Dynamischer Multiprogrammbetrieb von Parallelrechnern

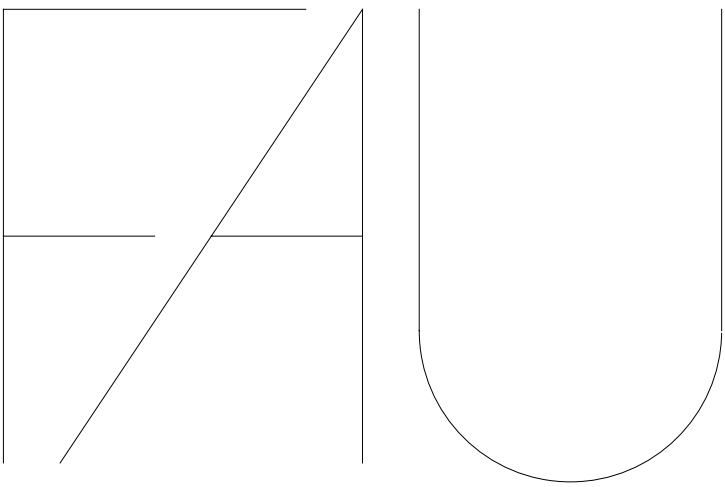
B. Rotzoll

November 1993 TR-I4-16-93

Interner Bericht

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)



Dieser Report wurde veröffentlicht in:

B. Rotzoll: “Dynamischer Multiprogrammbetrieb von Parallelrechnern”; In: H. Wedekind [Hrsg.] *Verteilte Systeme,* Grundl. und zukünft. Entw. aus der Sicht des SFB 182; Bibliographisches Institut, Mannheim; erscheint 1994

Dynamischer Multiprogrammbetrieb von Parallelrechnern

Babette Rotzoll

E-Mail: rotzoll@informatik.uni-erlangen.de

Teilprojekt B2
Entwurf und Implementierung eines an Hardwarearchitektur und Aufgabenklassen adaptierbaren Multiprozessorsystems

Zusammenfassung

Parallelrechner mit verteiltem Speicher werden üblicherweise im sog. *Space-Sharing*-Modus betrieben, d.h., eine Anwendung behält die ihr zugeteilten Knotenrechner exklusiv bis zu ihrer Terminierung. Ein Kennzeichen dieser Vorgehensweise ist, daß bei ihr die Zeit, die Jobs auf den Beginn ihrer Ausführung warten müssen, unabhängig vom Rechenzeitbedarf der Jobs ist.

Hier wird ein alternatives Job-Management-Konzept vorgestellt, das *Coscheduling auf Time-Sharing-Basis* genannt wird und mit dessen Hilfe es möglich ist, die Wartezeit der Jobs derart zu ihrem Rechenzeitbedarf in Beziehung zu setzen, daß kürzere Jobs eine geringere Wartedauer haben als langlaufende Jobs.

1 Motivation

Bei Parallelrechnern mit verteiltem Speicher, die im folgenden auch als *Multicomputer* bezeichnet werden, läßt sich eine ähnliche Entwicklung wie bei Monorechnern feststellen. Zuerst hat man Multicomputer nur im Batch-Betrieb genutzt, d.h., zu einem gegebenen Zeitpunkt lief nur eine einzige Anwendung im System. Später ist man dazu übergegangen, einen Multicomputer mehreren Benutzern gleichzeitig zugänglich zu machen, indem man die Ressourcen eines Multicomputers statisch unter den Benutzern aufgeteilt hat. Bei diesem auch als *Space Sharing* bekannten Konzept wird für einen Job vor der Ausführung eine geeignete Anzahl von Knotenrechnern und anderen Betriebsmitteln reserviert. Die am Anfang zugeteilten Betriebsmittel stehen dem Job dann für die gesamte Dauer seiner Ausführung zur Verfügung.

Das Space-Sharing-Konzept, das auch heutzutage üblicherweise bei Parallelrechnern mit verteiltem Speicher zur Verwaltung der Job-Abwicklungen eingesetzt wird, ist ursprünglich für Parallelrechner entworfen worden, bei denen die einzelnen Knotenrechner von einer Art Minimal-Betriebssystem gesteuert wurden. Die frühen Knotenrechner-Betriebssysteme unterstützten nicht das Vorhandensein mehrerer geschützter virtueller Adreßräume auf einem Knotenrechner und damit auch nicht einen Mehrbenutzer-Betrieb.

Die Parallelrechner-Entwicklungen der vergangenen Jahre haben einige Systeme hervorgebracht, bei denen neben den Kommunikationsverbindungen auch die einzelnen Knotenrechner immer leistungsfähiger wurden und bei denen die Peripherie-Hardware, vor allem der Hinter-

grundspeicher, immer mehr dezentralisiert wurde. Zugleich ist die Funktionalität der Knotenrechner-Betriebssysteme bei einigen Parallelrechnern um eine Reihe von Mechanismen erweitert worden, so daß nun z.B. bei vielen Systemen ein Mehrbenutzer-Betrieb auf jedem einzelnen Knotenrechner möglich ist. Ein Vertreter für ein derartig flexibles und leistungsfähiges Parallelrechnersystem ist die an der Universität Erlangen-Nürnberg entwickelte *MEMSY*-Architektur.

Vor dem Hintergrund des mächtigen, auf *UNIX* basierenden MEMSY-Knotenrechner-Betriebssystems [StTT93] stellt sich die Frage, ob es vielleicht lohnend wäre, vom statischen Konzept des Space Sharing abzurücken und bei der Job-Abarbeitung eine Form von *Time Sharing* vorzusehen. Mit Hilfe eines Time Sharing auf der Ebene paralleler Jobs wäre es möglich, die Wartezeiten der Jobs in Beziehung zu setzen zu ihrem Rechenzeit-Bedarf. Es ist möglich, dafür zu sorgen, daß kurze Jobs eine geringere Wartezeit haben als langlaufende Jobs. Dies könnte eine reizvolle Eigenschaft in Mehrbenutzer-Umgebungen sein, bei denen die Ausführungszeiten der Jobs stark unterschiedlich sind, z.B. wenn neben verhältnismäßig langen Abwicklungen von bereits fest installierten Software-Produkten auch kürzere Testläufe von sich noch im Entwicklungsstadium befindenden Programmen auftreten.

Nach einer genaueren Schilderung der betrachteten Problemstellung werden im dritten Kapitel Konzepte aus der Literatur zur dynamischen Job-Verwaltung auf Parallelrechnern erläutert. Im vierten Kapitel werden Ansätze für einen Time-Sharing-Betrieb auf einem Parallelrechner mit verteiltem Speicher und zweidimensionaler Gitter-Topologie (2-D-Gitter) vorgestellt. Die dargelegten Konzepte beziehen sich insbesondere auf MEMSY-Rechnersysteme.

2 Ausführung von parallelen Jobs: Annahmen und Zielsetzungen

Für die vorliegenden Betrachtungen sind nur parallele Jobs von Interesse. Der Begriff *Job* wird hier gleichbedeutend mit dem Terminus *Applikation* in [StTT93] verwendet. Die Ausführung eines parallelen Jobs besteht aus einer Menge von Prozessen, die zur Bewerkstelligung einer gemeinsamen Aufgabe miteinander kooperieren. Hier wird davon ausgegangen, daß der Anwender für jeden seiner Jobs den Knotenrechner-Bedarf und die gewünschte Verbindungstopologie angibt. Dabei wird zum einen angenommen, daß sich die Zahl der einer Anwendung zugeteilten Knotenrechner während ihrer Ausführung nicht ändert und zum anderen, daß die angeforderten Knotenrechner in einer rechteckförmigen Gittertopologie angeordnet sein müssen, d.h., Knotenrechner-Anforderungen haben die Form m x n. Die Aufgabe des Betriebssystems ist es, eine geeignete Knotenrechner-Teilmenge für jeden Job zu finden *(Knotenrechner-Allokation)* und den Prozessen der Jobs CPU-Zeit zuzuteilen *(Scheduling)*. Hier wird vorausgesetzt, daß der Rechenzeitbedarf der hereinkommenden Jobs i.a. nicht im voraus bekannt ist.

Bei der einfachsten Form der Job-Verwaltung auf Time-Sharing-Basis werden die Prozesse eines Jobs auf die Knotenrechner verteilt und jeder Knotenrechner, der gleichzeitig die Prozesse von mehreren Applikationen enthalten kann, entscheidet unabhängig von allen anderen Knotenrechnern, welchen Prozeß er jeweils für ein Zeitquantum zur Ausführung bringt. Bei diesem Ansatz würde man einen Multicomputer quasi als ein verteiltes System betrachten, das aus einer

Reihe miteinander über ein Netzwerk verbundener Einzelrechner besteht, die miteinander kommunizieren können, wobei aber die Betriebssysteme der Einzelrechner unabhängig voneinander arbeiten.

Viele parallele Anwendungen für Multicomputer sind dadurch gekennzeichnet, daß ihre auf verschiedenen Knotenrechnern residierenden Teilprozesse mit einer derart hohen Frequenz kommunizieren, daß es aus Leistungsaspekten erforderlich ist, alle Teilprozesse zugleich auszuführen. Für derartige Jobs muß ein sog. *Coscheduling* realisiert werden, d.h., es ist dafür zu sorgen, daß sich die Aktivitätsträger eines Jobs auf verschiedenen Prozessoren jeweils zugleich in Ausführung befinden, so daß z.B. ein einzelner Prozeß eines Jobs, der periodisch die Teilergebnisse von anderen Prozessen des Jobs kennen muß, um mit seiner Ausführung fortfahren zu können, nicht lange auf die Teilergebnisse der anderen Prozesse zu warten hat, da diese zur gleichen Zeit abgearbeitet werden.

Die hier verfolgte Zielsetzung besteht darin, einerseits ein vom Rechenzeitbedarf der Jobs abhängiges Antwortzeitverhalten und andererseits eine effiziente Abarbeitung solcher paralleler Jobs, deren Teilprozesse häufig miteinander interagieren, zu erreichen. Mit anderen Worten: es soll ein Coscheduling auf Time-Sharing-Basis entwickelt werden.

Dabei ist es sicherlich wünschenswert, daß eine Strategie, die ein Coscheduling auf Time-Sharing-Basis realisiert, kein schlechteres mittleres Antwortzeitverhalten aufweist als Space-Sharing-Strategien.

Zur weiteren Darstellung der vorliegenden Problematik sollen nun in der Literatur zu findende Ansätze zur Knotenrechner-Allokation und zum Scheduling von parallelen Applikationen in einem Multicomputer selektiv dargelegt werden. Hinsichtlich verdrängender Scheduling-Strategien für parallele Anwendungen auf Parallelrechnern seien hier zwei für die vorliegenden Zwecke besonders relevante Ansätze aufgeführt. Die meisten erwähnten Strategien für verdrängendes Scheduling auf Parallelrechnern beziehen sich auf Multiprozessoren mit gemeinsamem Speicher [Blac90] und sind nicht auf Parallelrechner mit verteiltem Speicher anwendbar.

3 Knotenrechner-Zuteilungsverfahren und Scheduling-Strategien für parallele Jobs

Bei der Time-Sharing-Abarbeitung paralleler Jobs auf Parallelrechnern ist das Zuteilen von Knotenrechnern an die Jobs (Knotenrechner-Allokation) sehr eng verbunden mit dem Bestimmen der Jobs, die als nächste Rechenzeit erhalten (Scheduling). Nach Ousterhout [Oust82] ist es wichtig, daß das Scheduling effizient ist, weil es häufig stattfindet, wohingegen Knotenrechner-Allokationen weniger oft vorkommen und daher auch komplexere Berechnungen erfordern können.

3.1 Knotenrechner-Allokation für den Space-Sharing-Betrieb

Die in der Literatur (u. a. [ChLe91], [DuHa91], [KiDa91], [KLRa91] und [LiSt88]) erwähnten Algorithmen zur Knotenrechner-Zuteilung für Parallelrechner beziehen sich vorwiegend auf Hypercubes. Die Knotenrechner-Zuteilungsverfahren für Hypercubes vergeben i. d. R nur Knotenrechner-Mengen mit Subkubus-Topologie. Für einen Job, der einen Subkubus der Größe 2^k benötigt (das Hypercube-System habe dabei 2^n Knotenrechner, $0 \leq k \leq n$), wird versucht, einen entsprechenden freien, d. h. von keinem anderen Job belegten, Subkubus im Rechnersystem zu finden. Diese Aufgabenstellung ist i. a. nicht mit einem nur polynomial mit der Problemgröße wachsenden Zeitbedarf optimal lösbar. Daher muß man den Suchvorgang auf eine sinnvolle Teilmenge aller in Frage kommenden Subkuben beschränken. Hierfür werden eine Reihe von Strategien vorgestellt, wobei Bottom-up- (z.B. sog. Gray-Code-, Buddy- und A^k-Algorithmen) und Top-down-Verfahren genannt werden. Die Buddy-Verfahren beispielsweise berücksichtigen dabei mit $2^{(n - k)}$ Subkuben von allen betrachteten Verfahren die kleinste Teilmenge von Subkuben.

Einige der in der Literatur genannten Allokationsalgorithmen lassen sich auf 2 D-Gitter-verbundene Rechnersysteme übertragen und könnten so als Knotenrechner-Allokationen-Komponente des Job-Management für MEMSY eingesetzt werden. Dabei ist allerdings zu beachten, daß nahezu alle in der Literatur erwähnten Zuteilungsverfahren Teilrechnersysteme nur in Größen vergeben, die sich durch Zweierpotenzen (mit ganzzahligem, positiven Exponenten) darstellen lassen. Dies kann zu einem unerwünscht hohen internen Verschnitt *(internal fragmentation)* führen, wenn beliebige Job-Größen vorliegen.

3.2 Zentrale Knotenrechner-Allokationsverfahren und Job-Scheduling-Strategien

In seinem “Matrix-Methode” genannten Algorithmus zur Job-Knotenrechner-Zuteilung und zum Scheduling von Jobs auf einem Parallelrechner geht Ousterhout [Oust82] davon aus, daß ein Rechnersystem aus P Knotenrechnern besteht, von denen jeder bis zu Q Prozesse multiplexen kann. Daher enthält der Prozeßraum P x Q Prozeß-”Slots”, denen Prozesse zugeteilt werden können. Bei der Matrix-Methode wird der Raum aller Prozeß-Slots logisch als eine Matrix mit Q Zeilen und P Spalten betrachtet. Jede Spalte enthält die Prozeß-Slots eines Prozessors und jede Zeile der Matrix enthält einen Prozeß-Slot von jedem Prozessor.

Die Knotenrechner-Zuteilung für einen parallelen Job (von Ousterhout als *Task Force* bezeichnet) läuft dann bei der Matrix-Methode darauf hinaus, jedem der Prozesse des Jobs einen leeren Prozeß-Slot innerhalb einer Matrix-Zeile zuzuweisen. Mit der ersten Zeile beginnend werden der Reihe nach soviele Zeilen durchgegangen, bis man eine Zeile gefunden hat, in der genug unbenutzte Prozeß-Slots zur Erfüllung des Knotenrechner-Bedarfs des neuen Jobs vorhanden sind.

Das Scheduling besteht aus der Anordnung der Ausführungsprioritäten der Prozesse auf jedem Prozessor. Ousterhout macht dabei die Annahme, daß die Slot-Zuteilung eines Prozesses während seiner Lebenszeit nicht verändert wird. Der *Scheduling*-Algorithmus der Matrix-Methode benutzt einen Round-Robin-Mechanismus, um das System zwischen den verschiedenen Zeilen der Matrix zu multiplexen. In der Zeitscheibe 1 wird jedem Prozeß in der Zeile 1 die höchste Ausführungspriorität gegeben, dadurch wird für alle Task-Forces in dieser Zeile ein Coscheduling realisiert. In der Zeitscheibe 2 werden alle Prozesse der Zeile 2 zur Ausführung gebracht usw.

Ousterhout ist in seinen Überlegungen davon ausgegangen, daß die von einer Task Force benötigte Knotenrechner-Menge keinen Topologie-Beschränkungen unterliegt. Im Rahmen dieser Arbeit soll aber angenommen werden, daß einer Anwendung, die einen Bedarf von m x n Knotenrechnern hat, ein rechteckig angeordnetes Subrechnersystem mit der Länge m und der Breite n zugeteilt wird. Das heißt, wenn die Matrix-Methode auf die hier untersuchte Problematik angewendet werden soll, ist es erforderlich, für die Bestimmung der zuzuteilenden Knotenrechner in einer Matrix-Zeile einen geeigneten Zuteilungsalgorithmus zu haben. Ousterhout nennt noch zwei weitere Scheduling-Algorithmen, die logisch gesehen die Zeilen der Knotenrechner-Prozeß-Matrix nebeneinander stellen und so eine Knotenrechner-Zuteilung über Zeilengrenzen hinweg gestatten. Dies läßt sich allerdings nicht geeignet auf ein 2-D-Gitternetzwerk übertragen.

3.3 Hierarchisch verteilte Scheduling- und Allokationsstrategien

Zentrale Knotenrechner-Allokationsverfahren und Job-Scheduling-Strategien sind nur bis zu einer gewissen Knotenrechner-Anzahl brauchbar. Für den allgemeinen Fall mit einer beliebig großen Knotenrechner-Zahl bietet es sich an, die Aufgabe des Job-Managements auf mehrere Rechner zu verteilen. Diese müssen ihre Job-Management-Entscheidungen zur Erzielung von Coscheduling geeignet koordinieren.

Einen verteilten Ansatz für die dynamische Betriebsmittelverwaltung auf Parallelrechnern, der die Knotenrechner in einer hierarchischen Kontrollstruktur anordnet, haben D. Feitelson und L. Rudolph [FeRu90] entwickelt. Ihr Konzept der *verteilten hierarchischen Kontrolle* bezieht sich allerdings auf Parallelrechner mit gemeinsamem Speicher. Das Konzept der verteilten hierarchischen Kontrolle ist logisch gesehen ein Master-Slave-Ansatz, nur daß die Funktion des Masters hier über mehrere Rechner verteilt ist. Ein Parallelrechner wird durch eine virtuelle Maschine repräsentiert, die die Form eines Binärbaums hat. Die Knoten in der untersten Ebene werden "Slaves" genannt. Sie führen Benutzer-Prozesse aus. Die Knoten in den darüberliegenden Ebenen stellen die Prozessoren des Master, die sog. *Controllers*, dar. Die Controllers sind zuständig für Knotenrechner-Allokationen und das Coscheduling von Prozeß-Gruppen (z. B. die Prozesse eines Jobs).

Ein Controller in einer gegebenen Ebene kann Anforderungen nach so vielen Slaves erfüllen, wie in seinem Unterbaum angesiedelt sind. Die Controllers in höheren Ebenen kümmern sich also um Aktivitäten, die einen großen Parallelitätsgrad aufweisen, wobei der Wurzelknoten ver-

antwortlich ist für Aktivitäten, die alle Slaves zusammen koordinieren müssen. Die Forderung nach Coscheduling impliziert, daß die Prozesse einer Prozeß-Gruppe verschiedenen Prozessoren zuzuteilen sind. Jeder Controller ist verantwortlich für das Scheduling der Prozeß-Gruppen, die ihm zugeordnet worden sind. Jedem Controller wird pro Scheduling-Runde eine Zeitscheibe zugeteilt, während derer er seine Prozeß-Gruppen auf seinen Slave-Prozessoren zur Ausführung bringt. Jede Scheduling-Runde startet beim Wurzel-Controller und setzt sich nach unten in der Baumhierarchie der Controllers fort.

Diesen Ansatz zur Prozeß-Gruppen-Verwaltung auf Multiprozessoren mit gemeinsamem Speicher kann man nicht ohne Modifikationen auf das MEMSY-System übertragen. Die Darstellung eines Rechnersystems als Binärbaum würde bei Vorliegen von beliebigen Job-Größen eine hohe Zersplitterung des Rechnersystems nach sich ziehen. Ferner kann das beschriebene Konzept nicht die im Rahmen dieser Arbeit erwünschte Topologie-Restriktion (rechteckförmige Subtopologien) berücksichtigen.

4 Entwicklung eines verdrängenden Job-Scheduling auf Coscheduling-Basis

Eine Job-Scheduling-Komponente für ein Parallelrechnersystem läßt sich unter den beiden Aspekten der Knotenrechner-lokalen Unterstützung für ein Job-Scheduling und der Koordinierung der Job-Verwaltung über Knotenrechner-Grenzen hinweg betrachten. Mit dem ersteren Gesichtspunkt sind die Mechanismen gemeint, mit denen auf lokaler Knotenrechner-Ebene garantiert wird, daß in einem gegebenen Zeitraum nur die Prozesse einer gewissen Teilmenge der vorhandenen Applikationen zur Ausführung gelangen. Welche Applikationen dabei gerade favorisiert werden, soll von einem oder mehreren Job-Scheduling-Strategie-Servermodulen bestimmt werden.

Die Zahl der eingesetzten Job-Scheduling-Strategie-Servermodule richtet sich dabei nach der Größe des vorliegenden Parallelrechnersystems. Bei kleineren Systemen ist es vorteilhaft, wenn man eine zentrale Job-Scheduling-Strategie-Komponente hat. Bei größeren Systemen, bei welchen ein zentrales Job-Management zum Flaschenhals werden könnte, erscheint es vielversprechend, ähnlich wie in [FeRu90], eine baumhierarchische Ordnung über den Scheduling-Strategie-Servermodulen einzuführen und das globale Job-Scheduling hierarchisch (aber verteilt) zu steuern. Jedes einzelne Scheduling-Strategie-Modul arbeitet genauso wie das Scheduling-Strategie-Modul im zentralen Fall, abgesehen davon, daß es (außer dem Server an der Baumwurzel) nur für eine gewisse Teilmenge der Knotenrechner des Gesamtsystems verantwortlich ist. Hier soll im weiteren nur auf das zentrale Job-Management eingegangen werden.

4.1 Knotenrechner-lokale Mechanismen für das Job-Scheduling

Die vorliegenden Betrachtungen beziehen sich insbesondere auf MEMSY-Knotenrechner. Ein MEMSY-Knotenrechner besteht aus mehreren Prozessoren mit gemeinsamem Arbeitsspeicher *(shared memory multiprocessor)*. Das Betriebssystem eines Knotenrechners muß als Basis eines globalen Applikationsscheduling die Funktionalität bereitstellen, auf das Kommando einer "befugten Instanz" hin einen *Jobkontextwechsel* durchzuführen. Bei einem Jobkontextwechsel werden die Prozesse der bis dahin favorisierten Applikation von der Ausführung suspendiert, und die Prozessoren des Knotenrechners werden den Prozessen einer anderen Applikation exklusiv bereitgestellt.

Dazu ist im Betriebssystem der MEMSY-Knotenrechner ein Systemaufruf namens *do_group_switch(applid)* eingeführt worden. Dieser bewirkt, daß alle anderen Prozesse zugunsten der Prozesse der Applikation mit der Identifikation *applid* von der Ausführung zurückgestellt werden.

Bei der Realisierung ist zu berücksichtigen gewesen, daß es auf einem Knotenrechner auch Prozesse gibt, die keiner Benutzerapplikation zugeordnet sind. Solche Prozesse stellen z.B. der Swapper, Kommunikationsserverprozesse und der Page Daemon dar, d.h. für die Funktion des Systems sehr wichtige Prozesse, für die gewährleistet werden muß, daß sie hinreichend oft bzw. schnell zur Ausführung gelangen. O.B.d.A. wird im folgenden davon ausgegangen, daß alle Benutzer-Prozesse zu Applikationen gehören. Alle Prozesse, die mit keiner Applikation assoziiert sind, werden als "Systemprozesse" bezeichnet.

Um zu garantieren, daß die Systemprozesse in akzeptabler Weise bedient werden, ist von der zentralen Warteschlange für laufbereite Prozesse *(Run Queue)* des Knotenrechner-Betriebssystems Abstand genommen worden. Dafür werden jetzt zwei Run Queues eingerichtet: *sysrunq* und *applrunq*. In die *sysrunq* werden alle laufbereiten Systemprozesse eingegliedert und in die *applrunq* alle laufbereiten Applikationsprozesse. Der *sysrunq* wird ein fester Prozessor (*sysprocessor* genannt) eines MEMSY-Knotenrechners zugeordnet. Alle übrigen Prozessoren sind für die Abarbeitung der *applrunq* zuständig. Wenn in der *sysrunq* gerade keine laufbereiten Prozesse eingetragen sind, kann der *sysprocessor* auch einen Prozeß aus der *applrunq* zur Ausführung bringen.

Den Applikationen wird jeweils eine Priorität zugeordnet. Zum Setzen der Priorität einer Applikation wird ein Systemaufruf namens *set_appl_pri(applid, pri)* eingeführt. Für jede Applikationspriorität *applpri* gibt es eine eigene Eintragliste $applrunq_{applpri}$ in *applrunq*. Jedem Applikationsprozeß sind zwei Prioritätswerte zugeordnet: seine Applikationspriorität und seine Prozeßpriorität. Die Prozeßpriorität entspricht den von UNIX vergebenen Prozeßprioritätswerten. Die Applikationspriorität ist für alle Prozesse einer Applikation gleich. Ein Applikationsprozeß mit der Applikationspriorität *applpri* und der Prozeßpriorität *ppri* wird in $applrunq_{applpri}$ eingetragen. Die Einreihung innerhalb von $applrunq_{applpri}$ geschieht basierend auf der Prozeßpriorität *ppri*. Die laufbereiten Prozesse der Applikation mit der höchsten Priorität sollen zur Ausführung gelangen. Innerhalb einer Applikation werden die Prozesse gemäß dem herkömmlichen UNIX-Scheduling zur Ausführung gebracht. Wenn gerade kein Prozeß der Applikation

mit der höchsten Priorität laufbereit ist, so kann vorgesehen werden, daß dann ein Prozeß der Applikation mit der zweithöchsten Priorität bzw. ein Job, dessen Teilprozesse nicht so häufig miteinander kommunizieren müssen, zur Ausführung ausgewählt wird.

4.2 Scheduling von Zeitscheiben-Klassen

Das Rahmengerüst der hier vorgestellten Job-Management-Komponente für Multicomputer mit 2-D-Gitter-Verbindungstopologie orientiert sich grob an Ousterhouts Matrix-Algorithmus. Es sind allerdings eine Reihe sehr wichtiger Erweiterungen und Modifikationen nötig. Ein wesentlicher Aspekt ist dabei, daß Ousterhouts Matrix-Algorithmus keine Topologie-Restriktionen für die den Jobs zugeteilten Knotenrechner-Mengen kennt. Im vorliegenden Fall sollen allerdings Topologie-Beschränkungen berücksichtigt werden: die den Jobs zugeordneten Knotenrechner müssen innerhalb der Gesamt-Topologie ein Rechteck bilden.

Ousterhouts Definition einer Zeile in der Knotenrechner-Prozesse-Matrix entspricht hier das Konzept einer *Zeitscheiben-Klasse*. Alle Jobs, denen im selben Zeitintervall CPU-Leistung von Knotenrechnern zugeteilt wird, gehören zu derselben Zeitscheiben-Klasse. Anders als im Matrix-Algorithmus ist es hier möglich, daß ein Job mit mehreren Zeitscheiben-Klassen assoziiert sein kann. Da davon ausgegangen wird, daß kein Prozeß-Migrationsalgorithmus zur Verfügung steht, sind einem Job in jeder Zeitscheiben-Klasse, der er angehört, dieselben Knotenrechner zugeteilt.

Das Job-Scheduling läuft in Runden ab. Jede Scheduling-Runde beginnt mit der ersten Zeitscheiben-Klasse und bedient der Reihe nach für die Dauer einer Zeitscheibe alle vorhandenen Zeitscheiben-Klassen.

Die für das Job-Management relevanten Ereignisse, auf die es zu reagieren gilt, sind die Ankunft eines neuen Jobs, die Terminierung eines Jobs, der Beginn einer neuen Scheduling-Runde, das Ablaufen der Zeitscheibe für eine Zeitscheiben-Klasse und der Beginn der Ausführung der Jobs einer Zeitscheiben-Klasse. Im folgenden wird die Reaktion der hier entwickelten Job-Management-Strategie, die *MPJ-CoSched&TS* (Management paralleler Jobs auf der Basis von Coscheduling und Time Sharing) genannt wird, auf die einzelnen Ereignisse dargelegt.

MPJ-CoSched&TS-Reaktion bei Ankunft eines neuen Jobs *j*:

> Wenn es in *nmplist* (Liste aller Jobs, denen noch keine Knotenrechner zugeteilt worden sind) keinen Job gibt, der bereits *RetryLim* Male bei der Knotenrechner-Allokation zugunsten “jüngerer” Jobs übergangen worden ist, soll unter Verwendung der *MPJ-CoSched&TS-Allokationen*-Komponente versucht werden, eine geeignete Menge von Knotenrechnern für *j* zu finden.

Die Suche beginnt bei der aktuell bedienten Zeitscheiben-Klasse. Es werden der Reihe nach alle verfügbaren Zeitscheiben-Klassen durchgegangen, bis eine Zuteilungsmöglichkeit gefunden wird bzw. bis alle Zeitscheiben-Klassen erfolglos sondiert worden sind. Wenn keine geeignete Knotenrechner-Menge für *j* gefunden werden konnte, wird *j* in *nmplist* eingetragen.

MPJ-CoSched&TS-Reaktion bei Terminierung eines Jobs *j*:

(1) Es wird überprüft, ob die betroffene Zeitscheiben-Klasse ZSK_a jetzt keinen Job mehr enthält. Falls dies zutrifft, wird die Zeitscheiben-Klasse eliminiert, und in der Scheduling-Runde wird mit der nächsten anstehenden Zeitscheiben-Klasse fortgefahren.

(2) Falls die Zeitscheiben-Klasse ZSK_a unter Schritt 1 nicht eliminiert worden ist, werden die in *nmplist* aufgeführten Jobs der Reihe nach durchgegangen. Mit Hilfe der *MPJ-CoSched&TS-Allokationen*-Komponente wird versucht, für jeden betrachteten Job eine Knotenrechner-Zuteilung in ZSK_a zu finden.

(3) Falls es noch freie Knotenrechner in ZSK_a gibt, wird versucht, diese durch Jobs aus anderen Zeitscheiben-Klassen zu besetzen (*Alternativ-Belegungen*).

MPJ-CoSched&TS-Reaktion bei Ablauf der Zeitscheibe für eine Zeitscheiben-Klasse ZSK_a:

(1) Es wird festgestellt, ob alle in der betrachteten Zeitscheiben-Klasse ZSK_a abgebildeten Jobs noch Alternativ-Belegungen in anderen Zeitscheiben-Klassen haben. Falls dies zutrifft, wird ZSK_a eliminiert.

(2) Die Scheduling-Kontrolle wird nun weitergereicht an die nächste Zeitscheiben-Klasse.

MPJ-CoSched&TS-Reaktion bei Beginn der Zeitscheibe für eine Zeitscheiben-Klasse ZSK_a:

Die CPUs der Knotenrechner des Parallelrechnersystems werden den Prozessen der Jobs von ZSK_a zugeteilt.

MPJ-CoSched&TS-Reaktion bei Beginn einer neuen Scheduling-Runde:

(1) Es wird festgestellt, ob es Jobs in *nmplist* gibt. Falls dies zutrifft, werden soviele neue Zeitscheiben-Klassen erzeugt, bis für alle Jobs aus *nmplist* eine Knotenrechner-Zuteilung in einer Zeitscheiben-Klasse gefunden werden kann bzw. bis das Limit von n_{max} Zeitscheiben-Klassen erreicht worden ist. Danach wird sondiert, ob noch Alternativ-Belegungen von anderen Zeitscheiben-Klassen möglich sind. Die neuen Zeitscheiben-Klassen werden am Anfang der Liste der Zeitscheiben-Klassen eingereiht.

(2) Bei der ersten Zeitscheiben-Klasse anfangend wird die neue Scheduling-Runde gestartet.

Die Einordnung der Jobs in *nmplist* erfolgt in der Reihenfolge des absteigenden Knotenrechner-Bedarfs der vorliegenden Jobs. Dies hat sich in Simulationen als günstig erwiesen. Um Aushungerungseffekte zu vermeiden, gibt es die Größe *RetryLim*. Einem Job, der bereits *RetryLim* Male bei der Knotenrechner-Vergabe durch "jüngere" Jobs übergangen worden ist, müssen als nächstem Job Knotenrechner zugeteilt werden.

Würde man bei jedem neu hereinkommenden Job, für den es bei den bestehenden Zeitscheiben-Klassen keine Zuordnungsmöglichkeit gibt, gleich eine neue Zeitscheiben-Klasse erzeugen, so könnte man keinen Vorteil ziehen aus freiwerdenden Knotenrechnern von anstehenden Job-Terminierungen in der aktuellen Scheduling-Runde. Hohe Performanz-Verluste wären die Folge, da es dann viele wenig ausgelastete Zeitscheiben-Klassen geben würde. Deshalb werden neue Zeitscheiben-Klassen nur zu Beginn einer Scheduling-Runde erzeugt. Da die Knotenrechner nicht für beliebig viele Applikationen multiplexbar sind, muß ein Limit (n_{max}) bestimmt werden für die Anzahl der zugleich vorhandenen Zeitscheiben-Klassen.

4.3 Das Zuteilen von Knotenrechnern in Zeitscheiben-Klassen

Die Allokationskomponente von *MPJ-CoSched&TS* ist dafür verantwortlich, für einen Job in einer gegebenen Zeitscheiben-Klasse nach einer Knotenrechner-Zuordnung zu suchen. Da hier davon ausgegangen wird, daß die Jobs dynamisch, zu vorher nicht bekannten Zeitpunkten im System eintreffen und terminieren, ist es nicht möglich, zu jedem Zeitpunkt einen optimalen Zuordnungszustand zu erhalten. Die Problematik erinnert an den Bereich der dynamischen Freispeicherverwaltung.

Die meisten der in der Literatur erwähnten Knotenrechner-Zuordnungsalgorithmen treffen die Annahme, daß der Knotenrechner-Bedarf jedes Jobs sich in der Form $2^n \times 2^n$ $(n \geq 0)$ darstellen läßt. Der hier vorgestellte Algorithmus soll aber auch den allgemeinen Fall berücksichtigen, daß beliebig große Rechtecke angefordert werden dürfen.

Da der Zuordnungszustand in einer Zeitscheiben-Klasse einer dynamischen, durch vorher nicht bekannte Ereignisse bewirkten Änderung unterworfen ist und dadurch eine vorher getroffene optimale Zuordnung rasch zunichte gemacht werden kann, wurde davon ausgegangen, daß ähnlich wie bei der Freispeicherverwaltung ein First Fit-Algorithmus bei der Suche nach geeigneten Knotenrechnern für einen Job ausreichend ist.

Für den eingesetzten zweidimensionalen First Fit-Allokationsalgorithmus FF-2G repräsentiert sich ein 2-D-Gitter-Parallelrechnersystem als eine Matrix, wobei jedes Matrixelement einem Knotenrechner entspricht. FF-2G geht bei seiner Suche nach Knotenrechnern für einen Job, der ein m x n großes Knotenrechner-Feld benötigt, die Matrix-Elemente in einer geeigneten Reihenfolge durch und prüft, ob der jeweils betrachtete Knotenrechner der linke untere Knotenrechner eines m x n großen freien Knotenrechner-Feldes ist. Im Erfolgsfall bricht der Suchvorgang ab. Für die Reihenfolge, mit der die Matrix-Einträge sondiert werden, gibt es verschiedene Möglichkeiten. Denkbar wären z.B. ein zeilenweises Vorgehen oder ein Absuchen gemäß der

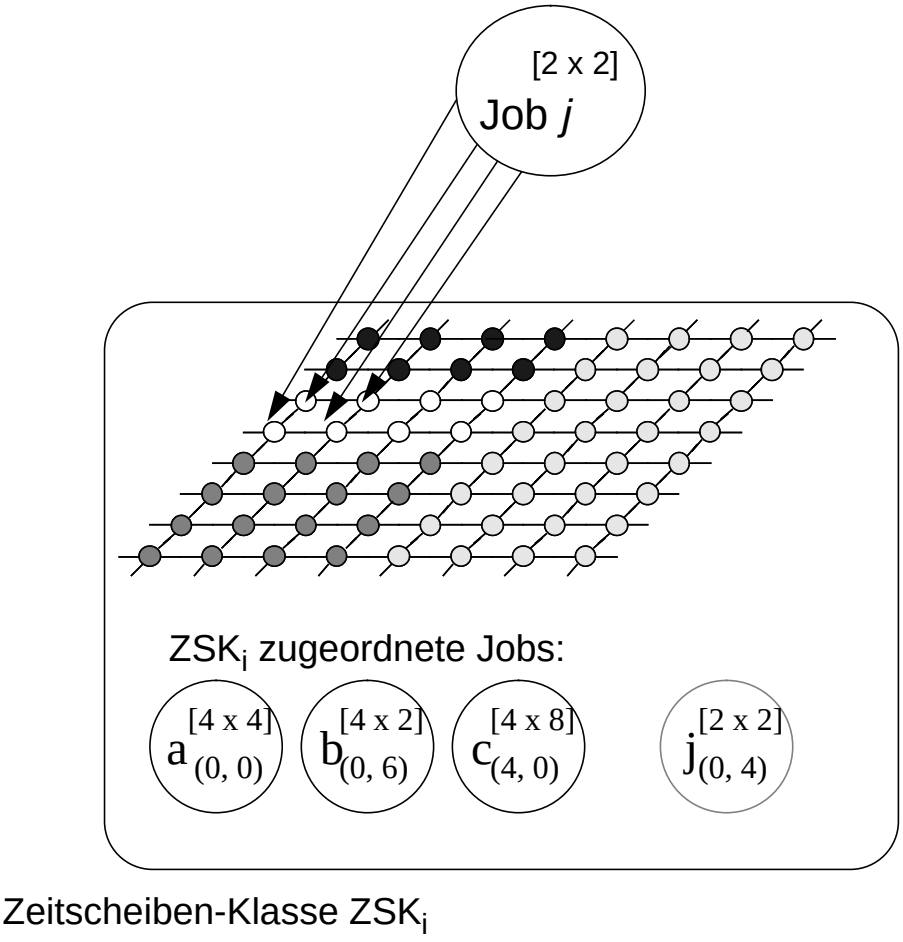


Abb. 4.1 Knotenrechner-Zuteilung für einen neuen Job innerhalb einer Zeitscheiben-Klasse

Reihenfolge, in der ein Buddy-Zuteilungsalgorithmus seine Einheiten bildet. Es ist eine Implementierung von FF-2G mit einem Zeitbedarf Θ (r x s) entwickelt worden (dabei bezeichne r x s die Größe des Parallelrechnersystems).

Beim Vorgang der Knotenrechner-Zuteilung in *MPJ-CoSched&TS* wird zunächst versucht, eine Knotenrechner-Zuordnung für einen neuen Job zu erlangen, ohne Alternativ-Zeitscheiben-Klassen-Belegungen von anderen Jobs zu verdrängen. Falls dies nicht möglich ist, wird nach einer Allokation für die um 90 Grad gedrehte Knotenrechner-Anforderung gesucht und bei erneutem Fehlschlag wird - falls vorhanden - eine geeignete Alternativ-Belegung eliminiert und durch den neuen Job ersetzt.

4.4 Simulationsergebnisse

Das Verhalten der beschriebenen Knotenrechner-Allokations- und Job-Scheduling-Strategie *MPJ-CoSched&TS* muß noch in ausführlichen Tests ergründet werden. Dies soll mit Hilfe von Simulationsexperimenten geschehen.

In bisherigen Simulationsexperimenten ist der Fall untersucht worden, daß alle Jobs Knotenrechner-Anforderungen haben, deren Größe sich in der Form 2^n x 2^n mit $n \geq 0$ (d.h., es werden nur quadratische Subtopologien der Größen 1, 4, 16 usw. vergeben) darstellen läßt. Modelliert wurde ein Rechnersystem, das aus 8 x 8 Knotenrechnern besteht. Die verschiedenen Möglichkeiten für die Job-Größen (1, 4, 16 und 64) werden als gleich wahrscheinlich angesehen. Die Ankunftszeiten werden als exponentiell verteilt angenommen. Bei den Bedienzeiten sind Exponential-, Normal- und Hyperexponential-Verteilungen betrachtet worden. In den Simulationsläufen ist *MPJ-CoSched&TS* mit einem Space-Sharing-Algorithmus verglichen worden, der auf Knotenrechner-Zuteilung wartende Jobs in der Reihenfolge absteigender Job-Größen betrachtet und ein *RetryLim* von 16 bzw. 64 (für Auslastungen von 70 % bei normalverteilten Bedienzeiten) verwendet. (Diese Space-Sharing-Ausprägung hat in den durchgeführten Simulationsexperimenten ein günstiges Verhalten gezeigt.) Es wird davon ausgegangen, daß die Bedienzeit unabhängig vom Knotenrechner-Bedarf eines Jobs ist.

In den Simulationsexperimenten hat sich gezeigt, daß mit *MPJ-CoSched&TS* das anvisierte Hauptziel erreicht worden ist, das darin besteht, daß die auftretenden Wartezeiten abhängig vom Rechenzeitbedarf der Jobs sein sollen, d. h., daß Jobs mit geringem Bedienzeitbedarf kleinere Wartezeiten als Jobs mit relativ hohem Bedienzeitbedarf haben. Zudem hat *MPJ-CoSched&TS* in keinem Fall eine schlechtere mittlere Antwortzeit aufgewiesen als der Space-Sharing-Algorithmus. Es kann sogar beobachtet werden, daß sich bei Erhöhung der Auslastung des Rechnersystems die Antwortzeit-Einsparung vergrößert, die mit *MPJ-CoSched&TS* im Vergleich zum Space-Sharing-Algorithmus bei derselben Auslastung erzielt werden kann (gemeint ist hier die mittlere Antwortzeit über alle Jobs). Die Simulationsergebnisse sind in den Abbildungen 4.2 und 4.3 für normal- und exponentialverteilte Bedienzeiten angegeben. Als Größe der Zeitscheibe wurde hier jeweils die Zeitspanne gewählt, innerhalb derer der Bedienzeit-Bedarf von 50 % aller Jobs liegt.

Als nächsten Schritt gilt es, das Verhalten von *MPJ-CoSched&TS* für den Fall zu betrachten, daß die angeforderten Knotenrechner-Teilmengen beliebige Rechtecke (nicht nur Quadrate nach Art des Buddy-Algorithmus) darstellen können.

5 Zusammenfassung

Parallelrechner mit verteiltem Speicher werden in der Praxis gewöhnlich im sog. *Space-Sharing*-Modus betrieben, d.h., die einer Anwendung einmal zugeteilten Ressourcen stehen dieser für die Dauer ihrer Ausführung exklusiv zur Verfügung. Ein Merkmal des Space-Sharing-Ansatzes ist, daß bei ihm die Wartezeiten der Jobs im System unabhängig vom Rechenzeitbedarf der Jobs sind.

Hier wurde ein alternatives Job-Management-Konzept vorgestellt, das Coscheduling auf Time-Sharing-Basis genannt wird und das im Gegensatz zum Space-Sharing-Ansatz bewirkt, daß Jobs mit einem geringeren Rechenzeitbedarf kleinere Wartezeiten haben als Jobs mit einem größeren Rechenzeitbedarf.

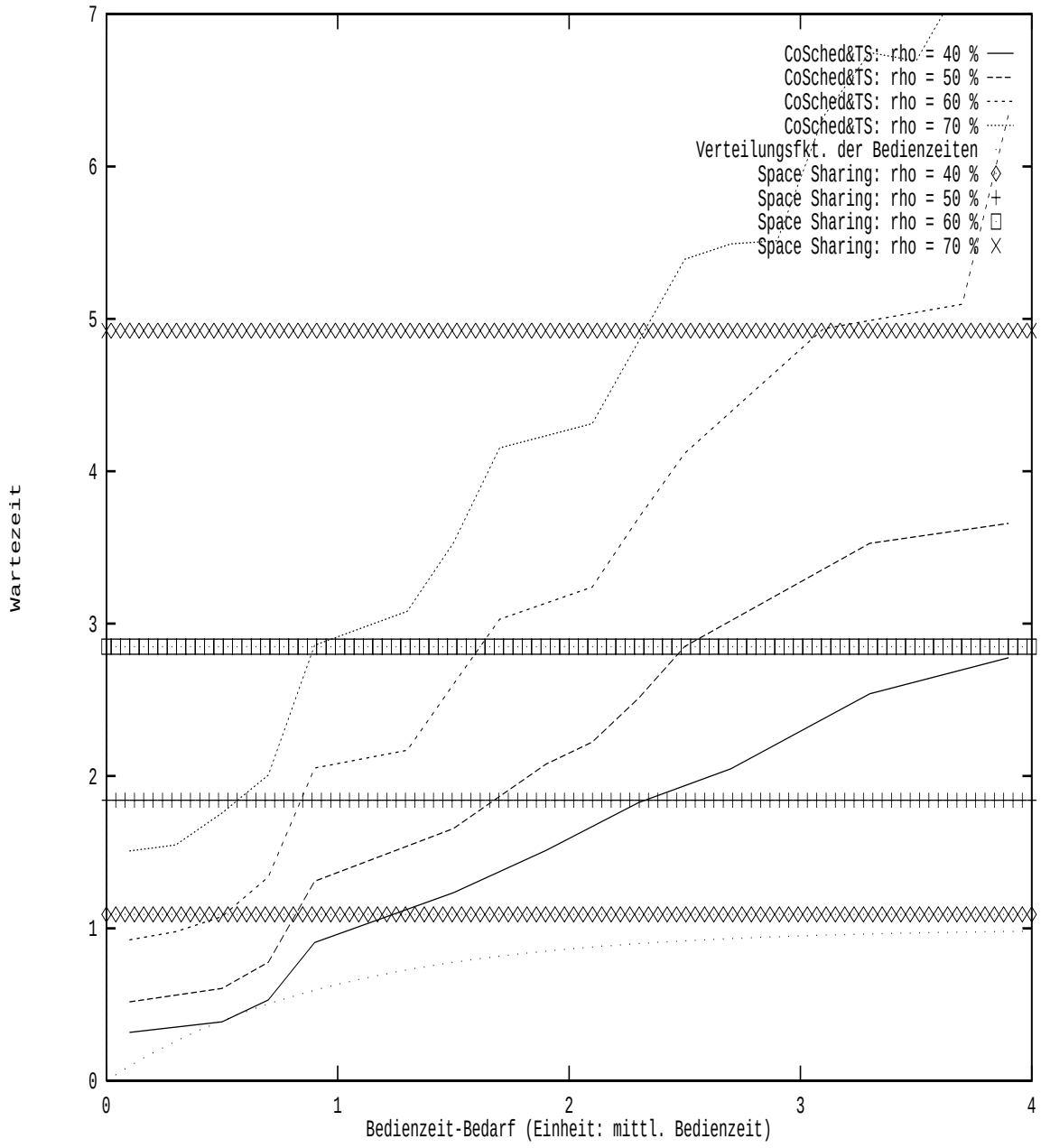


Abb. 4.2 Antwortzeit-Verhalten von *MPJ-CoSched&TS* und Space Sharing bei verschiedenen Rechnersystemauslastungen (rho) und Vorliegen einer Exponentialverteilung für den Rechenzeitbedarf der Jobs (d. h., 75 % aller Jobs haben keinen größeren Bedienzeit-Bedarf als 1,4 Zeiteinheiten)

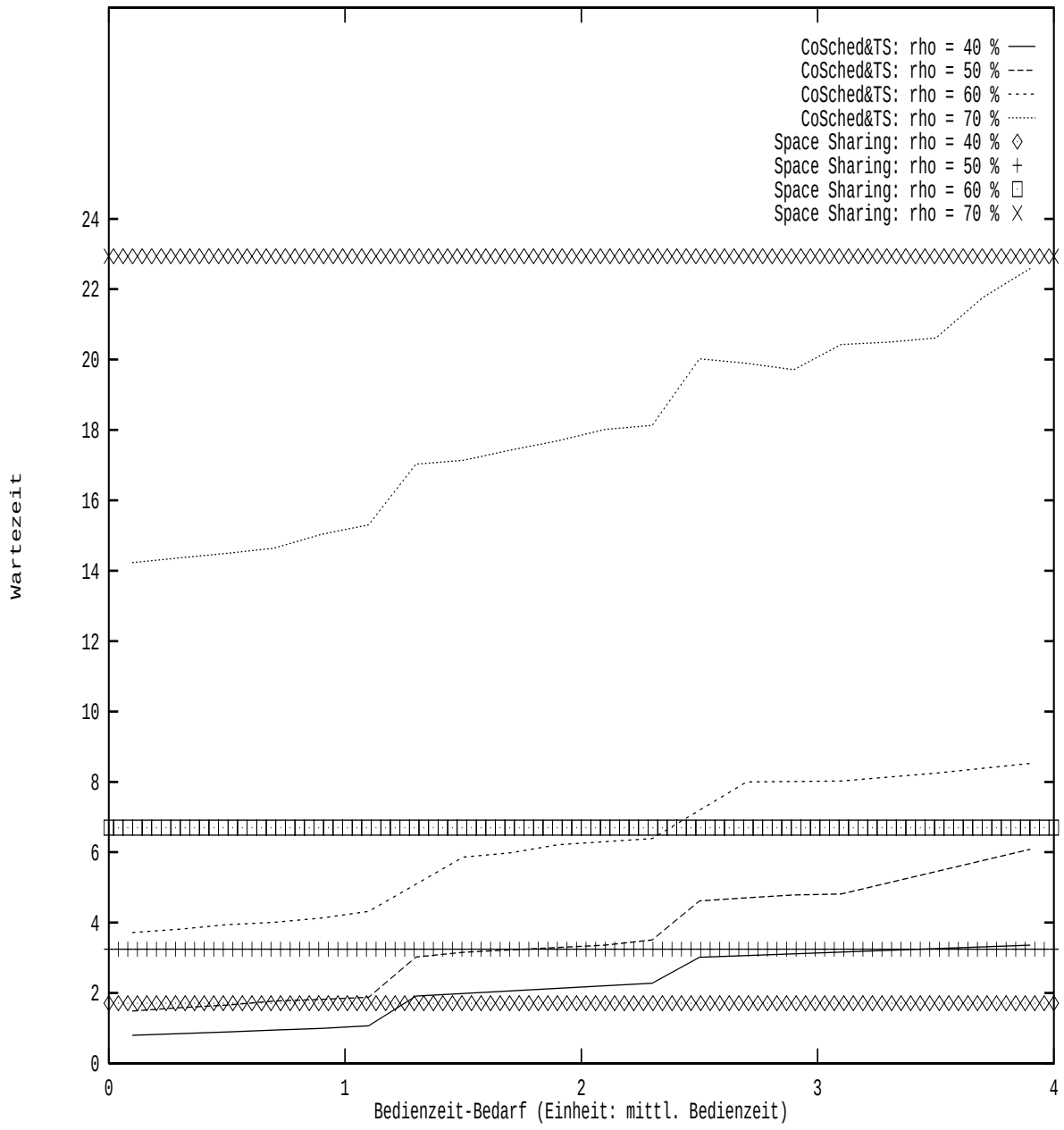


Abb. 4.3 Antwortzeit-Verhalten von *MPJ-CoSched&TS* und Space Sharing bei verschiedenen Auslastungen (rho) und Vorliegen einer Normalverteilung (mit dem Wert 1 für den Variationskoeffizienten) für den Rechenzeitbedarf der Jobs (d. h., 75 % der Jobs haben keinen höheren Bedienzeit-Bedarf als 1,8 Zeiteinheiten)

Literatur

AlGo89 George S. Almasi, Allan Gottlieb: “Highly Parallel Computing”, Benjamin/ Cummings division of Addison Wesley Inc., 1989

Blac90 David L. Black: “Scheduling and Resource Management Techniques for Multiprocessors”, Carnegie Mellon University, Diss., Juli 1990

ChLe91 K. Cheng, K. Lee: “Job Scheduling in a Partitionable Mesh Using a Two-Dimensional Buddy System Partitioning Scheme”, IEEE, Trans. on Parallel and Distributed Systems, Oktober 1990

DuHa91 Sh. Dutt, J. P. Hayes: "Subcube Allocation in Hypercube Computers", IEEE Transactions on computers, pp. 341 - 352, März 1991

FeRu90 D. Feitelson, L. Rudolph: “Distributed Hierarchical Control for Parallel Processing”, IEEE Computer, Mai 1990

JaCh98 Jai Eun Jang, Sung Woon Choi, Won Kyung Cho: “A New Approach to Processor Allocation and Task Migration in an N-Cube Multiprocessor”, Proc. Supercomputing ’89, November 1989, pp. 314 - 325

KiDa91 Jong Kim, Chita R. Das, Woei Lin: “A Top-Down Processor Allocation Scheme for Hypercube Computers”, IEEE Transactions on Parallel and Distributed Systems, 1/ 1991, pp. 20 - 30

KLRa91 Phillip Krueger, Ten-Hwang Lai, Vibha A. Radiya: "Processor Allocation vs. Job Scheduling on Hypercube Computers", Int’l Conf. Dist. Computing Systems, pp. 394 - 401

LiSt88 M. Livingston, Q. Stout: “Fault Tolerance of Allocation Schemes in Massively Parallel Computers”, Proc. Frontiers of Massively Parallel Processing, 1988

Oust82 John K. Ousterhout: “Scheduling Techniques for Concurrent Systems”, Int’l Conference on Distributed Processing, 1982

StTT93 Stukenbrock, W.; Thiel, Th.; Turowski, S.: Das Multiprozessorsystem MEMSY. In: H. Wedekind [Hrsg.] *Verteilte Systeme,* Grundl. und zukünft. Entw. aus der Sicht des SFB 182; Bibliographisches Institut, Mannheim; 1994