemessenen Leistung.
- Im Mehrprogrammbetrieb von parallelen Programmen entsteht nur ein geringer zusätzlicher Aufwand, der für die Messung mit acht vollständig parallelen PAR-Bench-Jobs bei 5.7% liegt. Dieser Aufwand wächst jedoch nicht linear mit der Anzahl der Prozessoren, sondern wird mit steigender Anzahl der Jobs geringer. Befinden sich nur ein oder zwei parallele Jobs gleichzeitig in Ausführung, so werden diese deutlich beschleunigt ausgeführt. Dabei wird ein *service*-Faktor von bis zu 3.2 erreicht.
- Bei der Ausführung von parallelen Programmen mit niedrigem Parallelisierungsgrad entstehen keine meßbaren Leerlaufzeiten. Der Scheduling-Aufwand für eine parallele Region ist von der Anzahl der parallelen Jobs in der Hintergrundlast abhängig. Bei einer geringen Anzahl von parallelen Jobs entsteht ein Aufwand von ca. 8.4 ms, während bei einer hohen Anzahl von parallelen Jobs ein Aufwand von ca. 3.6 ms verursacht wird.
- Im Mehrprogrammbetrieb mit ausreichender Hintergrundlast führt die parallele Ausführung von speicherintensiven Programmen zu keinem höheren Aufwand als die parallele Ausführung von rechenintensiven Programmen.

Parallele Programme können auf der CRAY Y-MP8/832 also auch im Mehrprogrammbetrieb effizient ausgeführt werden. Eine geringe Anzahl von parallelen Jobs

können auch im Mehrprogrammbetrieb deutlich beschleunigt werden, ohne das der Durchsatz der Y-MP8/832 signifikant reduziert wird.

5.5 Gegenüberstellender Vergleich der Meßergebnisse

Die hier zu vergleichenden Computersysteme erreichen im dedizierten Betrieb mit einem vollständig parallelen Programm eine hohe Effizienz. Im Mehrprogrammbetrieb führen diese Rechner das Scheduling paralleler Programme jedoch nach unterschiedlichen Prinzipien durch:

- Die Alliant FX/2816 verwendet eine statische Scheduling-Strategie, bei der die Prozessoren von der System-Administration in *cluster* eingeteilt werden müssen, die für die parallele Ausführung von Programmen genutzt werden können. Die als *cluster* konfigurierten Prozessoren führen parallele Programme bevorzugt aus. Stehen keine parallelen Programme zur Ausführung bereit, so zerfällt der *cluster* in seine Prozessoren, die dann sequentielle Programme verarbeiten.

- Der Scheduler von CONVEX verteilt freie Prozessoren dynamisch auf parallele Programme. Parallele Programme bekommen nur dann zusätzliche Prozessoren zugeteilt, wenn diese nicht durch andere Programme genutzt werden können. Der Scheduler basiert auf dem ASAP-Prinzip und wird durch Hardware unterstützt.

- Das Betriebssystem UNICOS 6.1 stellt einen dynamischen Scheduler zur Verfügung, der nicht benötigte Prozessoren nach Ablauf der *slave hold time* freiwillig wieder abgibt (CPI, siehe Abschnitt 2.2.2). Im Gegensatz zum Scheduler von CONVEX führt der UNICOS-Scheduler Programme auch dann parallel aus, wenn der Rechner mit sequentiellen Programmen ausgelastet ist.

Diese unterschiedlichen Scheduling-Strategien haben Auswirkungen auf die Ausführungsdauer von parallelen Programmen im Mehrprogrammbetrieb. In Abbildung 5.21 wird der durchschnittliche *service*-Faktor der parallelen Programme in Abhängigkeit von der Anzahl der gleichzeitig ausgeführten parallelen Programme dargestellt. Die Arbeitslast besteht in den Messungen für die CONVEX C3840, die CRAY X-MP/416 und die CRAY Y-MP8/832 aus 16 Jobs, von denen maximal 8 parallel ausgeführt werden. Für die Alliant FX/2816 mit 16 Prozessoren ist die Arbeitslast auf 24 Jobs erhöht, von denen ebenfalls bis zu 8 Jobs parallel ausgeführt werden. Da die typische Verarbeitung von parallelen Programmen untersucht werden soll, besitzen die sequentiellen und die parallele Jobs die gleiche Priorität.

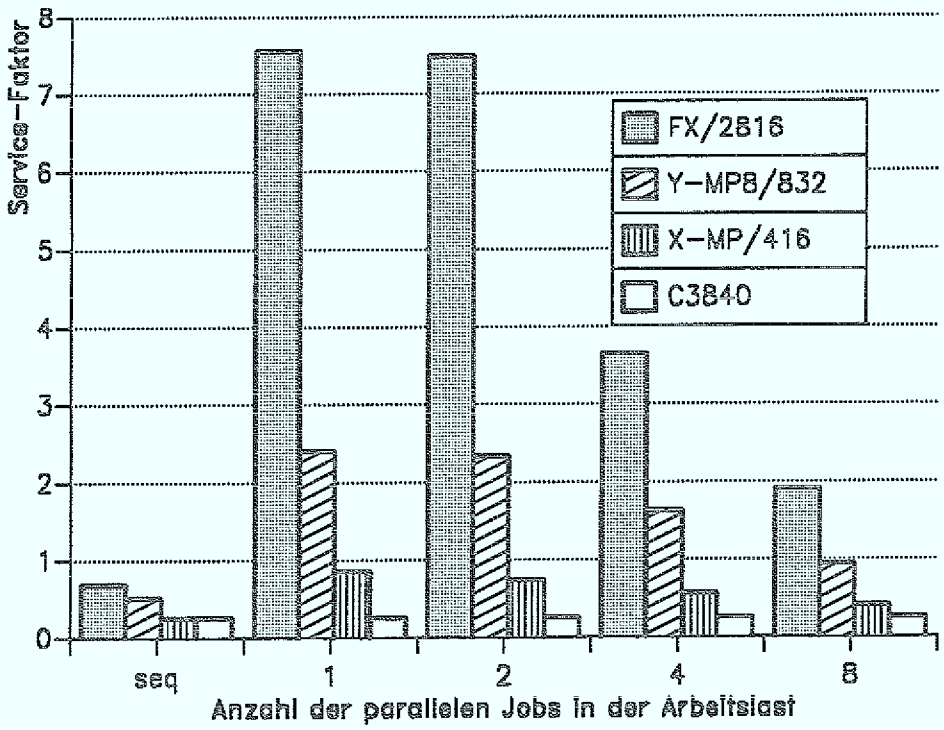


Abbildung 5.21: Der durchschnittliche *service-Faktor* eines parallelen Jobs in Abhängigkeit von der Anzahl der parallelen Jobs in einer Arbeitslast von 16 Jobs

Service-Faktor

Die Alliant FX/2816 ist in den Messungen als zwei *cluster* mit acht Prozessoren konfiguriert, was zur Folge hat, daß keine paralleler Job einen *service*-Faktor größer als acht erreichen kann. Dafür erreichen zwei parallele Programme beinahe denselben *service*-Faktor wie ein paralleles Programm. Bei mehr als zwei parallelen Jobs muß die Rechenzeit der zwei *cluster* auf die Jobs verteilt werden, mit der Folge, daß der *service*-Faktor abnimmt. Der *service-Faktor* der parallelen Jobs für die CRAY-Rechner ist aufgrund der niedrigeren Anzahl von Prozessoren geringer und erreicht bei der CRAY Y-MP8/832 mit 8 Prozessoren ein Maximum von 2.4 und bei der CRAY X-MP/416 ein Maximum von 0.86. Der gemessene *service*-Faktor fällt bei den CRAY-Rechnern ebenfalls mit steigender Anzahl von parallelen Jobs in der Arbeitslast. Im Gegensatz dazu ist der *service*-Faktor der parallelen Jobs bei der CONVEX C3840 konstant und entspricht dem *service-Faktor* eines sequentiell ausgeführten Jobs (erste Spalte in Abbildung 5.21). Bei der ausgewählten Arbeitslast erreichen also nur die Alliant FX/2816 und die CRAY Y-MP8/832 einen *service-Faktor* größer als eins und damit eine Beschleunigung von nicht höher priorisierten parallelen Programmen.

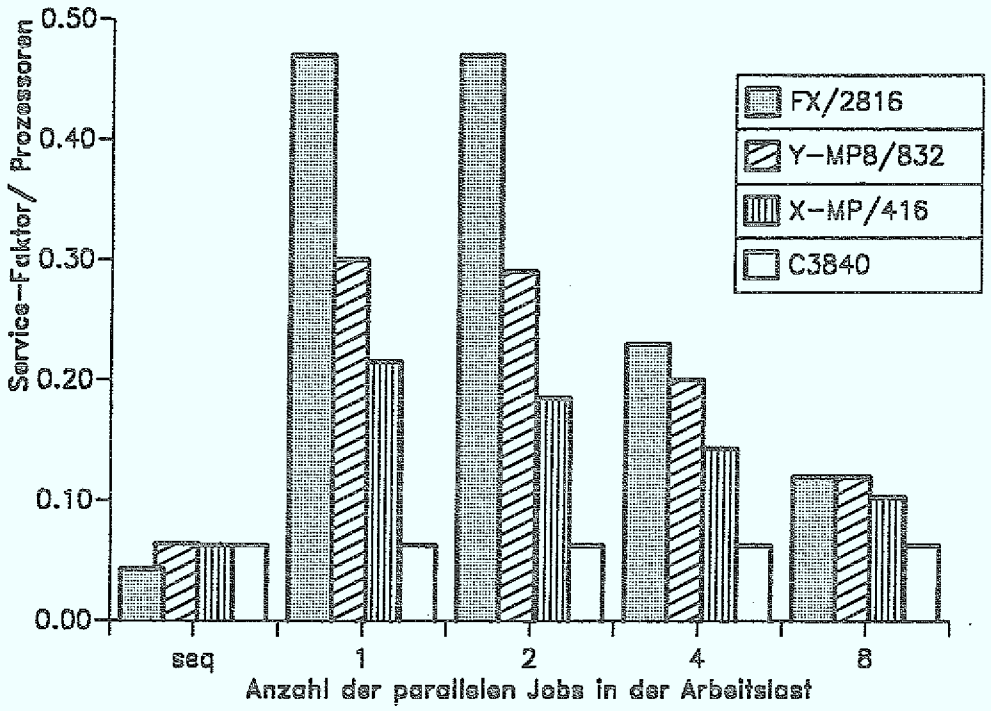


Abbildung 5.22: Der durchschnittliche normierte *service-Faktor* eines parallelen Jobs in Abhängigkeit von der Anzahl der parallelen Jobs in einer Arbeitslast von 16 Jobs

Normierter Service-Faktor

Ein Vergleich der Scheduling-Strategien der Rechner wird dadurch erschwert, daß in den Multiprozessorsystemen eine unterschiedliche Anzahl von Prozessoren installiert sind. Während der maximal erreichbare *service*-Faktor der CONVEX C3840 vier beträgt, liegt dieser Wert bei der Alliant FX/2816 bei sechzehn. In Abbildung 5.22 ist der *service*-Faktor deshalb normiert dargestellt, d.h. der erreichte *service*-Faktor wird durch die Anzahl der Prozessoren des Rechners dividiert. Dieser normierte *service*-Faktor gibt den Anteil am gesamten Computersystems wieder, der einem Job zugeteilt wird. Dieser Anteil beträgt für die Alliant FX/2816 für einen Job maximal 0.5, da ein *cluster* mit acht Prozessoren nur die halbe *Leistung* des Computersystems nutzen kann. Bei zwei parallelen Jobs werden beide *cluster* und damit die ganze FX/2816 durch parallele Jobs genutzt. Auf der CRAY Y-MP8/832 erreicht ein paralleler Job einen normierten *service*-Fakter von ungefähr 0.3, was etwas weniger als einem Drittel des gesamten Rechners entspricht. Der normierte *service-Faktor* der parallelen Jobs ist für die CRAY X-MP/416 geringer als für die CRAY Y-MP8/832, obwohl beide das gleiche Betriebssystem verwenden. Bei beiden ist der erreichte Wert für parallele Programm jedoch deutlich höher als für

sequentielle Programme. Parallele Jobs werden auf den Rechnern Alliant FX/2816, CRAY X-MP/416 und CRAY Y-MP8/832 bevorzugt ausgeführt, während sie auf der CONVEX C3840 die gleiche Prozessorzeit zugeteilt bekommen wie sequentielle Jobs.

Speichersystem

Das Speichersystem hat in den Messungen auf den Rechner CONVEX C3840, CRAY X-MP/416 und CRAY Y-MP8/832 keinen wesentlichen Einfluß auf die Ausführung von parallelen Jobs im Mehrprogrammbetrieb. Auf der Alliant FX/2816 ist dieser Einfluß aus zwei Gründen erheblich höher. Zum einen ist das Speichersystem leistungsschwach und empfindlich gegenüber einer hohen Belastung. Zum anderen werden parallele Jobs auch bei einer sequentiellen Hintergrundlast vollständig parallel ausgeführt. Bei der parallelen Ausführung eines speicherintensiven Jobs wird der Speicher deshalb durch alle Prozessoren eines *clusters* hoch belastet. In einer Messung wird der Durchsatz durch die Parallelisierung von speicherintensiven Jobs um 27.4% reduziert. Ein solcher Effekt konnte auf den übrigen Systemen nicht provoziert werden.

Leerlaufzeiten

Bei der parallelen Ausführung von Programmen mit niedrigem Parallelisierungsgrad bzw. ungünstiger *load balance* laufen Prozessoren auf einem dedizerten Rechner leer. Im Mehrprogrammbetrieb mit dynamischem Scheduling der Prozessoren wie auf der CONVEX und den CRAY-Rechnern werden leerlaufende Prozessoren für die Ausführung anderer Programme genutzt. Eine Reduzierung des Durchsatzes aufgrund von Leerlaufzeiten bei der parallelen Ausführung von Programmen mit niedrigem Parallelisierungsgrad läßt sich auf diesen Rechner nicht nachweisen. Im Gegensatz dazu können leerlaufende Prozessoren auf Computersystemen mit einer statischen Scheduling der Prozessoren wie der Alliant nicht genutzt werden. Bei einer ungünstigen Lastverteilung (*load balance* LB von 70%) wird der Durchsatz um 36% reduziert. Derselbe Effekt ist auch für parallele Programme mit niedrigem Parallelisierungsgrad zu erwarten. Aufgrund einer Inkonsistenz im Compiler wird der *on-chip*-Cache-Speicher in den mit der Compiler-Direktive *cvd$ noconcur* gekennzeichneten sequentiellen Regionen nicht mehr genutzt. Dadurch steigt in der Messung die Ausführungszeit des Programms um den Faktor 10, so daß die Leerlaufzeiten nicht quantifiziert werden konnten. Abschließend läßt sich feststellen, daß die bei einer statische Zuordnung von parallelen Prozessoren entstehenden Leerlaufzeiten einen drastischen Einfluß auf den Durchsatz eines Multiprozessorsystems haben können.

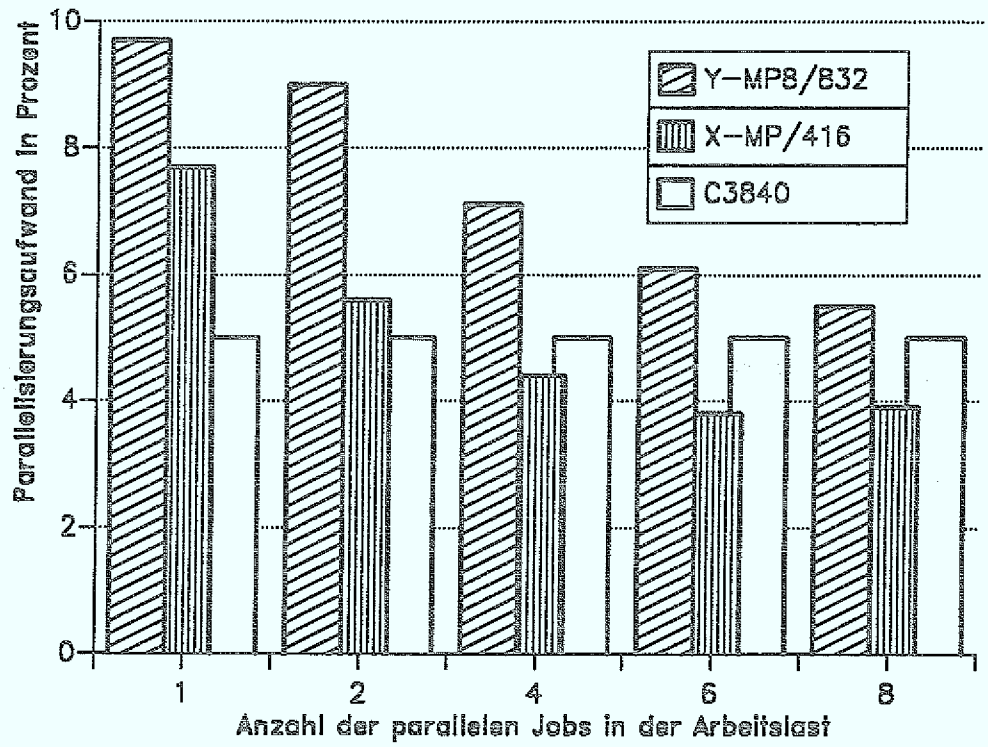


Abbildung 5.23: Zusätzlicher Aufwand bei der parallelen Ausführung der PAR-Bench-Jobs im Mehrprogrammbetrieb in Relation zum Prozessorzeitbedarf des Jobs bei sequentieller Ausführung

Scheduling-Aufwand

Der Scheduling-Aufwand konnte nur für die Rechner CONVEX C3840, CRAY X-MP/416 und CRAY Y-MP8/832 ermittelt werden, da die Meßergebnisse auf der Alliant FX/2816 einen zu großen Meßfehler aufweisen. In Abbildung 5.23 wird der Scheduling-Aufwand für einen hochparallelen Job in Arbeitslasten mit einer unterschiedlichen Anzahl von parallelen Jobs dargestellt. Der Aufwand wird in der Abbildung in Relation zur Summe aus CPU-time und System-time der Testjobs bei sequentieller Ausführung und gleicher Hintergrundlast abgetragen. Der Parallelisierungsaufwand, der bei der parallelen Ausführung eines Jobs auf der CONVEX C3840 entsteht, ist unabhängig davon, ob in der Hintergrundlast weitere Jobs parallel ausgeführt werden. Da dieser zusätzliche Aufwand weder bei der Prozessorzeit noch bei der Systemzeit abgerechnet wird, ist es für ein Benutzer in einer nicht dedizierten Umgebung unmöglich, diesen Aufwand zu bestimmen. Dieser Aufwand beträgt bei der CONVEX ungefähr 5% und entsteht, obwohl die Jobs der Arbeitslast nicht parallel ausgeführt werden. Bei den CRAY-Rechner hängt nicht nur der *service*-Faktor der Jobs von der Hintergrundlast ab, sondern auch der Parallelisierungsaufwand. Dieser Aufwand liegt für die gemessenen Arbeitslasten bei der CRAY

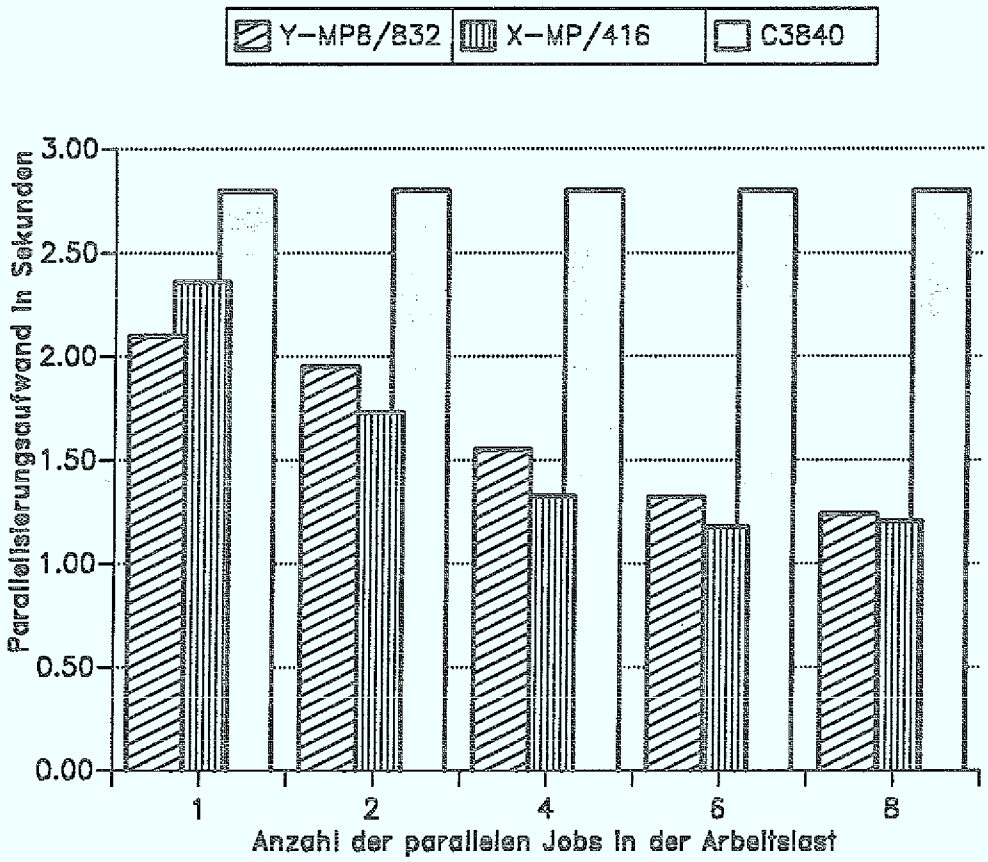


Abbildung 5.24: Zusätzlicher Aufwand bei der parallelen Ausführung der PAR-Bench-Jobs im Mehrprogrammbetrieb in Sekunden

X-MP/416 zwischen 4% und 8% und bei der CRAY Y-MP8/832 zwischen 5% und 10%. Der relative Parallelisierungsaufwand ist bei der Y-MP8/832 nicht wesentlich höher, obwohl die parallelen Jobs die doppelte Anzahl von Prozessoren anfordern.

Bei dem Vergleich des Parallelisierungsaufwands muß die unterschiedliche Ausführungsdauer der Arbeitslast auf den verschiedenen Rechnern berücksichtigt werden. Der oben betrachtete Aufwand ist daher kein Maß für Leistung der Multiprozessorsysteme in bezug auf die Durchführung von Scheduling-Operationen. In Abbildung 5.24 wird deshalb der absolute Scheduling-Aufwand in Sekunden graphisch veranschaulicht. Der parallele PAR-Bench-Job, der für diese Messungen verwendet wird führt 305 parallele Regionen aus. Die CONVEX C3840 benötigt zur Ausführung dieses parallelen Jobs ein Zeit von ca. 2.8 Sekunden. Auf der CRAY Y-MP8/832 fällt dieser Aufwand von ca. 2.1 Sekunden bei einem parallelen Job auf ca. 1.3 Sekunden bei acht parallelen Jobs. Für die CRAY X-MP/416 wurde in allen Messungen, bis auf der Messung mit einem parallelen Job, ein etwas geringerer Scheduling-Aufwand gemessen. Der Scheduling-Aufwand fällt bei der CRAY X-MP/416 von ca. 2.3 Sekunden auf ca. 1.2 Sekunden bei acht parallelen Jobs ab.

Der Parallelisierungsaufwand hängt von der Anzahl der parallelen Regionen ab, die von den Programmen ausgeführt werden. Diese Abhängigkeit wurde in Messungen mit unterschiedlichem Parallelisierungsgrad quantifiziert. Auf der CONVEX C3840 wird durch die Ausführung eines PAR-Bench-Jobs ein konstanter Aufwand von ca. 8 ms für jede parallele Region verursacht. Im Gegensatz dazu ist der Aufwand für eine parallele Region auf den CRAY-Rechnern von der Anzahl der von den allen parallelen Jobs der Arbeitslast angeforderten Prozessoren abhängig. Dieser Aufwand ist im Meßbereich für die CRAY X-MP/416 noch hinreichend linear, um einen Durchschnittswert von 4 ms zu berechnen. Bei der CRAY Y-MP8/832 ergibt sich ein Maximum von ca. 8 ms und ein Minimum von ca. 4 ms. Diese Werte zeigen, daß der absolute Aufwand, der durch die Parallelisierung auf den CRAY-Rechnern entsteht, geringer als der auf der CONVEX C3840 entstehende Aufwand ist. Diese Messungen zeigen jedoch insbesondere, daß der Scheduling-Aufwand bei einer niedrigeren Anzahl von parallelen Region geringer ist.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurde der zusätzliche Aufwand untersucht, der durch die Abarbeitung von parallelen Programmen auf Multiprozessorsystemen mit gemeinsamem Speicher entsteht. Leider gibt es generell weder genaue Herstellerinformationen noch Programmierwerkzeuge, die sowohl qualitativ als auch quantitativ eine Abschätzung dieses Aufwands erlauben. Deshalb wurden für vier sehr unterschiedliche Multiprozessorsysteme Kostenabschätzungen für die Bearbeitung von parallelen Programmen im Mehrprogrammbetrieb vorgenommen.

Die Messungen zeigen, daß auf Rechnern mit einer statischen Zuteilung der Prozessoren wie auf der Alliant FX/2816 parallele Programme nur dann einen geringen Einfluß auf den Durchsatz haben, wenn sie nahezu vollständig parallel und lastbalanciert abgearbeitet werden können. Ist dies nicht der Fall, so kann der Durchsatz des Systems durch die parallele Ausführung von Programmen signifikant reduziert werden. Bei der parallelen Ausführung von Programmen mit niedrigem Parallelisierungsgrad können Leerlaufzeiten auftreten, die höher als der Prozessorzeitbedarf des sequentiellen Programms sind; in einer Messung wurde mit einer ungünstigen Lastverteilung ein Anstieg der Prozessorzeit um ca. 36% beobachtet. Im Gegensatz dazu wird der Durchsatz auf Multiprozessorsystemen mit einer dynamischen Zuteilung der Prozessoren auch durch die parallele Ausführung von Programmen mit nicht idealen Eigenschaften nur geringfügig beeinträchtigt. Selbst in durchaus ungünstigen Fällen beträgt der auf den Systemen CRAY X-MP/416 und CRAY Y-MP8/832 gemessene Aufwand nur 4% bis 10% der Ausführungsdauer eines parallelen Programms. Dies gilt auch für das dynamische Scheduling der CONVEX C3840, das auf dem ASAP-Prinzip basiert. Da der durch die parallele Ausführung verursachte Aufwand jedoch weder bei der Prozessorzeit noch bei der Systemzeit abgerechnet wird, entsteht der Eindruck, daß keine zusätzlichen Kosten anfallen. In aufwendigen Meßreihen konnte jedoch der durchschnittliche zusätzliche Aufwand für ein vollständig parallelisiertes Programm auf ca. 5% quantifiziert werden. Dieser Aufwand hängt sowohl bei der CRAY X-MP/416 und der CRAY Y-MP8/832 als auch bei der CONVEX C3840 von der Anzahl der parallelen Regionen ab, die in einem Pro-

gramm ausführt werden. In günstigen Fällen mit nicht zu kurzen sequentiellen und parallelen Regionen entsteht deshalb auf den Rechnern mit dynamischem Scheduling nahezu kein zusätzlicher Aufwand. Dieses Ergebnis zeigt, daß schon mit heutigen *multitasking*-Implementationen sehr gute Ergebnisse erzielt werden können, die den universellen Einsatz in Produktionsumgebungen erlauben. Dies gilt insbesondere für Programme, bei denen die Granularität der parallelen Regionen ausreichend groß ist. In der Praxis hat sich jedoch gezeigt, daß für parallel ausgeführte Programmteile mit sehr kleiner Granularität in Einzelfällen deutlich höhere Effekte zu beobachten sind. In diesen Fällen kann dann durch manuelle Parameterwahl eine sequentielle Ausführung des Programms erzwungen werden. Diese Problemstelle muß in zukünftigen Realisierungen entweder durch deutlich verbesserte Analyseansätze im Compiler oder aber durch ein erweitertes Scheduling-Konzept, bei dem die *multitasking*-Implementation mit dem Betriebssystem-Scheduler Informationen austauscht, beseitigt werden.

Abschließend läßt sich sagen, daß das System PAR-Bench gut auf andere Multiprozessorsysteme mit gemeinsamem Speicher und parallelisierendem FORTRAN-Compiler portierbar ist. Den größten Aufwand verursachen dabei fehlende Standards bei einigen Systemaufrufen (Echtzeituhr) und bei den zur Parallelisierung benötigten Compiler-Direktiven. Eine Portierung von PAR-Bench auf weitere Rechner wie die z.B. die IBM ES/9000, die CRAY Y-MP C90 oder auch die NEC SX-3 würde das Rechnerspektrum für einen Vergleich noch erweitern. Die Flexibilität von PAR-Bench ermöglicht es, weitere Aspekte bei der Ausführung von parallelen Programmen zu analysieren. So könnte zum Beispiel der Einfluß der Ein-/Ausgabe oder der von interaktiven Teillasten auf die Ausführung von parallelen Programmen analysiert werden. Weiterhin können parallele Applikationen in die synthetischen Arbeitslasten mit eingebunden und so die Ausführung der Applikationen mit variierenden Hintergrundlasten untersucht werden.

Literaturverzeichnis

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *USENIX Technical Conference and Exhibition*, pages 93–113, 1986.

[AGS88] R. E. Anderson, R. G. Grimes, and H. D. Simon. Performance comparison of the CRAY X-MP/24 with SSD and the CRAY-2. *The Journal of Supercomputing*, 1(4):409–419, 1988.

[AHH88] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, 1988.

[All91] Alliant Computer Systems Corporation. *FX/2800 System Description*, March 1991.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullmann. *Compilerbau.* Addison-Wesley, 1988.

[BCG+91] D. Bradley, G. Cybenko, H. Gao, J. Larson, F. Ahmad, et al. Supercomputer workload decomposition and analysis. In *Proceedings 1991 International Conference on Supercomputing*, pages 458–467, 1991.

[BCK+89] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[BEK90] T. Bönniger, R. Esser, and D. Krekel. CRAY Y-MP, NEC SX-3 und SIEMENS VP-S. Vergleichende Darstellung von Höchstleistungscomputern. *Wirtschaftsinformatik*, 32(3):273–286, 1990.

[BH90] R. Berrendorf and J. Helin. Evaluating the basic performance of the intel iPSC/860 parallel computer. Technical Report KFA–ZAM–IB–9102, Forschungszentrum Jülich, 1990.

[Bie87] M. Bieterman. The impact of microtasked applications in a multiprogramming environment. In *Cray User Group Meeting (Fall)*, pages 144–148, 1987.

[Bla90] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–41, May 1990.

[BLR91] F. Barriuso, S. LaCroix, and S. Reinhardt. Exploiting fine-grain parallelism in a multiprogramming environment on CRAY Y-MP computer systems. To be published, 1991.

[BMN89] W. Brantley, K. P. McAuliffe, and T. A. Ngo. RP3 performance monitoring hardware. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 35–47. Addison-Wesley Publishing Company, 1989.

[Car89] R. J. Carpenter. Performance measurement instrumentation at NBS. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 159–184. Addison-Wesley Publishing Company, 1989.

[CB88] D. A. Calahan and D. H. Bailey. Measurement and analysis of memory conflicts on vector multiprocessors. In J. L. Martin, editor, *Performance Evaluation of Supercomputers*, pages 83–106. Elsevier Science Publishers B. V., 1988.

[CH91] T. M. Conte and W. W. Hwu. Benchmark characterization. *IEEE Computer*, 24(1):48–56, January 1991.

[CKM88] R. Cytron, S. Karlovsky, and K. McAuliffe. Automatic management of programmable caches. In *Proceedings 1988 International Conference on Parallel Processing*, volume 2, pages 223–230, 1988.

[CON90] CONVEX Computer Corporation. *CONVEX FORTRAN Language Reference Manual*, November 1990.

[Cra86] Cray Research, Inc. *CRAY X-MP Computer Systems: Four-Processor Mainframe Reference Manual*, 1986. HR–0097.

[Cra89] Cray Research, Inc. *CRAY Y-MP, CRAY X-MP EA, CRAY X-MP Multitasking Programmer's Manual*, 1989. SR–0222 F.

[Cra91] Cray Research, Inc. *CF77 Compiling System, Volume 4: Parallel Processing Guide*, 1991. SG–3074 5.0.

[CV90] H. Cheong and V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–47, June 1990.

[DI90] R. T. Dimpsey and R. K. Iyer. Performance degradation due to multiprogramming and system overhead in real workloads: Case study on a shared memory multiprocessor. In *Proceedings 1990 International Conference on Supercomputing*, pages 227–238, 1990.

[EHJ+91] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and W. Padua. Restructuring fortran programs for Cedar. In *Proceedings 1991 International Conference on Parallel Processing*, volume 1, pages 57–66, 1991.

[Gai89] B. Gaither. Instrumentation for future parallel systems. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 111–120. Addison-Wesley Publishing Company, 1989.

[Gel88] E. Gelenbe. *Multiprozessor Performance.* John Wiley & Sons Ltd., 1988.

[GSS87] E. F. Gehringer, D. P. Siewiorek, and Z. Segall. *Parallel Processing — The Cm* Experience.* Digital Press, 1987.

[GT90] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[Gul88] J. Gulbins. *UNIX: Eine Einführung in Begriffe und Kommandos von UNIX - Version 7, bis System V.3.* Springer Compass. Springer–Verlag, 1988.

[Hay88] J. P. Hayes. *Computer Architecture and Organisation.* McGraw-Hill, 1988.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[Her90] M. Herlihy. A methology for implementing highly concurrent data structures. *SIGPLAN Notes*, 25(3):197–206, 1990.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2.* Adam Hilger, 1988.

[HKN89] F. Hoßfeld, R. Knecht, and W. E. Nagel. Multitasking: Experience with applications on a CRAY X-MP. *Parallel Computing*, 12:259–283, 1989.

[Hof90] G. Hofemann. Vergleich der Speicherarchitekturen der Mehrprozessor-Vektorrechner CRAY X-MP und CRAY Y-MP anhand vektorisierender Algorithmen. Technischer Bericht Jül–Spez–555, Forschungszentrum Jülich, 1990.

[Hoß91] F. Hoßfeld. "Grand Challenges" – wie weit tragen die Antworten des Supercomputing? Interner Bericht KFA–ZAM–IB–9117, Forschungszentrum Jülich, November 1991.

[Int90] Intel Corporation. *i860 64-Bit Microprocessor Programmer's Reference Manual*, 1990.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, Inc., 1991.

[KTV+91] J. Konicek, T. Tilton, A. Veidenbaum, C. Zhu, et al. The organization of the Cedar system. In *Proceedings 1991 International Conference on Parallel Processing*, volume 1, pages 49–56, 1991.

[Lin90] M. A. Linn. Eine Programmierumgebung zur Messung der wechselseitigen Einflüsse von Hintergrundlast und parallelem Programm. Technischer Bericht Jül–2416, Forschungszentrum Jülich, 1990.

[LNVD91] M. A. Linn, W. E. Nagel, M. Vaeßen, and U. Detert. UNICOS 5.1 vs. UNICOS 6.0: Performance evaluation of autotasking. In *Cray User Group Meeting (Fall)*, pages 321–330, 1991.

[LV90] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessors scheduling. In *ACM SIGmetrics Conference*, pages 226–236, 1990.

[Mal89] A. D. Malony. Multiprocessor instrumentation: Approaches for Cedar. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 1–34. Addison-Wesley Publishing Company, 1989.

[Nag90a] W. E. Nagel. Parallelism on CRAY multiprocessor systems: Concepts of multitasking. In *Proceedings Ninth International Conference on Computing Methods in Applied Sciences and Engineering*, pages 393–412, 1990.

[Nag90b] W. E. Nagel. Performance Evaluation of Multitasking in a Multiprogramming Environment. Technical Report KFA–ZAM–IB–9004, Forschungszentrum Jülich, May 1990.

[Nag90c] W. E. Nagel. Prinzipien der Parallelverarbeitung auf Rechnern mit gemeinsamem Speicher. In A. Reuter, editor, *Proceedings 20. GI-Jahrestagung 1990, Informatik Fachbericht 257*, pages 403–417. Springer Verlag, 1990.

[Nel89] H. Nelson. Experience with performance monitors. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 201–208. Addison-Wesley Publishing Company, 1989.

[NL91a] W. E. Nagel and M. A. Linn. Benchmarking parallel programs in a multiprogramming environment: The PAR-Bench system. *Parallel Computing*, 17:1303–1321, 1991.

[NL91b] W. E. Nagel and M. A. Linn. Parallel programs and background load: Efficiency studies with the PAR-Bench system. In *Proceedings 1991 International Conference on Supercomputing*, pages 365–375, 1991.

[Oed85] W. Oed. Untersuchungen von Zugriffen auf den Arbeitsspeicher in Vektorrechnersystemen. Technischer Bericht Jül–1994, Forschungszentrum Jülich, 1985.

[PBG+85] G. F. Pfister, W. Brantley, D. George, S. Harvey, et al. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[PN85] G. F. Pfister and V. A. Norton. "Hot Spot" contention and combining in multistage interconnection networks. In *Proceedings 1985 International Conference on Parallel Processing*, pages 790–797, 1985.

[Pol88a] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[Pol88b] C. D. Polychronopoulos. Toward auto-scheduling compilers. *The Journal of Supercomputing 2*, 2:297–330, 1988.

[Pol89] C. D. Polychronopoulos. Multiprocessing versus multiprogramming. In *Proceedings 1989 International Conference on Parallel Processing*, volume 2, pages 223–230, 1989.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29:1184–1201, 1986.

[Reg89] H. Reger. Ein Vergleich der Multitasking-Implementierungen auf CRAY X-MP und IBM 3090. Technischer Bericht Jül–Spez–542, Forschungszentrum Jülich, 1989.

[Rei88] S. Reinhardt. Two parallel processing aspects of the CRAY Y-MP computer system. In *Proceedings 1988 International Conference on Parallel Processing*, volume 1, pages 311–314, 1988.

[Rei90] M. H. Reilly. *A Performance Monitor for Parallel Programs*. Academic Press, Inc., 1990.

[RT86] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, 29(12):1202–1212, December 1986.

[SIG89] SIGPLAN Special Interest Publication on Fortran. *Fortran 8X Draft*, May 1989.

[Smi82] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(4):473–530, 1982.

[SPG91] A. Silberschatz, J. L. Peterson, and P. Galvin. *Operating System Concepts.* Addison-Wesley Publishing Company, 1991.

[ST91] J. E. Smith and W. R. Taylor. Accurate modelling of interconnection networks in vector supercomputers. In *Proceedings 1991 International Conference on Supercomputing*, pages 264–273, 1991.

[Ste90] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23:12–24, June 1990.

[Sto90] H. S. Stone. *Computer Architecture and Parallel Processing.* Addison-Wesley Publishing Company, 1990.

[Tan90] A. S. Tannenbaum. *Betriebssysteme.* Carl Hanser Verlag, 1990.

[TC88] R. H. Thomas and W. Crowther. The uniform system: An approach to runtime support for large scale shared memory parallel processors. In *International Conference on Supercomputing*, pages 245–254, 1988.

[Tel89] P. J. Teller. Common memory working group summary. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 233–237. Addison-Wesley Publishing Company, 1989.

[Tel90] P. J. Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[TG89] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings 12th ACM Symp. Operat. Syst. Principles*, pages 159–166, 1989.

[TR86] A. Tevanian and R.F. Rashid. Mach: A basis for future unix development. In *Proceedings EUUG Autumn '86 Conference*, pages 407–411, 1986.

[Tuc86] S. G.. Tucker. The IBM 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[Uni89] J. Uniejewski. Characterizing system performance using application level benchmarks. In *Proceedings BUSCON Conference*, pages 159–167, 1989.

[vdPvK91] R. van der Pas and J. van Kats. Parallelism in a multi-user environment. *Parallel Computing*, 17:285–296, 1991.

[Wei90] R. P. Weicker. An overview of common benchmarks. *IEEE Computer*, 23(12):65–75, December 1990.

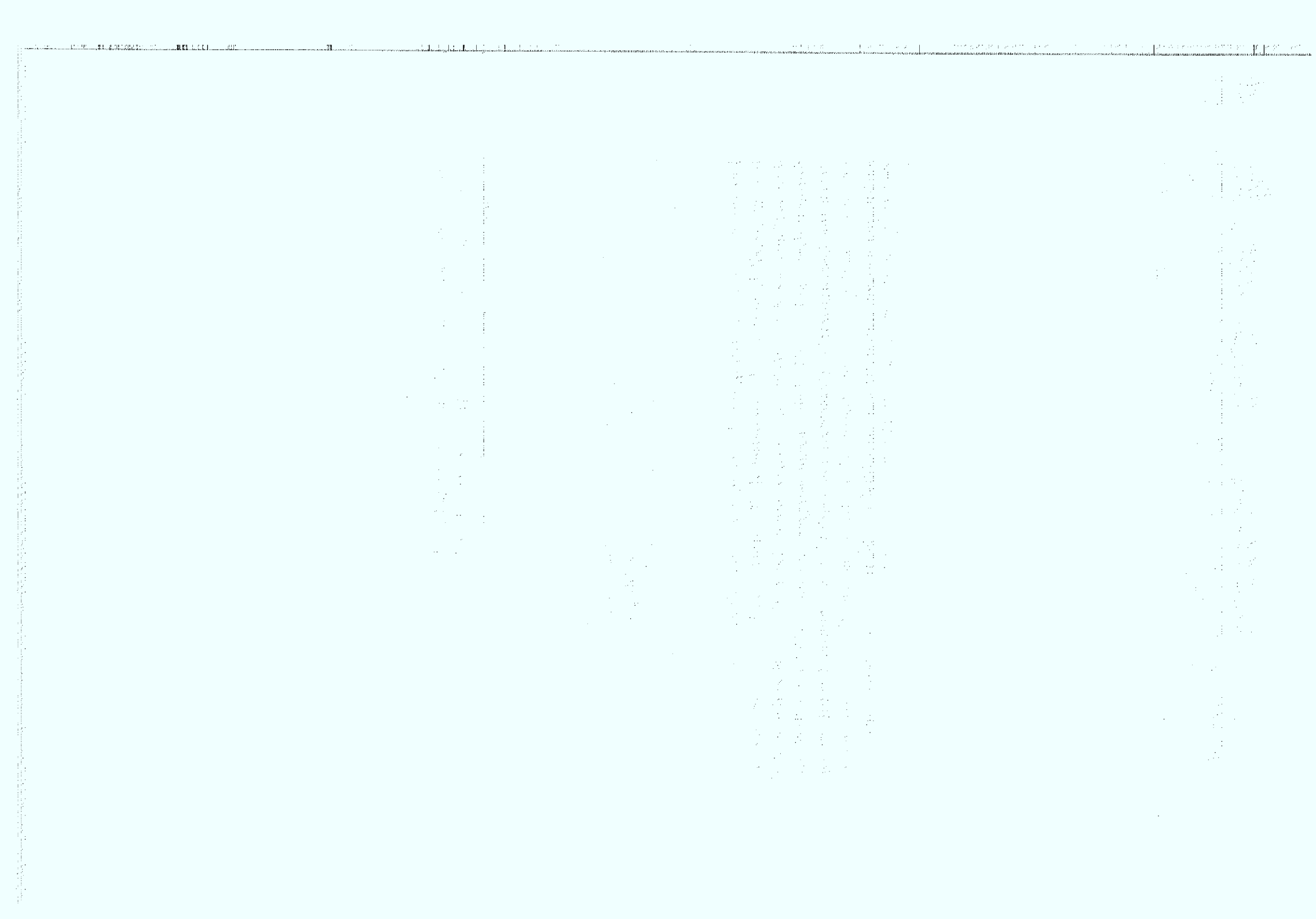
[Wil88] E. Williams. The effects of operating systems on supercomputer performance. In J. L. Martin, editor, *Performance Evaluation of Supercomputers*, pages 69–81. Elsevier Science Publishers B. V., 1988.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, 1990.

Diese Arbeit wurde als Diplomarbeit am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen angefertigt. Dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich zu erstellen.

Die durch die Institutsmitglieder erfahrene Unterstützung war eine wesentliche Voraussetzung für die Anfertigung dieser Arbeit. Mein besonderer Dank gilt Herrn W.E. Nagel für die intensive Betreuung und die vielfältigen Anregungen. Ebenfalls danken möchte ich Herrn Dr. N. Attig und Frau M. Sichelschmidt für die technische Betreuung bei den Messungen auf den CRAY-Systemen des ZAM.

Für die gute Unterstützung der Messungen auf der Alliant FX/2816, die im Labor für parallele Systeme der Gesellschaft für Mathematik und Datenverarbeitung durchgeführt wurden, möchte ich insbesondere Herrn Dr. T. Brandes und Herrn Springstubbe danken. Die Leistungsmessungen auf der CONVEX C3840 wurden durch die Geschäftsstelle Düsseldorf der CONVEX Computer GmbH ermöglicht, und mein Dank gilt dort den Herren C.-D. Kirst und Dr. U. Meier. Weitere Untersuchungen wurden durch die auf der C3840 des Deutschen Klimarechenzentrums in Hamburg zur Verfügung gestellten Ressourcen möglich; hier bin ich in besonderem Maß den Herren W. Sell, Dr. V. Gülzow und W. Stahl zu Dank verpflichtet.



FORSCHUNGSZENTRUM JÜLICH GmbH

KFA

Jül-2671
August 1992
ISSN 0366-0885