Einsatz von algorithmischen Skeletten im Scheduling massiv paralleler Systeme

Dissertation

Schriftliche Arbeit zur Erlangung des Doktorgrades des Fachbereichs Mathematik/Informatik an der Universität Gesamthochschule Paderborn

Bodo Kalthoff

Paderborn, den 15. Oktober 2002

Vorwort

Die vorliegende Arbeit entstand in wesentlichen Teilen während meiner Tätigkeit als wissenschaftlicher Mitarbeiter an der Universität-Gesamthochschule Paderborn.

Ich möchte Herrn Prof. Dr. F. J. Rammig für seine Unterstützung und zahlreichen Anregungen danken, die es mir ermöglichten, diese Arbeit parallel zu meiner späteren beruflichen Tätigkeit zu Ende zu führen.

Herrn Prof. Dr. W. Hauenschild danke ich für die Übernahme des Zweitreferates und das damit bekundete Interesse an dieser Arbeit.

Ferner danke ich Herrn Andreas Steffen für seine Hilfe bei der Implementierung der algorithmischen Skelette und den Mitarbeitern der AG Rammig für viele lebhafte Diskussionen.

Mein besonderer Dank gilt meiner Familie, insbesondere meiner Frau Antje für Ihre Geduld und Ihr Verständnis, ohne die diese Arbeit nie entstanden wäre.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Betrachtet man die Entwicklung der Parallelrechner, so erkennt man, daß die Zahl der Rechner mit mehreren hundert Prozessorelementen ständig zunimmt. Schaut man dagegen auf die Menge der auf solchen Rechnern lauffähigen Programme, so muß man feststellen, daß nur wenige Programme diese massiv parallelen Systeme effizient nutzen können. Je größer die Prozessorzahl eines Parallelrechners wird, desto kleiner wird die Zahl der Anwendungen, die eine sinnvolle Rechenlast darstellen. Stehen nicht genügend Anwendungen zur Verfügung, die ein hohes Maß an innerer Parallelität besitzen, ergibt der Einsatz eines massiv parallelen Systems aus ökonomischer Sicht keinen Sinn.
Ein vielversprechender Ansatz, diesem Dilemma zu entkommen, ist das Multiprogramming paralleler Programme. Beim Multiprogramming können mehrere parallele Anwendungen oder Jobs gleichzeitig auf einem Parallelrechner ablaufen. Der Einsatz von Schedulingverfahren ermöglicht es, die einzelnen Anwendungen räumlich und zeitlich zu koordinieren und somit die Performanz des Parallelrechners zu verbessern. Leider leistet die Systemsoftware vieler Parallelrechner keine hinreichende Unterstützung für das Multiprogramming und Scheduling. Ihre Zugangsverwaltungssysteme stellen oft nur eine Erweiterung klassischer Batchsysteme dar. Die Verwaltungssoftware für Parallelrechner des Paderborner Centers of Parallel Computing ist eine der wenigen Ausnahmen. Das dort entwickelte System versucht für einen kontinuierlichen Strom von paralleler Anwendungen ein Schedule zu finden, das einerseits eine hohe Auslastung der Parallelrechner, andererseits aber auch eine möglichst kurze Wartezeit garantiert. Als Grundlage zur Ermittlung des Schedules dienen Belegungsanforderungen der Nutzer, die sowohl Prozessorzahl als auch die gewünschte Belegungszeit angeben müssen. Informationen über den strukturellen Aufbau des Programms oder andere Programmparameter werden bei der Berechnung des Schedules nicht berücksichtigt. Diese

strikte Kapselung der parallelen Anwendung ermöglicht es, beliebige Programmiersprachen und Progammierpradigmen für die Anwenderprogramme zu nutzen. Die Qualität und Effizienz der so erzeugten Schedules kann verbessert werden, indem man diese enge Kapselung auflöst und eine stärkere Interaktion zwischen Scheduler und Anwenderprogrammen realisiert. Dieser Ansatz erfordert einen verstärkten Informationsfluß vom parallelen Programm zum Scheduler, sowie auf Seiten des Schedulers die Fähigkeit, Resourceanforderungen einer Anwendung zu beinflussen. Um dies zu erreichen, müssen die Anwendungen flexibel bezüglich der benötigten Resourcen sein. Dehnt man die Interaktion auf die Applikationsebene aus, sodaß benachbarte Applikationen Daten über ihren Berechnungsstatus austauschen können, ergiebt sich ein weiteres Steigerungspotential durch Synergieeffekte bei der Evaluation der Anwendungen. Dies kann zu einer weiteren Verbesserung der Effizienz führen. Bei der Entwicklung solch interagierender parallelen Programme ist es notwendig, geignete Programmierparadigmen auszuwählen, die eine Erzeugung von Schedulinginformationen unterstützen. Man erkauft sich die mögliche Effizienzsteigerung somit durch eine Einschränkung bei der freien Wahl des Programmiermodels bzw. der parallelen Programmiersprache. Auf Seiten der Schedulingsoftware müssen zudem Verfahren zur Verfügung stehen, die vorliegende Schedulinginformationen analysieren und darauf aufbauend optimierte Schedules generieren können. Schwerpunkt dieser Arbeit ist es, den Einsatz von Algorithmischen Skeletten in diesem Kontext zu motivieren. Dabei soll dem Leser nahegebracht werden, wie die Eigenschaften der hier vorgestellten Skelette genutzt werden können, um Schedulinginformationen zu generieren und so flexible parallele Anwendungen zu entwickeln, die mit geeigneten Schedulingverfahren interagieren können. Algorithmische Skelette bieten den Vorteil, daß in ihrer Implementierung alle Komponenten zur Interaktion mit anderen Anwendungen integriert werden können und somit völlig transparent für den Programmentwickler sind. Damit ist es für Entwickler möglich, ohne zeitlichen Mehraufwand parallele Programme zu entwickeln, die sich effizienter in ein Schedule integrieren lassen.

1.1 Strukturierung der Arbeit

Aufbauend auf einer grundlegenden Einführung der verwendeten Begriffe und Definitionen in Kapitel 2, stellt Kapitel 3 verschiedene Schedulingverfahren vor und diskutiert ihre Eigenschaften. Kapitel 4 schließt sich mit einer Einführung in die skelettbasierte parallele Programmierung an. Basierend auf diesen Grundlagen wird in Kapitel 5 ein Schedulingverfahren präsentiert, das die Eigenschaften skelettgenerierte Applikationen zur Optimierung

von Schedules nutzt. Kapitel 6 stellt die prototypenhafte Realisierung eines Schedulingsystems dar, das die Ansätze aus Kapitel 5 aufgreift und mittels skelettbasierter Informationen verschiedene unabhängige Optimierungsverfahren einsetzt. Anschließend wird eine Erweiterung des skelettbasierten Schedulings für heterogene Rechnernetze erörtert. Kapitel 7 schließt mit einer Zusammenfassung der Arbeit ab.

Kapitel 2

Effiziente Parallelverarbeitung

In diesem Kapitel werden verschiedene Ansätze zur Nutzung von Parallelrechnern vorgestellt und ihre Eignung für die effiziente Parallelverarbeitung diskutiert. Hierzu ist es notwendig, geeignete Effizienzmaße für die verschiedenen Ansätze zu definieren bzw. zu selektieren. Bei der Auswahl eines geeigneten Maßes spielt der gewählte Betrachtungspunkt eine starke Rolle. Je nach Sichtweise steht dabei die schnelle Abarbeitung eines parallelen Programmes (Nutzersicht) bzw. eine hohe Systemauslastung (Betreibersicht) im Vordergrund. Die verschiedenen Sichten können bei bestimmten Ansätzen zu widersprüchlichen Zielen führen. In diesen Fällen ist es notwendig, einen Kompromiß zu finden.

2.1 Single-Programming

Beim Single-Programming haben wir es mit dem klassischen Fall zu tun, daß eine einzelne Applikation auf einem Prozessornetz ausgeführt werden soll. Die Problematik besteht darin, die Applikation so auf das Prozessornetz abzubilden, daß die Ressource Rechenkapazität möglichst gut genutzt wird. Beim Single-Programming verschmelzen Betreiber- und Nutzersicht, da in diesem Fall eine schnelle Programmabarbeitung eine hohe Systemauslastung impliziert. Dies wird im nächsten Abschnitt verdeutlicht.

2.1.1 Speedup als Maß für Effizienz

Um die Effizienz der Berechnung einer Anwendung auf einem Parallelrechner zu beurteilen, wird im allgemeinen der *Speedup* bei paralleler Abarbeitung betrachtet. Der

$$\text{Speedup: } SP(p) = \frac{time(1)}{time(p)}$$

bezeichnet das Verhältnis von Rechenzeit einer Anwendung auf einem Einprozessorsystem $time(1)$ zur Rechenzeit $time(p)$ derselben Anwendung auf einem Parallelrechner mit p Prozessoren. Ein hoher Speedup ist gleichbedeutend mit einer hohen Auslastung des Parallelrechners und einer im Vergleich zur seriellen Abarbeitung kurzen Rechenzeit. Der maximale Speedup ist normalerweise durch die Anzahl der Prozessorelemente begrenzt. In einigen Fällen kann es allerdings vorkommen, daß der Speedup größer als die Zahl der Prozessoren ist. Dies ist meist durch einen zusätzlichen Berechnungsaufwand für die sequentielle Ausführung bedingt, der durch Speicherplatzbegrenzung und den dadurch entstehenden zusätzlichem Aufwand für Swapping bzw. Garbagge Collection auftreten kann. Da die Höhe des Speedups nur zusammen mit der Zahl der parallel eingesetzten Prozessoren eine sinnvolle Aussage über die Auslastung des Parallelrechners ergibt, wird oft der

$$\text{normierte Speedup: } SP(p)/p$$

benutzt. Dieses Maß wird oft als Effizienz bezeichnet, es besitzt jedoch nur im Kontext des Single-Programming diese Bedeutung.

2.1.2 Effizientes Single-Programming auf Parallelrechnern

Effizientes Single-Programming erzielt man, indem man den Speedup einer Anwendung optimiert. Hierzu ist ein möglichst gutes Mapping von Prozessen auf Prozessoren, bzw. Datenstrukturen auf Speicherelemente des Parallelrechners zu ermitteln. Bei vielen Problemen verändert sich die Rechenlast einzelner Programmteile während der Abarbeitung. Dadurch kann es passieren, daß die Rechenlast ungleichmässig verteilt ist und so der maximal mögliche Speedup nicht erreicht wird. In diesen Fällen ist es sinnvoll Lastausgleichsverfahren einzusetzen, um eine Rebalancierung der Rechenlast zu erreichen. Für die Durchführung bzw. die Integration solcher Verfahren ist meist der Anwender verantwortlich. Effizientes Single-Programming hängt damit primär von der Struktur der Anwendungen ab.

2.1.3 Grenzen des Single-Programming

Da die Effizienz durch den erzielten Speedup einer Anwendung bestimmt wird, ergeben sich natürliche Grenzen für die Auslastung des Parallelrechners. Wird der maximale Speedup einer Anwendung schon bei einer Prozessorzahl erreicht, die kleiner ist, als die Zahl der vorhandenen Prozessoren, so

kann durch den Einsatz zusätzlicher Prozessoren keine weitere Verbesserung erzielt werden. Sinkt der Speedup bei erhöhter Prozessorzahl, so kommt es sogar zu einer Verschlechterung der Effizienz. Das Single-Programming bietet somit aus Betreibersicht nur eingeschränkte Möglichkeiten, um einen Parallelrechner effizient zu nutzen.

2.2 Paralleles Scheduling

Im Fall des parallelen Scheduling haben wir es nicht mit einer einzelnen, sondern mit einer Menge von parallel ausführbaren Anwendungen zu tun, die abgearbeitet werden sollen. Diese Programme sind voneinander unabhängig. Zu ihrer korrekten Abarbeitung ist somit keine Kommunikation zwischen den Anwendungen notwendig. Unter dem Begriff des parallelen Schedulings wollen wir alle Verfahren zusammenfassen, die den Parallelrechner räumlich oder zeitlich so aufteilen, daß mehrere Applikationen gleichzeitig bzw. zeitlich verschränkt abgearbeitet werden. Die hierzu benutzten Verfahren hängen stark von der Struktur der abzuarbeitenden Jobs, dem benutzten Parallelrechner, sowie den Zielen des Schedulings ab. Als Hauptziel soll dabei die effiziente Nutzung der Rechenresourcen im Vordergrund stehen.

2.2.1 Effizienz und andere Ziele

Neben der effizienten Nutzung des Parallelrechners können Schedulingverfahren noch andere Ziele verfolgen. Diese Anforderungen an das Scheduling können dabei entweder vom Benutzer oder vom Betreiber vorgegeben werden. Die Tabelle 2.1 listet, angelehnt an die Klassifikation von Rudolph und Feitelson [FR96], die gängigsten Ziele auf. Man kann hierbei zwischen qua-

	Benutzer orientiert	System orientiert
Metrik	mittlere Antwortzeit	maximaler Durchsatz, Auslastung
Ziel	Zuteilung der gewünschten Ressourcen	Resourcenzuteilung nach administrative Präferenzen

Tabelle 2.1: Klassifikation von Scheduler Zielen

litativen und quantifizierbaren Zielen unterscheiden. Quantifizierbare Ziele sind solche, die mittels einer geeigneten Metrik gemessen werden. Die Metriken stellen dabei ein Kostenmaß zur Verfügung, das vom Scheduler optimiert werden soll. Die qualitativen Ziele eines Schedulers stellen dagegen Nebenbedingungen dar, die vom Scheduler garantiert werden müssen und so die

Generierung eines Schedules erschweren. Die direkten Auswirkungen qualitativer Ziele auf das Schedule hängen stark vom benutzten Schedulingverfahren ab. Daher werden sie in diesem Abschnitt nicht näher betrachtet, sondern später diskutiert. Wir wollen uns an dieser Stelle auf das Primärziel Effizienz konzentrieren. Dazu definieren wir analog zu [FRS+96] die folgenden Notationen:

t_i	Terminierungszeit von Job i in einem Schedule
s_i	frühest mögliche Startzeit für Job i
w_i	Gewichtung von Job i

Die Termininierungszeit t_i spezifiziert die Zeit, an der die Abarbeitung eines Jobs beendet ist. Sie bildet damit die Summe aus der reinen Rechenzeit und der Wartezeit bis zum Start der Abarbeitung dieses Jobs. Die frühest mögliche Startzeit s_i gibt dabei den Beginn der Wartezeit an. Sie beschreibt den Zeitpunkt, an dem der Scheduler Informationen über den neuen Job erhält. In vielen Schedulingansätzen wird für alle Applikationen $s_i = 0$ gefordert. Dies bedeutet, daß alle Applikationen vorab bekannt sein müssen. Die Gewichtung von Jobs w_i ermöglicht es, unabhängig von dem Prozessor- und Zeitbedarf einer Anwendung, bestimmte zu Jobs zu priorisieren. Damit ist es möglich von außen in die Berechnung des Schedules einzugreifen.

Auslastung

Aus Sicht des Betreibers wird ein Parallelrechner effizient genutzt, wenn alle Prozessoren kontinuierlich an der Abarbeitung der Applikationen beteiligt sind. Die Auslastung des Parallelrechners ergibt sich dann aus dem Prozentsatz der Prozessorzeit, die im Laufe des kompletten Schedules den Anwendungen zugeteilt wird. Bei dieser Sichtweise spielt es keine Rolle, wie effizient eine Anwendung die zugeteilte Prozessorzeit nutzt. Eine solche Definition der Auslastung findet man oft in kommerziellen Systemen, die dem Nutzer, die Belegungszeit für seine Applikationen in Rechnung stellen. Existiert kein solches Abrechnungswesen, d.h. wird die Benutzung des Parallelrechners z.B. pauschal abgerechnet, so ist es sinnvoll, die Auslastung auch auf Applikationsebene zu betrachten. Man definiert dann die Auslastung als den Prozentsatz der Prozessorzeit, die wirklich zur Berechnung von Applikationen benutzt wird. Verfolgt der Betreiber die Maximierung der Auslastung nach der ersten Definition, so kann dies zu einer Bevorzugung von bestimmten Applikationen führen. Es existiere z.B. eine Menge A von Anwendungen, die den kompletten Parallelrechner benötigen, sowie lange Rechenzeiten besitzen. Es sei zudem eine Menge B von Anwendungen gegeben, die nur einen Teil des Rechners auslasten und bei deren Abarbeitung Prozessorfragmente

übrig bleiben, die von keiner weiteren Anwendung genutzt werden können. Geht man vom Szenario aus, daß kontinuierlich neue Jobs der Mengen A und B erzeugt werden, so kann dies dazu führen, daß Anwendungen aus B nie bearbeitet werden, da der Scheduler versucht die Fragmentierung zu verringern und nur Jobs aus der Menge A wählt. Bei Benutzung der zweiten Definition für die Auslastung, werden Applikationen bevorzugt, die ihre zugeteilte Prozessorzeit effizient nutzen. Da die Effizienz einer Applikation sich meist mit steigernder Prozessorzahl verschlechtert, erhalten hier Applikationen den Vorrang, die wenig Prozessoren benötigen und gleichzeitig eine geringe Fragmentierung erzeugen. Applikationen mit hohem Prozessorbedarf und mäßiger Effizienz geraten dabei ins Hintertreffen. Dies wird besonders klar, wenn man sich das optimale Schedule mit maximierter Auslastung betrachtet. Dieses besteht aus Applikationen, die jeweils nur einen einzelnen Prozessor benutzen. In diesem Fall ergibt sich die maximale Effizienz pro Anwendung. Fragmentierungsprobleme tauchen nicht auf, da jeder Prozessor permanent genutzt wird. Dieses Szenario hat jedoch mit dem Scheduling paralleler Applikationen nicht mehr viel gemeinsam. Zur genauen Evaluation der Auslastung muß auf Prozessorebene die reale Rechenzeitzuteilung mitprotokolliert werden. Die genaue Ermittlung der Auslastung ist damit sinnvoll nur nach Ausführung des Schedules möglich. Sie stellt damit keine probate Metrik dar, um die Qualität verschiedener Schedules a priori zu beurteilen.

Maximierter Durchsatz

Beim Durchsatz wird die Zahl der Anwendungen gemessen, die pro Zeitintervall abgearbeitet werden können. Aus technischen Gründen wird der Durchsatz meist indirekt bestimmt, indem man die Zeit für die komplette Abarbeitung eines Schedules von n Jobs mißt. Hierzu werden die Terminierungszeiten t_i aller Jobs in einem Schedule herangezogen. Das Maximum der Terminierungszeiten stellt gerade die Zeit dar, die vom Start des Schedules bis zur Terminierung des letzten Jobs vergeht. Dieses Maß wird allgemein mit *makespan* bezeichnet:

$$\textit{makespan: } max\{t_i, 1 \leq i \leq n\}$$

Eine Minimierung von *makespan* führt somit zu einer Maximierung des Durchsatzes.

Verbesserung der mittleren Terminierungszeit

Für den Anwender des Parallelrechners sind andere Kriterien von Interesse. Ein Anwender ist zumeist daran interessiert, möglichst schnell die Ergebnisse seiner Anwendung zu erhalten. Es ist es für ihn prinzipiell zweitrangig, wie sich die Terminierungszeit seiner Applikation aus Wartezeit bis zum Start der Applikation und reiner Rechenzeit der Anwendung zusammensetzt. Eine Ausnahme bilden hierbei Applikationen, die primär der Erforschung von parallelen Algorithmen dienen. Bei diesen Anwendungen liegt der Schwerpunkt auf der reinen Rechenzeit, somit ist eine kurze Terminierungszeit nicht zwingend notwendig. Aus diesen Gründen wird oft die

$$\textit{mittlere Terminierungszeit: } \frac{1}{m}\sum_{i=1}^{M} w_i t_i$$

aller Jobs, bzw. deren nicht normierte Variante die

$$\textit{completion time: } \sum_{i=1}^{M} t_i$$

als Effizienzmaß benutzt. Hier wird davon ausgegangen, daß alle Jobs gleichzeitig vorliegen.Trifft dies nicht zu, so liefert die completion time ein verzerrtes Bild. In solchen Fällen werden Jobs mit einem Wert für $s_i > 0$ mit zu hohen Kosten belegt. Abhilfe schafft in diesen Fällen das um die Komponente s_i erweiterte Kostenmaß, die

$$\textit{Antwortzeit: } \sum_{i=1}^{M} (t_i - s_i).$$

Ein Problem kann auftreten, wenn die Rechenzeiten der einzelnen Jobs stark variieren. Haben zwei Applikationen denselben Prozessorbedarf, aber stark unterschiedliche Rechenzeiten, so ist es nach dem Effizienzmaß gleich, welche dieser Anwendungen zuerst abgearbeitet wird. Auf die subjektive Wartezeit der Anwender hat die Reihenfolge der Abarbeitung jedoch einen deutlichen Einfluß. Einen möglichen Ausweg stellt die Gewichtung von Jobs dar, mit deren Hilfe man, wie schon angesprochen, Anwendungen priorisieren bzw. die Anwendungen bzgl. der Rechenzeit normieren kann. Man erhält so die Maße

$$\textit{weighted completion time: } \sum_{i=1}^{M} w_i t_i$$

bzw.

$$\textit{gewichtete Antwortzeit: } \sum_{i=1}^{M} w_i (t_i - s_i).$$

2.2.2 Klassifikation von Jobs

Für die Auswahl geeigneter Schedulingverfahren ist es notwendig, einen genaueren Blick auf die zu bearbeitenden Jobs zu werfen. Prinzipiell können die Jobs in die folgenden Klassen [FR96] eingeteilt werden.

Entscheidungsträger	statische Zuweisung	dynamische Zuweisung
Benutzer	fest	variabel
System	skalierbar	reskalierbar

Tabelle 2.2: Klassifikation von Jobs basierend auf dem Prozessorbedarf

Jobs mit festem Prozessorbedarf

Jobs, die zu dieser Gruppe gehören, benötigen eine vom Anwender festgelegte Anzahl von Prozessoren. Sie sind so aufgebaut, daß sie einerseits nicht mit einer geringeren Anzahl von Prozessoren lauffähig sind, andererseits zusätzliche Prozessoren nicht nutzen können. Aus Sicht des Schedulers stellen Jobs mit festem Prozessorbedarf eine Black Box dar, die jede Information über den Aufbau des Programms verdeckt. Die Programmstruktur ist dabei meist unflexibel und für eine bestimmte Kombination von Problemgröße und Prozessoranzahl optimiert. Viele Schedulingverfahren basieren auf dem Modell von Jobs mit festem Prozessorbedarf. Bei ihnen gehören per Definition alle Jobs zu dieser Anwendungsklasse, selbst wenn z.B. die eigentliche Anwendung vom Programmaufbau her skalierbar ist.

Jobs mit variablem Prozessorbedarf

Bei einem Job mit variablem Prozessorbedarf ändert sich die Zahl der benötigten Prozessoren während der Abarbeitung. Die aktuell benötigte Prozessorzahl wird vom Job selbstständig ermittelt und dem System mitgeteilt. Hierbei gliedert sich die Berechnung in verschiedene Berechnungsphasen, die einen unterschiedlichen Grad an Parallelität besitzen können. Im Gegensatz zu Jobs mit festem Prozessorbedarf besteht bei Jobs mit variablem Prozessorbedarf die Möglichkeit, zu bestimmten Zeitpunkten der Berechnung Prozessoren an das System zurückzugeben, um diese für die Abarbeitung anderer Jobs zu nutzen, bzw. zusätzliche Prozessoren anzufordern, die ansonsten ungenutzt bleiben würden

Jobs mit skalierbarem Prozessorbedarf

Es gibt parallele Anwendungen, die flexibel bezüglich der Zahl von zugeteilten Prozessoren sind und es dem System erlauben, die genaue Anzahl an Prozessoren festzulegen. Wir unterscheiden hierbei Jobs, die nur zu Beginn ihrer Programmausführung skaliert werden können, von dynamisch skalierbaren Anwendungen, bei denen die Prozessorzahl auch während der laufenden Berechnung geändert werden kann. Zur Abkürzung bezeichnen wir die erste Gruppe als *skalierbare Jobs*, die Jobs mit dynamisch skalierbarem Prozessorbedarf als *reskalierbare Jobs*. Skalierbare Jobs bieten dem Scheduler die Möglichkeit, die Prozessoren so auf die Anwendungen zu verteilen, daß die Gesamteffizienz des Schedules verbessert wird. Er kann hierzu Informationen über die Zusammensetzung des Schedules nutzen, die dem einzelnen Nutzer nicht zur Verfügung stehen. Dieser Ansatz wird als *adaptive Partitionierung* bezeichnet [RSD+94].

reskalierbare Jobs

Reskalierbare Jobs stellen für den Scheduler die ideale parallele Applikation dar. Im optimalen Fall kann die Prozessorzahl einer solchen Anwendung zu einem beliebigen Zeitpunkt geändert werden. Dies erfordert einen parallelen Aufbau, der aus einer Vielzahl unabhängiger Tasks besteht. Bei einer Reskalierung kann es nun passieren, daß die Berechnung einzelner Tasks abgebrochen werden muß, wenn sich die Zahl der Prozessoren verringert. Die so gestopptem Teilberechnungen müssen dann vom Programm selbsttätig auf die verbleibenden Prozessoren umverteilt bzw. ihre Berechnung auf einen späteren Zeitpunkt im Programmablauf verschoben werden. Hierbei können kritische Zustände auftreten, wenn ein Task abgebrochen wird, der ein Betriebsmittel hält. Unterstützt das zugrunde liegende Betriebssystem keinen Entzug von Betriebsmitteln, so bleibt die Resource für andere Applikationen gesperrt. Diese Problematik kann umgangen werden, wenn eine Applikation eine Reskalierung nur zu bestimmten Zeitpunkten der Berechnung zuläßt.

2.2.3 Methoden des Schedulings

Zur Realisierung von Schedules stehen einem Schedulingverfahren potentiell verschiedene Methoden zur Verfügung. Welche davon zum Einsatz kommen, hängt neben den Schedulingzielen auch von der Struktur der Anwendungen und dem benutzten Parallelrechner ab. Dies zeigt sich in den folgenden Abschnitten, die auf die verschiedenen Methoden eingehen.

Partitionierung

Das naheliegendste Verfahren, mehrere Applikationen gleichzeitig abzuarbeiten, besteht darin, das Prozessornetz des Parallelrechners zu partitionieren. Die dabei entstehenden Teilnetze, die selbst wieder Parallelrechner verkörpern, werden dann den einzelnen Applikationen zugeordnet. Es handelt sich hierbei normalerweise um eine exklusive Zuteilung der Prozessorresourcen. Ein Prozessorteilnetz wird in diesem Fall genau einer Applikation zugeordnet. Die Anzahl der zugeteilten Prozessoren wird dabei entweder von der Applikation bestimmt, wenn es sich um Anwendungen mit fixem Prozessorbedarf handelt, oder vom Scheduler, falls die Applikationen dies zulassen. Die Partitionierung bietet den Vorteil, daß auf einem Prozessorelement normalerweise nur der Code und die Daten einer einzelnen Applikation im Speicher gehalten werden müssen. Dies vermeidet weitgehend die Problematik eines zu kleinen Hauptspeichers bei Systemen mit verteiltem Speicher. Ein entscheidender Nachteil der Partitionierung liegt darin, daß bei der Zuteilung von Prozessorpartitionen meist Fragmente übrig bleiben, die von keiner weiteren Anwendung genutzt werden können.

Time-Slicing

Beim Time-Slicing wird jeder Applikation der komplette Parallelrechner exklusiv für einen festgelegten Zeitabschnitt zur Verfügung gestellt. Reicht das Zeitintervall nicht für die vollständige Berechnung aus, so wird die Anwendung unterbrochen und die Prozessorresourcen an die nächste Applikation übergeben. Aufgrund der Verwandtschaft zum Multiprozeß-Scheduling sequentieller Rechner lassen sich die dort erprobten Verfahren leicht auf das Time-Slicing übertragen. Ein weiterer Vorteil besteht darin, daß für jede Applikation der gesamte Parallelrechner zur Verfügung steht. Damit wird das Mapping nicht durch zusätzliche Nebenbedingungen eingeschränkt. Vielmehr kann eine Applikation optimal auf dem Parallelrechner angeordnet werden und im Optimalfall den kompletten Parallelrechner ausnutzen. Neben diesen positiven Eigenschaften müssen jedoch auch ein paar Nachteile in Kauf genommen werden. So sollten im Idealfall alle Applikationen gleichzeitig im Hauptspeicher geladen sein, um die Prozesswechselkosten möglichst klein zu halten. Dies führt oft zu Speicherplatzproblemen, da Prozessorknoten von Parallelrechnern meist mit erheblich weniger Speicherplatz ausgestattet sind, als vergleichbare Einprozessorsysteme. Diese Problematik läßt sich zwar durch Swapping lösen, jedoch verfügen wenige Parallelrechner über die dazu sinnvollen lokalen Festplatten. Beim Time-Slicing bleiben oft einzelne Prozessoren bzw., Prozessorgruppen zu bestimmten Zeitintervallen unbenutzt,

da oftmals mehr Prozessoren zur Verfügung stehen, als die gerade berechnete Anwendung nutzen kann. Dies führt zu einer Auslastungsverschlechterung des Parallelrechners. Ein weiteres Problem tritt auf, wenn nach Ablauf eines Zeitintervalls nicht alle Kommunikationsvorgänge zwischen den Subprozessen einer Anwendung abgeschlossen sind. In diesem Fall können einzelne Nachrichten der Applikation noch im Kommunikationsnetzwerk des Parallelrechners vorhanden sein und so den korrekten Ablauf der neuen Applikation stören. Forschungen haben ergeben [PS95], daß Time-Slicing trotz seiner Kosten, die beim Prozeßwechsel auftreten, ein gutes Verfahren ist, um die mittleren Antwortzeiten der Anwendungen zu minimieren. Die Qualität der Verbesserung hängt hierbei direkt von der Varianz der durchschnittlichen Rechenzeiten der Anwendungen ab. Je höher die Streuung bei den Rechenzeiten der Anwendungen ausfällt, desto bessere Ergebnisse liefert Time-Slicing im Vergleich zu anderen Verfahren. Eine Erweiterung des Time-Slicings besteht darin, die Prozeßwechsel auf Prozessoren, auf denen Teile derselben parallelen Applikation laufen, zu synchronisieren. Dieses Verfahren wird i.a. mit dem Begriff *Gang Scheduling* bezeichnet. Die synchrone Abarbeitung aller Subprozesse einer Anwendung ermöglicht eine feingranulare Interaktion und synchronen Datenaustausch zwischen den einzelnen Subprozessen. Es wird so vermieden, daß ein Subprozeß eines Prozessors A auf eine Nachricht eines Subprozesses auf Prozessor B wartet, dieser Prozeß aber nicht mehr antworten kann, da seine Zeitscheibe auf Prozessor B schon abgelaufen ist.

Remapping

Das Remapping ermöglicht dem Schedule, einen laufenden Job ganz oder teilweise auf andere Prozessoren zu verlagern. Es stellt somit ein Verfahren dar, das benutzt werden kann, um die Zuordnung von Applikation zu Prozessoren zu ändern. Es ist damit möglich, Anwendungen Prozessoren zu entziehen, die diese für ihre weitere Berechnung nicht mehr benötigen, bzw. nur schwach auslasten. Der intensive Einsatz von Remapping im Scheduling wirft für Systeme mit verteiltem Speicher jedoch einige Probleme auf. Dies liegt vor allem daran, daß die Durchführung des Remappings meist die Kopie einer großen Datenmenge vom lokalen Speicher eines Prozessors in den lokalen Speicher eines anderen Prozessors erfordert. Ein weiteres Problem ergibt sich, wenn innerhalb einer Anwendung eine intensive Kommunikation erfolgt. Hier kann, wie beim nicht synchronen Time-Slicing, der Fall eintreten, daß durch die Migration ein Kommunikationsvorgang abgebrochen wird, weil Sender oder Empfänger ihre Position im Prozessornetz geändert haben. Stellt das benutzte Betriebssystem bzw. die benutzten Kommunikationsverfahren nur eine prozessorgebundene Kommunikation zur Verfügung, so wird

die Migrationsfähigkeit von Applikationen stark eingeschränkt.

Kapitel 3

Scheduling paralleler Applikationen auf Parallelrechnern

3.1 Stand der Technik

Betrachtet man den Stand der Technik auf dem Gebiet des parallelen Schedulings, so stellt man fest, daß sich die Aktivitäten auf bestimmte Paradigmen konzentrieren. Ein Scheduling-Paradigma wird dabei durch ein Tupel beschrieben, welches aus der zugrundeliegender Jobklassifikation, dem Ziel des Schedules und der benutzten Schedulingmethode besteht. Die einzelnen Paradigmen sind untereinander schwer vergleichbar, da sie von verschiedenen Voraussetzungen ausgehen. In diesem Kapitel sollen die gängigsten Paradigmen vorgestellt und konkrete Projekte diskutiert werden, die typische Vertreter dieser Paradigmen sind.

3.1.1 Jobs mit festem Prozessorbedarf und variable Partitionierung

In der einfachsten Form des Schedulings fungiert der Scheduler als reiner Prozessor-Distributor. Seine Aufgabe besteht darin, für jeden eingehenden Job, die gewünschte Prozessorzahl exklusiv für einen bestimmten Zeitabschnitt zur Verfügung zu stellen. Er benötigt dabei keinerlei Information über die Struktur oder das Verhalten einer Anwendung. In der Literatur wird dieser Ansatz als *variable Partitionierung* oder *pure space sharing* bezeichnet. Man findet ihn vor allem bei großen Parallelrechnern, die verteilten Speicher benutzen. Die hohe Verbreitung dieses Verfahrens läßt sich auf die

Tatsache zurückführen, daß es sich mit relativ wenig Aufwand und damit in kurzer Zeit implementieren läßt. Dies ist ein Fakt, der bei den heutigen Programmentwicklung im kommerziellen Bereich eine entscheidende Rolle spielt. Dabei werden auch die Nachteile in Kauf genommen, die sich durch ungünstige Fragmentierung ergeben. Diese Problematik kann entschärft werden, wenn eintreffende Jobs in einer Warteschlange zwischengespeichert werden. Für den Scheduler besteht so die Möglichkeit, Applikationen aus der Schlange zu selektieren, die nur geringe Fragmentierung erzeugen. Es muß jedoch beachtet werden, daß die Fairness bei der Abarbeitung nicht verletzt wird und ein Job unendlich lange in der Warteschlange verbleibt.

3.1.2 Jobs mit festem Prozessorbedarf und Gang-Scheduling

Wie schon erwähnt bietet das Gang-Scheduling einige Vorteile gegenüber dem nicht synchronisierten Time-Slicing. Sein Einsatz erfordert keinerlei Information über den Berechnungsablauf oder die Berechnungszeit der Anwendungen. Dem Anwender bleibt zudem die freie Wahl des benutzten Programmiermodells. Dieses Verfahren eignet sich daher sehr gut für Jobs mit festem Prozessorbedarf. Aus diesem Grund stellt das Gang-Scheduling ein recht weit verbreitetes Verfahren dar. Oft finden sich Kombinationen von Time-Slicing und variabler Partitionierung, die auch als Gang-Scheduling bezeichnet werden [FJ96].

3.1.3 Skalierbare Jobs und adaptive Partitionierung

Wie schon erwähnt besitzt die variable Partitionierung einige Nachteile, die sich auf die Effizienz des Schedules auswirken. Diese entstehen, wie schon angesprochen, durch eine ungünstige Fragmentierung des Parallelrechners. Zudem ist bei der variablen Partitionierung nicht sichergestellt, daß ein Job, die ihm zugeteilten Prozessoren auch effizient nutzt. Der Scheduler kann dies nicht beurteilen, da er keine Informationen über die Struktur der Anwendung besitzt. Die adaptive Partitionierung versucht diese Probleme zu umgehen, indem sie auf mehr Flexibilität und Kooperation der Applikationen baut. Hierbei wird zugrunde gelegt, daß ein Job einerseits auf einer beliebigen Anzahl von Prozessoren lauffähig und damit skalierbar ist, andererseits dem System Informationen über seine prozessorabhängige Laufzeit liefert. Mit diesen Informationen ist es dem Scheduler möglich, je nach Zusammensetzung der vorliegenden Applikationen, deren Prozessorbedarf adaptiv anzupassen. Bei geringer Systemauslastung erhalten die einzelnen Applikationen eine hohe Prozessorzahl, wenn sich dadurch ihre Rechenzeit verringert.

Liegt dagegen eine hohe Systemauslastung vor, erhalten neue Jobs nur eine geringe Anzahl von Prozessoren, um so die Gesamteffizienz zu steigern [RSD+94, Sev94].

3.1.4 Reskalierbare Jobs und Repartitionierung

Den für das Scheduling flexibelsten Ansatz stellt die Kombination von reskalierbaren Jobs und Repartitionierung dar. Dieses Scheduling Modell erfordert von den Anwendungen, genaue Informationen über ihren Zeit- und Prozessorbedarf, sowie die Bereitschaft bei Änderungen der Systemlast, ihren Ressourcenbedarf anzupassen. Bei diesem Ansatz steht die Effizienz aus Betreibersicht deutlich im Vordergrund. Dies geht auf Kosten des Programmentwicklers, der in der Wahl des parallelen Programmiermodells durch die an eine Applikation gestellten Anforderungen stark eingeschränkt ist. Sieht man von diesen Tatsachen ab, so ermöglicht dieser Ansatz die bestmöglichste Auslastung des Parallelrechners, da der Scheduler jederzeit auf Änderungen im Schedule, wie dem Auftauchen neuer Jobs, reagieren kann. Der Scheduler versucht, zu jedem Zeitpunkt, basierend auf den ihm vorliegenden Informationen, eine optimale Verteilung der Prozessoren zu erreichen. Terminiert eine Applikation und liegt aktuell kein neuer Job vor, so werden die freigewordenen Prozessoren auf die verbleibenden Applikationen verteilt, um deren Abarbeitung zu beschleunigen. Trifft ein neuer Job ein, werden Prozessoren von den anderen Jobs abgezogen, sodaß die Gesamteffizienz des neuen Schedules optimiert wird. Die Problematik besteht, daß dieses Scheduling-Paradigma nur sinnvoll implementiert werden kann, wenn es vom zugrundeliegenden Betriebssystem und der benutzten Programmentwicklungsumgebung massiv unterstützt wird. Daher ist es sinnvoll, diese beiden Komponenten zusammen zu entwickeln, was eine eine sehr komplexe Aufgabe darstellt. Es sind zum aktuellen Zeitpunkt keine real existierenden Parallelrechner bekannt, die dieses Scheduling-Paradigma benutzen.

3.1.5 Theoretische Modelle

Shelf Scheduling

Beim Shelf Scheduling [UST94] handelt es sich um ein nicht preemptives Schedulingverfahren. Einzelne Applikationen können somit nicht mehr unterbrochen werden, nachdem sie einmal gestartet wurden. Das zugrundeliegende Jobmodell geht von einer Anwendungen mit fester Prozessorzahl aus, deren Zeitbedarf vorab bekannt ist. Zu jedem Job J_i ergibt sich ein Tupel (h_i, P_i), das die Ressourceanforderungen beschreibt. Dabei steht P_i für

die Zahl der benötigten Prozessoren und h_i für die gewünschte Rechenzeit. Schaut man sich die Visualisierung eines Schedules an, so sieht es aus, als ob man die einzelnen Anwendungen in ein Regal eingeordnet hätte. Hierher rührt auch der Name des Verfahrens Die Höhe der Regalbretter vom Boden stellt dabei die Startzeit einer Teilmenge der Applikationen, der Platzbedarf einer Applikation die Menge der zugeteilten Prozessoren dar.
Der SMART-Algorithmus [UST94]schildert die prinzipielle Arbeitsweise beim Shelf Scheduling.

1. Bestimme das kleinste k, für das gilt $2^k > max(h_i)$
2. Partitioniere die Anwendungen so in die k Klassen, daß gilt: $\forall i \in M$ in Partition j mit $2^{(j-1)} < h_i \leq 2^j$
3. Sortiere die Jobs primär nach Klasse und innerhalb der Klasse aufsteigend nach Prozessorgröße in die Shelfs
4. Sind für einen Job nicht mehr genügend freie Prozessoren vorhanden, so eröffne mit diesem Job den nächsten Shelf
5. Das Maximum der Berechnungszeiten aller Jobs innerhalb eines Shelfs bestimmt die Höhe eines Shelfs

Ein typisches Resultat des Algorithmus findet sich in Abbildung 3.1 Die Zielfunktion des Shelf Schedulings besteht darin, die mittlere Antwortzeit zu minimieren. Die Vorteile des Verfahrens liegen darin, daß eine gestartete Applikation nicht mehr unterbrochen werden muß. Es ist daher besonders für Parallelrechner geeignet, die aus Speicherplatzmangel oder anderen Gründen den Einsatz preemptiver Verfahren nicht erlauben. Shelf Scheduling benötigt zudem nur minimalen Berechnungsaufwand. Schwiegelshohn u.a. haben nachweisen können, das die Ausführungszeit eines so berechneten Shelf Schedules im ungünstigen Fall maximal um den Faktor 8 von der optimalen Lösung abweichen kann. Diese Tatsache beeinträchtigt ein wenig den praktischen Einsatz des Shelf Schedulings.

PBI-Scheduling

Ein interessantes preemptives Schedulingverfahren stellt der PBI Algorithmus [STW95] (Preemptive by Interleaving) von Schwiegelshohn dar. Hierbei dürfen einzelne Applikationen im Gegensatz zum Shelf Scheduling beliebig oft unterbrochen werden. Dadurch wird eine Verbesserung der Effizienz erreicht. Ein großer Vorteil des Verfahrens besteht darin, daß nur gleichzeitig maximal

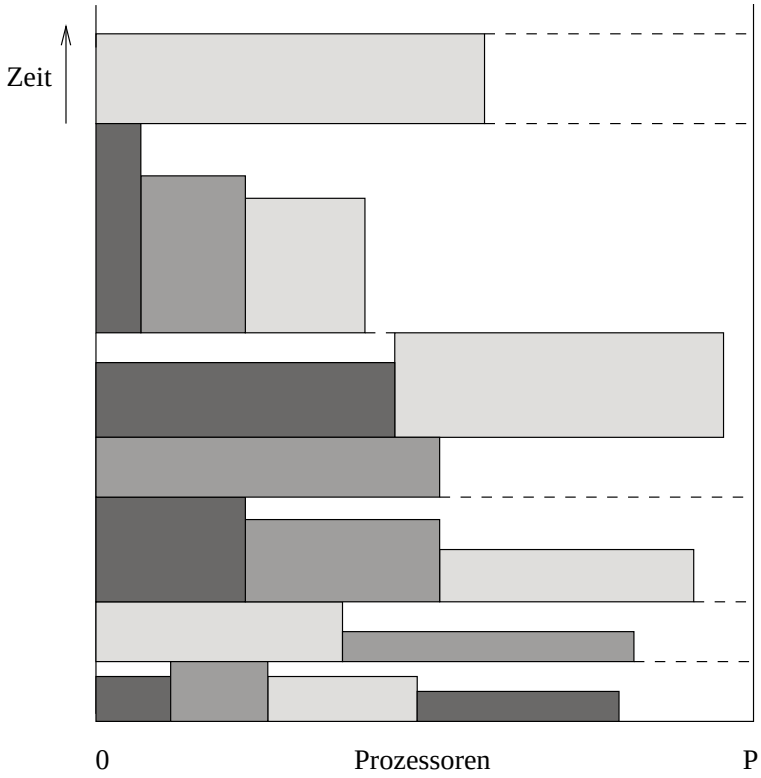


Abbildung 3.1: typisches Shelf Schedule

zwei Anwendungen auf derselben Mengen von Prozessoren abgearbeitet werden. Somit ist dieser Schedulingalgorithmus wie das Shelf Scheduling auch für parallele Plattformen geeignet, die über kein prozessororientiertes Virtual Memory verfügen. Das Verfahren von Schwiegelshohn benutzt als Effizienzkriterium die gewichteten Antwortzeiten aller Applikationen eines Schedules. Der PBI Algorithmus generiert zuerst zwei parallele Schedules S_1 und S_2, die jeweils die komplette Prozessorzahl zur Verfügung haben. In einem zweiten Schritt werden die beiden Schedules dann zum entgültigen Schedule zusammengefaßt . Der Algorithmus läuft dabei folgendermaßen ab:

1. Ordne alle Jobs in einer Prioritätsliste, sodaß Job i vor Job j steht, falls $\frac{1}{w_i h_i} < \frac{1}{w_j h_j}$
2. Bestimme die Zahl der aktuell verfügbaren Prozessoren P_{S_1} in Schedule S_1. Liegt zum selben Zeitpunkt in S_2 ein Job j vor mit $w_j < P_{S_1}$, so verschiebe die restliche Berechnung von j ins Schedule S_1
3. Nimm den nächsten Job i aus der Prioritätenliste. Falls $P_{S_1} > w_i$, füge

Job i in S_1 ein, ansonsten füge i in S2 ein.

4. Falls die Prioritätenliste nicht leer ist, fahre mit Punkt 2 fort.
5. Teile S_1 und S_2 in horizontale Streifen. Ein Streifenwechsel tritt immer dann auf, wenn die Berechnung eines Jobs von S_2 nach S_1 wechselt oder in S_2 abgeschlossen wurde.
6. geniere ein kombiniertes Schedule, indem abwechselnd Streifen aus S_1 und S_2 ausgeführt werden und das mit dem ersten Streifen aus S_1 startet

Ein typisches Beispiel für ein PBI-Schedule liefert Abbildung 3.2. Im Gegen-

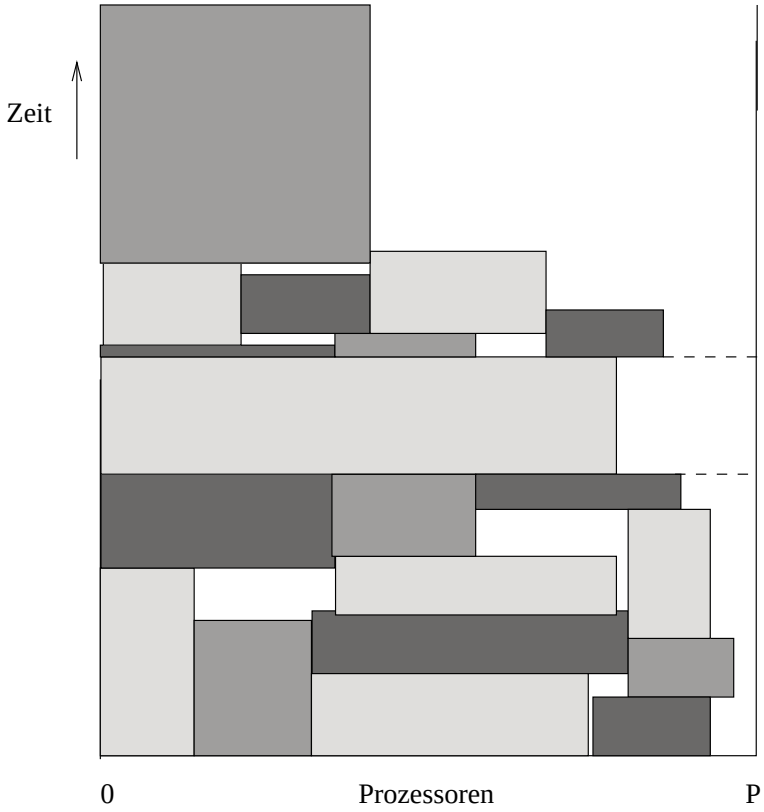


Abbildung 3.2: Preemptives, von PBI generiertes Schedule

satz zum Shelf-Scheduling bietet der PBI-Algorithums ein deutlich verbessertes Verhalten im Worst-case. In [STW95] konnte nachgewiesen werden, daß der Algorithmus weniger als den Faktor 2.5 vom optimalen Wert abweicht. Er bietet somit deutlich bessere Voraussetzungen für den praktischen Einsatz, falls der benutzte Parallelrechner das preemptive Abarbeiten mehrerer Jobs unterstützt.

Dynamic Equipartitioning

Dieser von Deng u.a. [DGBL96] vorgestellte Ansatz geht von der realistischen Prämisse aus, daß die wirkliche Berechnungszeit einer parallelen Anwendung erst nach Ende der Berechnung festgestellt werden kann. Sie liegt damit dem Scheduler nicht als Information zum optimierten Aufbau des Schedules vor. Das sich trotz dieser Einschränkung effiziente Schedules berechnen lassen, wird am Beispiel Dynamic Equipartitioning (DEQ) gezeigt. Dieses Verfahren verteilt die Prozessoren des Parallelrechners gleichmäßig auf alle Applikationen. DEQ läuft dynamisch beim Eintreffen neuer Applikationen ab und funktioniert nur mit Applikationen, die dynamisch reskalierbar sind. Es handelt sich damit um einen preemptiven Ansatz zum Scheduling. Das globale Ziel des DEQ liegt in der Minimierung der mittleren Antwortzeit aller Anwendungen und ist von daher gut mit anderen Verfahren vergleichbar. Der Algorithmus benötigt zur Brechnung eines Schedules die Gesamtzahl der verfügbaren Prozessoren P, die Menge der aktuell abzuarbeitenden Jobs J_1 bis J_n, sowie die von ihnen maximal einsetzbare Anzahl an Prozessoren P_1 bis P_n. Das Equipartitioning benutzt das folgende Zuteilungsverfahren:

1. Falls für alle $i : 1 \leq i \leq n$ gilt $P_i > \frac{P}{n}$, dann gib jedem Job $J_i : 1 \leq i \leq nfracPn$ Prozessoren
2. Ansonsten gib jedem Job J_i mit $P_i < \frac{P}{n}$ P_i Prozessoren.
3. Ermittle die Zahl n der Jobs ohne Prozessorzuweisung und die Zahl P der verbliebenen Prozessoren
4. Falls $n > 0$ starte rekursiv DEQ.

Um die Güte des Verfahrens nachzuweisen, haben Deng u.a. DEQ mit dem optimalen Schedule verglichen, bei dem sowohl der Prozessorbedarf, als auch die Rechenzeit für jeden Jobs bekannt ist. Beim Vergleich wird davon ausgegangen, daß für jede Applikation J_i deren maximal nutzbare Zahl an Prozessoren P_i und die sich daraus resultierende Rechenzeit h_i vorliegt. Erhält eine Applikation vom Scheduler $p_i < P_i$ Prozessoren, so wird angenommen, daß sich die Rechenzeit proportional zu $\frac{P_i}{p_i} h_i$ verlängert. Unter obigen Voraussetzungen konnte im Vergleich nachgewiesen werden, daß für eine Menge von n Anwendungen DEQ maximal um den Faktor $2 - \frac{2}{n-1}$ von der optimalen Lösung abweicht. Dieser Faktor stellt dabei mathematisch gesehen das Optimum dar und kann nur verbessert werden, wenn das Schedulingverfahren auch über Rechenzeitinformationen verfügt. Der Einsatz von DEQ verspricht im praktischen Einsatz gute Schedulingergebnisse. Die Vorteile liegen in seiner simplen Berechnung, die das Verfahren uneingeschränkt zum Online

Einsatz befähigt, und in der Tatsache, daß keinerlei Laufzeitinformationen notwendig sind. Die Probleme bei DEQ liegen auf Seiten des Anwenders und des Betriebssystems. Der Anwender wird gezwungen, reskalierbare Applikationen zu entwerfen, was eine große Einschränkung bei der Programmentwicklung bedeutet. Das Betriebssystem muß über die Möglichkeit der Preemption verfügen, um die korrekte Reskalierung zu realisieren. Auf grund dieser Tatsachen hat DEQ noch keine Verbreitung in der Praxis gefunden.

3.1.6 Scheduling in der Praxis

Schedulingverfahren im praktischen Einsatz zeichnen sich meist dadurch aus, daß Prozessorbedarf und Rechenzeit einer parallelen Applikation vom Anwender fest vorgegeben werden. Die folgenden Verfahren zeigen Methoden, wie trotz dieser Restriktion effizientes Scheduling möglich ist.

EASY-Loadleveler

EASY-Loadleveler stellt ein funktionsfähiges Schedulingsystem dar, das eine Kombination des EASY-Jobschedulers für parallele Systeme und dem verteilten, netzwerkbasiertem Jobscheduler Loadleveler ist. Die urspüngliche Entwicklung von EASY basiert auf dem 128-Prozessor SP System des Argonne National Laboratory. Für diesen Parallelrechner wurde ein Schedulingsystem gesucht, daß die folgenden Bedingungen erfüllt:

- Jeder Job wir abhängig von seiner Entstehungszeit in einer Warteschlange abgelegt.
- Die Warteschlange wird normalerweise nach dem FIFO-Prinzip abgearbeitet.
- Ein Job wird erst dann abgearbeitet, wenn die gewünschte Zahl an Prozessoren verfügbar ist.
- Kann ein Job nicht abgearbeitet werden, weil noch nicht genügend Prozessoren frei sind, darf eine nachfolgende Applikation die aktuell freien Prozessoren nutzen.
- Die Berechnung einer wartenden Applikation wird nicht durch eine zeitlich nachfolgende Applikation verzögert.

Die Realisierung dieser Prinzipien führte David Lifka [SCZL96] zur Entwicklung von EASY, welches die Vorgaben mittels der Strategie des Backfillings

umsetzt. Beim Backfilling werden Anwendungen mit geringerem Prozessorbedarf aber späterem Zeitstempel zeitlich im Schedule vorgezogen, um freie Prozessoren zu nutzen. Ergibt sich ein möglicher Start der in der Schlange wartenden Applikation, so werden die vorgezogenen Anwendungen abgebrochen, falls sie den Start verzögern würden. Die Entwicklung von Loadleveler basiert auf dem von IBM entwickeltem kommerziellen System Condor, das ursprünglich zum Scheduling von Workstation-Clustern konzipiert wurde. Loadleveler stellt als Erweiterung dieses Ansatzes einen Scheduler für massiv parallele Systeme, insbesondere dem IBM SP dar. Seine Aufgabe ist es, die Rechenlast zu balancieren, vorhandene Resourcen effizient zu nutzen, sowie eine faire Abarbeitungsreihenfolge zu garantieren. Das System besitzt eine weltweite, von IBM unterterstützte Verbreitung. Die Ankopplung von Easy an Loadleveler bietet für Easy den Vorteil, effiziente Systemfunktionen zum Prozeßstart, Prozessorüberwachung und anderer Monitorfunktionen nutzen zu können. Die Kombination beider Verfahren realisiert ein effizientes Schedulingsystem für IBM SP Parallelrechner, das sich durch Robustheit und Fairness in der Abarbeitung auszeichnet. Das eingesetzte Backfilling garantiert eine verbesserte Auslastung des Parallelrechners und zudem eine kurze Antwortzeit für Anwendungen mit kleinem Prozessorbedarf und kurzer Rechenzeit. Das Verfahren stellt somit ein praktikables Verfahren dar, das mit minimalem Einsatz eine gute Effizienzverbesserung erreicht. Der Ansatz stößt jedoch an seine Grenzen, wenn die Zusammensetzung der Jobs keine geeigneten Kandidaten fürs Backfilling enthält. In diesem Fall verhält sich der EASY-Scheduler genau wie jedes andere FIFO-Verfahren.

Der PC^2 Scheduler

Das Paderborner Zentrum für paralleles Rechnen kurz PC^2, beschäftigt sich schon seit Jahren mit der Entwicklung und dem Betrieb von Schedulingsystemen für Parallelrechner. Im Laufe der Zeit wurde dabei das komplexe Softwaresystem CCS [GR96] entwickelt, das dem Benutzer eine einheitliche Benutzerschnittstelle zum Zugriff auf verschiedenste Parallelrechner zur Verfügung stellt. Eine Applikation erhält dabei exklusiven Zugriff auf die gewünschte Prozessorpartion für einen festgelegten Zeitraum. Die Abarbeitung paralleler Anwendungen wird in zwei Phasen abgewickelt. Im ersten Schritt werden neue Jobs im sogenannten *Warteraum* abgelegt. Er besteht aus einem hierarchischem System von Warteschlangen und dient als Zwischenpuffer für das eigentliche Scheduling. Im zweiten Schritt vollzieht sich das eigentliche Scheduling. Der Scheduler selektiert nach Priorität eine bestimmte Anzahl von Jobs aus jeder Warteschlange, um aus ihnen ein neues Schedule zu berechnen. Zur Ermittlung des eigentlichen Schedules dienen

verschiedene Algorithmen, zwischen denen adaptiv nach Rechenlast gewechselt wird. Die Auswahl eines Schedulers wird dabei vom implizenten Abstimmungssystem IVS gesteuert. Hiermit ist es möglich, den Scheduler an die einzelnen Lastzustände anzupassen und so die Qualität des Schedules zu verbessern. Als Schedulingziele dienen primär die Auslastung des Parallelrechners und sekundär die mittlere Antwortzeit der Jobs. Ein weiteres Ziel von CCS liegt in der möglichst genauen Prognose der Startzeit einer Anwendung. Dies ist vor allem für die Abarbeitung interaktiver Programme wichtig. Die Problematik liegt hierbei in der Organisation des Warteraums, der durch die Organisation der Warteschlangen, die Verzögerung von Jobs durch höher priorisierte Anwendungen erlaubt. Um das primäre Schedulingziel zu erreichen, wird als Metrik Makespan benutzt. Da ein Job durch eine feste Prozessorzahl P_i und Rechenzeit h_i charakterisiert wird, kann die Minimierung von Makespan auls Bin-Packing Problem modelliert werden. Zu dessen Lösung verfügt der Scheduler über verschiedene Algorithmen. Das einfachste Verfahren, welches im CCS eingesetzt wird, ist das First-Come-First-Serve (FSFS) Scheduling. Hierbei wird den einzelnen Jobs in der Reihenfolge des Eintreffens, die gewünschten Partitionen zugeordnet. Diese Methode wird jedoch nur benutzt, solange der betrachtete Parallelrechner nicht vollständig belegt ist. Trifft dies nicht zu, kommen zwei weitere Schedulingverfahren zum Zug. Hierbei wird das Schedule so aufgebaut, daß zu bestimmten Zeitpunkten im Schedule der Jobstart synchron auf allen belegten Prozessoren erfolgt. Die Zeitpunkte werden als Schedulingpunkte bezeichnet und sind vergleichbar mit den Positionen der einzelnen Shelfs beim Shelf Scheduling. Der First-Fit-Decreasing-Height(FFDH) Algorithmus stellt das zweite Schedulingverfahren dar:

- Sortiere n Jobs absteigend nach Rechenzeit in die Request-Liste RL
- $t_{start} = 0$, $P_{frei} = P$, $t_{neu} = 0$
- Solange $i < n$ entferne J_i aus RL und mache das folgende:
  * Falls $h_i + t_{start} > t_{neu}$, setze $t_{neu} = h_i + t_{start}$
  * Falls zum Zeitpunkt t_{start} $P - i < P_{frei}$ gilt , setze $t_{start} = t_{neu}$ und $P_{frei} = P$ weise J_i P_i Prozessoren zum Zeitpunkt t_{start} zu , $P_{frei} = P_{frei} - P_i$

Da dieser Algorithmus Anwendungen mit langen Rechenzeiten bevorzugt, werden kurze Jobs über Gebühr verzögert. Dies schlägt sich in einem schlechten Ergebnis für das Sekundärziel des Schedules, der mittleren Antwortzeit

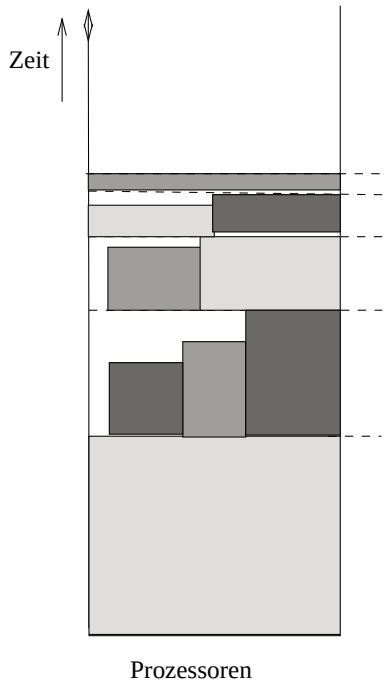


Abbildung 3.3: FFDH Schedule

nieder. Das Verfahren eignet sich daher besser für den Batch-Betrieb, bei dem die mittlerer Antwortzeit keine Rolle mehr spielt. Sortiert man die Jobliste RL aufsteigend nach Rechenzeit und führt dann das eigentliche Scheduling durch, so erhält man das zweite eingesetzte Verfahren, den First-Fit-Increasing-Height(FFIH) Algorithmus. Seine Resultate besitzen große Ähnlichkeit mit den Schedules des Shelf Schedulings. Betrachtet man Durchsatz und mittlere Antwortzeit, so bleibt die Gesamtlänge des Schedules konstant, die mittlere Antwortzeit verbessert sich aber deutlich, da zeitlich kurze Jobs nun bevorzugt abgearbeitet werden. Weitere Verbesserungen der beiden Algorithmen werden erzielt, wenn man die zeitliche Plazierung nicht nur zu Schedulingzeitpunkten erlaubt. Eine Applikation darf dabei zeitlich beliebig hinter einer anderen Applikation plaziert werden, solange ihre Rechenzeit nicht den nächsten Schedulingpunkt überschreitet. Damit ist es möglich, vergleichbar dem Shelf-Scheduling, kurze Jobs dynamisch in ein schon generiertes Schedule zu integrieren. Hiermit wird eine weitere Steigerung des Durchsatzes erreicht. Die so entstandenen Verfahren werden als FFDH*- und FFIH*-Algorithmus bezeichnet. Die Wahl des aktuellen Schedulingverfahrens wird, wie schon erwähnt, vom IVS gesteuert, welches dazu die folgenden Regeln benutzt:

- Falls der Parallelrechner nicht vollständig belegt ist, schalte in den

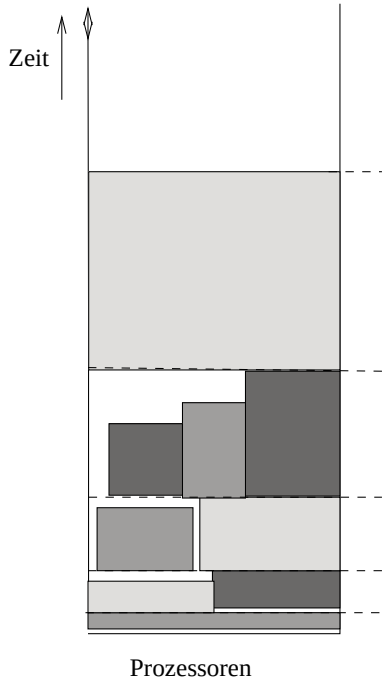


Abbildung 3.4: FFIH Schedule

FCFS Modus und weise die Partionen direkt zu.

- Wähle den FFDH* Modus, um die Auslastung zu maximieren, falls überwiegend Batch-Jobs vorliegen und der Rechner schon vollständig belegt ist
- Ansonsten wähle den FFIH* Modus, um die mittlere Antwortzeit zu minimieren.

Das komplexe Schedulingmodell des CCS ermöglicht die adaptive Anpassung auf das aktuelle Jobprofil. Es garantiert eine gute Auslastung des Parallelrechners und benötigt keine weiteren Informationen über die Struktur der Anwendungen. Der Anwender ist damit nicht auf die Verwendung bestimmter Programmierparadigmen beschränkt. Insgesamt gesehen stellt das CCS ein System dar, das unter den gegebenen Randbedingungen eine effiziente parallele Nutzung realisiert.

Gang-Scheduling am LLNL

Im Computing Center des Lawrence Livermore National Laboratory (LLNL) [FJ96] werden schon seit längerer Zeit Schedulingsysteme benutzt, die auf Gang Scheduling beruhen. Diese Verfahren werden dabei auf einer BBN

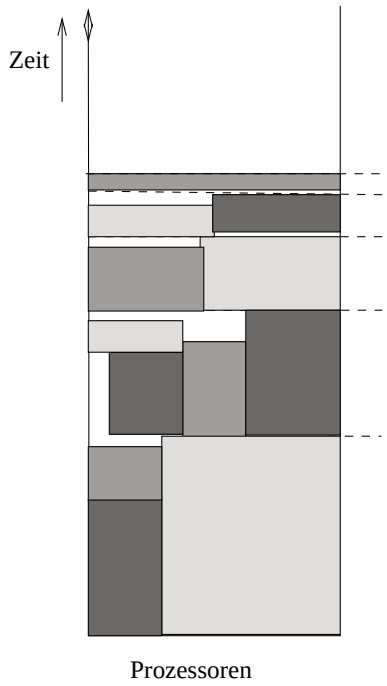


Abbildung 3.5: FFDH* Schedule

TC2000, einer CRAY T3D, sowie einer Digital Alpha 8400 eingesetzt. An dieser Stelle soll exemplarisch auf das Gang Scheduling der CRAY T3D eingegangen werden. Beim hier benutzten Verfahren handelt es sich um kein reines Time-Slicing, sondern um eine Kombination dieses Verfahrens mit variabler Partitionierung. Ziel des Schedulers ist es, die unterschiedlichen Anforderungen verschiedener Benutzergruppen möglichst optimal zu erfüllen. Der Scheduler erhält dazu neben der benötigten Prozessorzahl auch eine Klassifikation der Anwendung. Eine Angabe der Rechenzeit ist, wie schon erwähnt, beim Gang Scheduling nicht notwendig. Beim Scheduling werden die folgenden Klassen unterschieden:

- Interaktive Jobs: Hier wird eine kurze Antwortzeit erwartet. Ein Job darf in seiner Ausführung unterbrochen werden.
- Debug Jobs: Die Jobs dieser Klasse dürfen nicht unterbrochen werden. Der Benutzer erwartet zudem eine kurze Antwortzeit, um das Debugging durchführen zu können.
- Production Jobs: Diese Jobs präsentieren parallele Berechnungen, die für den Batchbetrieb geeignet sind. Bei ihrer Abarbeitung wird auf einen hohen Durchsatz Wert gelegt.

Abbildung 3.6: FFIH* Schedule

- Benchmark Jobs: Eine kurze Antwortzeit ist nicht erforderlich, jedoch darf die Anwendung nicht unterbrochen werden.
- Standby Jobs: Diese Klasse enthält alle Jobs niedriger Priorität, die abgearbeitet werden, wenn Prozessoren ansonsten idle bleiben würden.

Die Eigenschaften einer Jobklasse werden durch ein 4-Tupel modelliert, welches die folgenden Parameter besitzt:

- Priorität: Hierdurch wird die Priorität einer Klasse festgelegt.
- Wartezeit: Die maximale Wartezeit bis zum Start der Anwendung bzw. dem Start der nächsten Zeitscheibe einer Anwendung.
- Zeitmultiplikator: Die minimale Rechenzeit einer Zeitscheibe pro zugeteiltem Prozessor
- Prozessorlimit: Die maximale Zahl der Prozessoren, die einer Applikation dieser Klasse zugeordnet werden kann

Von besonderem Interesse ist der prozessorabhängige Parameter Zeitmultiplikator. Die CRAY T3D erlaubt einen Prozeßwechsel nur in Form eines

Jobdumps auf ein sekundäres Speichermedium. Da dieses Medium nicht lokal verfügbar ist, ergibt sich beim Prozeßwechsel ein Overhead, der sich proportional zur Anzahl der zugeteilten Prozessoren verhält. Somit ist es sinnvoll die Mindestlänge einer Zeitscheibe auch von der Anzahl der Prozessoren abhängig zu machen. Der Zeitmultiplikator erfüllt diese Aufgabe. Basierend auf diesen Informationen läuft das Scheduling prinzipiell wie folgt ab. Im ersten Schritt wird eine Liste von Jobs erzeugt, die ihre klassenbezogene Wartezeit überschritten haben. Die Reihenfolge der Jobs in der Liste richtet sich dabei nach den Parametern. Mittels einer Kostenfunktion, die als Parameter die Priorität, die Anzahl der Prozessoren sowie die verbleibende Mindestrechenzeit eines Jobs benutzt, wird der kostengünstigste Kandidat gesucht, dessen Ressourcen für den neuen Job freigegeben werden. Jobs aus der Benchmark- oder der Debug-Klasse, sowie höher priorisierte Jobs scheiden als Kandidaten prinzipiell aus. Der eigentliche Wechsel erfolgt, sobald die Mindestrechenzeit des kostengünstigsten Jobs abgelaufen ist. Mit Einsatz des Gang-Schedulings konnte die Auslastung der CRAY T3D um ca. 30 - 40 Prozent gegenüber des ursprünglichen Schedulers gesteigert werden. Die durchnittliche wöchentliche Auslastung beträgt dabei über 90 Prozent. Dieser Effekt beruht einerseits auf der Tatsache, daß Gang Scheduling bei einer heterogen zusammengesetzten Jobmenge die besten Ergebnisse im Vergleich zur reinen Partitionierung liefert, andererseits durch Modifikation der Klassenparameter ein Feintuning durchgeführt werden kann. Betrachtet man die Tatsache, daß bestimmte Applikationen nie unterbrochen werden und die einzelnen Zeitscheiben recht lang sind, so wirft sich die Frage auf, ob die Klassifikation des Verfahrens als Gang Scheduling wirklich sinnvoll ist.

Kapitel 4

Algorithmische Skelette zur Modellierung von parallelen Programmen

4.1 Motivation

Die Entwicklung und der Einsatz paralleler Programme wird vom Wunsch getrieben, das zugrundeliegende Berechnungsproblem möglichst schnell zu lösen bzw. bei der Berechnung einen vorgegebenen Zeitrahmen nicht zu überschreiten. Ein Schlüsselproblem stellt dabei die Kommunikation und Synchronisation sowie das Mapping der parallelen Programmkomponenten dar. Die korrekte und effiziente Lösung dieser Aufgaben erfordert ein hohes Maß an Programmieraufwand. Die zugrunde liegende parallele Hardware und das benutzte Betriebssystems üben einen massiven Einfluß auf die Struktur der Lösung aus. Ein so entstandenes Programm ist dadurch meist auf einen Parallelrechnertyp zugeschnitten und kann nicht universell auf verschiedenen Rechnern eingesetzt werden. Der Versuch, ein paralleles Programm möglichst effizient zu gestalten, führt somit zu einer Einschränkung der Universalität bei der Auswahl des Parallelrechners. Der Wunsch flexiblere parallele Programme zu entwickeln, die auf einer Vielzahl paralleler Plattformen lauffähig sind, führte zur Entwicklung vereinheitlichter Kommunikationsbibliotheken wie PVM und MPI. Mit diesen Bibliotheken ist es möglich, parallele Programme zu entwickeln, die ohne große Probleme auf andere Rechnerstrukturen zu portieren sind. Hierzu werden die Kommunikationsroutinen verschiedener Parallelrechner in einheitliche abstrakte Funktionen gekapselt. Die zugrunde liegende Systemroutine wird dadurch transparent für den Benutzer. Dieser sieht nur noch den abstrakten Aufruf, ohne sich um die Art und

Weise der Implementierung kümmern zu müssen. Die Kapselung der Systemroutinen realisiert die gewünschte Portabilität, erfordert aber erhöhte Kommunikationskosten, da die abstrakten Systemaufrufe eine zusätzliche Schicht zwischen Programm und System darstellen. Vereinheitlichte Kommunikationsroutinen stellen zwar ein probates Mittel dar, um die Portabilität von parallelen Programmen zu unterstützen, das Mapping der einzelnen Programmkomponenten muß jedoch weiterhin vom Entwickler auf den entsprechenden Parallelrechner angepaßt und optimiert werden. Ausschlaggebend ist dabei das zugrundeliegende Programmierparadigma. Der strukturelle Aufbau des parallelen Programms bestimmt die notwendige Kommunikation, deren Kosten wiederum von der Art des Mappings und dem Kommunikationsnetzwerk des Parallelrechners abhängig sind. Universelle Portabilität paralleler Programme kann nur erreicht werden, wenn Programmierparadigmen, Kommunikation und Mapping zusammen betrachtet werden. Dieser Ansatz führt direkt zum Konzept der Algorithmischen Skelette.

4.2 Algorithmische Skelette

Algorithmische Skelette stellen einen Ansatz dar, parallele Programmierung auf einer hohen Abstraktionsebene zu betreiben. Die Abstraktion wird dadurch erreicht, daß dem Benutzer ein Programmrahmen zur Verfügung gestellt wird, der den strukturellen Aufbau des parallelen Programms realisiert und über wohldefinierte Schnittstellen benutzerdefinierte Funktionen integriert. Aus funktionaler Sicht können algorithmische Skelette als Metafunktionen oder HOFs (Higher Order Functions) betrachtet werden, die als Parameter Benutzerfunktionen sowie Datenstrukturen für die Ein- und Ausgabe besitzen. Algorithmische Skelette verkörpern Programmierparadigmen, wie z.B. Divide&Conquer, Branch&Bound oder Farming, die entweder schon parallele Lösungsansätze darstellen oder gut parallelisierbar sind. Oft wird dabei das Ziel verfolgt, die parallelen Strukturen des Verfahrens für den Benutzer transparent zu machen. Der einzige Punkt in der Programmentwicklung, an dem der Entwickler normalerweise mit Parallelität in Berührung kommt, ist die Auswahl eines Programmierparadigmas und damit des Skeletts, das für die Problemlösung geeignet ist. Die Problematik des parallelen Programmierens wird dabei, wie schon erwähnt, von einer geringen Abstraktionsebene (Kommunikation, Synchronisation, Mapping) auf eine hohe Abstraktionsebene (Auswahl des Skelettes) verlagert. Algorithmische Skelette können in vielfältiger Form auftreten. Das Spektrum reicht dabei von universellen Skeletten bis zu Programmierrahmen für dedizierte Spezialanwendungen. Der mögliche Einsatzbereich skelettbasierter Programmierung richtet sich dabei

nach Art und Anzahl der zur Verfügung stehenden Skelette.

4.3 Abstraktion durch Programmklassen

Der Einsatz algorithmischer Skelette in der Entwicklung von parallelen Programmen zwingt den Entwickler zu einem strukturierten Programmentwurf. Dabei werden die folgenden Entwurfschritte durchlaufen:

1. Selektion des Skeletts
2. Design der notwendigen Datenstrukturen
3. Entwicklung der anwenderspezifischen Funktionen

4.4 Stand der Technik

Betrachtet man das Spektrum der skelettorientierten Programmierung, so lassen sich die folgenden Forschungsschwerpunkte erkennen:

- Allgemeine Programmierskelette
  Versucht man mit einer geringen Zahl von Skeletten ein möglichst großes Anwendungungspektrum abzudecken, so muß der Aufbau der einzelnen Skelette sehr allgemein gehalten werden. Parallelität wird hierbei auf einer hohen Abstraktionsebene betrachtet, um das Skelett für eine Vielzahl von Problemstellungen nutzen zu können. Klassischer Vertreter dieser Gattung sind z.B. die von Cole vorgeschlagenen Skelette für Divide&Conquer und Iterative Combination oder das Farming-Skelett. Sie bieten auf der einen Seite Universalität beim Einsatz, können hierdurch jedoch nicht die Performanz einer parallelen Speziallösung bieten.

- Spezialskelette
  Hierbei handelt es sich z.B. um Skelette, die für ein eng umrissenes Einsatzgebiet konzipiert und optimiert werden. Typische Einsatzgebiete dieser Skelette sind die parallele Bildverarbeitung, bzw. allgemeine Matrizenoperationen, bei denen Datenparallelität genutzt werden kann. Weiterhin können zu dieser Gruppe auch Skelette gezählt werden, die auf eine spezielle Parallelrechnerstruktur zugeschnitten sind.

- Kompositionelle Ansätze
  Ein anderer Weg zur parallelen Datenverarbeitung wird von den kompositionellen Ansätzen verfolgt. Im Gegensatz zu den vorher erwähnten Skeletten bieten kompositionelle Modelle keine festgelegten Programmierrahmen, sondern verschiedene Konstruktoren, um den Ablauf paralleler Programme zu strukturieren und zu synchronisieren. Vertreter dieses Paradigmas sind z.B. die parallele Kompositionssprache PCN und P^3L.

In den folgenden Abschnitten werden typische Vertreter der verschiedenen Skeletttypen vorgestellt und ihre Eigenschaften diskutiert.

4.4.1 Allgemeine Programmierskelette

Cole: Algorithmic Skeletons

Das von Cole [Col89] vorgestellte Konzept der *Algorithmic Skeletons* stellt einen allgemeinen Ansatz in der skelettbasierten Programmierung dar. Die vom Autor diskutierten Skelette z. B. für Divide&Conquer oder iterative Lösungsverfahren stellen allgemeine Programmierparadigmen dar, die ein hohes Maß an inhärenter Parallelität besitzen. Der Aufbau der Programmierrahmem ist so gestaltet, daß parallele Aktionen möglichst transparent für den Benutzer sind. Cole versucht durch diese Transparenz unnötige Details vom Programmierer fernzuhalten, damit sich dieser besser auf die anwendungsspezifischen Probleme fokussieren kann. Neben dieser Eigenschaft spielt die Effizienz eine zentrale Bedeutung in Coles Ansatz. Die von ihm für die Skelette gewählten Programmierparadigmen erfordern zur effizienten Realisierung nur einfache Ringstrukturen bzw. Prozessorgitter. Dies ermöglicht eine gute Portierbarkeit auf verschiedene Parallelrechnerarchitekturen.

Higher Order Functions (HOF)

Der von Tore Bratvold [Bra94a], [Bra94b] verfolgte Ansatz zielt auf eine effiziente automatische Parallelisierung funktionaler Programme hin. Dabei sollen in funktionalen Programmen, basierend auf einer Teilmenge der Programmiersprache ML, HOFs identifiziert werden und deren implizite Parallelität durch eine geeignete parallele Implementation realisiert werden. Zu diesem Zweck steht eine wohldefinierte Menge von HOFs zur Verfügung, die je nach vorhandenem Parallelrechner auf unterschiedliche parallele Implementationen abgebildet werden können. Parallelität wird in diesem Ansatz nicht explizit ausgedrückt, sondern wird implizit durch den Gebrauch der

HOFs modelliert. Da sowohl HOFs als auch die Anwenderroutinen in einer deklarativen funktionalen Sprache geschrieben werden, begünstigt dies mögliche Transformationen des Programms während der Übersetzungsphase. Als Grundlage zur Übersetzung dienen die folgenden zwei Mengen, die die unterstützten HOFs sowie deren parallele Implementation auf verschiedenen Zielplattformen repräsentieren.

1. $\mathcal{P}$- einer Menge von Programmmustern $\{p_1, p_2, \cdots, p_n\}$, zu denen effiziente parallele Implementationen mit korrespondierenden Performanzmodellen, für die jeweilige Hardwareplattform zur Verfügung stehen. Hierzu gehören z.B. Ansätze wie die Prozessorfarm.

2. $\mathcal{H}$- einer Menge von HOFs $\{h_1, h_2, \cdots, h_m\}$, zu denen korrespondierende Muster in $\mathcal{P}$ existieren. HOFs aus $\mathcal{H}$ werden kurz als $\mathcal{H}$-HOFs bezeichnet. Als Beispiele für $\mathcal{H}$-HOFs sein hier die funktionalen Konzepte *map* und *filter* genannt.

Der Ansatz von Bratvold läßt sich gut anhand des Ablaufs einer Übersetzung beschreiben. Dabei laufen nacheinander die folgenden Schritte ab:

- Lexikalische-,Syntaxanalyse und Typkontrolle:
  Die in dieser Phase durchgeführte Typkontrolle ermittelt die Größe der Objekte, die zwischen parallelen Prozessen ausgetauscht werden.

- HOF Identifikation und Extraktion:
  Bei der Analyse des Programmgraphen wird versucht, implizite und explizite $\mathcal{H}$-HOFs mittels verschiedener Transformationen und Matchingstrategien zu erkennen.

- Ermittlung nützlicher Parallelität:
  Basierend auf Ergebnissen statischer Analysen und des Profilings wird für jede $\mathcal{H}$-HOFs bestimmt, inwieweit eine parallele Abarbeitung einen Laufzeitvorteil verspricht.

- Mapping, Lastausgleich und Optimierung:
  In diesem Schritt werden die Rechnerresourcen mittels Transformationen so auf die einzelnen Programmkomponenten abgebildet, daß die erwartete Programmlaufzeit minimiert und der Speedup maximiert wird.

- Programmerzeugung:
  Im letzten Schritt werden alle $\mathcal{H}$-HOFs, die parallel abgearbeitet werden sollen, durch ihre korrespondierenden parallelen Implementationen $\mathcal{P}$ ersetzt und diese auf die zugeteilten Prozessormengen gemappt. Anwendungsspezifischer Code wird in sequentiellen ausführbaren Code übersetzt und an entsprechender Stelle eingefügt.

Das Konzept der HOFs bietet für den Benutzer die vollkommene Transparenz von Parallelität, da er aus seiner Sicht nur mit der Entwicklung eines funktionalen Programmes beschäftigt ist. Alle weiteren Aktivitäten übernimmt der Compiler, der selbstständig parallele Strukturen erkennt und möglichst effizient auf den vorhandenen Parallelrechner abbildet. Ein Nachteil bei diesem komplexen Ansatz besteht darin, daß Parallelität nur implizit modelliert und nur in bestimmten Konstrukten vom Übersetzer erkannt werden kann. Dies bedeutet für den in der parallelen Programmierung erfahrenen Entwickler eine unnötig starke Einschränkung in der Programmentwicklung.

4.4.2 Spezialskelette

Algorithmische Skelette für mathematisch-technische Anwendungen

Der Skelettansatz von Stoltze [Sto94] spezialisiert sich auf das Gebiet mathematisch-technischer Anwendungen. Hierbei wurde für eine funktionale Programmiersprache versucht, die Effizienz paralleler Anwendung insbesondere in Anwendungsfeldern der Numerik zu verbessern und gleichzeitig die Parallelität für den Anwender transparent zu halten. Basierend auf der zu diesem Zweck entworfenen funktionalen Programmiersprache FPS wurden verschiedene algorithmische Skelette entwickelt, die das datenparallele Arbeiten auf Array-Strukturen ermöglichen. Um das Arbeiten mit Arrays weiter zu unterstützen verfügt FPS über Sprachelemente, die eine effiziente Generierung, Verwaltung und Manipulation dieser Datenstrukturen ermöglichen. Die mathematisch-technische Ausrichtung von FPS wird deutlich, wenn man einen Blick auf die verfügbaren Skelette wirft. So existieren für die in funktionalen Sprachen meist anzutreffenden Funktionen *map* und *fold* mit den Skeletten *map-array* und *fold-array* spezielle Versionen für die effiziente parallele Abarbeitung auf verteilten Arraystrukturen. Zum Datenaustausch innerhalb eines Arrays stehen die Skelette *rotate-column* und *rotate-row* zur Verfügung. Weiterhin besitzt FPS Skelette zur Permutation, zum Kopieren und zur Multiplikation von Feldern. Durch die skelettorientierte Modellierung erreicht FPS eine grobgranulare Parallelität, die in mathematischen Einsatzgebieten zu einer erheblich höheren Effizienz im Vergleich zu nicht skelettbasierten parallelen funktionalen Programmiersprachen führt.

$\mathcal{N}$-Graphen für Transputernetzwerke

Das Konzept der $\mathcal{N}$-Graphen wurde von S. Gorlatch und C.Lengauer [GL95] für Transputernetzwerke entwickelt. Ziel war es ein Skelett für Divide&-

Conquer zu entwickeln, das sich gut auf ein Prozessornetzwerk fester Größe abbilden läßt und zudem eine gute Lastbalancierung, sowie einen möglichst hohen Anteil an lokaler Kommunikation besitzt. Die Problematik dieser Forderungen wird klar, wenn man mögliche Implementationen eines binären D&C-Verfahrens betrachtet. Hierzu zählen die Einbettung eines Binärbaums bzw. eines Polynomialbaums auf das Transputernetzwerk. Diese beiden Ansätze haben jedoch auch ihre Nachteile. Bei der Binärbaumeinbettung arbeiten nur die Blattprozessoren aktiv an der Lösung der Teilprobleme mit. Die anderen Prozessoren arbeiten nur als Datenrouter bzw. kombinieren die Teillösungen. Hierdurch ergibt sich in den meisten Anwendungsfällen ein nicht ausgeglichenes Verhältnis von Kommunikation und Berechnung pro Rechenknoten. Die Binomialeinbettung ist unter diesem Aspekt zwar besser, jedoch erfordert sie schon bei kleineren Bäumen einen Grad von mehr als 4 für die oberen Knoten im Baum. Diese Tatsache erschwert die Einbettung von Binomialbäumen in Transputernetzwerke, da diese auf den Grad 4 pro Knoten beschränkt sind. Eine Lösung dieses Problems stellen die $\mathcal{N}$-Graphen dar. Sie bieten einerseits die Balance zwischen Rechnen und Kommunizieren und sind andererseits durch ihre grundlegende Struktur auf den Grad 4 beschränkt. Auf Basis dieses Graphenmodells ist es damit möglich, effiziente Implementierungen von Divide&Conquer auf Transputernetzwerken oder Parallelrechnern mit ähnlichen Restriktionen zu realisieren. subsectionkompositionelle Ansätze

PCN

Die Programmiersprache *PCN* (Parallel Programming Notation) [FOT92], [FT91],[Fos94] stellt einen kompositionellen Ansatz dar, der seine Wurzeln in der Welt der parallelen logischen Programmiersprachen besitzt. PCN wurde von Steve Tuecke und Ian Foster in Kooperation am Argonne National Laboratories und am Caltech (California Institute of Technology) entwickelt. Ziel war es, eine Programmiersprache zu entwickeln, mit der es möglich ist, parallele Programmierung auf einer hohen Abstraktionsebene zu betrachten und zu organisieren. PCN ist ein Nachfahre der Programmiersprache STRAND [FT89] und des parallelen Programmiermodells UNITY [CM88]. Wie schon Strand bietet PCN die Möglichkeit, externe C– und Fortranfunktionen in das Programm zu integrieren. Beiden Sprachen ist zudem der intensive Gebrauch von Rekursion gemeinsam. Im Gegensatz zu Strand benutzt PCN jedoch keine unstrukturierten logischen Klauseln, sondern einen funktionalen Ansatz in Kombination mit einer C-ähnlichen Syntax für die Beschreibung eines parallelen Programms. Zur Konstruktion von PCN-Programmen werden die drei Basis-Kompositoren

- parallele Komposition (parallel composition)
- sequentielle Komposition (sequential composition)
- Fallunterscheidung (choice composition)

zur Verfügung gestellt. Durch Verschachtelung der entstandenen Komponenten lassen sich komplexere Programmstrukturen erzeugen. Die Kommunikation zwischen einzelnen funktionalen Komponenten erfolgt über das Prinzip gemeinsamer logischer Variablen. Die gemeinsame Variable dient dabei als abstrakter Kommunikationskanal zwischen zwei Prozessen, der nicht prozessorgebunden ist. PCN bietet die Möglichkeit, in Kombination mit den Kompositionsoperatoren parallele Programmrahmen zu definieren, deren Korrektheit unabhängig vom Mapping ist. Die eigentliche parallele Implementation wird erreicht, indem man die einzelnen Prozesse auf die verfügbaren Prozessoren abbildet. Hierzu stellt PCN eine Gruppe von Mapping-Operatoren zur Verfügung, die einem Prozeß als Annotation im Programmtext hinzugefügt werden können. Ein weiteres wichtiges Sprachelement von PCN stellen die sogenannten *Metacalls* dar. Der Metacall ist ein abstrakter Funktionsaufruf, der als Parameter einer PCN-Funktion auftreten kann. Hiermit ist es auf einfache Art möglich, dynamisch Anwenderroutinen in ein Programmskelett zu integrieren. PCN bietet so die Möglichkeit auf einer hohen Abstraktionsebene parallele Skelette zu modellieren.

P^3L

Das P^3L-System (Pisa Parallel Programming Language) [DDM+92], [Pel93], [BD+93] verfolgt einen strukturierten parallelen Ansatz, bei dem algorithmische Skelette und kompositionelle Ansätze zum Einsatz kommen. P^3L verfügt über keine, dem Benutzer zugänglichen, primitiven Konstrukte zum Aufspannen von parallelen Prozessen oder zur Prozeßkommunikation. Der Anwender wird so gezwungen, zur Modellierung von Parallelität auf die vorgegebenen parallelen Skelette zurückzugreifen. Es besteht jedoch die Möglichkeit mittels Komposition aus bestehenden Skeletten neue Programmierrahmen zu generieren. Als Basis hierzu bietet P^3L dem Anwender z.B. die Skelette *map*, *pipe*, *reduce*, *farm*, *loop*. Zu jedem dieser Skelette existiert eine Bibliothek von Implementationsrahmen. Ein Rahmen besteht dabei aus einer Menge parametrisierbarer Prozesse, die für eine gegebene parallele Architektur eine optimierte Implementation des zugrundeliegenden Skeletts darstellen. Jedem Implementationsrahmen steht ein Performanzmodell zur Seite, das zur Laufzeitprognose eines skelettbasierten Programmsegments

genutzt wird. Das Performanzmodell stützt sich dabei auf die grundlegenden Strukturen des benutzten Parallelrechners, wie z.B. Struktur des Verbindungsnetzwerks oder Aufbau der Prozessorknoten und zieht daneben die Kosten für Kommunikation, Scheduling, Datenzugriff und anderer Basisoperationen des Parallelrechners in Betracht. Ein P^3L-Programm läßt sich als Prozeßbaum darstellen, bei dem die einzelnen Knoten die verwendeten Skelettkonstrukte, die Blätter sequentielle benutzerdefinierte Funktionen und die Kanten den Datenfluss zwischen den Skeletten bezeichnen. Da P^3L eine rein funktionale Semantik besitzt und die einzelnen parallelen Konstrukte als Datenflußmodule betrachtet werden können, ist es möglich, mittels geeigneter Transformationen den Prozeßbaum eines Programms zu modifizieren, ohne die Korrektheit des Programms zu verletzen. Diese Transformationen werden vom P^3L -Compiler in Kombination mit den Performanzmodellen eingesetzt, um das Programm so auf die Prozessoren abzubilden, daß Rechenlast und Kommunikationsaufkommen der einzelnen Prozessorknoten ausbalanciert sind und die erwartete Programmlaufzeit möglichst minimal wird. Die Optimierung mittels Transformationen kann vom Compiler zur Zeit jedoch nur durchgeführt werden, wenn die Eingabedaten zum Zeitpunkt der Übersetzung bekannt sind und der Compiler somit die Datenmenge und Kommunikationskosten berechnen kann. Eine Erweiterung des Compilers für dynamische Datenstrukturen befindet sich momentan in der Entwicklung. P^3L stellt einen sehr interessanten, in sich geschlossenen Ansatz zur skelettbasierten Progammierung dar, der den Anwender weitgehend von den typischen Problemen der parallelen Programmierung befreit. Die einzelnen Skelette verkörpern im Gegensatz zu anderen Ansätzen, wie z.B. den Skeletons von Cole keine allzu komplexen Programmrahmen. So existiert z.B. in P^3L kein direktes Skelett für Divide&Conquer, sondern muß aus den Basisskeletten *map* und *reduce* geeignet generiert werden. Dies ist jedoch kein gravierender Nachteil, da in P^3L aus hierarchisch zusammengefügten Skeletten automatisch eine gültige und effiziente Implementation generiert werden kann.

SCL

Ein funktionaler, datenparalleler Ansatz wird von der *Structured Coordination Language* SCL [DT93],[DGTY95] verfolgt. Sie wurde von John Darlington u.a. am Imperial College entwickelt und benutzt skelettorientierte Konstrukte auf verschiedenen Ebenen der Programmierung. Hierbei werden drei Typen von Skeletten unterschieden:

1. Configuration Skeletons:

Um auf ein Datenobjekt effizient parallel zugreifen zu können, muß das Objekt sinnvoll partitioniert und die Partitionen auf die einzelnen Prozessoren gemappt werden. Dieser Vorgang wird in SCL mit Hilfe sogenannter Configurations beschrieben. Eine Configuration ist dabei mit den Compileroptionen zum Datenmapping beim High Performance Fortran (HPC) vergleichbar. Sie werden in SCL jedoch mit agorithmischen Skeletten, den Configuration Skeletons realisiert.

2. Elementary Skeletons:
   Analog zur datenparallelen Strukturierung von Objekten mittels Skeletten bietet SCL einen Skelettsatz von Funktionen, die auf diesen Objekten parallel ausgeführt werden können. Für verteilte Arrays stehen die Funktionen *map*, *imap*, *fold* und *scan* zur Verfügung. Die Kommunikation zwischen Prozessoren wird durch den Austausch von Array-Elementen realisiert. Hierbei wird zwischen regulärer und irregulärer Kommunikationsskeletten unterschieden. Im regulären Fall stehen neben einem Broadcast verschiedene Versionen der Funktion *rotate* zur Verfügung mit denen einzelne Elemente, Zeilen oder Spalten vertauscht werden können. Für den irregulären Fall stehen die Skelettfunktionen *send* und *fetch* zur Verfügung, mit denen es möglich ist, Skelette zur komplexere Datenkommunikationen zu generieren.

3. Computational Skeletons:
   Zur Steuerung des parallelen Programmablaufs dient in SCL die Klasse der Computational Skeletons. Ihre Elemente verkörpern, wie in der skelettorientierten Programmierung üblich, weit verbreitete parallele Programmieransätze. SCL bietet dem Anwender die Programmierrahmen *farm* (Farming), *SPMS* (Single Program Multiple Data) und *iterateUntil* bzw. *iterateFor* (Iteration).

Die Skelette stellen aus Sicht des Programmierers Funktionen dar und können, auf Grund des funktionalen Ansatzes von SCL, beliebig kombiniert werden. Damit ist es möglich, komplexere neue Skelette zu generieren. SCL bietet für die so entstandenen Skelette einen Satz von Transformationsregeln an, mit denen die Granularität von Kommunikation und Parallelität eines Skelettes modifiziert und es so für verschiedene Plattformen optimiert werden kann. Da SCL im wesentlichen nur Sprachmittel zur Steuerung des Programmablaufs und der Datenabbildung bietet, ist es notwendig, SCL mit einer Basissprache zu kombinieren, in der der Anwender problemspezifische Routinen sequentiell modellieren kann. Die Kombination einer Basissprache *X* mit SCL wird als *structured parallel programming scheme SPP(X)* bezeichnet. Ein Beispiel hierfür ist *Fortran-S*, eine Kombination aus SCL

und Fortran. Durch den Einsatz von Programmskeletten auf verschiedenen Ebenen des Programmentwurfs und der Möglichkeit Skelette beliebig zu kombinieren, bietet SCL einen universellen Programmieransatz im Bereich der datenparallelen Programmierung. Die aus SCL abgeleitete Programmiersprache Fortran-S steht somit in ihrer Flexibilität Sprachen, wie HPF, in keiner Weise nach. Ihre skelettorientierte Programmierung bietet zudem ein hohes Maß an Wiederbenutzbarkeit von parallelen Ablaufstrukturen.

4.5 Algorithmischen Skeletten im Scheduling

Jeder der in diesem Kapitel vorgestellten Programmieransätze besitzt Potential, das Scheduling paralleler Applikationen zu unterstützen.

- Die Gruppe der allgemeinen algorithmischen Skelette bietet den Vorteil, daß ihre fest vorgegebenen parallelen Ablaufstrukturen statisch analysiert und diese Information fürs Scheduling genutzt werden kann. Auch lassen sich in den Implementationen von allgemeinen Skeletten leicht Funktionen zum Monitoring der Berechnung integrieren, die als weitere Informationsquelle genutzt werden können.

- Kompositionelle Ansätze bieten Unterstützung bei der Modellierung von parallelen Strukturen, Kommunikationskanälen innerhalb einer parallelen Applikation, sowie zwischen Applikation und Schedulingverfahren.

- Spezialskelette bieten dieselben positiven Eigenschaften allgemeiner algorithmischer Skelette. Sie erkaufen sich ihre hohe Effizienz jedoch mit einer starken Einschränkung des Einsatzbereiches.

Läßt man die Spezialskelette wegen ihres eingeschränkten Einsatzbereichs beiseite, so ergeben sich durch die Kombination der beiden verbleibenden Gruppen interessante Möglichkeiten fürs Scheduling. Ein vielversprechender Ansatz besteht darin, allgemeine algorithmische Skelette einzusetzen, die selbst wieder mittels eines kompositionellen Programmieransatzes realisiert werden. Durch dieses zweischichtige Verfahren können einerseits Informationen über die Ablaufstrukturen allgemeiner Skelette genutzt werden, andererseits erleichertert der kompositionelle Programmieransatz dieImplementierung der Skelette. Das im folgenden Verlauf vorgestellte Verfahren basiert in der ersten Schicht auf den Skeletten von Cole, die ihrerseits in der zweiten Schicht in PCN realisiert worden sind.
Coles Ansatz bietet neben den schon angeführten Eigenschaften für die vorgestellten Skelette den Vorteil, sich effizient auf einfachen Prozessorgittern

implementieren zu lassen. Zudem lässt sich die Zahl der zugeteilten Prozessoren von außen z.B. durch ein Schedulingverfahren vorgeben. Die algorithmischen Skelette von Cole stellen einen offenen Programmieransatz dar, der sich leicht in eine Schedulingumgebung integrieren läßt. Anders sieht es bei den HOFs von Bratfold aus. Hier handelt es sich um einen Vertreter eines in sich geschlossenen Programmieransatzes, der keine externe Prozessorzuteilung vorsieht, da die Nutzung paralleler Strukturen allein vom zugehörigen Compiler bestimmt wird. Es ist auch nicht einfach möglich, Funktionen zum Monitoring in die Applikationen zu integrieren, da hierzu der Compiler um die notwendigen Funktionen erweitert werden müßte. Ähnlich sieht es bei der Wahl von PCN für die zweite Schicht aus. PCN ist im Gegensatz zu P^3L ein offenes System, das sich durch seine parallelen Kontrollstrukturen und Metafunktionen gut zur Implementierung von allgemeinen algorithmischen Skeletten eignet. Weiterhin unterstützt PCN durch virtuelle Prozessortopologien die Erstellung von topologieunabhängigen parallelen Applikationen. P^3L dagegen bietet keine solchen Elemente, da hier die parallele Abarbeitung komplett vom P^3L -System kontrolliert wird. Dieser ähnlich den HOFs in sich geschlossene Ansatz hat seine Vorteile bei der automatischen parallelen Abarbeitung von einzelnen Applikationen. Er ist jedoch durch die mangelnde äußere Einflußnahme auf die Abarbeitung paralleler Strukturen weniger für den Einsatz im Scheduling geeignet. SCL hat die Einschränkung, nur datenparallele Programmierparadigmen zu unterstützen. Da bei der Implementierung algorithmischer Skelette nicht auf parallele Prozeßstrukturen verzichtet werden kann, fällt SCL, und damit alle Vertreter einer rein datenparallelen Programmierung, für die weitere Betrachtung aus.

Kapitel 5

Schedulingverfahren mit algorithmischen Skeletten

Der Einsatz algorithmischer Skelette ermöglicht es, ohne zusätzlichen Aufwand bei der Programmentwicklung, Anwendungen zu entwickeln, die eng mit dem Scheduler kooperieren können. Das Kooperationspotential einer Anwendung hängt dabei direkt von der Wahl des zugrundeliegenden Skelettes ab. Dessen Eigenschaften lassen sich besonders gut zur statischen und dynamischen Optimierung von Schedules einsetzen. Ziel der statischen Optimierung ist es, vor dem Start der Abarbeitung eines Schedules die Ressourceanforderungen einzelner Anwendungen so zu modifizieren, daß das resultierende Schedule bzgl. der benutzten Kostenfunktion optimiert wird. Dies funktioniert umso besser, je mehr im Schedule enthaltene Applikationen skalierbar sind. Die dynamische Optimierung basiert auf dem Prinzip des Prozessorremappings zwischen verschiedenen Anwendungen des Schedules, um so während der Abarbeitung des Schedules die Verteilung der Prozessoren zu optimieren und die Berechnungskosten zu minimieren. Welche Optimierungsmethoden zum Einsatz kommen können, hängt stark von der Zusammensetzung der Jobmenge und damit von den eingesetzten algorithmischen Skeletten ab. Um dies zu verdeutlichen, werden zu Beginn dieses Kapitels drei algorithmische Skelette vorgestellt und ihre unterschiedlichen Eigenschaften im Kontext des Schedulings diskutiert. Im Zentrum der Betrachtungen steht dabei die Skalierbarkeit und die Fähigkeit zum dynamischen Remapping. Im weiteren Verlauf des Kapitels wird detailiert geschildert, wie diese Eigenschaften genutzt werden können, um Schedules zu optimieren bzw. effiziente Schedules zu generieren.

5.1 Skelett für Divide & Conquer

Das erste betrachtete Skelett ist eine Variante des Divide-and-Conquer--Paradigmas (vgl. [AHU74, Wei92]), das sich allgemein wie folgt formulieren läßt:

```
if    Probleminstanz klein genug
then  löse Problem direkt
else
      Divide   : zerlege das Problem in mehrere Teilprobleme
      Conquer  : löse jedes Teilproblem rekursiv
      Combine  : berechne aus den Teillösungen die Gesamtlösung
endif
```

Abbildung 5.1: allgemeine Beschreibung des Divide-and-Conquer Paradigmas

5.1.1 Spezifikation des D&C-Skelettes

Die hier betrachtete Variante des Divide-and-Conquer geht von der Einschränkung aus, daß bei der Zerteilung eines Problems jeweils genau k Subprobleme entstehen. Der Ablauf des D&C-Skelettes läßt sich gut als Berechnungsbaum darstellen. Dabei werden die folgenden Schritte durchlaufen.

Schritt 1: übergebe der Wurzel des Baumes die initiale Probleminstanz

Schritt 2: Falls das Problem zerlegbar ist, teile das Problem in k Teilprobleme auf und übergebe diese zur Lösung an die k Söhne weiter

Schritt 3: wiederhole Schritt 2 rekursiv, solange bis die erzeugten Teilprobleme direkt gelöst werden können oder bis der gewünschte Parallelitätsgrad (Zahl der Prozessoren, die Teillösungen berechnen erreicht ist.

Schritt 4: berechne die elementaren Teilprobleme, die den Blättern des Baumes zugeordnet sind und reiche die Ergebnisse an die Elternknoten weiter

Schritt 5: alle inneren Knoten des Baumes, die Teillösungen von ihren Söhnen empfangen haben, vereinigen diese zu einer neuen Teillösung und geben diese dann ebenfalls an ihren Vaterknoten weiter

Schritt 6: wiederhole Schritt 5 solange, bis die Gesamtlösung die Wurzel des Baumes erreicht hat

Sind die Probleminstanzen vom Typ *prob* und Lösungen vom Typ *sol*, so sind für den Einsatz eines *dc*-Skelettes mit Verzweigungsgrad k vom Benutzer die folgende Funktionen zu spezifizieren:

$$\begin{aligned} indivisible &: prob \rightarrow bool \\ base\_func &: prob \rightarrow sol \\ split_k &: prob \rightarrow [prob_1, \ldots, prob_k] \\ join_k &: [sol_1, \ldots, sol_k] \rightarrow sol \end{aligned}$$

Die Funktion *indivisible* entscheidet für eine gegebene Probleminstanz, ob diese rekursiv weiterbearbeitet werden muß oder direkt durch Anwendung der Funktion $base\_func$ gelöst werden kann. Durch die Funktion $split_k$ wird ein Problem in k Teilprobleme zerlegt. Die Funktion $join_k$ beschreibt, wie k Teillösungen zu einer neuen Lösung zusammengesetzt werden.

5.1.2 Eigenschaften des Skelettes

Bei der Implementation des D&C-Skelettes kann in den verschiedenen Berechnungsphasen die inhärente Parallelität dieses Programmiermodells genutzt werden. Die maximal nutzbare Parallelität steigt dabei solange, bis das Gesamtproblem in atomare Subprobleme aufgeteilt worden ist. Während der anschließenden Lösungsphase bleibt die Parallelität konstant, um dann während des Zusammenfügens der Teillösungen wieder abzunehmen. In Abbildung 5.2 wird dies am parallelen Profil einer D&C-Anwendung mit dem Verzweigungsgrad $k = 2$ verdeutlicht. Bei der parallelen Abarbeitung kann der Berechnungsbaum des D&C–Skelettes auf verschiedene Weise auf die zugeteilten Prozessoren abgebildet werden. Die Anzahl der nutzbaren Prozessoren ist dabei nur durch die Anzahl der generierten Subprobleme begrenzt.

Skalierbarkeit

Wird jedes Subproblem auf einen einzelnen Prozessor abgebildet, so wird die maximale Parallelität erreicht. Eine solche Zuordnung ist jedoch nur sinnvoll, wenn die Granularität der erzeugten Subprobleme hinreichend groß ist und

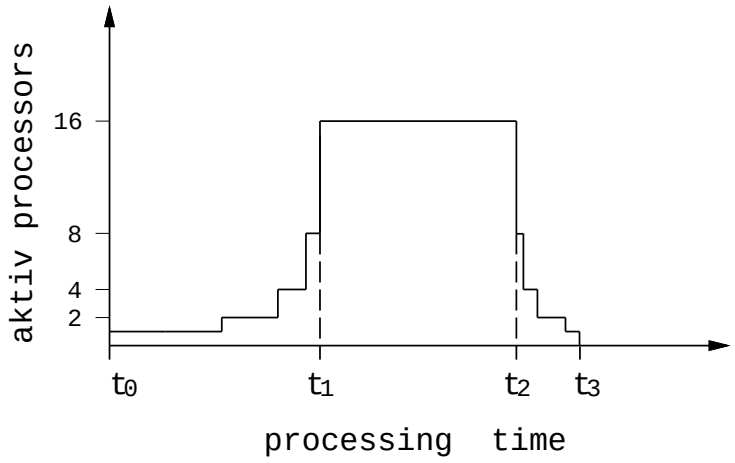


Abbildung 5.2: Paralleles Profil einer D&C–Anwendung

die Berechnungskosten eines Teilproblems deutlich größer als die anfallenden Kommunikationskosten sind.
In den meisten Fällen ist es jedoch sinnvoll, Gruppen von Subproblemen in Clustern zusammenzufassen bzw. mehrere Subprobleme auf denselben Prozessor zu mappen, um so die Granularität zu erhöhen. Für die parallele Abarbeitung einer skelettbasierten Applikation ist es erforderlich, dem Skelett Informationen über die zugeteilte Prozessormenge zu übergeben. Basierend auf diesen Informationen kann die Implementation des D&C–Skelettes so gestaltet werden, daß die erzeugten Subprobleme möglichst gleichmäßig auf die zugeteilten Prozessoren verteilt werden. Alle Applikationen, die auf einer solchen Implementation des D&C–Skelettes aufbauen, sind damit skalierbar.

Dynamisches Remapping

Um den Overhead beim dynamischen Remapping gering zu halten, ist es sinnvoll eine Änderung der Prozessorzahl nur an Zeitpunkten durchzuführen, bei denen ein Eingriff in die laufende Berechnung nur geringe Kosten erfordert. Wirft man einen genaueren Blick auf Abbildung 5.2, so stellt man fest, daß zusätzliche Prozessoren nur während der ersten Berechnungsphase, der Verteilung der Subprobleme zwischen t_0 und t_1 ohne Probleme integriert werden können. Hat die Berechnung den Punkt t_2, dem Ende der Lösungsberechnung erreicht, nimmt die Parallelität wieder ab. Die dann freiwerdenden Prozessoren werden bis zum Zeitpunkt t_3, dem Ende der Berechnung, nicht mehr benötigt und können damit anderen Applikationen zur Verfügung gestellt werden. Die Struktur des D&C–Skelettes erlaubt es, in einer Implementation des Skelettes die einzelnen Berechnungsphasen zu ermitteln und eine

aktuelle Liste der nicht mehr benötigten Prozessoren zu generieren. Diese Informationen ermöglichen es dem Scheduler, ein effizientes dynamisches Remapping zu realisieren. Hierauf wird im Verlauf des Kapitels noch detailierter eingegangen.

5.2 Skelett für Iterative Combination

Das Skelett für Iterative Combination stellt einen Rahmen dar, mit dem Greedy-Algorithmen parallelisiert werden können. Es wird parallel zu jedem Objekt s einer gegebenen Objektmenge S der optimale Partner gesucht und beide verschmolzen. Dieser Vorgang wird iterativ wiederholt bis alle Objekte zu einem einzigen Objekt verschmolzen sind.

5.2.1 Spezifikation des *ic*-Skelettes

`while` ($|S| \neq 1$ `and` weitere Berechnung möglich) `do`
`begin`
- `forall` $s \in S$
  finde den „optimalen Partner“ t für s
- vereinige alle „optimale Partner“-Paare

`end`

Abbildung 5.3: allgemeine Beschreibung der iterativen Vereinigung

Eine einzelne Iteration besteht aus der sequentiellen Ausführung der beiden folgenden Berechnungsabschnitte, die jeweils parallel über alle Objekte ausgeführt werden:

- Test&Select-Phase
  Während dieser Phase wird für jedes Objekt der optimale Partner bestimmt. Hierzu wird eine Funktion benötigt, die einen Richtwert für eine mögliche Vereinigung von zwei Objekten zurückliefert. Zusätzlich muß eine weitere Funktion bereitgestellt werden, mit der anhand solcher Bewertungskriterien das gesuchte Objekt bestimmt werden kann.

- Combine-Phase
  In dieser Phase werden die Objekte entsprechend der Wahl ihrer optimalen Partner vereinigt. Um dies realisieren zu können, wird eine Funktion benötigt, die das Zusammenfügen von Objekten ermöglicht.

Um das Skelett auf ein spezielles Problem zuzuschneiden, muß der Anwender zunächst die Typ-Beschreibung eines Objektes festlegen. Hierin müssen alle Informationen enthalten sein, die für die Durchführung der einzelnen Phasen benötigt werden. Wurde hierzu derTyp *obj* spezifiziert, so sind für den Einsatz des *ic*-Skelettes vom Benutzer folgende Funktionen zu definieren:

$$\begin{aligned} combine &: obj \times obj \rightarrow obj \\ value &: obj \times obj \rightarrow val \\ accept &: val \times val \rightarrow bool \end{aligned}$$

Die Funktion *combine* ermöglicht dabei das Zusammenfassen zweier Objekte zu einem neuen Objekt. Über die Funktion *value* erhält man ein Kostenmaß vom Typ *val*, welches das Ergebnis einer möglichen Vereinigung der beiden Objekte widerspiegelt. Eine Entscheidung darüber, welches Maß den optimalen Nutzen repräsentiert (d. h. ob das erste Kostenmaß dem zweiten vorzuziehen ist oder nicht), kann über die Funktion *accept* getroffen werden. Es gibt eine Reihe von Anwendungen für die iterative Vereinigung, bei denen die „optimale Partner"-Relation (im folgenden oP-Relation genannt) keine symmetrische Struktur besitzt; d. h., daß ein Objekt s_1, welches der optimale Partner eines Objektes s_2 ist, selbst wiederum ein anderes Objekt s_3 als eigenen optimalen Partner besitzen kann.[1]Solche Objekte müssen natürlich zu einem einzigen Objekt zusammengefaßt werden, wobei für die Korrektheit des Ergebnisses auf die „richtige"Reihenfolge der Vereinigungschritte zu achten ist. Objekte dürfen nur dann zusammengefaßt werden, wenn zwischen ihnen eine direkte oP-Relation existiert oder wenn sie schon mit weiteren Objekten vereinigt wurden, durch die eine solche Beziehung impliziert wird. Diese Eigenschaft muß bei jeder parallelen Implementierung sichergestellt werden. Da die Parallelisierung der iterativen Vereinigung im Vergleich zum Divide-and-Conquer Paradigma eine komplexere Aufgabe darstellt, sind noch weitere Punkte für eine Realisierung zu beachten:

- Repräsentation der gültigen Informationen
  Die Menge der aktuellen Objekte verändert sich von Iteration zu Iteration. Während einige Objekte ihre Gültigkeit verlieren, werden andere neu kreiert. Nach jeder Iteration muß eine konsistente Darstellung der aktuellen Objekte gewährleistet sein.

- Gemeinsame Nutzung von Daten
  Eine parallele Berechnung erlaubt die simultane Ausführung ähnlicher Arbeitsschritte (z. B. das Finden des optimalen Partners für mehrere Objekte), wobei ein einzelnes Objekt an mehreren Aktionen gleichzeitig beteiligt sein kann. Es muß daher versucht werden, die aufgrund gemeinsamer Nutzung von Informationen auftretenden Probleme durch geeignete Zugriffsverfahren oder durch Duplizieren der verwendeten Daten zu minimieren.

[1]Ein Beispiel hierfür ist das Problem zur Bestimmung eines minimalen Spannbaumes.

- Vermeidung multipler Datenströme
  Durch die simultane Durchführung der Vereinigungsschritte können mehrere Kopien eines Objektes entstehen. Es muß daher sichergestellt werden, daß jedes neue Objekt zu Beginn der nächsten Iteration genau einmal vorhanden ist.

- Terminierungserkennung
  Nach Beendigung jeder Iteration muß entschieden werden, ob noch eine weitere durchzuführen ist. Eine offensichtliche Lösung dieses Problems besteht darin, die Anzahl der aktiven Objekte zu bestimmen und diesen Wert mit der Objektanzahl der vorangegangenen Iteration zu vergleichen. Das Resultat des Vergleichs muß dann allen Prozessoren mitgeteilt werden.

- Lastverteilung
  In jeder Iteration wird i. a. die Anzahl und Größe der verbleibenden Objekte verändert. Die daraus entstehenden Lastsituationen können erheblichen Einfluß auf das Leistungsverhalten des Programms besitzen. Es sollte daher versucht werden, eine hohe Effizienz der Programmausführung durch dynamische Lastverteilungsstrategien zu bewahren.

5.2.2 Eigenschaften des Skelettes

Die parallele Implementation des *ic*-Skelettes läßt sich unter Berücksichtigung der oben genannten Kriterien leicht auf einem Prozessorring realisieren. Die Ringstruktur bietet den Vorteil, sich einfach auf beliebige Gitterstrukturen abbilden zu lassen. Somit ist eine Applikation, die auf einer solchen Implementation des *ic*-Skelettes beruht, flexibel beim Mapping und läßt sich leichter in einem Schedule anordnen.

Skalierbarkeit

Zu Beginn der Berechnung wird jedes Objekt auf ein Prozessorelement abgebildet. Existieren mehr Objekte als Prozessoren, so werden die Objekte auf virtuelle Prozessorelemente abgebildet. Diese bilden dann einen virtuellen Prozessoring, der auf die realen Prozessoren gemappt wird. Hierdurch kann einerseits für verschiedene Objektzahlen ein einheitliches Berechnungsverfahren in der Implementation benutzt werden, andererseits realisiert das Konzept der virtuellen Prozessoren auf einfache Weise die Skalierbarkeit von *ic*-Anwendungen. Bei der Skalierung muß jedoch darauf geachtet werden, daß die Anzahl der virtuellen Prozessoren ein ganzzahliges Vielfaches der

Zahl der realen Prozessoren ist, da ansonsten die Performanz durch ungleiche Prozessorbelastung verschlechtert wird.

Dynamisches Remapping

Wegen der zugrundeliegenden Ringstruktur, können während der Berechnung ohne eine komplette Restrukturierung keine neuen Prozessoren integriert werden. Anders sieht es bei der vorzeitigen Freigabe von Prozessoren aus. Tritt während der Berechnung der Fall ein, daß die Zahl der Objekte nur noch halb so groß wie die Zahl der Prozessoren ist, so ist es sinnvoll, die Hälfte der Prozessoren freizugeben. Hierzu bestimmt man zuerst die Menge der Prozessoren, die freigegeben werden soll. Danach werden alle Objekte auf die verbleibenden Prozessoren abgebildet und diese Prozessoren zu einem neuen Ring verschaltet. Dieses Vorgehen hat den Vorteil, daß der neue Prozessorring nicht über freigegebene Prozessoren geführt werden muß. Das prinzipielle Vorgehen dieser internen Restrukturierung wird noch einmal durch Abbildung 5.4 verdeutlicht. Anwendungen, die auf dieser Implementierung des *ic*-Skelettes beruhen, können in einem dynamischen Remappingprozeß aus den genannten Eigenschaften nur Prozessoren abgeben und somit einzig als Prozessorquelle fungieren.

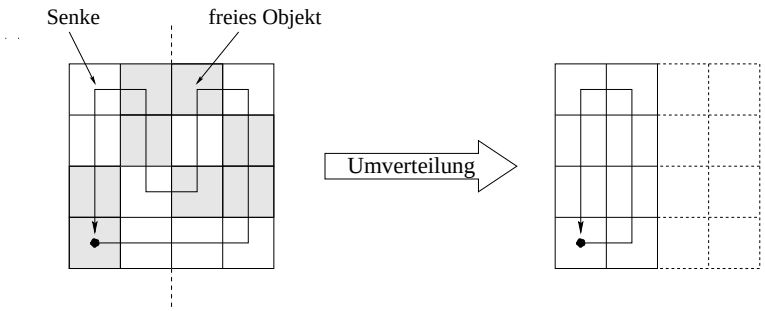


Abbildung 5.4: Interne Restrukturierung beim *ic*-Skelett

5.3 Skelett für Farming

Das Skelett für Farming stellt einen parallelen Rahmen für Anwendungen dar, bei denen ein gegebenes Problem in eine Menge von unabhängigen Teilproblemen zerlegt und aus deren Teillösung die Gesamtlösung berechnet werden soll. Im Gegensatz zum Divide & Conquer wird hier nicht von einer hierarchischen Problemzerlegung ausgegangen.

5.3.1 Spezifikation des Farming-Skelettes

Die Arbeitsweise des Farming Skelettes wird aus Sicht des Benutzers durch den folgenden Algorithmus beschrieben:

Nimm die von generate_subproblems erzeugten Subprobleme
und lege sie in der Liste P ab;
while P $\neq$ []
    {
    hole subproblem aus P;
    solve_subproblem;
    Leg Ergebnis in Liste S ab;
    }
übergib S an die Funktion combine_solutions ;

Das Skelett generiert zuerst alle Subprobleme und speichert sie in einer Liste zwischen. Danach werden sukzessiv alle Subprobleme gelöst, zwischengespeichert und abschließend die Gesamtlösung berechnet. Diese sequentielle algorithmische Sicht auf das Skelett, ermöglicht es dem Benutzer anwendungsspezifische Funktionen zu entwickeln ohne mit den Problemen paralleler Programmierung kämpfen zu müssen. Die parallele Ausführung des Skelettes ist in der Implementation gekapselt und transparent für den Benutzer. Der Anwender hat die folgenden Funktionen zu ergänzen, um mit dem Farming Skelett eine parallele Applikation zu erstellen:

$$\begin{aligned} \mathit{generate\_subproblems} &: \mathit{prob} \rightarrow [\mathit{prob}_1, \ldots, \mathit{prob}_k \\ \mathit{solve\_subproblem} &: \mathit{prob} \rightarrow \mathit{sol} \\ \mathit{combine\_solutions} &: [\mathit{prob}_1, \ldots, \mathit{prob}_k] \rightarrow \mathit{sol} \end{aligned}$$

Die Funktion *generate_subproblems* dient dazu, das initiale Problem in Teilprobleme zu zerlegen. Die generierten Teilprobleme werden dabei in einer

Liste abgelegt. Die Funktion *solve_subproblem* stellt den Rahmen für die Lösung einzelner Teilprobleme dar. Die Kombination der Teillösungen muß vom Benutzer mittels der Funktion *combine_solutions* spezifiziert werden. Zum Datenaustausch wird analog zur Funktion *generate_subproblems* eine Listenstruktur eingesetzt, die zur Akkumulation der Teillösungen dient.

5.3.2 Eigenschaften des Farming-Skelettes

Das Farming-Skelett verkörpert, worauf schon der Name hindeutet, das parallele Programmierparadigma des Farmings. Hierbei kooperiert ein zentraler Kontrollprozessor mit einer Menge voneinander unabhängiger Arbeitsprozessoren. Der Kontrollprozessor, auch *Master* genannt, erhält zu Beginn des Berechnungsablaufes das zu lösende Problem. Dieses wird von ihm in unabhängige Teilprobleme zerlegt und an die einzelnen Arbeitsprozessoren, als *Worker* bezeichnet, verteilt. Die Lösungen der Teilprobleme fließen zum Master zurück und werden von diesem zur Gesamtlösung zusammengefügt. Die Unabhängigkeit der zu berechnenden Teilprobleme stellt das wesentliche Charakteristikum aller Anwendungen dar, die mittels Farming berechnet werden können. Applikationen mit datenabhängigen Teilproblemen erfordern ein hohes Maß an Kommunikation, die bei einem zentral gesteuerten Ansatz wie dem Farming, zu einem Kommunikationsengpaß und somit zu einer ineffizienten Programmabarbeitung führen kann.

Skalierbarkeit

Das grundlegende Konzept des Farmings erlaubt es, die Zahl der Worker-Prozessoren in einem weiten Rahmen beliebig zu wählen und bietet somit eine sehr gute Skalierbarkeit. Die maximale nutzbare Prozessorzahl ist analog dem D&C-Paradigma durch die Anzahl der Subprobleme beschränkt. Aus Effizienzgründen ist es jedoch meist sinnvoll, weniger Worker einzusetzen, um die Rechenlast pro Arbeiter zu erhöhen.

Dynamisches Remapping

Die Struktur des Farming Skelettes bietet die Möglichkeit, während einer laufenden Berechnung Worker-Prozessoren zur Farm hinzuzufügen oder zu entfernen. Dazu muß die Programmevaluation weder reorganisiert noch unterbrochen werden. Um einen Prozessor als Worker hinzuzufügen, ist es nur notwendig, ihn beim Master anzumelden. Die Entfernung eines Workers kann erfolgen, wenn dieser die Bearbeitung eines Subproblems fertiggestellt hat und ein neues Problem anfordert. Der entsprechende Worker erhält dabei

statt eines neuen Subproblems eine Terminierungsbotschaft. Diese veranlaßt ihn, sich aus der Prozessor-Farm abzumelden. Das Farming Skelett ist, vom Standpunkt des dynamischen Remapping aus gesehen, am universellsten einsetzbar, da es bei einem Prozessorremapping sowohl Prozessoren freigeben als auch aufnehmen kann. Seine Universalität wird durch die relativ einfache Kontrollstruktur erreicht und aus Sicht der Programmentwicklung mit einem eingeschränkten Einsatzbereich erkauft.

5.4 Optimierung von Schedules mittels skalierbarer paralleler Programme

5.4.1 Speedup-Prognose skelettbasierter Programme

Die Reskalierung einer Anwendung läßt sich nur effizient zur Optimierung von Schedules einsetzen, wenn für die neue Prozessorzahl auch die zugehörige neue Rechenzeit bekannt ist. Die Zeiten müssen dazu entweder vom Benutzer zur Verfügung gestellt oder näherungsweise vom Scheduling–System generiert werden. Die hierbei gewonnenen Ergebnisse sind je nach vorliegenden Informationen nur Näherungswerte, die von der realen Rechenzeit abweichen können. Die Auswirkungen von Fehlprognosen können vermindert werden, wenn man konservative Abschätzverfahren verwendet, die bei der Reskalierung vom worst-case ausgehen. Es läßt sich so eine Unterschätzung der realen Laufzeit und deren Auswirkung auf den Ablauf des restlichen Schedules vermeiden. Zu Beginn des Abschnitts stehen Verfahren im Mittelpunkt, die solch konservative Laufzeitprognosen realisieren. Neben diesen Verfahren werden noch andere Methoden erörtert, die auf dem Gebiet der Laufzeitprognose ihren Einsatz finden. Stehen zur Laufzeitprognose nur Verfahren zur Verfügung, die keine verläßliche Abschätzung der Maximallaufzeit erlauben, so kann man beim Aufbau des Schedules versuchen, die Auswirkungen von Fehlprognosen zu verringern. Hierauf wird im Abschluß des Abschnitts eingegangen.

Konservative Rechenzeitschätzung

Betrachtet man die Speedup-Kurven typischer paralleler Anwendungen, so stellt man fest, daß der Speedup zwar ein monoton wachsendes Verhalten zeigt, sich jedoch meist asymptotisch einem absoluten Maximum nähert. Geht man von einer Applikation J_i aus, die diese Eigenschaft besitzt und mißt man für sie die Rechenzeit $T_i(p_i)$ beim Einsatz der maximal nutzbaren Prozessorzahl p_i, so läßt sich mittels dieser Meßwerte eine verläßliche konservative Rechenzeitabschätzung realisieren. Man definiert hierzu die maximale Rechenzeit für Job i bei Einsatz von j Prozessoren mit $j \in [1, ..., p_i]$ als $T_i(j) = T_i(p_i) * \frac{T_i}{j}$. Diese Prognose liefert nie eine Überschätzung der Rechenzeit, da sie auf einer linearen Speedup-Kurve beruht, die maximal die Speedup-Werte der realen Kurve erreichen, aber diese nie überschreiten kann. Abbildung 5.5 verdeutlicht dies.
Benutzt man die ermittelten Daten zur Reskalierung, so wird garantiert, daß die geschätzten Joblaufzeiten auch real eingehalten werden können. Es erfolgt damit eine Optimierung für den worst-case. Die konservative Schätzung

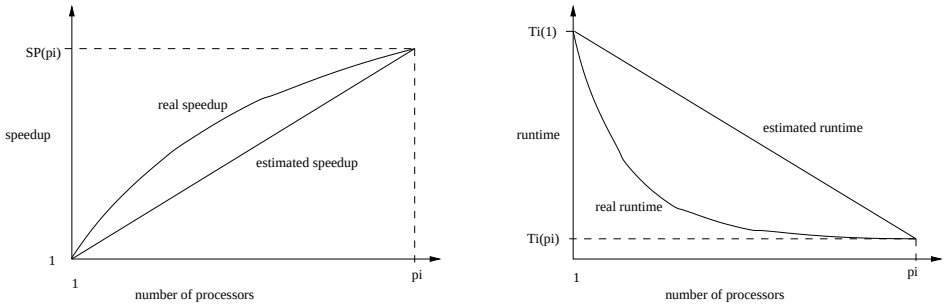


Abbildung 5.5: konservative Schätzung

bietet sich also vor allem für zeitkritische Schedules an, bei denen bestimmte Zeitschranken zwingend eingehalten werden müssen. Eine solche Laufzeitschätzung läßt sich in ihrer Qualität noch weiter verbessern, wenn die Laufzeiten eines Jobs für verschiedene Prozessorzahlen vorliegen. Verbindet man die einzelnen Meßwerte mittels linearer Interpolation, so erhält man wiederum eine konservative Schätzung mit den bekannten positiven Eigenschaften. Abbildung 5.6 verdeutlicht dieses Vorgehen. Eine automatische iterative Verbesserung der konservativen Schätzung läßt sich elegant in die algorithmischen Skelette integrieren. Jede Skelettbasierte Applikation erhält dazu ein automatisch eingefügtes History-Modul, das für jeden Programmlauf die Anzahl der zugeteilten Prozessoren und die tatsächlich benötigte Rechenzeit mitprotokolliert. Die so generierten Daten können dann vor dem

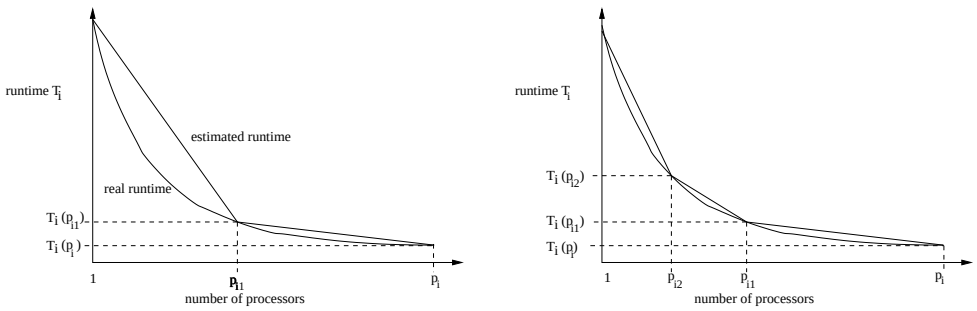


Abbildung 5.6: Iterative Verbesserung der Schätzung

neuen Programmstart dem Scheduler zur Verfügung gestellt werden, um im Fall einer Reskalierung die neue Rechenzeit zu prognostizieren.
Eine korrekte Laufzeitschätzung kann jedoch nur erfolgen, wenn sich für die verschiedenen Programmläufe die Eingabedaten nicht ändern. Trifft dies

nicht zu, so wird die automatische Laufzeitprognose erheblich komplexer, wenn nicht unmöglich.

Laufzeitprognose mittels Programmklassen

Eine weitere Möglichkeit Information über mögliche Programmlaufzeiten zu ermitteln besteht darin, die einzelnen Applikationen in Laufzeitklassen einzuteilen. Ein solcher Ansatz wird von R. Gibbons [Gib96] propagiert. Mit diesem Ansatz wird jedoch nur eine grobgranulare Laufzeiteinteilung erreicht. Eine weitere Verbesserung der Klassifikation kann erzielt werden, wenn man neben den Applikationen auch die Eingabedaten klassifiziert. Dazu ist es notwendig, geeignete Maße zur Quantifizierung der Eingabedaten zu definieren. Jede Kombination von Programm und Eingabedaten kann nun einer Kombination von Programm- und Eingabeklasse zugeordnet werden. Diese Klassifikation dient dann als Grundlage zur Laufzeitprognose. Je mehr Programm- und Eingabeklassen zur Verfügung stehen, desto genauer wird die Laufzeitprognose. Die besten Ergebnisse können erzielt werden, wenn man die zugrundeliegenden Datenstrukturen des Programms analysiert und ihren Einfluß auf die Programmlaufzeit bestimmt. Diese Technik wurde in [CGW96] benutzt, um eine optimierte Evaluation einer Fuzzy-Library auf verschiedenen Hardwareplattformen zu erreichen.

5.4.2 Einsatz vordefinierter Datentypen

Stellt man einem Benutzer Algorithmische Skelette und vordefinierte Datenstrukturen als Sprachobjekte zur Verfügung, so ist es möglich, eine weitere Verbesserung bei der Laufzeitanalyse zu erzielen. Kosten für den Transport, Vergleich und anderer Basisoperationen von Datenobjekten können dann vorab statisch analysiert werden. Auf diesen Grundinformationen basierend kann mittels der schon erwähnten Quantifizierung der Eingabedaten eine genauere Prognose erfolgen. Die prinzipiellen Ideen eines solchen Ansatzes wurden in [CG96] betrachtet, um eine automatische Partitionierung und das Scheduling paralleler agentenbasierter Systeme zu realisieren.

Exakte Programmanalyse

Die qualitativ besten Prognoseergebnisse lassen sich mit einer detailierten Programmanalyse erreichen. Hierzu ist ein detailliertes Modell des zugrundeliegenden Parallelrechners, sowie des darauf laufenden Betriebssystems notwendig. Eine detaillierte Programmanalyse erfordert einen hohen zeitlichen manuellen Aufwand. Sie ist damit primär nur für Programme sinnvoll, die

eine sehr lange Rechenzeit besitzen bzw. sehr oft ausgeführt werden müssen. Für Programme, die nur selten abgearbeitet werden oder deren Eingabedaten einen nicht quantifizierbaren Einfluß auf die Programmrechenzeiten haben, spielt eine detailierte Programmanalyse aus Kostengründen eine untergeordnete Rolle.

Effizienzverbesserung von Schedules mit grobgranularer Laufzeitprognose

Bei grobgranularen Laufzeitprognosen können die realen Laufzeiten einzelner Jobs ihre zugehörigen Laufzeitprognosen im Gegensatz zur konservativen Laufzeitschätzung deutlich übertreffen. Hierdurch kann der Ablauf eines Schedules stark verzerrt und die Prozessorauslastung deutlich verringert werden. Siehe dazu Abbildung 5.7. Um diesen Einfluß zu minimieren, ist es sinnvoll, die Auswirkungen von lokalen Fehlern auf das Gesamtschedule zu begrenzen. Man kann dies mittels verschiedener Ansätze realisieren.

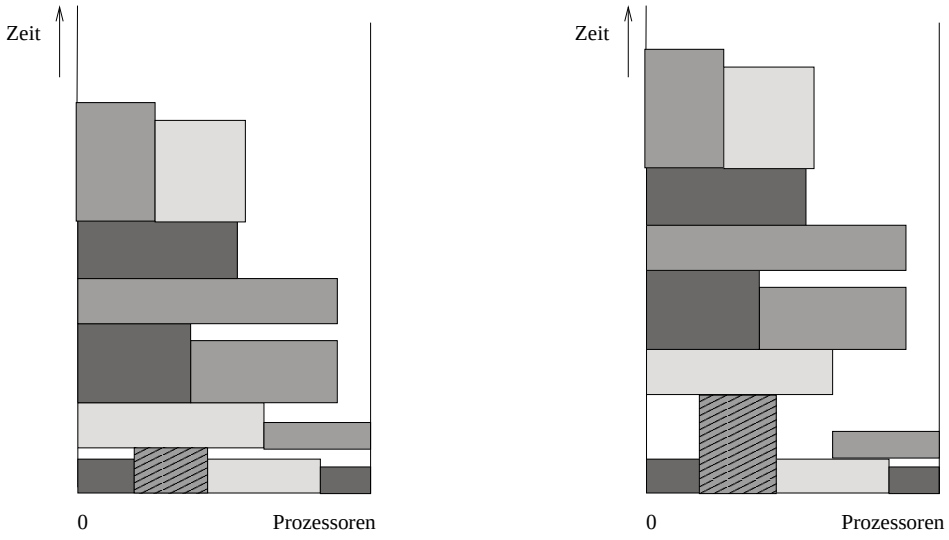


Abbildung 5.7: Auswirkung eines Jobs mit schlechter Prognose bei ungünstiger Plazierung

1. Zeitlich späte Plazierung einer kritischen Applikation:
   Je größer die erwartete Laufzeitvarianz bezogen auf den Prognosewert ist, desto später wird die Applikation ausgeführt. Hierdurch wird erreicht, daß möglichst wenige Applikationen durch eine Laufzeitverzögerung von J_i beeinflußt und gegebenenfalls zu spät gestartet werden.

2. Minimierung der Anzahl abhängiger Applikationen:
   Neben der Methode der späten Jobausführung kann auch mit anderen Mitteln versucht werden, die Anzahl der von einem Job beeinflußten Applikationen zu verringern. Man erreicht dies, indem man die einzelnen Jobs so im Schedule anordnet, daß es aus möglichst vielen voneinander unabhängigen vertikalen Partitionen besteht. Kommt es in einer dieser Partitionen zu einer unerwünschten Verlängerung der Laufzeit eines Jobs, so wirkt sich dieses Problem nur auf die nachfolgenden Jobs in der zugehörigen Partition aus. Damit bleibt die Störung lokal und ihre Auswirkungen auf das Schedule begrenzt. Eine vertikale Partitionierung des Schedules ist jedoch meist nicht ohne Reskalierung einzelner Applikationen möglich. Kann diese nicht sinnvoll durchgeführt werden, so muß das Schedule so organisiert werden, daß hinter Applikationen mit grobgranularer Zeitprognose möglichst nur solche Anwendungen plaziert werden, die eine räumliche Ausbreitung der Störung möglichst klein halten. Die hierzu notwendigen Prinzipien werden im Kapitel über dynamisches Remapping näher erläutert.

Wirft man einen abschließenden Blick auf die Thematik der Laufzeitprognose, so stellt man fest, daß es möglich ist durch den Einsatz von vordefinierten Programmrahmen und Datenstrukturen verläßliche Resultate zu erhalten. Je mehr der Programmentwickler auf diese Hilfsmittel zurückgreift, desto genauer werden dabei die Ergebnisse. Für den Anwender bedeutet dies keinen zusätzlichen Aufwand für die Laufzeitanalyse, er bezahlt jedoch die exakte Prognose durch eine zum Teil starke Einschränkung bei der Wahl der Programmierparadigmen und Datenstrukturen.

5.4.3 Optimierung durch Reskalierung

Bei der Generierung von Schedules für eine Menge paralleler Applikationen tritt oft der Fall auf, daß das erzeugte Schedule zwar eine optimale Anordung der einzelnen Applikationen bezüglich Ausührungszeit realisiert, die Auslastung des Parallelrechners jedoch nicht besonders hoch ist. Die Menge der Resource-Anforderungen der einzelnen Applikationen besitzt in diesen Fällen eine ungünstige Zusammensetzung, die dazu führt, daß keine effizientere Plazierung möglich ist. Abbildung 5.4.3 zeigt ein typisches Beispiel für diesen unerwünschten Fall. Hier verhindert der Prozessorbedarf der einzelnen Applikationen weitgehend die gleichzeitige Abarbeitung mehrerer paralleler Anwendungen.

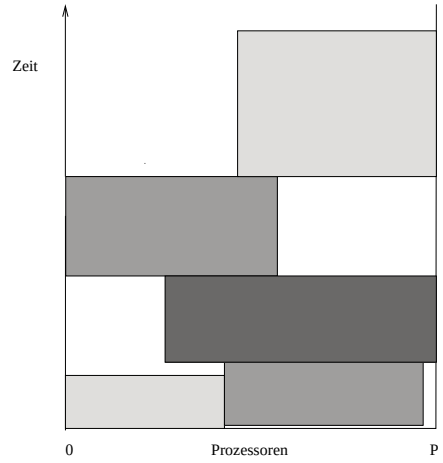


Abbildung 5.8: Schedule mit ungünstiger Zusammensetzung

Ein besseres, effizienteres Schedule kann in diesem Fall nur dann ermittelt werden, wenn entweder

- die Zusammensetzung der Menge der Applikationen modifiziert wird oder
- die Resource-Anforderungen einzelner Applikationen geändert werden.

Der erstgenannte Ansatz kann nur dann zum Einsatz kommen, wenn zusätzliche Anwendungen zum Zeitpunkt der Berechnung des Schedules zur Verfügung stehen und aus den Anwendungen eine geeignete Teilmenge zur Berechnung des Schedules gewählt werden kann. Ist dies nicht der Fall, so muß auf das Eintreffen weiterer Anwendungen gewartet werden. Im Worst-Case ist

das unter Einbeziehung des neuen Jobs berechnete Schedule nicht besser als das alte. Daher ist es nicht sinnvoll mit der Berechnung des Schedules auf das Eintreffen neuer Applikationen zu warten, sondern zu versuchen, neue Jobs in das laufende Schedule zu integrieren, wie es z.B. beim Back-filling geschieht. Der zweite Ansatz die Resouceanforderungen einzelner Applikationen eines Schedules zu modifizieren, bietet deutlich mehr Möglichkeiten zu einem effizienteren Schedule zu gelangen. Dies ist jedoch nur bei Applikationen realisierbar, deren Prozessorzahl skalierbar ist. Weiterhin ist es, wie im vorhergehenden Abschnitt beschrieben, notwendig, für die neue Prozessorzahl eine verläßliche Laufzeitprognose anzugeben. Die Problematik läßt sich jedoch, wie schon erwähnt, durch den Einsatz algorithmischer Skelette lösen. Verlagert man die Skalierbarkeit von der eigentlichen Applikation in die Realisierung des Skeletts, so erben alle Applikation, die dieses Skelett benutzen, die Fähigkeit zur Reskalierung. Zur Laufzeitprognose bietet sich die im vorhergehenden Abschnitt vorgestellte iterative Version der konservativen Laufzeitabschätzung an. Da dieser Ansatz vom worst-case ausgeht, ist sichergestellt, daß eine Reskalierung, die theoretisch die Gesamtkosten verbessert, das Ziel auch in der Praxis erreicht. Zudem fällt die reale Laufzeit einer Anwendung meist kürzer als die konservative Schätzung aus, ergibt sich dadurch in der Praxis noch eine zusätzliche Effizienzsteigerung. Die Optimierung von Schedules mittels skalierbarer Anwendungen, kann leicht mit bestehenden Schedulingverfahren gekoppelt werden und arbeitet ohne Probleme mit den verschiedenen Bewertungsfunktionen zusammen. Dies wird erreicht, indem der Optimierungvorgang aus Sicht des Schedulingverfahrens total transparent gestaltet wird.
Der Scheduler geht von einem homogenen Jobmodell aus, bei dem die Resourceanforderungen der einzelnen Jobs aus seiner Sicht statisch sind. Die eigentliche Reskalierung einzelner Anwendungen wird von einem Transformationsmodul ausgeführt, das die Ressourceanforderungen einzelner Anwendungen modifiziert und die neuen Anforderungen wieder dem Scheduler übergibt. Die eigentliche Optimierung des Schedules ergibt sich durch die iterative Kombination von Transformation und Berechnung des Schedules. Der Algorithmus in Abbildung 5.4.3 verdeutlicht dies schematisch.

5.4.4 Einfluß skalierbarer Applikationen

Da in der Regel nicht alle Applikationen skalierbar sind, stellt sich die Frage, ob in diesem Fall eine Optimierung mittels Reskalierung sinnvoll ist und den zusätzlichen Aufwand rechtfertigt. Um diese Frage zu beantworten, wurden in einer Simulation Jobmengen mit einer unterschiedlichen Anzahl skalierbarer Applikationen betrachtet. Ziel der Untersuchung war es festzustellen,

geg : Jobmenge $J_1, \ldots, J_n$
ges : optimiertes Schedule S_{opt}

Berechne initiales Schedule S
Setze $S_{alt} = S$
Iteriere k mal:
    Selektiere skalierbaren Job J_i
    Generiere neue Prozessorzahl P_i und neue Laufzeit h_i
    Berechne neues Schedule S_{neu}
    Falls $cost(S_{neu}) < cost(S_{alt})$
        Setze $S_{alt} = S_{neu}$
Setze $S_{opt} = S_{alt}$
Gib optimiertes Schedule S_{opt} aus

Abbildung 5.9: Optimierung durch Reskalierung

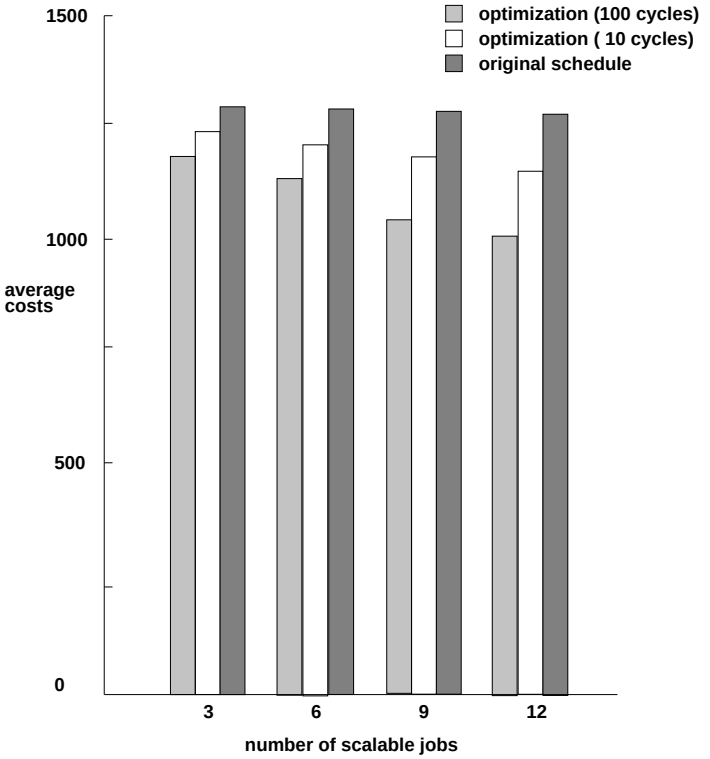


Abbildung 5.10: Auswirkung der Optimierung mittels Reskalierung

wie sich der prozentuale Anteil skalierbarer Funktionen auf eine Optimie-

rung des Gesamtschedules auswirkt. Die betrachtete Jobmenge setzte sich dabei aus 12 Applikationen zusammen, deren Prozessorbedarf und Laufzeiten zufällig gewählt wurden. Für jeweils 1000 Jobkombinationen wurde mit dem PBI-Schedulingverfahren ein Schedule berechnet und die durchschnittlichen Kosten bestimmt. Zudem wurden die einzelnen Schedules durch iterative Optimierung zu verbessert. Um die Rechenzeit der Optimierung möglichst gering zu halten, wurde die Zahl der Iterationszyklen bewußt klein gehalten. Die Ergebnisse der Simulation finden sich in Abbildung 5.10. Wie zu erwarten, liefert die Optimierung mit steigender Zahl skalierbarer Applikationen oder wachsender Zahl von Iterationsschritten bessere Ergebnisse.
Beachtlich ist, daß bei einem Anteil von 25 % an skalierbaren Anwendungen schon eine Kostenverbesserung um ca. 10 %. erzielt werden kann. Die Optimierung durch Reskalierung ist damit unter ungünstigen Randbedingungen ein probates Mittel zur Effizienzverbesserung von Schedules.

5.5 Dynamisches Remapping von Prozessoren

Beim dynamischen Remapping werden abhängig vom Berechnungszustand Prozessoren zwischen zwei Applikationen ausgetauscht. Dabei wird die Zielsetzung verfolgt, die Qualität des berechneten Schedules während der Laufzeit zu verbessern. Die hierzu notwendigen Daten über den Berechnungszustand einzelner Anwendung kann leicht vom verwendeten Skelett generiert werden. Hierzu dienen Monitorfunktionen, die in die Skelettimplementation integriert sind und eine wohldefinierte Schnittstelle besitzen. Bei den vorgestellten Skeletten wird intern der aktuelle Zustand der Berechnung gespeichert. Eine mögliche Prozessorabgabe wird dem System dadurch signalisiert, daß die freiwerdenden Prozessoren in eine Abgabeliste übernommen werden. Prozessornummern, die in dieser Liste auftauchen, werden von der Applikation nicht mehr benötigt und können auch nicht mehr von der Anwendung zurückgenommen werden.

5.5.1 Integration von nicht skelettbasierten Programmen

Da skelettbasierte Programmierung die Universalität des Programmaufbaus zu Gunsten einer komfortableren zeiteffizienten Entwicklung von parallelen Programmen aufgibt, besteht der Wunsch, auch nicht skelettbasierte Programme ins dynamische Scheduling integrieren zu können. Hierzu wird das Dummy-Skelett *fixedsized* benutzt. Dieses Skelett kennzeichnet die Menge von Applikationen, für die die folgenden Bedingungen gelten:

- Die Zahl der benötigten Prozessorelemente bleibt während der gesamten Berechnung konstant bzw. es bleibt dem System unbekannt, ob vorzeitig Prozessoren freigegeben oder zusätzliche Prozessoren in die laufende Berechnung integriert werden können.
- Die Prozessorelemente müssen vollständig zu Beginn der Berechnung zur Verfügung stehen.

Auf Grund dieser Eigenschaften werden solche Anwendungen mit dem Begriff *fixedsized* bezeichnet. Die Einbettung in das Dummy-Skelett dient einerseits dazu, dem Scheduler diese Informationen zur Verfügung zu stellen und andererseits eine einheitliche Modellierung aller Anwendungen zu realisieren. Hierdurch kann der Aufbau des Schedulers vereinfacht werden, da keine

formale Unterscheidung zwischen skelettbasierten und nicht skelettbasierten Anwendungen getroffen werden muß.

5.5.2 Horizontales Remapping

Beim Prozessor-Remapping zwischen zwei gleichzeitig laufenden Anwendungen erfolgt eine Umverteilung entlang der horizontal verlaufenden Prozessorachse des Schedules. Aus diesem Grund wird das Verfahren hier als horizontales Remapping bezeichnet. Ein solches Remapping von Applikation A_1 zu Applikation A_2 ist nur möglich, wenn A_1 während des Programmlaufs Prozessoren entzogen und der Applikation A_2 hinzugefügt werden können. Um mittels horizontalem Remapping eine Laufzeitverbesserung des gesamten Schedules zu erreichen, müssen die folgenden Punkte erfüllt sein:

1. Applikation A_1 besitzt Prozessoren, die bis Ende der Berechnung nicht mehr benutzt werden
2. Applikation A_2 erreicht durch die Hinzunahme dieser freien Prozessoren eine kürzere Programmlaufzeit.
3. Die Kosten für eine Umverteilung dürfen den erreichten Laufzeitgewinn nicht übersteigen.

Diese Voraussetzungen werden nur in sehr seltenen Fällen erreicht. Ein sinnvolles Beispiel hierfür ist der Prozessoraustausch zwischen einer D&C Anwendung und einer Farming Applikation. Während des Ablaufs der D&C Applikation werden sukzessiv Prozessoren frei, die als zusätzliche Worker in der Farming Anwendung integriert werden können. Da geeignete Partner zum horizontalen Remapping aber äußerst selten anzutreffen sind, wird auf dieses Verfahren nicht weiter eingegangen.

5.5.3 Vertikales Remapping

Beim vertikalen Remapping findet der Prozessoraustausch im Gegensatz zum horizontalen Remapping nicht entlang der Prozessor-, sondern der Zeitachse statt. Hierbei erfolgt ein Remapping nur zwischen Anwendungen, die in direkter zeitlicher Abfolge stehen. Ziel ist es, freiwerdende Prozessoren möglichst frühzeitig einer nachfolgenden Anwendung zur Verfügung zu stellen. Dadurch soll erreicht werden, daß deren Berechnungszeit verkürzt bzw. deren Terminierungszeitpunkt vorzeitig erreicht wird. Dafür ist es notwendig, daß

Vorgänger:	D&C	Farming	Fixed	*ic*
Nachfolger:				
D&C	0	+	-	0
Farming	+	++	-	+
Fixed	0	+	-	0
ic	0	+	-	0

Tabelle 5.1: Kombinationen von Skeletten

1. Applikationen vorzeitig die benötigten Prozessorresourcen erhalten bzw.
2. vorzeitig mit einem Teil der ihnen zugeteilten Prozessoren starten können.

Welcher dieser beiden Fälle beim vertikalen Remapping genutzt werden kann, hängt stark vom Aufbau der Anwendung bzw. des verwendeten algorithmischen Skeletts ab. Welche Kombinationen beim vertikalen Remapping erfolgversprechend sind, läßt sich aus Tabelle 5.1 und den folgenden Beispielen entnehmen.

- **Remapping zwischen einer D&C Applikation und einer Fixed–sized Anwendung**
  Da Fixed–sized Anwendungen erst starten können, wenn alle zugewiesenen Prozessorelemente verfügbar sind, besteht das Ziel des Remapping hier darin, die Prozessoren möglichst früh zur Verfügung zu stellen. Das wird in diesem Beispiel dadurch erreicht, daß die D&C Anwendung zuerst die von der Fixed–sized Applikation benötigten Prozessoren zurückgibt. Dazu muß die Berechnung der D&C Anwendung so strukturiert werden, daß die benötigten Prozessoren eine Teilmenge der Blätter bzw. der unteren Schichten des D&C–Berechnungsbaums bilden. Hierzu wird dem D&C–Skelett zum Zeitpunkt des Programmaufrufs eine Liste der Prozessoren übergeben, die zuerst freigegeben werden sollen.

- **Remapping zwischen einer D&C Applikation und einer Farming Anwendung**
  Dieses Beispiel spiegelt den Idealfall des vertikalen dynamischen Remappings wider. Die von der D&C Anwendung freigegebenen Prozessoren können direkt der Farming Applikation übergeben werden. Dadurch wird erreicht, daß kein Prozessor auf Arbeit warten muß und sich die Berechnungszeit für das Farming durch vorzeitigen Start verkürzt.

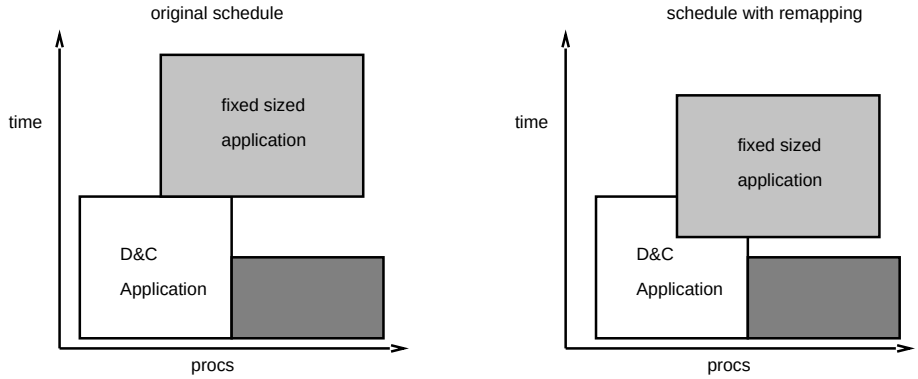


Abbildung 5.11: Remapping zwischen einer D&C Applikation und einer Fixed–sized Anwendung

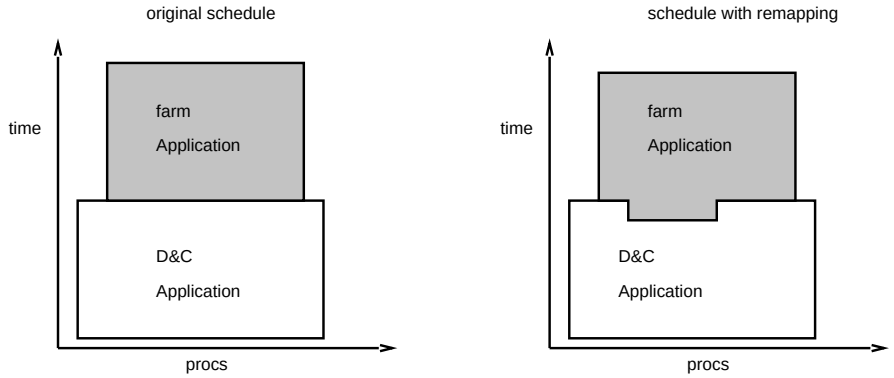


Abbildung 5.12: Remapping zwischen einer D&C Applikation und einer Farming Anwendung

- **Dynamischer Start einer Farming Applikation**
  Applikationen, die nicht darauf angewiesen sind, zum Programmstart alle zugeteilten Prozessorresourcen zu besitzen, können Ihre Programmlaufzeit dadurch verkürzen, daß sie ihre Berechnung mit einer Teilmenge der zugewiesenen Prozessoren starten. Die Startzeit der Anwendung verlagert sich dabei von der eigentlichen Startzeit Ts zu einem früheren Zeitpunkt Ts'. Hierdurch können Prozessoren aktiviert werden, de ansonsten bis zum Zeitpunkt Ts inaktiv bleiben würden. Anwendungen, die mittels des Farming Skelettes realisiert wurden, sind für diese Art des dynamischen Mapping hervorragend geeignet. In Bild 5.5.3 ist dies im Zusammenspiel mit zwei anderen Applikationen zu sehen. Die Farming Anwendung startet, sobald freie Ressourcen zur Verfügung stehen.

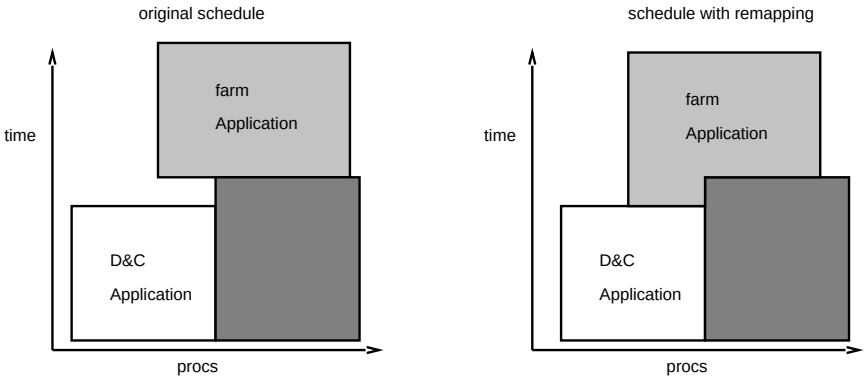


Abbildung 5.13: Dynamischer Start einer Farming Applikation

Das vertikale Remapping bietet damit ideale Möglichkeiten, um mit einem minimalen Overhead an Verwaltungskosten, den Ablauf eines Schedules zu verbessern. Durch die Integration des Remappings in die Skelette ergibt sich für den eigentlichen Scheduler kein zusätzlicher Aufwand während der Abarbeitung der Applikationen.

5.6 Das Kostenmodell für das Scheduling

Beim Einsatz von Skeletten im Scheduling paralleler Applikationen wächst das Potential zur Effizienzsteigerung mit der Zahl der skelettbasierten Applikationen. Für den Betreiber eines Parallelrechners ist es daher sinnvoll den Anwender zum Einsatz von Skeletten zu motivieren. Dies ist besonders in den Fällen von Interesse, wo eine skelettbasierte Anwendung eine höhere Rechenzeit besitzt als eine Realisierung ohne Skelett. Eine solche Motivation läßt sich sehr gut über ein Bonussystem bei der Kostenabrechnung bzw. eine Priorisierung beim eigentlichen Scheduling realisieren. Ist der Betrieb des Parallelrechners als Profit-Center organisiert, so bietet sich ein Rabatt bei den Belegungskosten an. Dabei ist es sinnvoll, den Nachlaß skelettabhängig zu gestalten. Je flexibler und kooperativer eine Anwendung aufgrund ihres Skelettes ist, desto geringere Kosten sind pro Prozessor und Zeiteinheit zu zahlen. Wird für die Nutzung des Parallelrechners nur eine Pauschale berechnet, so ist es sinnvoll, den Anwendungen, je nach Skelett, eine Priorisierung zu geben. Die beiden Bonussysteme können auch integriert werden, wenn für alle Applikationen ein Abrechnungswesen eingeführt wird, bei dem der Benutzer selbst die Prioritätsklasse seiner Anwendung festlegt. Die Kosten, die dem Anwender in Rechnung gestellt werden, lassen sich dann z.B. mittels der Funktion

$$cost_{Anwender}(J_i) = sk_i * prio_i * h_i * P_i$$

definieren, die nur zur Abrechnung dient. Dabei sind h_i und P_i wie bekannt definiert. Der Faktor sk_i mit $0 < sk_i \leq 1$ stellt den Einfluß des benutzten Skelettes dar. Für nicht skelettbasierte Applikationen wird sk_i fest auf 1 gesetzt. Mit dem Faktor $prio_i \in [0.5, 1]$ wird die Priorität der Anwendung festgelegt. Um die Priorisierung auch bei der Berechnung des Schedules einfließen zu lassen, muß dort statt der Funktion

$$cost(J_i) = h_i * P_i$$

die Funktion

$$cost_{prio}(J_i) = prio_i * h_i * P_i$$

benutzt werden.

5.7 Optimierung mittels Back-Filling

Die bislang hier vorgestellten Schedulingverfahren gehen von der Restriktion aus, daß vor Berechnung des Schedules alle Jobs vorliegen. Geht man von der realitätsnahen Annahme aus, daß neue Jobs kontinuierlich generiert werden, so treten Probleme beim Einsatz der vorgestellten Verfahren auf. Die zeitlichen Abstände, in dem die neuen Jobs eintreffen, sowie deren Resourceanforderungen sind a priori unbekannt. Arbeitet das Verfahren mit einer festen Zahl an Jobs pro Schedule, so kann dies zu langen Wartezeiten bis zur Berechnung des Schedules führen. Im worst case wartet der Scheduler unendlich lange auf das Eintreffen eines Jobs, in vielen anderen Fällen ist die Wartezeit länger als die Terminierungszeit des vorhergehenden Schedules. Trifft dies zu, so ergeben sich unnötige Idle-Zeiten des gesamten Parallelrechners. Es ist daher notwendig, die Zahl der Jobs pro Schedule variabel zu

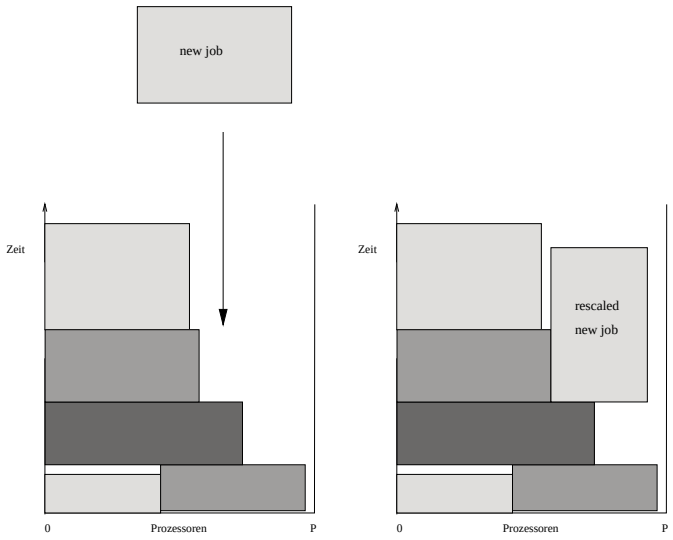


Abbildung 5.14: Back-Filling mit Skalierung

halten und die Terminierung des vorherigen Schedules als Zeit-Kriterium zur Berechnung des neuen Schedules heranzieht. Durch dieses Verfahren können jedoch recht ineffiziente Schedules entstehen, wenn die Anzahl der Jobs pro Schedule sehr klein ist und somit nur wenige Kombinationen für das Schedule möglich sind. Um dies zu vermeiden, ist es sinnvoll, Verfahren hinzuzufügen, die schon in Ausführung befindliche Schedules verbessern können.

Hierzu ist das in Kapitel 2 erwähnte Back-Filling gut geeignet. Trifft während der Ausführung eines Schedules ein neuer Job ein, so wird zuerst versucht,

diesen Job in das aktuelle Schedule zu integrieren. Die Ausführung der einzelnen Applikationen des Schedules bleibt dabei unberührt, da nur momentan ungenutzte Prozessorressourcen an den neuen Job vergeben werden. Weiterhin muß der neue Job spätestens am Ende des alten Schedules die Prozessoren wieder freigeben, da sich ansonsten zusätzliche Wartezeiten bis zum Start des nächsten Schedules ergeben würden.
Ein besonders interessantes Verfahren erhält man, wenn man Back-Filling in Kombination mit skalierbaren Applikationen anwendet. Ist der neu eintreffende Job skalierbar, so läßt sich leichter eine Position im laufenden Schedule finden, an der er integriert werden kann. Steht nur wenig Berechnungs- und Optimierungszeit zur Erstellung des Schedules zur Verfügung, so ist es sinnvoll die Optimierung mittels Reskalierung nur beim Back-Filling einzusetzen. Dadurch kann bei gleichem Berechnungsaufwand, die Qualität der Schedules deutlich verbessert werden.

Kapitel 6

Realisierung eines semi-distributivem Schedulingverfahrens in PCN

6.1 Einführung in PCN

Zur Prototyp–Implementation der Skelette und Schedulingverfahren wurde die parallele Programmiersprache PCN (Program Composition Notation [FOT92, FT91]) gewählt. Sie wurde in Kooperation am Caltech (California Institute of Technologie) und Argonne National Laboratories von Steve Tuecke und Ian Foster entwickelt. PCN ist ein Nachfahre der Programmiersprache STRAND [FT89] und des parallelen Programmiermodells UNITY [CM88]. Wie schon Strand bietet PCN die Möglichkeit, externe C– und Fortran-Funktionen in das Programm zu integrieren. Beiden Sprachen zeichnen sich durch den intensiven Gebrauch von Rekursion aus. Im Gegensatz zu Strand benutzt PCN jedoch keine unstrukturierten logischen Klauseln, sondern einen funktionalen Ansatz in Kombination mit einer C-ähnlichen Syntax fur die Beschreibung eines parallelen Programms. Bei PCN stehen, wie im Namen angedeutet, die Methoden im Vordergrund, mit denen Programmblöcke zu komplexen parallelen Programmen zusammengesetzt werden können [Fos96, Fos94, FK94]. In Abbildung 6.1 ist beispielhaft die Systematik der Programmkomposition illustriert. Für die Realisierung dieser Techniken werden drei Konstruktoren bereitgestellt, die in Abschnitt 6.1.2 noch genauer beschrieben werden.

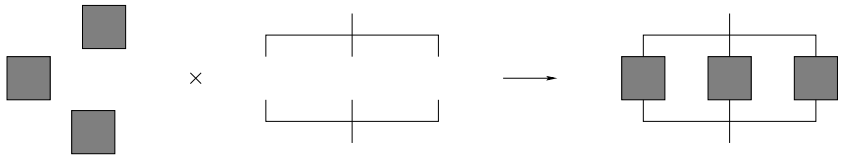


Abbildung 6.1: Komposition von Programmroutinen

6.1.1 Programmierung in PCN

Ein PCN-Programm besteht aus einer Menge von Prozeduren, von denen jede die folgende allgemeine Form besitzen muß ($k, l \geq 0$):

$$\begin{array}{l} name\,(arg_1, \ldots, arg_k) \\ declaration_1, \ldots, declaration_l \\ block \end{array}$$

Ein Block kann aus einer Kombination von Prozeduraufrufen und Zuweisungen bestehen, die mittels Kompositions–Operatoren verknüpft sind. Um Datenabhängigkeiten überschaubar zu halten, erfolgt die Parameterübergabe ausschließlich über den *call-by-value* Mechanismus. Aus demselben Grund sind in PCN Programmen nur lokale Variablen erlaubt. Jedes korrekte Programm besitzt eine Startprozedur *main*, die initial auf dem Prozessor mit Kennung 0 gestartet wird.

6.1.2 Die Basis-Mechanismen

Zur Konstruktion von PCN-Programmen werden die drei Basis-Mechanismen

- parallele Komposition (parallel composition)
- sequentielle Komposition (sequential composition)
- Fallunterscheidung (choice composition)

zur Verfügung gestellt. Durch Verschachtelung dieser drei Techniken lassen sich komplexere Programmstrukturen erzeugen. Eine Komposition besitzt die allgemeine Form

$$\{op \quad block_1, \ldots, block_k\} \qquad\qquad (k > 0)$$

wobei *op* einen der drei Kompositions-Operatoren „ || “ (parallel), „ ; “ (sequential) oder „ ? “ (choice) darstellt. Dieser Operator bestimmt, in welchem Kontext die Blöcke $block_1, \ldots, block_k$ ausgewertet werden.

Parallele Komposition (parallel composition)

Eine parallele Komposition besitzt die Form

$$\{||\quad block_1, \ldots, block_k\}$$

und spezifiziert, daß die angegebenen Blöcke simultan ausgewertet werden sollen (Abbildung 6.2). Die tatsächliche Ausführungsreihenfolge bei sequentieller Abarbeitung wird dabei als beliebig angenommen und kann daher nicht vorhergesagt werden. Eine parallele Komposition terminiert, wenn alle daran beteiligten Blöcke terminiert sind.

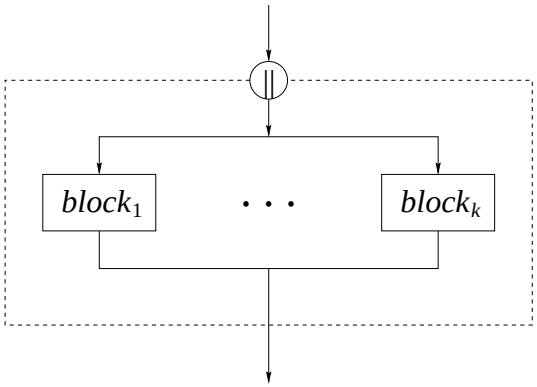


Abbildung 6.2: parallele Komposition

Sequentielle Komposition (sequential composition)

Eine sequentielle Komposition besitzt die Form

$$\{;\quad block_1, \ldots, block_k\}$$

und besagt, daß die angegebenen Blöcke sequentiell in der aufgeführten Reihenfolge auszuwerten sind (Abbildung 6.3).

Fallunterscheidung (choice composition)

Die dritte Art der Zusammensetzung von Programmkomponenten ermöglicht die Auswahl aus einer Menge von Alternativen und besitzt den allgemeinen Aufbau

$$\{?\quad guard_1 {-\!>}\ block_1, \ldots, guard_k {-\!>}\ block_k\}$$

Jeder Ausdruck $guard_i$ besteht aus der Konjunktion boolescher Testabfragen und spezifiziert die notwendigen Bedingungen, die zur Ausführung des

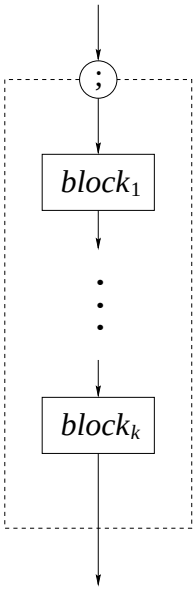


Abbildung 6.3: sequentielle Komposition

dazugehörigen Blocks erfüllt sein müssen. Es wird maximal einer der angegebenen Blöcke ausgeführt, der für den Fall, daß er nicht eindeutig bestimmt ist, nichtdeterministisch ausgewählt wird (Abbildung 6.4).

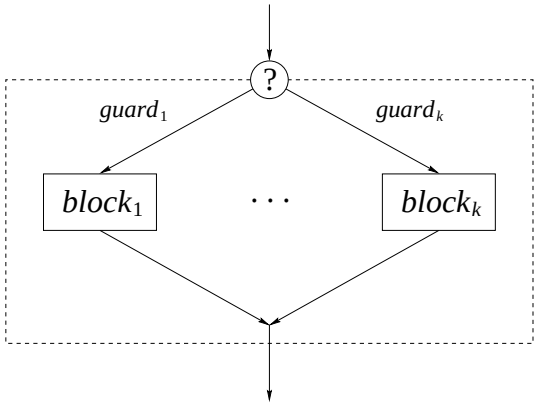


Abbildung 6.4: Fallunterscheidung

6.1.3 Datentypen und Variablen

In PCN existieren die drei einfachen Datentypen: Zeichen (`char`), Ganzzahl (`int`) und Fließkommazahlen mit doppelter Genauigkeit(`double`). Eindi-

mensionale Felder dieser Datentypen werden ebenfalls unterstützt. Weiterhin wird der komplexe Datentyp Tupel (`tuple`) zur Verfügung gestellt. Diese Datenstruktur ist eng verwandt mit speziellen Konstrukten, wie sie z. B. in den Programmiersprachen Prolog, Lisp und Strand verwendet werden. Sie soll aufgrund ihrer besonderen Bedeutung bei der Programmierung in PCN kurz beschrieben werden.

Der Datentyp `tuple`

Ein Tupel besitzt die allgemeine Form

$$\{term_1, \ldots, term_k\} \qquad (k \geq 0)$$

und wird dazu benutzt, komplexe Datenstrukturen $(term_1, \ldots, term_k)$zu konstruieren. Tupel können beliebig ineinander verschachtelt werden und Elemente verschiedener Typen enthalten. Insbesondere kann ein Tupel auch das Null-Tupel ({}) repräsentieren. Einige Beispiele sollen diesen Sachverhalt verdeutlichen:

```
{a, b}      {"abc"}      {}      {6, {7, {}}}      {a, 11.1,
                                                   "xyz"}
```

Einzelne Tupel-Elemente werden in der gleichen Weise wie Array-Elemente referenziert. Der match-Operator „`?=`“ kann dazu benutzt werden, um ein Tupel in seine wesentlichen Bestandteile zu zerlegen:

```
tupel ?= {term_1, ..., term_k}
```

Diese Anweisung testet, ob die Variable `tupel` als k-Tupel deklariert wurde und definiert im positiven Fall den Term `term_i` als Referenz für das i-te Element dieses Tupels. Eine Liste kann in PCN als verschachtelter 2er-Tupel der Form(`{h, t}`) aufgefaßt werden, wobei das erste Element den Kopf und das zweite Element den Rest der Liste darstellt (das Null-Tupel repräsentiert hierbei die leere Liste). Zur Vereinfachung der Schreibweise bietet PCN eine spezielle Listennotation. Die folgenden Darstellungen sind äquivalent:

```
{h, t}          ⇔  [h|t]
{1, {2, {3, {}}}}  ⇔  [1, 2, 3]
{1, {2, {3, tail}}}  ⇔  [1, 2, 3|tail]
```

Innerhalb eines PCN-Programms werden zwei Klassen von Variablen unterschieden. Dieses sind

- veränderliche Variablen (mutable variables)
- single-assignment Variablen (definitional variables)

Veränderliche Variablen können als Variablen des Datentyps `int`, `double` oder `char` deklariert werden und besitzen initial einen unbekannten Wert. Dieser Wert kann während ihrer Lebensdauer beliebig oft durch eine Zuweisung der Form

```
variable := expression
```

verändert werden. Sie werden bei Prozeduren mit imperativem Programmieransatz benötigt. Single-assignment Variablen sind für die Datentypen `int`, `double`, `char` und `tuple` zulässig und werden **nicht** deklariert. Jede Variable innerhalb einer Prozedur, die nicht explizit im Prozedurkopf deklariert wurde, ist somit automatisch eine definitional Variable. Diese Variablen besitzen initial einen speziellen undefinierten Wert und können, nachdem ihnen einmal ein Wert durch eine Anweisung der Form

```
variable = expression
```

zugewiesen wurde, nicht mehr verändert werden. Single-assignment Variablen dienen zur Kommunikation und Synchronisation simultan kooperierender Prozesse (vgl. Abschnitt 6.1.4). Der Datenaustausch zwischen zwei Prozessen erfolgt über das Beschreiben und Auslesen gemeinsamer definitional Variablen. Bei dem Versuch, eine undefinierte single-assignmentVariable zu lesen, wird der dazugehörige Prozeß solange suspendiert, bis der Variablen ein Wert zugewiesen wurde. In Tabelle 6.1 sind die Eigenschaften beider Variablenarten noch einmal zusammengefaßt.

6.1.4 Kommunikation und Synchronisation

Wie bereits in Abschnitt 6.1.3 erwähnt, erfolgt die Prozeß-Kommunikation zwischen simultan ausgeführten Prozessen über gemeinsame Variablen. Dabei ist es unabhängig, auf welchen Prozessoren sich die kommunizierenden Prozesse befinden. Einige Beispiele des Informationsflusses sollen hier kurz erläutert werden.

Single-assignment Variablen als einfache Kommunikationskanäle

Zur Darstellung eines einfachen Informationsaustausches zwischen zwei Prozessen seien die beiden folgenden Prozeduren gegeben, die über die gemeinsame Variable `x` miteinander kommunizieren:

	veränderliche Variable	single-assignment Variable
initialer Wert	willkürlicher Wert	spezieller undefinierter Wert
Zuweisungsoperator	`:=`	`=`
Lese-Operation	immer erfolgreich	wird blockiert bis Variable definiert ist
Anzahl Wertzuweisungen	beliebig viele	eine
unterstützt die parallele Ausführung von Prozessen	nein	ja
wird explizit deklariert	ja	nein
Datentypen	`int`, `double`, `char`	`tuple`, `int`, `double`, `char`

Tabelle 6.1: Gegenüberstellung der Variablen-Arten in PCN

```
producer (x)
{  x = "hello" }
```

```
consumer (x)
{? x == "hello" -> greet (),
   x != "hello" -> ignore ()
}
```

Das Beschreiben der Variablen `x` durch den `producer`-Prozeß übermittelt die Nachricht `"hello"` an den `consumer`. Dieser kann durch einfaches Auslesen der Variablen dann die entsprechenden Aktionen durchführen. Dieses Beispiel macht deutlich, wie single-assignment Variablen zur Kommunikation zwischen Prozessen eingesetzt werden können. Soll eine Kommunikationsverbindung mehr als einen Datensatz übertragen oder ergibt sich eine kontinuierliche Kommunikation (Data Stream), so kann man in PCN auf die aus der parallelen logischen Programmierung bekannte Technik der *Incomplete Messages* zurückgreifen.

```
{|| producer(a),
    consumer(a)
}
```

```
producer (x)
{||  x = [ "hello"| newx ] ,
producer (newx)}

consumer (y)
```

```
{?    y ?= [ "hello"| newy ] -> { ;greet(),
                                  consumer(newy) }
}
```

In diesem Beispiel sendet der Producer in einer endlosen Schleife Nachrichten über die Variable `a` an den Consumer. Jede Nachricht besteht aus der eigentlichen Botschaft "hello"und einer neuen Kommunikationsvariablen `newy`, mit der sich der Producer rekursiv aufruft. Da die Variable `newy` zu diesem Zeitpunkt noch nicht instanziert ist, spricht man von einer *Incomplete Message*. Der Consumer wartet bis er eine Nachricht erhält, die er in die ursprüngliche Botschaft und eine neue Kommunikationsvariable zerlegen kann und ruft sich mit dieser erneut auf.

6.1.5 Prozeß-Mapping

Die Ausführung eines parallelen Algorithmus auf einem Parallelrechner umfaßt die Zuordnung von einzelnen Teilaufgaben an die Prozessoren(Mapping). PCN verlangt vom Programmierer die Angabe des gewünschten Prozessorknotens. Wird keine Mappingdirektive angegeben, so plaziert PCN den neuen Prozeß auf dem Vaterknoten. PCN bietet das Konzept der *virtuellen Topologien* zur Unterstützung des Mappings. Eine virtuelle Topologie besteht aus mehreren Prozessoren (Knoten) und einer virtuellen Verbindungsstruktur, die die Organisation dieser Knoten widerspiegelt (Ring, Array, Gitter etc.). Die Zuweisung eines Prozesses an einen bestimmten Knoten innerhalb der Topologie muß explizit durch die Verwendung des Operators `@` erfolgen. Ein Prozeß kann dabei absolut durch Angabe der Prozessornummer bzw. relativ zur Position des Vaterprozesses erfolgen. Für die relative Positionierung auf einem zweidimensionalen Gitter stehen z.B. die Plazierungsdirektiven *north*, *south*, *east* und *west* zur Verfügung. Die Anwendung des Prozess-Mappings und den Aufbau von Kommunikationsstrukturen verdeutlichen die folgenden Beispiele. Das erste Beispiel zeigt die Komposition eines Prozessorring, bei dem jeder Prozeß auf einem neuen Prozessorknoten gemappt wird. Das zwei-

```
{|| prozess1(a1,a2)@vts:node(1);
   prozess2(a2,a3)@vts:node(2);
   prozess3(a3,a4)@vts:node(3);
   prozess4(a4,a1)@vts:node(4);
}
```

Tabelle 6.2: Definition eines Prozessorrings

te Beispiel zeigt, wie mittels Definitional-Variablen und des Prozeß-Mappings rekursiv ein Ring mit `n` Worker-Prozessen aufgespannt werden kann. Der Prozeß `buildring` wird solange rekursiv aufgerufen, bis `n` auf 1 heruntergezählt wurde. Bei jedem Aufruf wird ein Prozeß `worker` gestartet, dessen zwei Definitional-Variablen die Kommunikationskanten darstellen. Der letzte `worker` , der auf Prozessor 1 gestartet wird, schließt dann den Ring, indem er eine Verbindung mit dem ersten `worker` herstellt.

```
buildring (n, ringstart, ringend)
        {?
            n > 1− >
                { ||
                worker(ringstart,nextworker),
                buildring(n-1, nextworker, ringend)@vts:node(n)
                },
            n ==1 − >
                worker(ringstart,ringend)
        }
```

Tabelle 6.3: rekursive Ringdefinition

Metacalls

Die Möglichkeit zur sog. *higher-order Programmierung* wird in PCN mittels sogenannter *Metacalls* unterstützt. Innerhalb einer Prozedur können hierzu Funktionen durch Stringvariablen präsentiert werden. Diesen Variablen werden beim Prozeduraufruf (Metacall) konkrete Funktionsnamen zugeordnet. Mit diesem Konstrukt können z.B. benutzerdefinierte Funktionen an einen in PCN realisierten Skelettrahmen übergeben werden.

Wiederverwendung von Code-Segmenten

Durch das Einbinden von „fremden“ Code-Segmenten (Fortran, C) in PCN-Programme, ist es möglich, bereits existierende Techniken aus dem Bereich der sequentiellen Software-Entwicklung in die Programmerstellung miteinzubeziehen. Komplexere Applikationen können unter Verwendung von wiederverwendbaren parallelen Strukturen (*software cells*, *templates* [Fos92]) auf einfache Weise modular zusammengesetzt werden.

6.1.6 Das Ausführungsmodell von PCN

Das Ausführungsmodell von PCN basiert auf dem Konzept kommunizierender abstrakter Maschinen. Jede abstrakte PCN-Maschine besitzt eine eigene Kommunikations- und Schedulingkomponente um PCN-Prozesse abzuarbeiten. Dadurch ist es möglich, auch parallele Hardwareplattformen zu benutzen, die kein explizites Scheduling (z.B. T800 Transputer) unterstützen. Weiterhin bietet das Konzept der abstrakten Maschine die Möglichkeit, PCN leicht auf andere Hardwareplattformen zu portieren, da einzig die abstrakte Maschine sowie die Kommunikation zwischen einzelnen Maschinen realisiert werden müssen. Alle anderen Komponenten wie Compiler oder Debugger können weiterbenutzt werden, da der von ihnen benutzte Zwischencode universell ist. Im Vorfeld dieser Arbeit wurden Implementationen von PCN auf verschiedenen Transputersystemen realisiert. Als Betriebssystem wurde dabei PARIX verwendet.

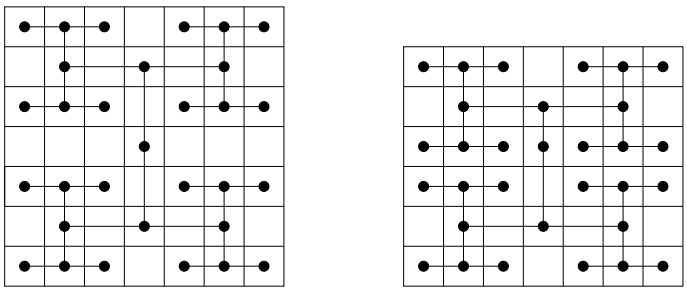
6.2 Implementierung der Skelette

6.2.1 Das d&c-Skelett

Im Rahmen einer Diplomarbeit [Ste96] wurden verschiedene Implementierungen für d&c-Skelette mit unterschiedlichem Verzweigungsgrad k in PCN realisiert und ihre Eigenschaften untersucht. Bei der Implementierung wurde ein Prozessorgitter zugrunde gelegt, in das mit verschiedenen Mappingmethoden der d&c-Prozeßbaum eingebettet wurde. An dieser Stelle soll exemplarisch auf die Implementierung eines d&c-Skeletts mit dem Verzweigungsgrad k=2 mittels H-Baum Einbettung eingegangen werden. Dabei wird gezeigt, wie sich ein H-Baum effizient auf verschiedene Gitterstrukturen abbilden läßt.

H-Baum Einbettung

Diese Form der Einbettung stellt eine Methode dar, um einen Binärbaum auf ein 2-dimensionales Gitter abzubilden. Das dabei entstehende fraktale H-förmige Erscheinungsbild besitzt eine regelmäßige Struktur, wobei die Pfade zwischen den Knoten einer Ebene im Binärbaum und deren direkten Vorgängern alle die gleiche Länge besitzen. Die H-Baum Einbettung hat den Nachteil, daß bei kleinen Gittern, die Ausnutzung der zugeteilten Prozessoren (Abbildung 6.5 (a)) nicht optimal ist. Dieses Verhalten kann durch die optimierte H-Baum Einbettung (Abbildung 6.5 (b)) deutlich verbessert werden. Die Berechnungsformeln für den H-Baum und den optimierten H-Baum finden sich im Anhang A.2.



(a) (b)

Abbildung 6.5: H-Baum Einbettung einfach (a) und optimiert (b)

Realisierung der Einbettungen

Zur Einbettung des H-Baums ist es notwendig, die Kennungen der Prozessoren festzulegen, auf denen die Knoten des Berechnungsbaumes gemappt werden sollen. Diese Information kann entweder zum Zeitpunkt des Mappings berechnet oder der Skelettimplementierung als zusätzlicher Parameter mitgegeben werden. Bei der Implementierung wurden die folgenden Varianten realisiert:

- Explizite Berechnung
  Bei der expliziten Berechnung (siehe Anhang) wird auf Basis der aktuellen Prozessorkennung und der Gitterdimensionen das Mapping des nächsten Berechnungsknotens direkt ermittelt.

- Permutationsliste
  Bei diesem Mapping erhält die Implementierung des Skeletts beim Aufruf eine Liste, die eine Permutation der zugeteilten Prozessorkennungen enthält. Die entsprechenden Listen mit den Permutationen für die verschiedenen Prozessormengen sind hierzu vorab ermittelt worden. Sie stellen Kodierungen von optimierten H-Baumeinbettungen dar. Die ersten k-Einträge der Liste bestimmen, an welche Prozessoren die erzeugten Teilprobleme übergeben werden sollen. Die Restliste wird dann in k ungefähr gleichgroße Teillisten zerlegt und an die jeweiligen Prozessoren übergeben. Die Permutation der Prozessorkennungen wird dabei so konstruiert, daß das resultierende Mapping dasselbe Verhalten wie die explizite Berechnung zeigt. Abbildung 6.6 zeigt dieses Verfahren an einem Beispiel mit k=2. Die Erzeugung einer Permutationsliste erfolgt invers zur Abarbeitung im Skelett. Von den Prozessorkennungen der Blätter aus werden iterativ Teillisten der Sohnknoten erzeugt. An jedem Vaterknoten werden die entsprechenden Teillisten gemischt und die Kennungen der Söhne in die neue Liste vorne eingefügt. Das Konstruktionsverfahren läßt sich an Abbildung 6.6 gut nachvollziehen.

- lol-Struktur
  Dieses Verfahren benutzt eine ähnliche Struktur, wie der Permutations-Ansatz. Es wird jedoch zum Mapping statt einer einfachen Liste , eine verschachtelte Listenstruktur benutzt, die aus den zugewiesenen Prozessorkennungen und Trennsymbolen erzeugt wird. Diese lol-Struktur (*list-of-lists*) besitzt den Aufbau $[n_1, n_2, l_1, l_2]$, wobei n_i $(i = 1, 2)$ die Kennung des Prozessors darstellt, dem das i-te Teilproblem zugeordnet wird und l_i $(i = 1, 2)$ die rekursiv definierte lol-Struktur repräsentiert,

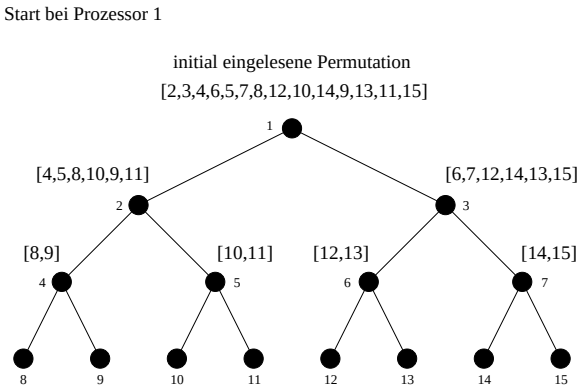


Abbildung 6.6: Einbettung über eine Permutation

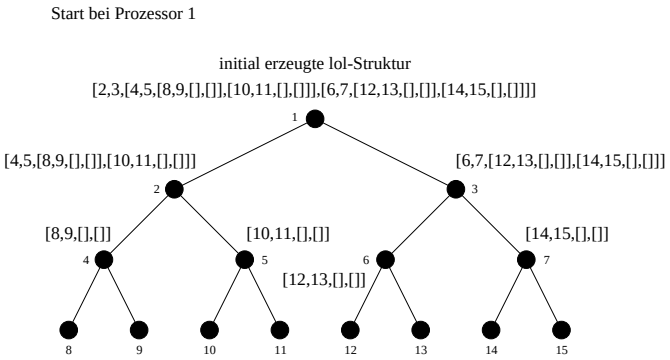


Abbildung 6.7: Einbettung über eine lol-Struktur

die dem Prozessor n_i für das weitere Prozeß-Mapping übergeben wird (Abbildung 6.7). Dieser Ansatz erspart bei einem Divide-Schritt das explizite Erzeugen der Teillisten, erfordert aber einen gewissen Mehraufwand bei der Erstellung der lol-Struktur.

Die Konstruktion der lol-Struktur erfolgt ähnlich der Konstruktion einer Permutationsliste. Ausgehend von den Blattknoten werden wieder iterativ Listenstrukturen erzeugt. Jeder Knoten erhält hierzu die Listen seiner Sohnknoten. Besitzt ein Knoten keine Söhne, so werden stattdessen 2 leere Listen an den Vaterknoten weitergegeben. Für jeden Vaterknoten wird eine neue Liste erzeugt, die als Elemente die Nummern seiner Söhne und deren schon erzeugte Listen enthält.

Der Einsatz von Permutationslisten oder lol-Strukturen bietet den Vorteil, daß so auf einfache Weise auch entartete H-Baum Einbettungen (Abbildung 6.8) realisiert werden können, die sich nicht mehr rekursiv berechnen lassen.

Zu einer vorgegebenen Prozessorzahl kann für jede mögliche Gitterstruktur ein Permutationsliste ermittelt werden, die eine optimierte Einbettung eines entarteten H-Baums beinhaltet. Benutzt man die entsprechende Permutationsliste für die vom Scheduler zugewiesene Prozessormenge, so erhält man automatisch eine effiziente Ausnutzung der Ressourcen. Abbildung 6.9 und 6.10 zeigen am Beispiel von n=16 Prozessoren, die Permutationslisten für die möglichen Prozessorgitter.

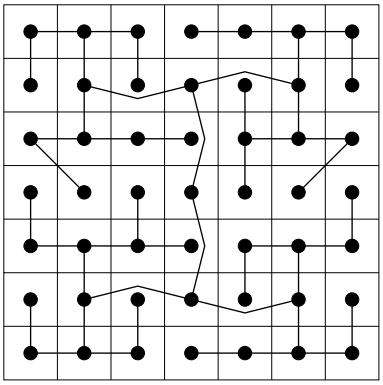


Abbildung 6.8: entarteter H-Baum

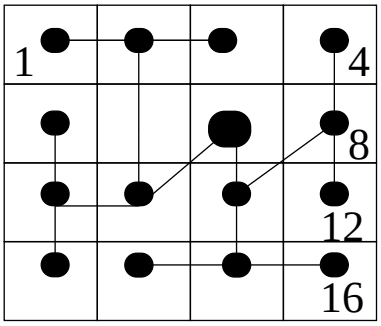


Abbildung 6.9: Permutation(10,11,2,15,9,8,1,14,5,4,3,16,13,12), 4 x 4 Gitter

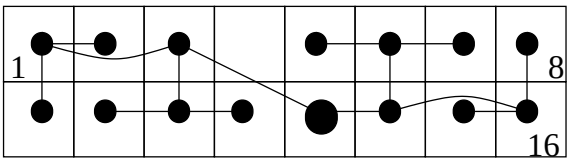


Abbildung 6.10: Permutation (3,14,1,6,11,15,2,5,10,14,9,7,12,16), 2 x 8 Gitter

6.2.2 Das *ic*-Skelett

Neben dem d&c-Skelett wurde auch das ic-Skelett im Rahmen der oben angegebenen Diplomarbeit betrachtet. Wiederum wurden verschiedene parallele Implementierungen dieses Skelettes in PCN realisiert, um je nach Anwendungsfall die bestgeeignete Implementierung nutzen zu können. Als Grundlage für die interne Kommunikation dient - wie schon in Abschnitt 5.2 erwähnt - eine Ringstruktur. Diese bietet den Vorteil, sich ohne Probleme in verschiedene Prozessorgitter einbetten zu lassen. Abbildung 6.11 verdeutlicht das an Beispielen.

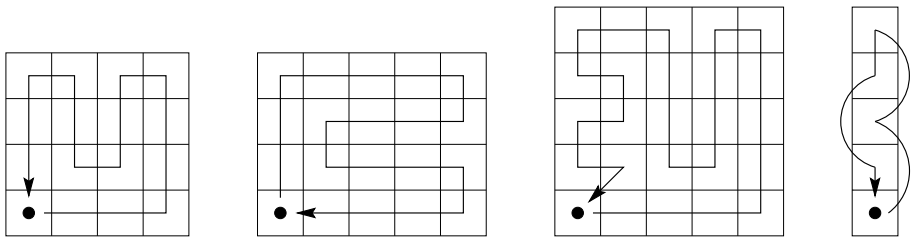


Abbildung 6.11: Bestimmung des Ringes bei verschiedenen Gitterdimensionen

Implementierung

Die eingesetzten Algorithmen gehen davon, daß jeder Prozessor im Ring initial genau ein Objekt besitzt. Daher muß zu Beginn der Berechnung eine Ringstruktur aufgebaut werden , bei der

$$\textit{Anzahl Objekte} = \textit{(Anzahl Prozessoren)}$$

gilt. Da diese Bedingung nicht immer erfüllt werden kann, sei es, weil die Anzahl der Objekte sehr groß ist oder der Scheduler weniger als die gewünschten Prozessoren zuteilt, wird hierzu zuerst ein virtueller Prozessorring aufgebaut, bei dem die obige Bedingung erfüllt ist. Dieses Vorgehen wird von PCN durch die Tatsache unterstützt, daß die Kommunikation zwischen Prozessen - und als solche kann man die virtuellen Prozessoren sehen - vollkommen unabhängig von der Plazierung auf den realen Prozessoren ist (siehe auch 6.1.4). Der virtuelle Ring wird dann segmentweise auf die zugeteilte Prozessormenge gemappt, sodaß die Kommunikationskanten zwischen den einzelnen Segmenten einen physikalischen Ring bilden (Abbildung 6.12).

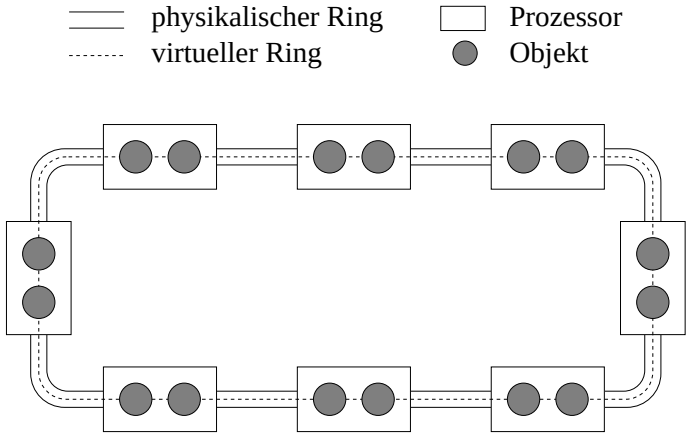


Abbildung 6.12: physikalischer und virtueller Ring

Test&Select-Phase In der Test&Select-Phase erstellt jeder Prozessor zunächst eine Kopie des eigenen Objekts. Diese Kopien werden dann in festgelegter Reihenfolge einmal durch den Ring geschickt. Während eines Durchlaufs kann jeder Prozessor sein Objekt mit allen anderen Objekten vergleichen und den besten Partner ermitteln. Jeder Prozessor merkt sich dabei immer die Kennung des aktuell besten Partners und hat am Ende des Ringdurchlaufs seinen besten Partner gefunden.

Combine-Phase Zum Ende eines Iterationschritts muß jedes Objekt mit seinem besten Partner vereinigt werden. Der Single-Tour Algorithmus realisiert dies in einem weiteren Ringdurchlauf. Jeder Prozessor erstellt hierzu ein Datenobjekt (Abbildung 6.13), das neben der eigentlichen Objektbeschreibung noch zwei weitere Informationsfelder enthält, die während des Ringdurchlaufs dynamisch verändert werden.

Objekt-beschreibung	Feld 1 (*was wurde schon mit dem Objekt vereinigt*)	Feld 2 (*was muß noch mit dem Objekt vereinigt werden*)

Abbildung 6.13: Informationsobjekt für den Single-Tour Algorithmus

Das erste Feld beschreibt, welche Objekte schon mit dem lokalen Objekt vereinigt worden sind. Das zweite Feld enthält Informationen über die Objekte, die noch mit dem lokalen Objekt vereinigt werden müssen. Wie bei

der Test&Select-Phase behält jeder Prozessor seine originale Objektbeschreibung und schickt das neu erstellte Datenobjekt durch den Ring. Bei jedem Schritt des Ringdurchlaufs wird von jedem Prozessor lokal getestet, ob er ein Objekt besitzt, das im Feld 2 vermerkt ist. Ist dies der Fall, so vereinigt der Prozessor das zugeschickte Objekt mit seinem lokalen Objekt, ändert entsprechend Feld 1 und Feld 2 und sendet eine Kopie des neuen Datenobjekts an den nächsten Prozessorknoten. Schlägt der Test Fehl, so wird das Datenobjekt unverändert im Ring weitergeschoben. Der Single Tour Algorithmus garantiert, daß am Ende des Ringdurchlaufs jedes zu einem Objekt korrespondierende Paket genau die Menge von Objekten beschreibt, mit denen das lokale Objekt zu einem neuen Objekt vereinigt wurde. Der Beweis hierzu findet sich in [Ste96]
Das folgende Beispiel veranschaulicht die Arbeitsweise. Als Grundlage dient ein Prozessorring bestehend aus den Prozessoren 1 bis 3. Initial befindet sich Objekt `a` auf Prozessor 1, Objekt `b` auf Prozessor 2 und Objekt `c` auf Prozessor 3. In der Test&Select-Phase wurden folgende beste Partner gefunden: `a` hat den Partner `b`, `b` hat den Partner `c` und `c` hat den Partner `a`. Der Ablauf der Combine-Phase wird in Tabelle 6.4 beschrieben. Man erkennt, daß nach einem kompletten Durchlauf drei Kopien des kombinierten Objekts `(a,b,c)` entstanden sind. Jeder Prozessor überprüft jetzt, ob sich in dem zusammengesetzten Objekt ein Teilobjekt befindet, das vor Ablauf der Combine-Runde auf einem Prozessor mit niedriger Prozessornummer gespeichert war. Trifft dies zu, so löscht der Prozessor seine lokale Kopie des zusammengesetzten Objekts. Somit verbleibt im Beispiel einzig die Kopie auf Prozessor 1. Die Berechnung ist damit abgeschlossen.
Der Single-Tour Algorithmus besitzt die positive Eigenschaft, daß nur ein weiterer Durchlauf zur Durchführung der Combine-Phase benötigt wird. Die Größe der Datenobjekte nimmt im Laufe der Rundreise durch den Ring eine stetig zu. Falls dies bei einer konkreten Anwendung zu Kommunikationsproblemen führen sollte, kann alternativ der in [Ste96] aufgeführte 4-Tour Algorithmus genutzt werden.

Internes Remapping

Ein Merkmal der iterativen Vereinigung ist die stetige Verringerung der Objektanzahl. Daher ergibt sich während der Berechnung zwangsläufig die Situation, daß die Zahl der Objekte die Zahl der Prozessoren unterschreitet. Es existieren ab diesem Zeitpunkt Prozessoren im Ring, die nicht mehr mit der iterativen Kombination, sondern einzig mit dem Weiterleiten von Objekten befaßt sind. Dies kann einen erheblichen Einfluß auf die Rechenzeit haben. Es ist daher sinnvoll, diese Prozessoren aus dem Ring zu entfer-

Objekt, (Position)	vereinigte Objekte	noch zu vereinigende Objekte	aktuelle Position	Berechnungs-schritt
a(1)	-	b	1	
b(2)	-	c	2	0
c(3)	-	a	3	
a(1)	a(1),b(2)	c	2	
b(2)	b(2),c(3)	a	3	1
c(3)	c(3),a(1)	b	1	
a(1)	a(1),b(2),c(3)	-	3	
b(2)	b(2),c(3),a(1)	-	1	2
c(3)	c(3),a(1),b(2)	-	2	
a(1)	a(1),b(2),c(3)	-	1	
b(2)	b(2),c(3),a(1)	-	2	3
c(3)	c(3),a(1),b(2)	-	3	

Tabelle 6.4: Ablauf einer Combine-Phase

nen und anderen Applikationen zur Verfügung zu stellen. Hierzu wird eine Restrukturierung des Prozessorrings durchgeführt, die immer dann durchgeführt wird, wenn die Bedingung

$$\textit{Anzahl Objekte} \leq \tfrac{1}{2}\ \textit{(Anzahl Prozessoren)}$$

erfüllt ist. Die freigegebenen Prozessoren werden dann von der Skelettimplementierung in eine Prozessorfreigabeliste eingetragen.

6.2.3 Das Farming-Skelett

Auch hier wurde versucht, das Farming-Skelett möglichst effizient auf ein beliebiges Prozessorgitter zu mappen. Erneut wird eine Permutationsliste mit Prozessorkennungen eingesetzt, die auf jedes Prozessorgitter abgestimmt ist. Die Kommunikationstopologie zwischen Master und Workern wird völlig losgelöst vom Mapping aufgebaut. Der Master der Farm erhält dazu als Eingabe eine Liste der Prozessorkennungen der Worker. Er startet zu Beginn den ersten Worker auf dem ersten Knoten aus der Prozessorliste und übergibt ihm die Restliste und eine Definitional-Variable als Kommunikationskanal zum Master . Der Workerprozeß entfernt wiederum den nächsten Eintrag aus der Prozessorliste und startet auf dem entsprechenden Prozessor den nächsten Workerprozeß und übergibt wieder Kommunikationskanal und Restliste an den neuen Worker. Dieser Vorgang wird solange wiederholt bis

die Prozessorliste leer ist und alle Worker inititialisiert sind. Jeder Worker ist nun mit dem Master verbunden und signalisiert dies, indem er eine Arbeitsanfrage an den Master schickt. Die Anfragen aller Worker werden mittels des merge-Konstruktors von PCN zu einer Anfrageschlange zusammengefaßt und vom Master der Reihe nach abgearbeitet. Zur Kommunikation stehen dem Master die folgenden Konstrukte zur Verfügung:

- `["problem ", work ]`

- `[ "terminate "]`
  Liegen keine offenen Teilprobleme mehr vor, so erhält ein Worker auf seine Arbeitsanfrage diese Nachricht. Sie veranlaßt ihn seine Arbeit einzustellen und den Master darüber zu informieren.

Der Worker nutzt zur Kommunikation die folgenden Botschaften:

- `[ "workrequest ", work]`
  Die erste Komponente in Form eines Strings kennzeichet die Botschaft als Arbeitsanfrage, die zweite Komponente stellt eine Definitionalvariable dar, die dem Master als Kommunikationsweg zum Worker dient und mit der Zuweisung `work = subproblem` instanziert wird.

- `[ "terminated ", procnumber]`
  Dieses Tupel wird vom Worker benutzt, um dem Master zu signalisieren, daß der Worker seine Abarbeitung beendet hat und aus der Farm entfernt werden kann. Die Definitionalvariable `procnumber` enthält die Prozessorkennung des Workers. Sie wird vom Master benötigt, um den Prozessor für andere Applikationen wieder frei zu geben. Erhält der Master die obige Nachricht, so trägt er die Prozessorkennung in die Prozessorfreigabeliste ein.

Integration von Prozessoren während der Berechnung

Die oben geschilderte Initialisierungsroutine ermöglicht es, auch während der Berechnung weitere Prozessoren in die Farm zu integrieren. Dies ist der Fall, wenn die Liste der Prozessorkennungen nicht abgeschlossen ist, sie also die Form `[p1,p2, ... pn|prest]` besitzt. Es seien `p1` bis `pn` schon mit Prozessornummern belegt, die Restliste `prest` jedoch eine nicht instanzierte Definitional Variable. In diesem Fall kann der Worker auf Prozessor `pn` keinen weiteren Worker mehr starten, da `prest` noch undefiniert ist. Der

Initialisierungsprozeß wird nun suspendiert und wieder gestartet, sobald ein neuer Eintrag in die Prozessorliste erfolgt. Da der Initialisierungsprozeß nebenläufig zu den anderen Prozessen des Workers ist, ergibt sich durch das Warten keine Verzögerung der anderen Aufgaben des Workers.

Freigabe von Prozessoren

Wird ein Worker nicht mehr benötigt, weil keine weiteren Teilprobleme mehr vorhanden sind, so gibt der Master diesen wieder frei. Er besitzt dafür eine Freigabeliste, in die er die Kennungen der freigewordenen Prozessoren einträgt. Eine vorzeitige Entfernung von Prozessoren aus der laufenden Berechnung ist nicht vorgesehen, da beim eingesetzten Schedulingverfahren nur Prozessoren freigegeben werden müssen , die von der Applikation nicht mehr benötigt werden. Dieses Feature kann jedoch einfach in die Skelettapplikation integriert werden. Hierzu ist es notwendig, daß jeder Workerprozeß bei einer Arbeitsanfrage zusätzlich seine Prozessorkennung verschickt. Über diese Kennung ermittelt der Master dann bei einer Anfrage, ob der Prozessor freigegeben werden soll. Trifft dies zu, so erhält der Worker statt eines Teilproblems eine Terminierungsbotschaft. Nachdem der Worker dann die erfolgreiche Terminierung gemeldet hat, kann der Master seine Freigabe über die Prozessorfreigabeliste melden.

6.2.4 Das Fixed-Sized Skelett

Um nicht skelettbasierte Applikationen abarbeiten zu können, ist es notwendig, sie in den Dummy-Skelettrahmen *Fixed-Sized* zu kapseln. Um dies zu ermöglichen, müssen jedoch die folgenden zum Teil technischen Bedingungen erfüllt werden:

- Das parallele Programm muß in PCN realisiert sein. Diese Restriktion ist notwendig, um eine Applikation in die PCN–Laufzeitumgebung einbetten zu können.
- Das Programm darf nur eine direkte Plazierung bei der Generierung von neuen Prozessen benutzen. Nummer bzw. der Name des Prozessors müssen direkt angegeben werden. Da dem Programmierer nicht bekannt ist, welche genaue Topologie die später zugeteilte Prozessorpartition besitzen wird, ist eine relative Plazierung eines Prozesses nicht möglich. Unter einer relativen Plazierung verstehen wir hier eine Anweisung wie "Starte Prozeß X auf dem rechten Nachbarn von Prozessor Y ".

Erfüllt eine Applikation diese Bedingungen, so kann sie automatisch in das Dummy–Skelett f*ixed–sized* eingebettet werden. Bei der Einbettung ist es notwendig, die Prozessorplazierung auf ein listenbasiertes Modell umzustellen. Jede Applikation erhält vom Scheduler eine Liste der Prozessornummern, die der Anwendung zugeteilt worden sind. Diese Liste muß von der Applikation selbstständig ausgewertet und die Prozessornummern den einzelnen Prozeßaufrufen zugeordnet werden. Bei Terminierung der Applikation muß die Liste der freien Prozessoren wieder übergeben werden. Die Programmtransformationen werden automatisch während der Einbettung durchgeführt. Die dazu notwendigen Schritte lassen sich gut an dem folgenden kleinen Beispiel demonstrieren:

```
fixed_sized()
{||  prozess1@vts:node(1);
      prozess2@vts:node(2);
      prozess3@vts:node(3);
      prozess4@vts:node(4);
}
```

Das Programm besitzt vier parallel auszuführende Prozesse. Auf Eingabe und Ausgabe, sowie einer genauen Spezifikation der Prozesse soll hier aus Gründen der Übersichtlichkeit verzichtet werden. Die folgenden Transformationen werden nun nacheinander durchgeführt:

1. Ersetze alle Prozessornummern durch neue Variablen:

   ```
   fixed_sized()
   {||  prozess1@vts:node(a);
         prozess2@vts:node(b);
         prozess3@vts:node(c);
         prozess4@vts:node(d);
   }
   ```

   Alle festen Prozeßplazierungen werden in variable Plazierungen umgewandelt. Hierzu werden die Definitionalvariablen a bis d eingeführt. Sie erhalten später die Werte aus der Prozessorliste.

2. Füge die Prozessorliste als zusätzlichen Übergabeparameter ein und werte die Liste aus.

```
fixed_sized(proc-in)
{? proc-in ?=[a,b,c,d] ->
{||
prozess1@vts:node(a);
prozess2@vts:node(b);
prozess3@vts:node(c);
prozess4@vts:node(d);
}
}
```

Das ursprüngliche Programm erhält als Übergabeparameter eine Definitionalvariable, die später vom Scheduler mit der Prozessorliste instanziert wird. Mit dem Choice-Operator *?* wird die Ausführung solange suspendiert, bis mittels des Match-Operators *?=* der Inhalt von *proc-in* in den Variablen *a* bis *d* zugewiesen werden kann.

3. Gib die freigewordenen Prozessoren nach Terminierung als Prozessorliste zurück.

```
fixed_sized(proc-in,proc-out)
{? proc-in ?=[a,b,c,d] -> {;
                 {||
                  prozess1@vts:node(a);
                  prozess2@vts:node(b);
                  prozess3@vts:node(c);
                  prozess4@vts:node(d);
                   },
                  proc-out=[a,b,c,d]}
```

Analog zur Abarbeitung der Eingangsliste wird aus den 4 Prozessornummern eine Liste erzeugt, die der Definitional-Variable *proc-out* zugewiesen wird. Der Sequenzoperator *;* sorgt dafür, daß dies erst geschieht, wenn *prozess1* bis *prozess4* abgearbeitet worden sind.

6.3 Wahl des Scheduling Verfahrens

Bei der Realisierung eines Schedulingverfahrens hat man die Wahl zwischen einer verteilten und einer globalen Implementation des Verfahrens. Dabei sind die folgenden Vor- und Nachteile gegeneinander abzuwägen:

- Globale Verfahren:
  Hier liegt die Kontrolle bei einem Master-Scheduler. Alle notwendigen Informationen sind zentral an einer Stelle verfügbar. Dadurch wird die Umsetzung des Schedulingalgorithmus erleichtert. Problematisch wird ein solcher Ansatz, wenn die Kommunikationsbandbreite zum zentralen Scheduler zu gering ist, oder die Berechnung und Kontrolle des Schedules die Rechenkapazität des Prozessorknotens übersteigt. In diesen Fällen ist ein zentraler Ansatz nicht geeignet.

- Verteilte Verfahren:
  Eine Distribution des Schedulingverfahrens auf eine Teilmenge oder die Gesamtmenge der Prozessoren vermeidet lokale Überlastungen im Kommunikationsbereich und bei Rechenlast einzelner Prozessoren. Als nachteilig erweist sich jedoch oft die Notwendigkeit, einen intensiven Austausch von Schedulinginformationen zwischen den einzelnen Prozessoren zu realisieren. Verteilte Implementationen bieten somit nur einen Vorteil, wenn die einzelnen Prozessoren zur Berechnung des Schedules möglichst nur lokale Informationen benötigen.

Kombiniert man globale mit verteilten Verfahren, so ist es möglich, die Vorteile beider Ansätze zu nutzen, ohne deren Nachteile in vollem Umfang in Kauf nehmen zu müssen. Hierzu kann der Schedulingvorgang in zwei Phasen zu unterteilt werden:

1. **Die initiale Berechnung des Schedules:**
   Während der Berechnung ist es sinnvoll, die komplette Information über das bisher berechnete Schedule, sowie die Informationen über die abzuarbeiteten Applikationen zentral zur Verfügung zu haben. Da die im Rahmen der Arbeit benutzten Verfahren zur Berechnung des initialen Schedules auf sequentiellen Algorithmen basieren, bietet sich für diese Phase ein zentraler Ansatz an.

2. **Die Abarbeitung des Schedules:**
   Zu Beginn der Abarbeitung eines Schedules ist für jede Applikation die

Prozessor und Zeitzuteilung bekannt. Zudem sind im initialen Schedule zu jeder Applikation eindeutige Vorgänger und Nachfolger berechnet worden. Alle Informationen, die zum korrekten Ablauf des Schedules benötigt werden, sind damit entweder statisch vorhanden (z.B. Prozessormenge, Startzeit) oder lassen sich dynamisch aus lokalen Informationen (z.B. vorzeitige Freigabe von Prozessoren) generieren. Einzig die Information über die Systemzeit muß mittels eines geeigneten Verfahrens (z.B. broadcast) während der Bearbeitung des Schedules propagiert werden. Für die Ablaufkontrolle eines Schedules bietet sich somit ein verteilter Ansatz an.

Die obigen Überlegungen führten zur Entwicklung eines zweiphasigen Schedulingverfahrens, das in den folgenden Abschnitten weiter beschrieben wird.

6.4 nichtpreemtives 2-Phasen Scheduling

Das hier vorgestellte Verfahren verlagert die Ablaufsteuerung des Schedules in die daran beteiligten parallelen Programme. Jede Applikation ist also selbst für die Kontrolle ihrer benötigten Systemressourcen verantwortlich. Die notwendigen Funktionalitäten werden von den zur Modellierung gewählten Skeletten zur Verfügung gestellt. Eine skelettbasierte Funktion entscheidet selbstständig über den Start und die Terminierung ihrer eigenen Berechnung. Die hier vorgestellte Realisierung eines zweistufigen Verfahrens arbeitet nicht preemptiv. Diese Einschränkung wurde nur aufgrund der beschränkten Systemressourcen (Hauptspeicher des Transputers) getroffen. Ein preemtives Verhalten, wie es z.B. der PBI Algorithmus fordert, kann durch Suspendierung einer Applikation bei Überschreiten einer gegebenen Zeitmarke realisiert werden. Hierauf wird in einem späteren Abschnitt noch kurz eingegangen.

6.4.1 Initiales Scheduling

In der initialen Phase des Schedulings, werden alle statisch verfügbaren Informationen ausgewertet und ein erstes vorläufiges Schedule für die vorhandenen Anwendungen berechnet. Die einzelnen parallelen Applikationen müssen dabei skelettbasiert sein. Nicht skelttbasierte Applikationen können, wie schon in Abschnitt 6.2.4 geschildert, mittels des fixed-sized-Skeletts in eine skelettbasierte Applikation transformiert werden. Ziel dieses ersten Schedules ist es, den Ansprüchen an Rechenzeit und Prozessorbedarf zu genügen. Dazu wird der Shelf–Scheduling Algorithmus von Schwiegelshohn benutzt. Als Kostenfunktion dient, wie schon in Kapitel 3 vorgestellt, die gewichtete Summe

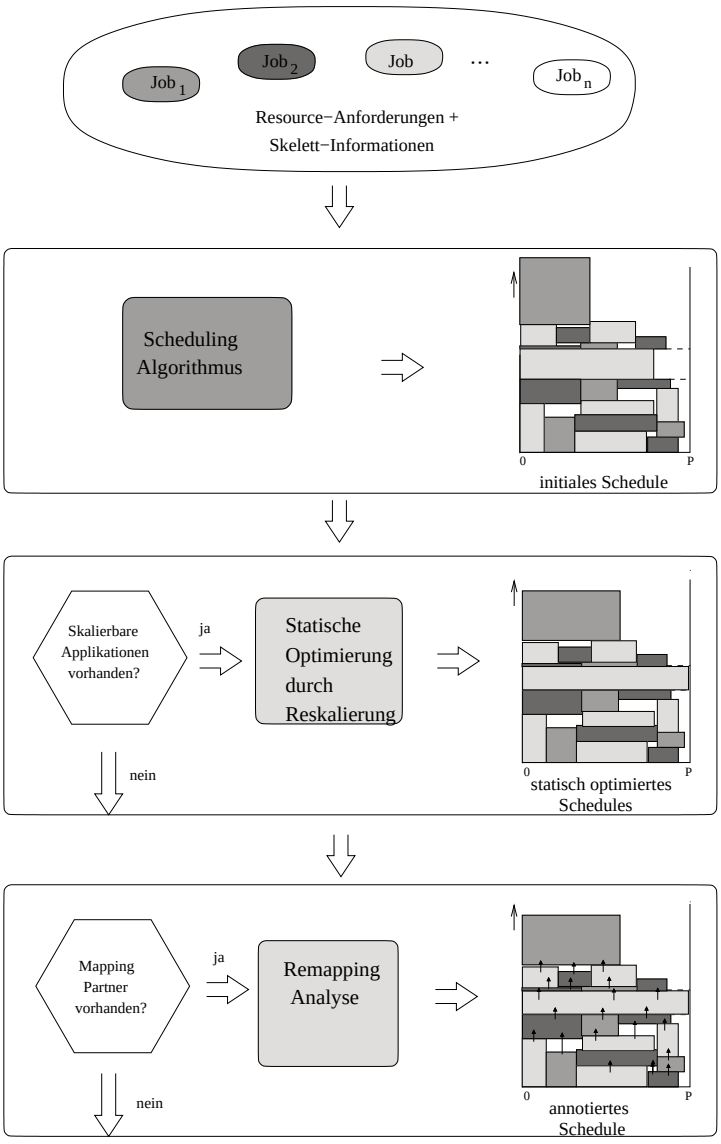


Abbildung 6.14: Generierung des Schedules

aller Antwortzeiten:

$$\frac{1}{m} \sum_{i=1}^{M} w_i t_i.$$

Im nächsten Schritt wird versucht, das berechnete Schedule mittels Reskalierung einzelner Applikationen zu optimieren und so die Kosten weiter zu minimieren. Dieser Optimierungsschritt kann jedoch nur durchgeführt werden, falls skalierbare Applikationen vorliegen. Das Programm zur Berechnung des initialen Schedules stützt sich dabei auf die Informationen, die von den verwendeten Skeletten geliefert werden und testet, ob skalierbare Applikationen vorliegen.
Ist dies der Fall, so kann die Optimierung durchgeführt werden. Dazu wird das in 5.4.3 vorgestellte Verfahren benutzt. Nach Ablauf einer vorgegebenen Anzahl an Iterationsschritten wird das beste Schedule selektiert und die Optimierungsphase abgeschlossen.
Im nächsten Schritt wird das Schedule einer Remapping-Analyse unterzogen. Hierbei werden potentielle Kandidaten für das vertikale Remapping bestimmt. Dazu ist es notwendig, für alle zeitlich direkt aufeinander folgende Applikationen deren zugrundeliegendes Skelett zu betrachten. Mit Hilfe dieser Information, ist es möglich Partner für ein dynamisches Prozessor Remapping zu finden. Für jedes potentielle Remapping–Paar wird dabei bestimmt, welche gemeinsame Prozessormenge die Applikationen besitzen und welche Prozessoren vorzeitig zwischen den Anwendungen transferiert werden sollen. Die so generierten Remapping–Instruktionen werden in einem annotierten Schedule gespeichert. Basierend auf diesem Schedule erhalten die einzelnen Applikationen ihre Instruktionen fürs Remapping. Hierzu gehört die zugeteilte Prozessormenge, sowie Start– und Terminierungszeitpunkt. Die Ausführung des Schedules wird verteilt von den einzelnen Applikationen gesteuert. Dieser Vorgang wird detailiert im nächsten Abschnitt beschrieben.

6.4.2 Verteilter Ablauf des Scheduling

Die verteilte Ablaufsteuerung und Synchronisation der parallelen Programme basiert auf dem Versenden von Prozessorlisten. Jedes parallele Programm verfügt über eine Kommunikationsschnittstelle, die als Ausgabe eine Liste der aktuell freigewordenen Prozessoren verschickt. Diese Liste wird dynamisch während der Berechnung des Programms J_i schrittweise aufgebaut und bei jeder Änderung mittels Broadcast an alle Programme verschickt, die in direkter zeitlicher Nachfolge eines Programms stehen und deren Prozessormenge

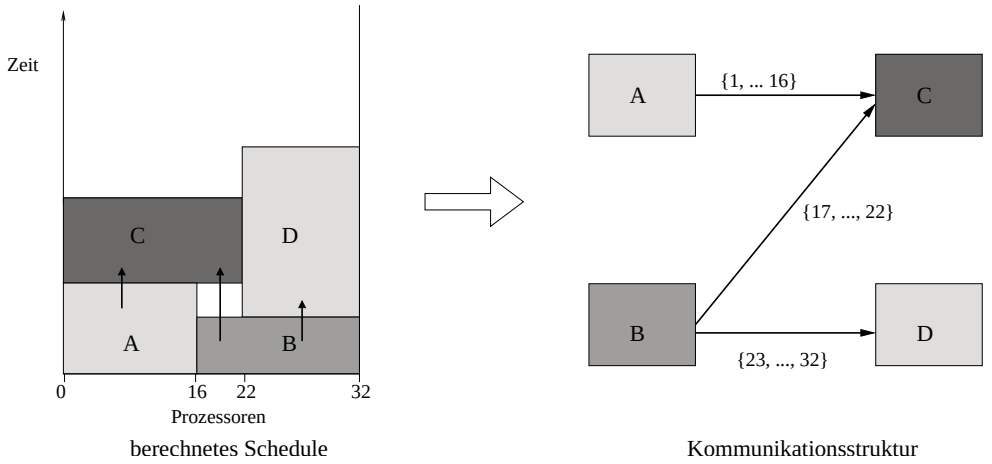


Abbildung 6.15: Remapping von Prozessoren

einen nicht leeren Schnitt mit J_i besitzt. Jedes Programm erhält vom Scheduler zudem eine Liste der zugeteilten Prozessorkennungen. Hiermit ist es möglich, aus der Menge freier Prozessoren die zugeteilten Prozessoren mittels ihrer Kennungen herauszufiltern. Jedes Programm kann so autonom feststellen, wann alle Prozessoren verfügbar sind, und die eigentliche Berechnung starten.

6.4.3 Verteiltes dynamisches Remapping

Das dynamische Remapping erfolgt entlang der Zeitachse zwischen aufeinanderfolgenden Programmen. Analog zur Ablaufsteuerung werden hierzu wiederum die Listen freier Prozessoren zur Kommunikation und zum Austausch von Ressourcen benutzt. Potentielle Kandidaten für den Prozessoraustausch sind schon während des initialen Schedulings vom Schedulingalgorithmus bestimmt worden. Die so ausgewählten Programme erhalten Remapping-Direktiven, mit denen sie selbständig das Remapping initiieren und kontrollieren. Die Direktiven lassen sich dabei in zwei Klassen aufteilen:

1. Anweisungen an den abgebenden Prozeß:
   Hierbei wird einem Programm mitgeteilt, welche Prozessorteilmenge von der nachfolgenden Applikation benötigt wird. Diese Direktive ermöglicht es, die Evaluation des Programmes so zu organisieren, daß gerade diese Prozessormenge zuerst frei wird. Das hierzu benötigte flexible interne Mapping des Programms ist, wie schon in Abschnitt

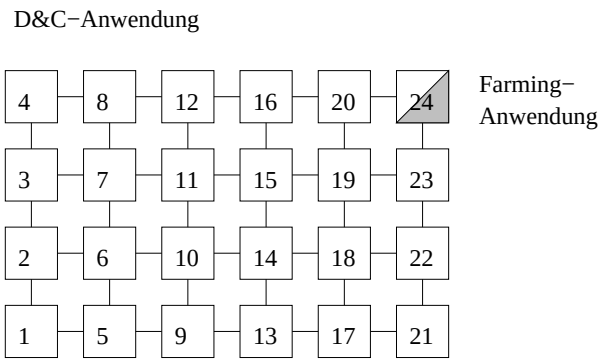
6.2 erwähnt, in die Implementation der Skelette integriert.

2. Anweisungen an den aufnehmenden Prozeß:
   Einem Prozeß wird hierbei mitgeteilt, ob der Start der Berechnung mit der vollständigen zugeteilten Prozessormenge oder schon bei der Verfügbarkeit einer Teilmenge erfolgt. Hierzu wird die Kombination zweier Prozessorlisten benutzt. Die Liste der zugeteilten Prozessoren stellt die Menge der initial zum Start benötigten Prozessoren dar. Eine Zusatzliste enthält die Kennungen von Prozessoren, die in die laufende Berechnung integriert werden können. Für Applikationen, die ihre gesamten zugeteilten Prozessoren zum Start benötigen, ist diese zweite Liste leer. Mittels dieses einfachen Listenkonstrukts ist es möglich, einer Applikation alle zum Remapping notwendigen Informationen zur Verfügung zu stellen.

Im folgenden Beispiel soll die Verwendung von Mapping-Direktiven zur Steuerung des Remappings zwischen zwei Applikationen verdeutlicht werden. Bei den abzuarbeitenden Programmen handelt es sich um eine Applikation j_i basierend auf dem D&C-Skelett und einer Anwendung j_k, die als Grundlage Farming benutzt. Weiterhin gilt $\mathcal{P}_i \subset \mathcal{P}_k$. Ein möglicher Start von j_k hängt somit einzig von freigewordenen Prozessoren der Applikation j_i ab. Da jede Applikation, die das Farming–Skelett nutzt, sinnvollerweise schon mit dem Kontrollprozessor und einem einzelnen Worker-Prozessor arbeiten kann, besteht die initiale Prozessormenge aus zwei Prozessoren. Die Kennung der Prozessoren wird vom Schedulingalgorithmus so gewählt, daß alle Worker-Prozessoren in enger topologischer Nachbarschaft liegen. Hierdurch wird ein Prozessormapping realisiert, das mit der Kommunikationstopologie des zugrundeliegenden Transputersystems harmoniert und die Kommunikationskosten innerhalb einer Applikation minimiert.

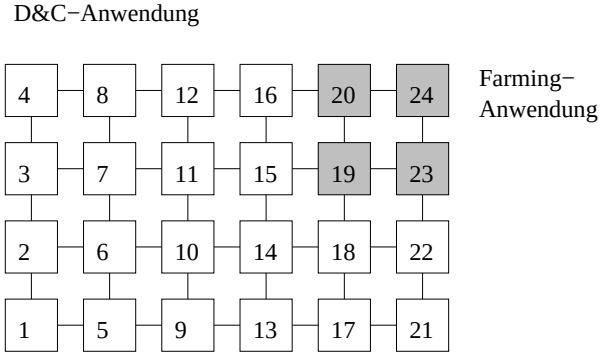
- initiales Mapping
  Der initiale Schnappschuß zeigt die aktuelle Prozessorbelegung bei Abarbeitung der D&C–Applikation. Die Applikation befindet sich im Zeitintervall zwischen den Zeitpunkten T1 und T2 ihres Parallelitätsprofils (siehe Kapitel 5.1). Die Anwendung befindet sich also in der Berechnungsphase, die die maximale Parallelität aufweist. Alle zugeteilten Prozessoren nehmen aktiv an der Berechnung teil. Zu diesem Zeitpunkt befindet sich die Farming–Applikation nicht in Ausführung. Der initiale Skelettprozeß fürs Farming befindet sich jedoch im suspendierten Zustand im Hauptspeicher von Prozessor 35. Wird vom D&C-Programm nun mindestens ein Prozessoren freigegeben, so wird dieser initiale Prozess wieder aktiviert. Seine Aufgabe ist es nun, festzustellen, ob die

freigegeben Prozessoren zur zugeteilten Prozessormenge gehören. Trifft dies zu, so wird die eigentliche Abarbeitung der Farming–Applikation begonnen, anderenfalls deaktiviert sich der initiale Prozeß wieder und wartet auf neue Prozessoren.



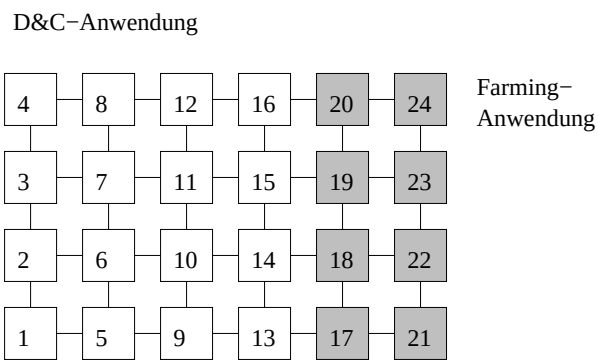

```
initialer Aufruf:

pIn1   = [1, ..., 24],
procfilter1 = [1, ..., 24],
procfilter2 = [13, ..., 24]],
div_and_conq(pIn1, pFilter1, pOut1, ... )@vts:node(1),
pIn2 = pOut1, farm(pIn2, pFilter2, pOut2, ...)@vts:node(24),

aktuelle Listeninhalte: pIn2 =[]
```

- Remapping (Schritt 1)
  Die D&C–Applikation hat den Zeitpunkt T2 passiert und schon einige Prozessoren freigegeben. Die Farming–Applikation hat diese Prozessoren zu einem vorzeitigen Programmstart genutzt.



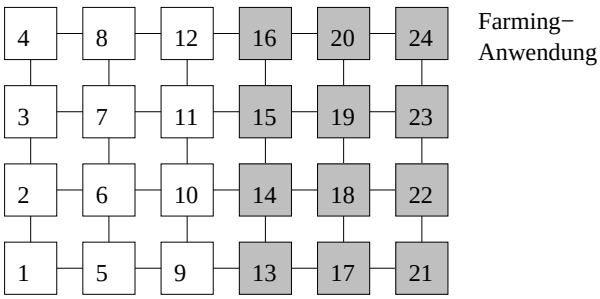
```
aktuelle Listeninhalte: pIn2 =[19,20,23,24]
```

- Remapping (Schritt 2)
  Analog zum vorhergehenden Schritt wurden weitere Prozessoren zwischen den beiden Applikationen migriert.




```
aktuelle Listeninhalte: pIn2 =[17,18,21,22]
```

- Remapping (Schritt 3)
  Die Farming–Applikation hat die letzten zugeteilten Prozessoren erhalten. Der Migrationsprozeß ist hiermit beendet. Beide Applikationen werden nun weiter voneinander unabhängig berechnet.




```
aktuelle Listeninhalte: pIn2 =[13,14,15,16]
```

6.5 Eingesetzte Optimierungsverfahren

6.5.1 Ausgewählte Optimierungsverfahren

Bei der Optimierung eines statischen Schedules müssen bei der Wahl des Optimierungsverfahrens die folgenden Punkte berücksichtigt werden:

- **die Beziehung zwischen Laufzeit und Qualität:**
  Beim praktischen Einsatz eines Optimierungsverfahrens ist darauf zu achten, daß die Laufzeit des Verfahrens in einem verträglichen Verhältnis zum erzielten Ergebnis der Optimierung steht. Bei der Optimierung von Schedules, die nur einmal abgearbeitet werden, dürfen die durchschnittlichen Berechnungskosten für die Optimierung nicht die durch die Optimierung erzielte durchschnittliche Zeitersparnis bei der Abarbeitung des Schedules übertreffen. Kann diese Bedingung nicht eingehalten werden, so ist das Optimierungsverfahren ungeeignet, da die Gesamtkosten für Optimierung und Ausführung des Schedules, die Ausführungkosten für ein nicht optimiertes Schedule überschreiten.

- **Konvergenzverhalten des Optimierungsverfahrens:**
  Aus der oben aufgeführten Beziehung zwischen Laufzeit und Qualität der Optimierung ergibt sich die Anforderung an das Optimierungsverfahren, schon nach möglichst kurzer Rechenzeit ein Optimierungsergebnis zu erzielen, das eine kosteneffektive Verbesserung des ursprünglichen Schedules darstellt. Bei Schedules mit kurzer Gesamtrechenzeit sind durch Optimierung meist nur kleine zeitliche Verbesserungen zu erreichen. In solchen Fällen muß das Optimierungsverfahren schon nach kurzer Rechenzeit ein brauchbares Ergebnis erzielen. Bei Schedules mit langer Gesamtrechenzeit ist das Potential für eine zeitliche Verbesserung deutlich größer als bei Schedules mit kurzer Rechenzeit. Hier darf das Optimierungsverfahren deutlich mehr Rechenzeit einsetzen, um ein effizientes Optimierungsergebnis zu erzielen.

Basierend auf den obigen Überlegungen wurden die folgenden drei Optimierungsverfahren als Testkandidaten ausgewählt und ihre Qualität in einer ausgedehnten Simulation getestet.

6.5.2 Hill-Climbing

Das sogenannte *Hill-Climbing* stellt ein einfaches Optimierungsverfahren dar, das versucht, durch zufällige Modifikationen eine Lösung zu optimieren. Da-

bei werden die folgenden Schritte durchlaufen:

1. Starte mit einer initialen Problemlösung
2. Modifiziere per Zufall die Problemlösung und stelle sicher, daß die neue Lösung korrekt ist.
3. Stelle die alte Lösung wieder her, falls sie besser als die neue Lösung ist.
4. Erhöhe den Iterationszähler
5. Halte an, falls die maximale Anzahl an Iterationen erreicht ist, ansonsten fahre fort mit Schritt 2

Eine Maximierung mittels Hill-Climbing ist, wie schon der Name andeutet, mit dem Erklimmen eines Berges zu vergleichen. Da wir es hier mit einem Minimierungsproblem zu tun haben, müßte der hier eingesetzte Algorithmus eigentlich Hill-Descending heißen. Da sich die prinzipielle Arbeitsweise nicht vom Hill-Climbing unterscheidet, verwenden wir weiter die gebräuchlichere Bezeichnung Hill-Climbing.

Einsatz im Scheduling

Wie schon in Abschnitt 5.4.3 beschrieben, wird das Hill-Climbing nicht direkt zur Optimierung des Schedules, sondern zur Optimierung der Ressourceanforderungen eingesetzt. Hierzu dient Schritt 2 vom Algorithmus. In Schritt 3 wird dann zunächst mittels PBI-Scheduling ein neues Schedule berechnet und dieses mit dem alten Schedule verglichen. Beim Test auf Korrektheit muß in Schritt 2 daher nur sichergestellt werden, daß die neue Ressourceanforderung für den modifizierten Job korrekt ist.

Vorteile:

Das Hill-Climbing hat den Vorteil, schnell ein lokales Optimum zu erreichen, da immer nur solche Veränderungen des Schedules beibehalten werden, die eine Verbesserung mit sich bringen.

Nachteile:

Damit das Hill-Climbing gut funktioniert, dürfen die inkrementellen Änderungen am Schedule nicht zu groß sein. Dies führt dazu, daß ein einmal

erreichtes lokales Optimum nicht mehr verlassen wird, falls alle möglichen weiteren Änderungen zu einer Verschlechterung des Ergebnisses führen. Je mehr Tasks ein Schedules enthält, desto größer ist die Zahl der lokalen Optima und damit die Gefahr, daß das Hill-Climbing in einem solchen Optimum hängen bleibt.

6.5.3 Genetische Algorithmen

Mit dem Begriff *Genetische Algorithmen* werden Verfahren bezeichnet, die zur Optimierung Methoden aus der Genetik wie *Mutation*, *Vererbung* und *Selektion* benutzen. Genetische Algorithmen starten ihre Optimierung mit einer festgelegten Menge von zufälligen Lösungen, die als Population bezeichnet wird. Eine einzelne Lösung wird dabei in einem Individuum kodiert. In einem iterativen Verfahren werden nun mittels Mutation oder Vererbung Generationen neuer Individuen generiert, die wiederum neue Lösungen repräsentieren. Zur Bewertung, welche Individuen sich fortpflanzen können, dient die sogenannte *Fitness-Funktion*. Ein Genetischer Algorithmus durchläuft die folgenden Schritte:

1. Initialisiere eine zufällige Generation 0 von x Individuen als Startpopulation
2. Berechne für alle Elemente der Generation ihren aktuellen Wert mittels der Fitness-Funktion
3. Wähle mittels der Fitness-Funktion Individuenpaare aus und erzeuge Nachkommen durch Rekombination der Gene.
4. Mutiere den Nachwuchs
5. Ersetze die Elterngeneration durch die Kindergeneration nach einer festgelegten Selektionsregel
6. Halte an, falls die Lösung ein vorgegebenes Kriterium erreicht hat, oder die maximale Anzahl an Generationen erreicht wurde. Ansonsten fahre fort mit Schritt 2.

Einsatz im Scheduling

Oft findet man Genetische Algorithmen, die komplett auf den Schritt 3 der Vererbung verzichten und den Nachwuchs nur durch Klonen und Mutation

erzeugen. Dieser Ansatz büßt zwar einige Flexibilität ein, jedoch erleichtert er den Test, ob ein neu erzeugtes Individuum eine korrekte Kodierung der Lösung enthält. Ein solch gestalteter Genitischer Algorithmus wurde als Basis für den Vergleichstest gewählt. Analog zum Hill-Climbing wird auch hier in einem Individuum kein Schedule, sondern nur die Ressourcen-Anforderungen der Jobs kodiert. In einem Mutationsschritt werden zufällig die Prozessoranforderungen eines Jobs modifiziert und die dadurch geänderte Rechenzeitanforderung automatisch angepaßt.

Vorteile:

Da in einer Population immer viele Optimierungsversuche gleichzeitig durchgeführt werden, ist die Gefahr gering, bei der Optimierung in einem lokalen Optimum hängenzubleiben. Dies kann zwar für einzelne Individuen der Population zutreffen, aber solche Individuen werden dann in der Regel durch Selektion in den nächsten Iterationsschritten aus der Population entfernt. Je mehr Individuen eine Population hat, desto geringer ist die Gefahr, in lokalen Optima hängenzubleiben.

Nachteile:

Im Gegensatz zum Hill-Climbing besitzen Genetische Algorithmen den prinzipiellen Nachteil, bezogen auf den Rechenaufwand, langsamer zu konvergieren. Dies rührt daher, daß die Gesamtzahl an Iterationsschritten auf die Gesamtpopulation aufgeteilt werden muß. Stehen insgesamt nur 1000 Iterationsschritte zur Verfügung, so kann jedes Individuum einer Population von 100 Individuen nur 10 Iterationschritte durchführen. Diese Eigenschaft schlägt sich auch in den praktischen Ergebnissen nieder.

6.5.4 Der Sintflut-Algorithmus

Der Sintflut-Algorithmus [Wal93] stellt ein Optimierungsverfahren dar, bei dem der zulässige Lösungsraum zufällig durchstreift und eine Lösung akzeptiert wird, falls Sie über einem bestimmten Schwellwert liegt. Dieser Schwellwert wird mit fortlaufender Simulation immer mehr angehoben. Die Maximierung mit dem Sintflut-Algorithmus ist mit einer Wanderung im Gebirge zu vergleichen, bei der der Wanderer zufällig herumwandert. Mit Beginn der Wanderung steigt der Wasserpegel kontinuierlich an. Der Wanderer darf nur solche Wege gehen, bei denen er nicht die Fluten durchschreiten muß. Diese Regel führt dazu, daß der Wanderer nach einer gewissen Zeit keinen Schritt

mehr machen kann, da alle Wege überflutet sind. Er hat in diesem Fall ein lokales Maximum erreicht.

Einsatz im Scheduling

Der Sintflut-Algorithmus ist vom Grundkonzept ein Maximierungsverfahren. Um ihn für das hier vorliegende Minimierungproblem einsetzen zu können, wurde mit einem sinkenden Schwellwert gearbeitet. Dabei werden die folgenden Schritte durchlaufen:

1. Setze den Schwellwert auf den 1,5-fachen Wert des nicht optimierten Schedules
2. Initialisiere das Schwellwert-Dekrement mit dem Quotient aus Schwellwert und Anzahl an Iterationszyklen
3. Modifiziere zufällig die Ressourcen-Anforderung eines Jobs
4. Berechne neues Schedule
5. Falls Wert für neues Schedule < Schwellwert, dekrementiere Schwellwert, ansonsten setze die Modifikation zurück
6. Falls gefundenes Minimum > Schwellwert, setze Schwellwert auf gefundenes Minimum
7. Falls Wert für neues Schedule < gefundenes Minimum, setze gefundenes Minimum auf Wert für neues Schedule
8. inkrementiere den Iterationszähler
9. Falls maximale Iterationszahl erreicht ist, gib gefundenes Minimum aus, ansonsten fahre mit Schritt 3 fort.

Solange der Algorithmus eine bessere Lösung findet, wird der Schwellwert kontinuierlich herabgesetzt. Wird der Punkt erreicht, bei dem der Schwellwert kleiner als das gefundene Minimum ist, so arbeitet der Algorithmus in einem reinen Hill-Climbing-Modus und versucht so das Ergebnis noch weiter zu verbessern.

Verfahren	Zahl der Iterationen					
	Ohne Optimierung	100	1000	2000	5000	10000
Hillclimbing:	7790,7	5849,2	5363,5	5357,5	5357,5	5357,5
Sintflut:	7790,7	6059,7	5135,3	4935,5	4748,5	4739,9
Gen. Alg. :	7790,7	7010,7	5531,8	5175,0	4907,8	4825,9

Tabelle 6.5: Ergebnisse der getesteten Optimierungsverfahren

Vorteile:

Da der Sintflut-Algorithmus während der Berechnung auch Lösungen akzeptiert, die schlechter als die zuletzt gefundene Lösung sind, ist hier im Gegensatz zum Hill-Climbing die Gefahr geringer in einem lokalen Optimum hängen zu bleiben.

Nachteile:

Bedingt durch die sehr hohe Toleranz in der Startphase des Algorithmus, fallen die Zwischenergebnisse nach geringer Anzahl an Simulationszyklen naturgemäß schlechter als beim Hill-Climbing aus.

6.5.5 Praktischer Vergleich der Verfahren

Die oben vorgestellten Verfahren wurden in einer Simulation auf ihre Eignung im Zusammenspiel mit den eingesetzten Schedulingverfahren getestet. Als Vergleichsbasis diente ein Menge von 1000 zufällig zusammengestellten Schedules, die jeweils aus 20 parallelen Jobs bestanden. Jedem Job wurden zwischen 4 und 128 Prozessoren und Laufzeiten zwischen 20 und 400 Zeiteinheiten zufällig zugeteilt. Alle so generierten Schedules wurden nacheinander mittels Hill-Climbing, dem Sintflut-Algorithmus und dem Genetischen Algorithmus optimiert. Nach 10000 Iterationsschritten wurde die Optimierung beendet, wobei nach 100, 1000, 2000 und 5000 Iterationen jeweils Zwischenergebnisse ermittelt wurden. Bei dem mitgetesteten Genetischen Algorithmus wurde eine Population von 50 Individuen gewählt. Um die Ergebnisse dieser Optimierung bzgl. Berechnungsaufwand mit den beiden anderen Verfahren vergleichen zu können, wurden hier jeweils nur 2, 20, 40, 100 und 200 Iterationen pro Individuum durchgeführt. Der Aufwand hierfür entspricht dann in etwa dem Aufwand für 100, 1000, 2000, 5000 und 10000 Iterationszyklen der anderen Verfahren.

Betrachtet man die Ergebnisse, die in der Tabelle 6.5 aufgelistet sind, so

stellt man das folgende Verhalten fest:

- Kurze Iterationszeiten (100 Iterationen):

  Bei sehr kurzen Simulationszeiten liefert das Hill-Climbing die besten Ergebnisse. Vergleicht man die Werte mit dem Sintflut-Algorithmus, so liefert dieser deutlich schlechtere Resultate. Dies rührt daher, daß der Pegelstand beim Sintflut-Algorithmus in der Startphase oft noch Verschlechterungen der Lösung zuläßt und somit das Verfahren langsamer gegen ein Minimum konvergiert. Der Genetische Algorithmus zeigt im Vergleich den schlechtesten Wert, da er bei einer Population von 50 Individuen nur jeweils 2 Optimierungsschritte pro Individuum ausgführt. Es konnte daher nur eine partielle Optimierung an maximal zwei Jobs im Schedule durchgeführt werden.

- Mittlere Iterationszeiten (1000 - 2000 Iterationen):

  Beim Wert nach 1000 Iterationen läßt sich erkennen, daß der Sintflut-Algorithmus ein signifikant besseres Ergebnis als seine Mitkonkurrenten liefert. Dieser Trend setzt sich auch bei 2000 Iterationen fort. Hier erkennt man, daß das Hill-Climbing schon bei 1000 Iterationen ein lokales Minimum erreicht hat. Eine Verbesserung des Ergebnisses ist nach dieser nur mehr möglich, falls die Ressoucren-Anforderungen von mindestens zwei Jobs geändert werden. Eine solche Veränderung läßt das Hill-Climbing nicht zu, da nach jeder Modifikation der Ressourcen-Anforderung eines einzelnen Jobs, immer getestet wird, ob sie eine Verbesserung darstellt. Trifft dies nicht zu, so wird die Änderung direkt verworfen. Es kann so keine Verbesserung des Schedules durch einen Synergie-Effekt mit einer weiteren Modifikation erreicht werden. Wie man an den Ergebnissen sieht, hat der Sintflut-Algorithmus nicht mit diesen Problemen zu kämpfen. Durch seine Toleranz gegenüber partiell schlechteren Lösungen, kann eine durch die Modifikation an einem Job erzielte verschlechterte Lösung akzeptiert werden, die dann im nächsten Optimierungsschritt zu dem beschriebenen Synergie-Effekt führen kann.

- Längere Iterationszeiten (5000 - 10000 Iterationen)
  Im Bereich der längeren Iterationszeiten kann man wieder die bekannten Probleme des Hill-Climbings mit lokalen Minima erkennen. Auch nach 5000 und 10000 Iterationschritten ist keine Verbesserung erreicht

worden. Der Sintflut-Algorithmus liefert dagegen aus den schon beschriebenen Gründen jeweils ein neues Minimum. Einzig der Genetische Algorithmus kann hier mithalten. Er zeigt jedoch für alle Werte ein leicht schlechteres Verhalten.

Aus den Ergebnissen der Simulation kann man das folgende Fazit ziehen:

- Für kurze Simulationszeiten ist das Hill-Climbing das beste der getesteten Verfahren.
- Für mittlere und längere Simulationszeiten liefert der Sintflut-Algorithmus sehr gute Ergebnisse.
- Der getestete Genetische Algorithmus liefert ähnliche gute Ergebnisse, wie der Sintflut-Algorithmus, wird aber von diesem immer übertroffen..

Basierend auf diesen Ergebnissen wurde für den weiteren praktischen Einsatz ein Verfahren gewählt, daß abhängig von der Zahl der Simulationszyklen entweder das Hill-Climbing oder den Sintflut-Algorithmus einsetzt. Zwar wäre hier auch eine Kombination aus Genetischen Algorithmus und Hill-Climbing sinnvoll gewesen, jedoch hätte man noch weiteren Aufwand ins Tuning investieren müssen, um die Qualität vom Sintflut-Algorithmus zu erreichen.

6.6 Ergebnisse des Schedulingverfahrens

An den in diesem Abschnitt aufgeführten Beispielschedules wurde untersucht, ob sich die Vorteile der Reskalierung und des dynamischen Remappings auch im praktischen Einsatz zeigen.

6.6.1 Laufzeitvergleich

Für jede in den Schedules benutzte Applikation wurde zuerst die Laufzeit für ihre maximal einsetzbare Prozessorzahl ermittelt, um Informationen für die Berechnung des initialen Schedules zu erhalten. Hierzu wurde die Applikation ohne weitere Partnerapplikationen ausgeführt und die Laufzeit ermittelt. Die Prozessorzuweisung erfolgte dabei analog zur Abarbeitung im Schedule über Prozessorlisten. Die Ergebnisse finden sich in Tabelle 6.6 bis Tabelle 6.9

Mergesort (D&C-Skelett)			
Job Nr.	Eingangswerte	Prozessoren	Laufzeit
M1	2000	16	81.2
M2	4000	16	163.7
M3	8000	16	325.4
M4	2000	64	40.3
M5	4000	64	82.2
M6	8000	64	166.6

Tabelle 6.6: Laufzeitvergleich Mergesort

Hanoi (D&C-Skelett)			
Job Nr.	Scheibenzahl	Prozessoren	Laufzeit
H1	17	64	69.3
H2	18	64	143.6
H3	19	64	292.5

Tabelle 6.7: Laufzeitvergleich Türme von Hanoi

Aus den oben angeführten Applikationen wurden 10 Testschedules erstellt. Hierzu wurden jeweils 10 Applikationen durch Zufall ausgewählt und mittels Shelf-Scheduling das initiale Schedule bestimmt. Tabelle 6.10 listet die Zusammensetzung der Schedules auf.
Die Ergebnisse des initialen Schedulings finden sich in den Abbildungen 6.16 bis 6.34 und in der Tabelle 6.11. Im nächsten Schritt wurde versucht,

Minimaler Spannbaum (ic-Skelett)			
Job Nr.	Objektzahl	Prozessoren	Laufzeit
MS1	40	32	82.4
MS2	50	32	151.2
MS3	60	32	249.0
MS4	70	32	353.1
MS5	40	4	178.2
MS6	50	4	312.4

Tabelle 6.8: Laufzeitvergleich Minimaler Spannbaum

Matrix-Vektor (farm-Skelett)			
Job Nr.	Matrixgröße	Prozessoren	Laufzeit
MV1	40 x 40	8	52.1
MV2	80 x 80	8	183.6
MV3	80 x 80	32	50.4
MV4	160 x 160	32	194.5

Tabelle 6.9: Laufzeitvergleich Matrix-Vektor Multiplikation

die einzelnen Schedules mittels Reskalierung zu verbessern. Dafür wurden die Ressourceanforderungen der Schedules in jeweils 100 Iterationen mittels Hill-Climbing optimiert. Die eigentliche Generierung der Schedules erfolgte weiterhin mittels Shelf-Scheduling. Im letzten Schritt wurden die optimierten Schedules auf ein Transputer-Gitter mit 96 Prozessorknoten abgebildet. Durch das dynamische Remapping erfolgt der Start einer Applikation spätetens nachdem alle zugeteilten Prozessoren verfügbar sind. Hierdurch wurde bei der Abarbeitung der synchrone Start der Applikationen in einem Shelf vermieden und die Ausführungszeit des Schedules verkürzt. Die Tabelle 6.11 beinhaltet für jedes Schedules neben der mittleren Anwortzeit auch die Terminierungszeit vor und nach Optimierung sowie die reale Abarbeitungszeit. Die errechneteTerminierungszeit nach Optimierung dient hier als Vergleichswert für die reale Laufzeit des Schedules. Durch diesen Vergleich können die Auswirkungen des dynamischen Remappings ermittelt werden. Betrachtet man die mittlere Antwortzeit vor und nach der Reskalierung, so stellt sich in den Beispielen eine durchschnittliche Verbesserung von 17 Prozent ein. Die positiven Ergebnisse rühren daher, daß die einzelnen Shelfs nach der Optimierung besser ausgenutzt werden. Dies hat auch direkte Auswirkungen auf die Terminierungszeiten der Schedules. Hier wird im Mittel

Test-Schedules	
Schedule Nr.	Zusammensetzung
1	MS6, M3, MV1, MS1, MS4, M5, MV3, H2, MV4, MV2
2	MS2, H1, MV3, MV2, M2, M6, MV1, H2, M1, MS5
3	MS2, MV1, MV2, MS3, M3, M4, MS6, MV4, MS4, H2
4	M3, MV1, MS3, M4, M5, MS2, MS6, MS1, MV4, H3
5	MV2, MS4, MV4, H2, M6, MV2, M5, MV1, MV3, H3
6	MS4, MV4, MV1, MS3, MS2, MV3, MS1, MV2, H2, H1
7	MS1, H3, H2, MS6, MV4, MV2, H1, MV2, MS2, M6
8	MV2, MS3, MV3, MS4, M1, MV4, MS2, MS1, MV1, M3
9	M1, MV1, MV3, MS2, MV4, M3, M4, MS6, H3, M2
10	MV3, M8, MV1, H3, MV4, M5, M1, MV1, MS4, M6

Tabelle 6.10: Zusammensetzung der Schedules

eine ca. 10 Prozent kürzere Laufzeit erzielt.
Wirft man einen Blick auf die Resultate beim dynamischen Remapping, so fallen die Einsparungen nicht ganz so hoch aus. Die mittlere Verbesserung beträgt 5 Prozent. Dies ist zum Teil darauf zurückzuführen, daß die Applikationen, die vom Remapping am meisten profitieren (MV1, .. , MV4), im Vergleich zu den anderen Applikationen deutlich geringere Laufzeiten aufweisen. Somit ist hier das dynamische Optimierungspotential begrenzt. Zudem ist zu erkennen, daß Schedules mit einem geringen Verschnitt wie z.B. Schedule 1 ein geringeres dynamisches Optimierungspotential aufweisen als z.B. die Schedules 4 und 8, die einen höheren Verschnitt besitzen.
Positiv ist zu bewerten, daß sich beim Einsatz des dynamischen Remappings keine Laufzeitverschlechterung einstellt. Am Aufbau der optimierten Schedules in den Abbildungen 6.17 bis 6.35erkennt man, daß die prinzipiellen Schwächen des Shelf-Schedulings nicht überwunden werden. Die Anordnung der Applikationen in den Shelfs wird weiter vom Shelf-Sheduling bestimmt. Eine Umgruppierung einer Applikation auf einen freien Platz in einem tieferen Shelf ist nicht möglich. Dies ist der Preis, der für die Optimierung auf Ressourcen-Ebene zu zahlen ist.

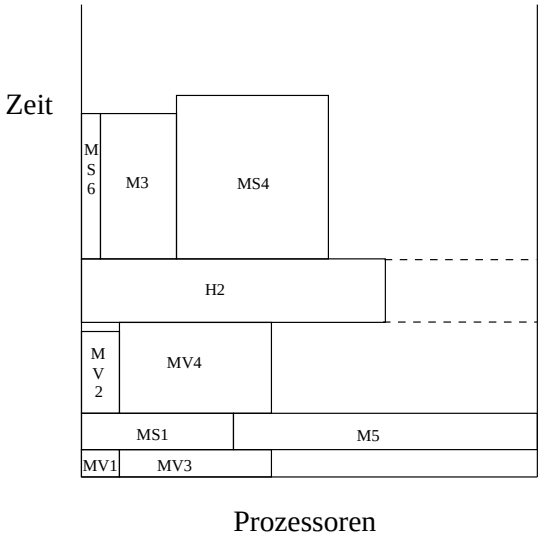


Abbildung 6.16: Schedule 1

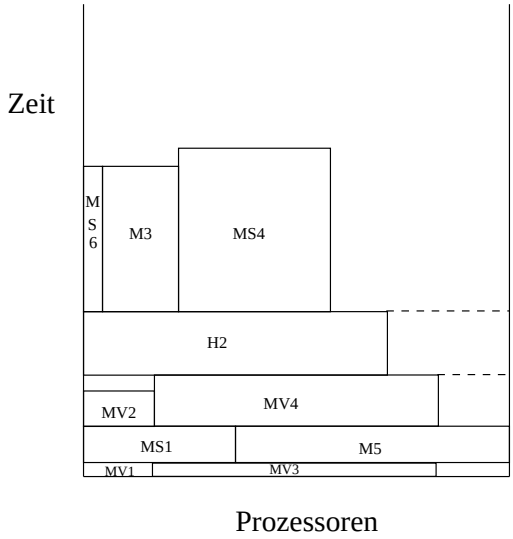


Abbildung 6.17: Schedule 1 nach Optimierung

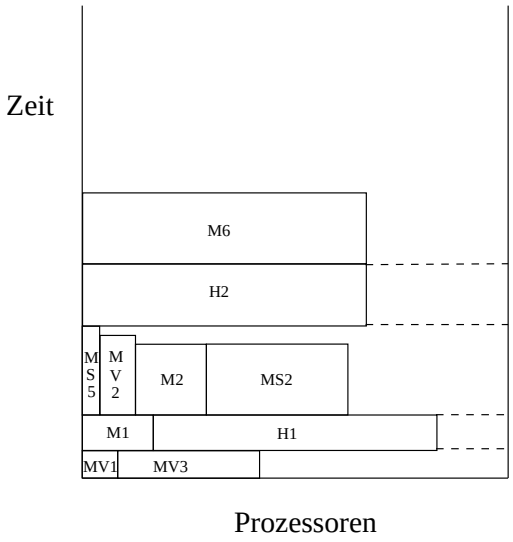


Abbildung 6.18: Schedule 2

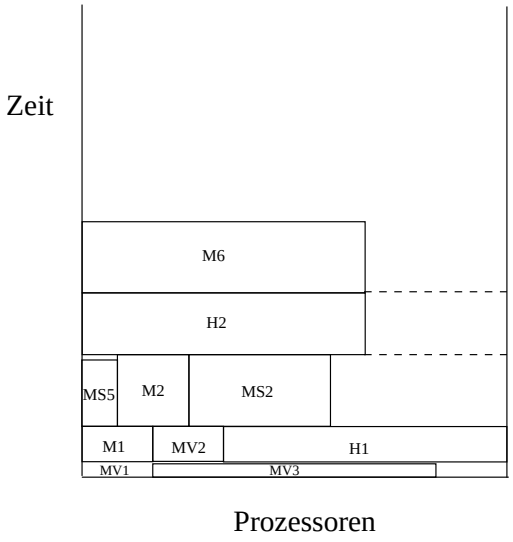


Abbildung 6.19: Schedule 2 nach Optimierung

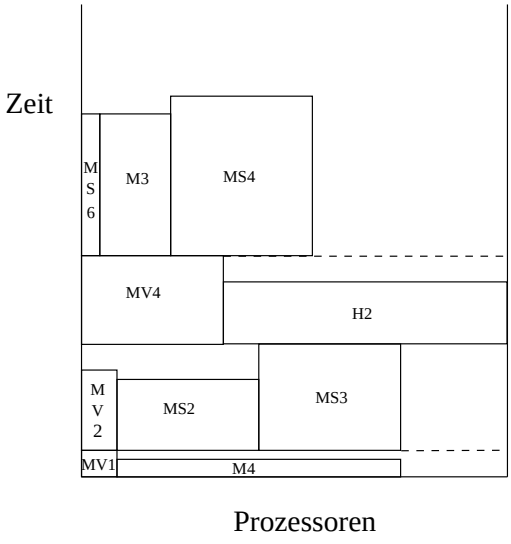


Abbildung 6.20: Schedule 3

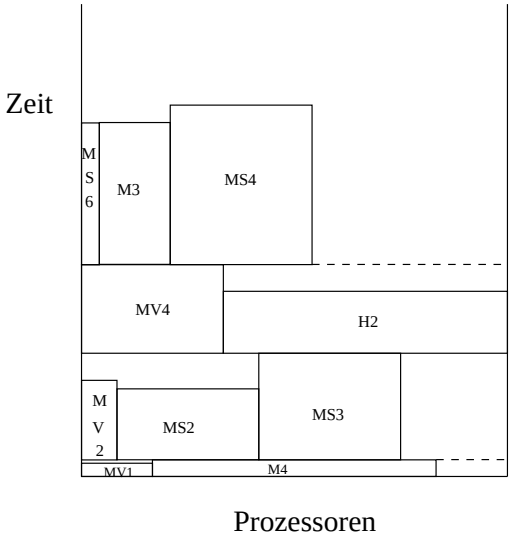


Abbildung 6.21: Schedule 3 nach Optimierung

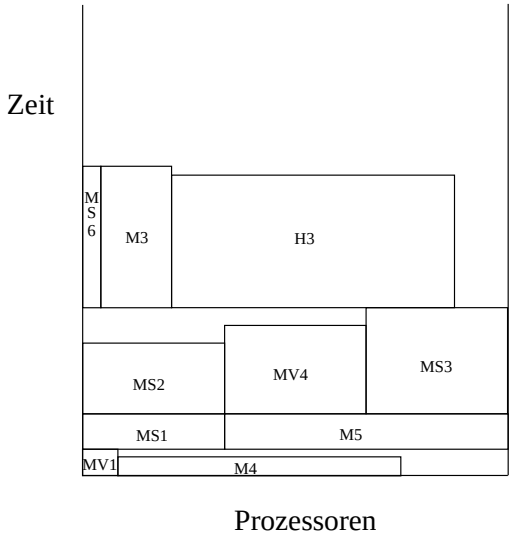


Abbildung 6.22: Schedule 4

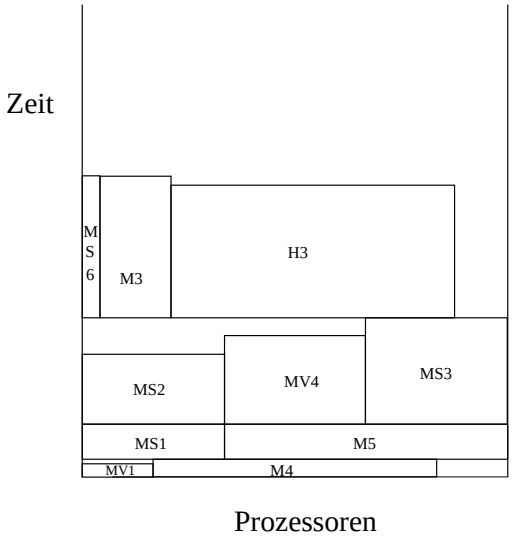


Abbildung 6.23: Schedule 4 nach Optimierung

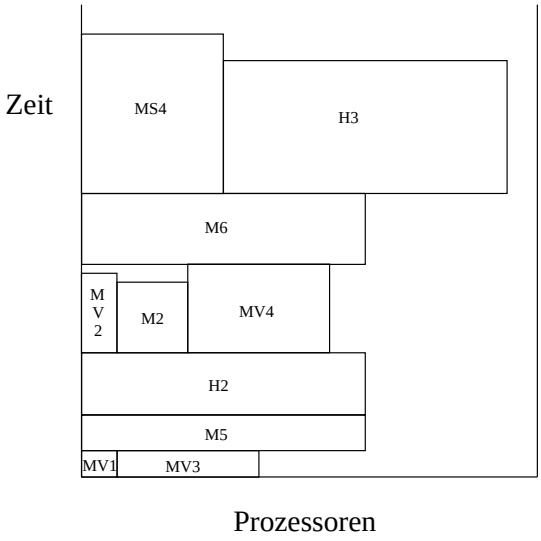


Abbildung 6.24: Schedule 5

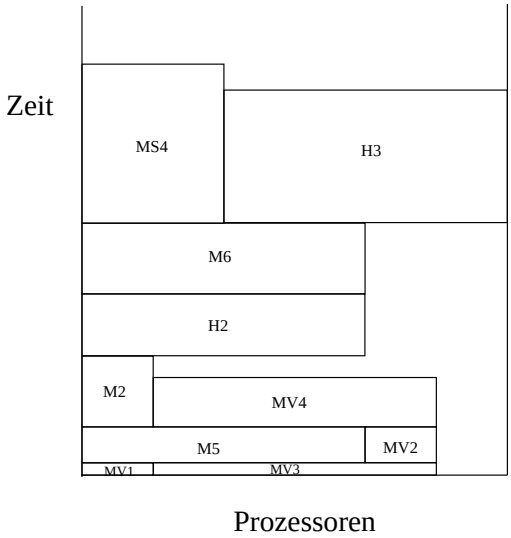


Abbildung 6.25: Schedule 5 nach Optimierung

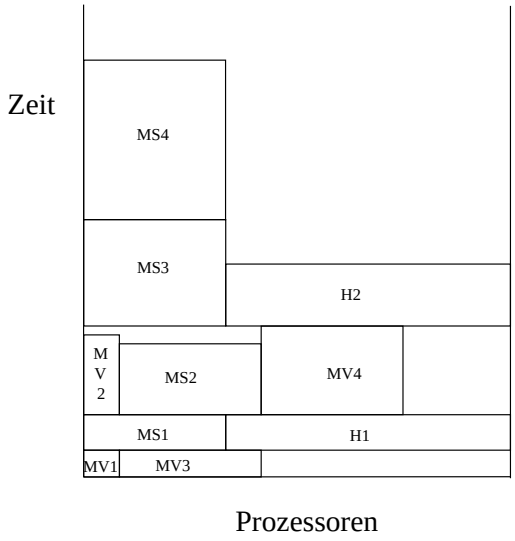


Abbildung 6.26: Schedule 6

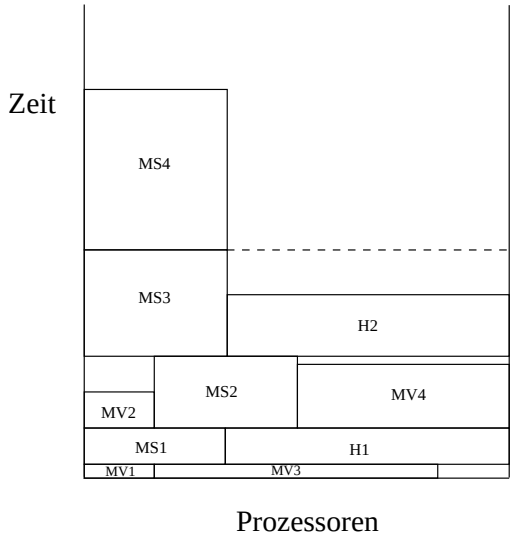


Abbildung 6.27: Schedule 6 nach Optimierung

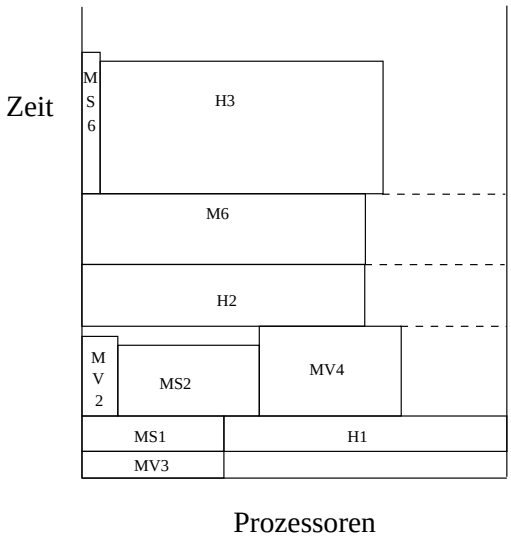


Abbildung 6.28: Schedule 7

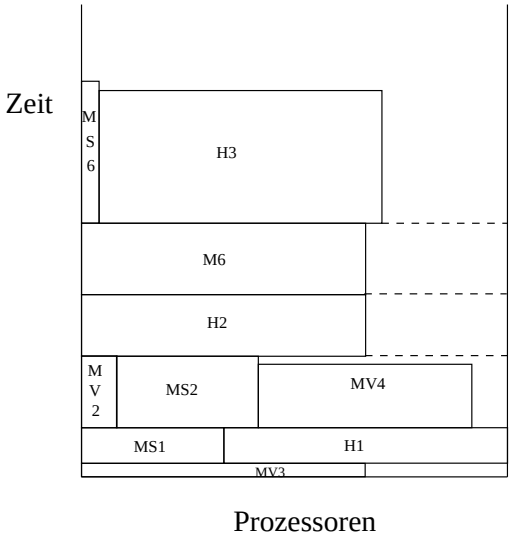


Abbildung 6.29: Schedule 7 nach Optimierung

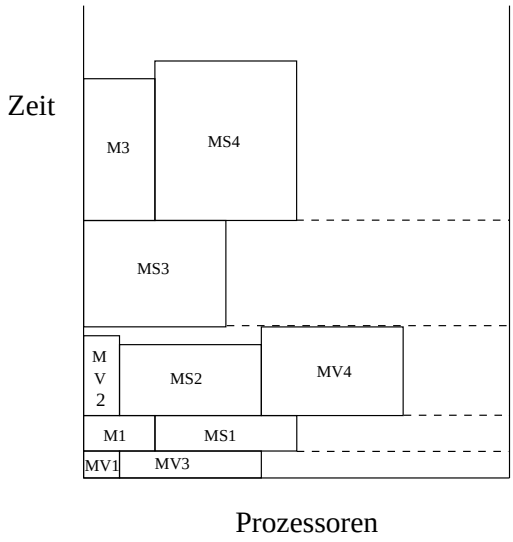


Abbildung 6.30: Schedule 8

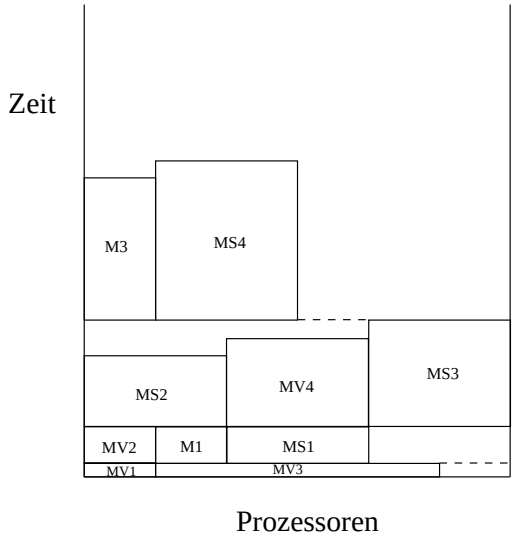


Abbildung 6.31: Schedule 8 nach Optimierung

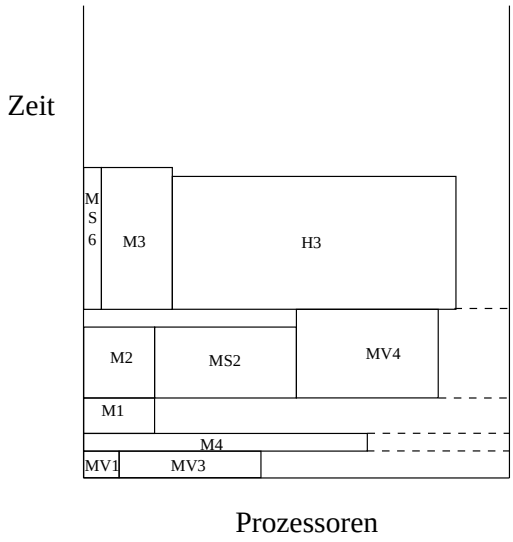


Abbildung 6.32: Schedule 9

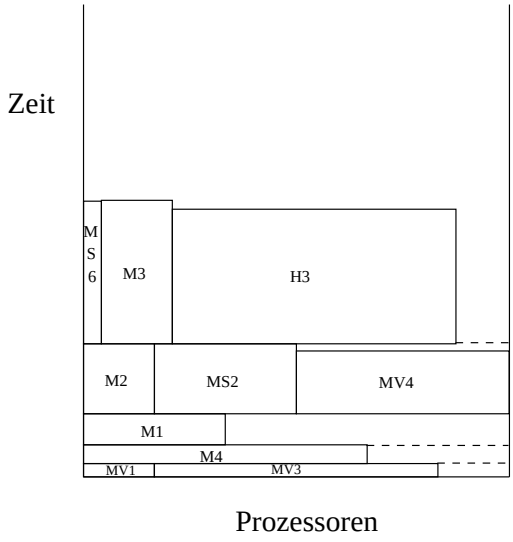


Abbildung 6.33: Schedule 9 nach Optimierung

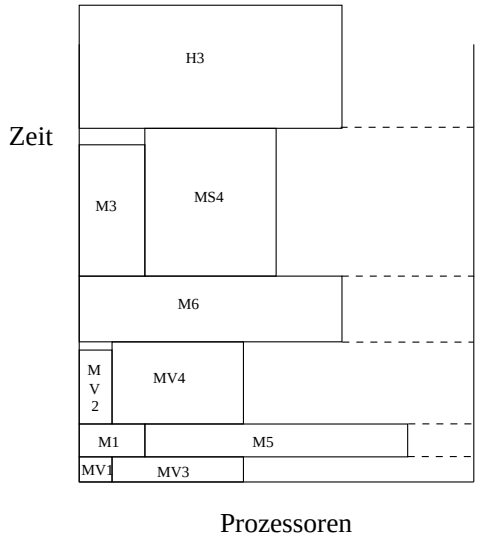


Abbildung 6.34: Schedule 10

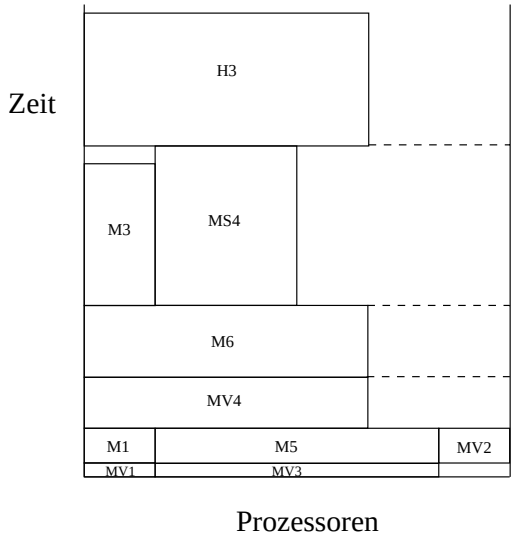


Abbildung 6.35: Schedule 10 nach Optimierung

	mittlere Antwortzeit		Terminierungszeit		
Nr.	initial	nach Reskalierung	initial	nach Reskalierung	mit dynamischem Remapping
1	389.97	262.26	824.7	702.4	686.94
2	264.39	218.23	621.7	581,15	561.97
3	420.05	413.01	848.7	836.9	786.0
4	344.02	331.98	708.9	697.1	609.7
5	445.28	364.52	992.1	944.85	912.11
6	327.29	297.44	931.1	861.75	837.62
7	421.89	385.72	949.9	913.8	864.45
8	371.56	269.83	931.1	710.55	656.05
9	343.33	273.32	693.5	596.35	588.00
10	431.65	339.76	1140.0	1016.7	989.0

Tabelle 6.11: Ergebnisse der Schedules

6.6.2 Laufzeitmaskierungen

Die folgenden Beispielmessungen zeigen, daß in einem Schedule mit vertikalem Remapping Applikationen z.T. "kostenlos"berechnet werden können. Dieser Fall tritt auf, wenn die Berechnung einer Applikation J_1 erst beendet wird, nachdem die Ausführung der Nachfolger-Applikation J_2 schon fertig ist. Die Prozessoren, die J_1 vorzeitig an J_2 übergibt, reichen hier aus, um J_2 zusätzlich während der Rechenzeit von J_1 zu evaluieren. Als Grundlage für die Messungen dienten eine Matrix-Vektor-Multiplikation, die auf dem Farming-Skelett basiert und eine Realisierung von Mergesort auf Basis eines binären D&C-Skeletts. Als Referenz wurden zuerst die Laufzeiten der beiden Applikationen für verschiedene Prozessorzahlen ermittelt. Die Ergebnisse sind in Tabelle 6.12 zusammengefaßt.

Laufzeitvergleich		
Prozessoren	Laufzeit Matrix-Vektor	Laufzeit Mergesort
8	6.1	14.1
16	3.5	11.7
32	2.4	7. 5

Tabelle 6.12: Laufzeiten der Einzelapplikationen

Anschließend wurde die Gesamtlaufzeit des Mini-Schedules ermittelt, in dem zuerst Mergesort startete und im Laufe der Berechnung freiwerdende Prozessoren an die Matrix-Vektor-Multiplikation übergab. Die Ergebnisse für verschiedene Prozessorzuweisungen finden sich in Tabelle 6.13.

Laufzeitvergleich				
Nr.	Prozessoren Matrix-Vektor	Prozessoren Mergesort	Laufzeit ohne Remapping	Laufzeit mit Remapping
1	8	8	20.2	13.9
2	8	16	17.8	11.7
3	16	16	15.2	11.7
4	16	32	12.0	8.9
5	32	32	9.9	7.8

Tabelle 6.13: Laufzeitvergleich: dynamisches Remapping - sequentielle Abarbeitung

Die Laufzeiten ohne Remapping ergeben sich dabei aus der Summe der jeweiligen Einzellaufzeiten aus Tabelle 6.12. Vergleicht man die Ergebnisse von Messung 2 und 3, so stellt man fest, daß sich die Gesamtlaufzeit mit Remapping nicht mehr verbessert, obwohl sich die Prozessorzahl der Matrix-Vektor-Multiplikation verdoppelt hat. Vergleicht man die Laufzeiten mit der Einzellaufzeit von Mergesort auf 16 Prozessoren, so ergibt sich dasselbe Ergebnis. Die Berechnung der Matrix-Vektor-Multiplikation findet somit komplett während der Berechnung vom Mergesort statt und ist schon beendet, bevor ihr ihre komplette Prozessormenge zugeteilt wurde.

6.7 Konzept einer preemptiven Erweiterung

Eine preemptive Version des vorgestellten Schedulingverfahrens läßt sich realisieren, indem man das preemptive Verhalten in die algorithmischen Skelette und damit in die parallelen Applikationen verlagert. Die Applikationen erhalten dann beim initialen Aufruf eine Liste ihrer Start- und Endzeiten der einzelnen Zeitscheiben, beginnend mit der Terminierungszeit des ersten Scheduling–Intervalls und endend mit der Terminierungszeit des letzten Intervalls. Erreicht ein Prozeß eine Zeitmarke, die das Ende einer Zeitscheibe markiert, so wechselt der Prozeß in einen Busy-Wait-Modus. Dieser Status wird erst wieder verlassen, wenn die Startzeit des nächsten Scheduling–Intervalls erreicht wird. Erreicht der Prozess die letzte Zeitmarke – markiert durch eine leere Restliste der Scheduling–Zeitmarken – so terminiert das Programm. Die Systemzeit wird in diesem Beispiel als kontinuierliche Zeitliste durch einen zentralen Clock–Prozeß zur Verfügung gestellt. Intern wird dabei auf eine Systemroutine zur Ermittlung der Zeit zurückgegriffen. Eine Zeitkontrolle sollte in der Implementation des Skeletts sinnvollerweise direkt vor und nach dem Aufruf einer Benutzer–Funktion erfolgen. Diese Funktionen werden damit als atomare Funktionsblöcke behandelt. Bei sehr rechenintensiven Benutzerfunktionen kann dies in einigen Fällen zu einem grobgranularen Zeitvergleich führen. Modelliert ein Anwender Benutzerfunktionen in reinem PCN, so setzt sich eine Funktion letztendlich aus feingranularen Basisfunktionen zusammensetzt. Es ist dann möglich, an beliebigen Punkten zwischen diesen Basisfunktionen neue Zeitkontrollpunkte einzufügen. Rechenintensive Benutzerfunktionen können somit ohne Einfluß auf ihre partielle Korrektheit automatisch durch eine zeitkontrollierte Sequenz von Subfunktionen ersetzt werden. Man erhält dadurch eine beliebig feingranulare Zeitsteuerung des Schedulings. In Anhang A.1 ist diese Transformation exemplarisch für eine sequentielle Benutzerfunktion durchgeführt worden. Setzt sich eine Benutzerfunktion jedoch aus einem sequentiellen C- oder Fortran-Programm zusammen, das in einer PCN-Funktion gekapselt ist, so kann eine solche Transformation nicht durchgeführt werden, da C- und Fortran-Programme aus Sicht von PCN atomare Prozesse darstellen.

6.8 Skelettbasiertes Scheduling auf Netzwerken heterogener Parallelrechner

Nach der Betrachtung des Schedulings auf homogenen Parallelrechnern soll an dieser Stelle noch ein kurzer Exkurs in die Welt der heterogenen Parallelrechner gemacht werden. Der Verbund verschiedener Parallelrechner zu einem heterogenen Netzwerk stellt eine zusätzliche Problematik fürs Scheduling dar. Aus Sicht des Schedulingverfahrens existiert nicht mehr ein einzelner Prozessorpool $\mathcal{P}$ sondern eine Menge $\mathcal{P}^1, ..., \mathcal{P}^m$ verschiedener Prozessorsets, wobei m der Anzahl der verschiedenen Parallelrechner entspricht. Da die einzelnen Parallelrechner unterschiedliche Verbindungsstrukturen besitzen können, ist es nicht garantiert, daß jedes Benutzerprogramm auf jedem Parallelrechner ausführbar ist bzw. die Ausführung im Bezug auf Effizienzkriterien sinnvoll ist. Weiterhin muß die Rechenzeit einer Applikation für alle nutzbaren Parallelrechner bekannt sein, da sonst die Anzahl möglicher Schedulevarianten extrem eingeschränkt und die Generierung eines effizienten Schedules quasi ausgeschlossen ist. Um diese Problematiken zu entzerren, lassen sich wiederum algorithmische Skelette mit ihren schon dargestellten Eigenschaften benutzen. Die in diesem Kapitel vorgestellten Konzepte zum Scheduling beruhen auf der Idee des *Slotting*. Hierbei setzt sich die Gesamtprozessormenge, wie schon erwähnt, $\mathcal{P}$ aus einzelnen Untermengen $\mathcal{P}^1, ..., \mathcal{P}^m$ zusammen. In der graphischen Darstellung (Abbildung 6.36) wird ein Schedule dabei in vertikale Streifen sogenannte *Slots* eingeteilt.

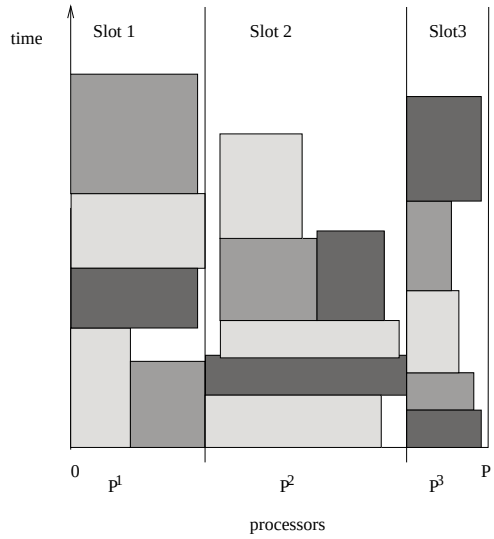


Abbildung 6.36: Slotting

Eine Applikation darf hier immer nur die Prozessoren eines einzigen Slots nutzen. Einzelne Slots stellen autonome Schedules der einzelnen Parallelrechner dar. Ist eine Applikation auf möglichst vielen Parallelrechnern des Verbundes ausführbar, so läßt sich die Qualität des Schedules in Worst–Case Situationen deutlich verbessern. Wir diskutieren dies am Beispiel eines einfachen heterogenen Parallelrechnerverbundes. Dieser Verbund bestehe aus den beiden Parallelrechnern PR_1 mit der Prozessormenge $\mathcal{P}^1 = \{1, .., m\}$ und PR_2 mit der Prozessormenge $\mathcal{P}^2 = \{m+1, .., 2*m\}$. Für jede Applikation i soll gelten, daß für jede Prozessormenge $\mathcal{P}_i^1 \subset \mathcal{P}^1$ und $\mathcal{P}_i^2 \subset \mathcal{P}^2$ mit $|\mathcal{P}_i^1| = |\mathcal{P}_i^2|$ die reine Rechenzeit h_i konstant bleibt. Die Kosten für die Berechnung eines Schedules sind also vom gewählten Slot unabhängig. Wir betrachten nun die beiden folgenden Schedule-Beispiele:

- Schedule1:
  Sei J eine Menge von $|J| = 2*n$ parallelen Anwendungen, $\mathcal{P}^1$ und $\mathcal{P}^2$ die oben definierten Mengen. Es gelte für alle Jobs i : Rechenzeit $h_i = l$ und Gewichtung $w_i = 1$. Job j mit $1 \leq j \leq n$ sei nur auf PR_1 mit Prozessorzahl $P_j = m/2 - 1$ ausführbar. Job k mit $n < k \leq 2*n$ sei nur auf PR_2 mit Prozessorzahl $P_k = m/2+1$ ausführbar. Damit ergibt sich das in Abbildung 6.37 dargestellte minimale Schedule. Die Jobs

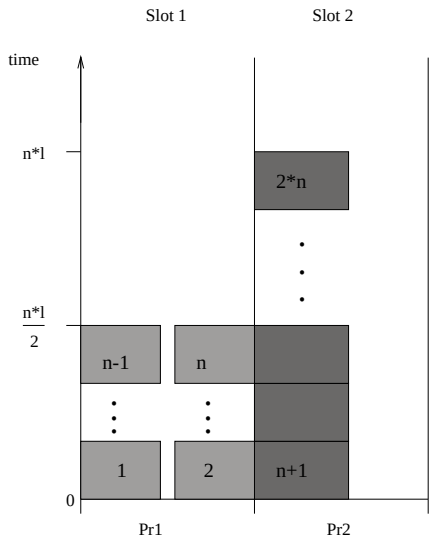


Abbildung 6.37: schlechte Ausnutzung

von Typ k benötigen immer mehr als die Hälfte der in PR_2 verfügbaren

Prozessoren. Damit können keine zwei Jobs gleichzeitig auf PR_2 abgearbeitet werden. Die Summe der Antwortzeiten berechnet sich damit wie folgt:

Schedule auf PR_1 :

$$2 * \sum_{i=1}^{n} w_i * t_i = 2 * \sum_{i=1}^{n} i * l = l * (n+1) * n$$

Schedule auf PR_2 :

$$\sum_{i=1}^{2n} w_i * t_i = \sum_{i=1}^{2n} i * l = l * (2n+1) * n$$

Aus der Summe beider Schedules ergibt sich somit die mittlere Antwortzeit:

$$\sum_{i=1}^{4n} w_i * t_i = \frac{n*(3n+2)*l}{4n} = \frac{(3n+2)*l}{4}$$

- Schedule2:
  Betrachten wir erneut die Applikationen und Prozessormengen aus Schedule1 und denselben Parallelrechnerverbund bestehend aus PR_1 und PR_2. Lassen sich die Anwendungen auf beiden Plattformen berechnen, so erhalten wir einen deutlich besseren minimalen Schedule, wie Abbildung 6.38 zeigt.

  Hier können die Jobs vom Typ j und Typ k so im Schedule angeordnet werden, daß auf jedem Parallelrechner jeweils 2 Jobs gleichzeitig abgearbeitet werden können.

  Als mittlere Antwortzeit erhalten wir hier:

$$\sum_{i=1}^{4n} w_i * t_i = \frac{2n*(n+1)*l}{4n} = \frac{(n+1)*l}{2}$$

Es zeigt sich damit, daß die Plattformunabhängigkeit einzelner Applikationen das Worst-Case Verhalten beim Scheduling deutlich verbessert. Da in einem Pallelrechnerverbund die Anzahl der Prozessoren von einem zum anderen Parallelrechner meist variieren, hängt die Ablauffähigkeit einer Applikation nicht allein vom Typ des Parallelrechners ab. Eine Applikation, die prinzipiell auf allen Rechnern des Verbunds ablauffähig ist, kann nur auf den Plattformen ablaufen, die hinreichend Prozessoren zur Verfügung stellen. Ein solches Problem läßt sich jedoch mittels Reskalierung lösen.

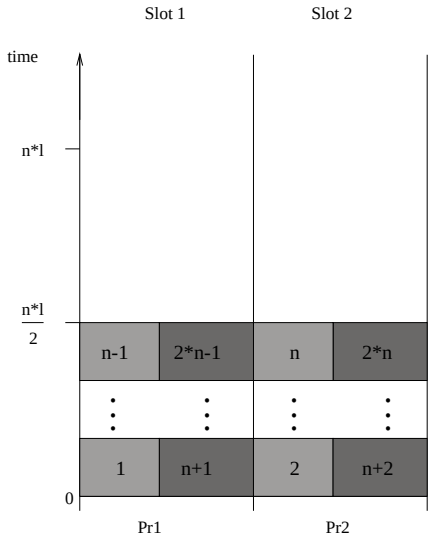


Abbildung 6.38: gute Ausnutzung

6.9 Erweitertes Schedulingverfahren für Parallelrechnernetze

Das hier vorgestellte Verfahren basiert auf dem in Abschnitt 5.4.3 vorgestellten Verfahren für homogene Prozessornetze. Es wird hier für jeden einzelnen Prozessorslot ein autonomes Schedule berechnet. Dazu ist es notwendig, die Menge der parallelen Applikationen auf die einzelnen Slots aufzuteilen. Ist eine Applikation auf mehr als einem Parallelrechner lauffähig, so wird sie zufällig einem Slot zugeordnet. Nach der initialen Berechnung aller Slot–Schedules, erfolgt die Optimierung der Schedules. Hierbei werden einzelne Applikationen in andere Slots migriert, um einen Lastausgleich zwischen den einzelnen Parallelrechnern zu erzielen. Als Lastkriterium wird die Terminierungszeit der Slot-Schedules benutzt. Ziel ist es, diese Zeiten anzugleichen, um eine möglichst zeitgleiche Terminierung der Slot–Schedules zu erreichen. Tabelle 6.14 beschreibt die Grundstruktur vom Algorithmus.

Bei der Migration der Applikationen ist oft eine Reskalierung der Prozessorzahl notwendig, um eine Applikation effizient in einen anderen Slot integrieren zu können. Hierbei können wieder die Eigenschaften algorithmischer Skelette genutzt werden, die eine plattformunabhängige Entwicklung paralleler Programme und die Realisierung skalierbarer Applikationen unterstützen.

Teile die Jobs auf die Slots S_1 bis S_n auf
Berechne für jeden Slot S_j initiales Schedule mittels RM1-Algorithmus
Berechne die durchschnittliche Terminierungszeit und speichere Kopie:
$term_{av} = 1/n * \sum_{j=1}^{n} term(S_j)$
$term_{save} = term_{av}$
Iteriere k mal:
  Wähle einen Slot S_j mit $term(S_j) > term_{av}$
  Wähle einen Slot S_k mit $term(S_k) < term_{av}$
  Wähle einen Job $i \in S_j$ mit $wi \leq |S_k|$
  Falls für alle Jobs $i \in S_j$ $wi > |S_k|$ ist
    Wähle einen Job $i \in S_j$ mit $skel_i$ ist skalierbar
    Generiere neue Prozessorzahl P_i mit $P_i \leq |S_k|$
  Migriere i von S_j nach S_k
  Berechne neue Schedules für S_j und S_k
  Berechne die neue durchschnittliche Terminierungszeit:
  $term_{av} = 1/n * \sum_{j=1}^{n} term(S_j)$
  Falls $(term_{av} > term_{save})$
    Restore alte Schedules für S_j und S_k

Tabelle 6.14: Scheduling Algorithmus mit Optimierung durch Reskalierung

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde gezeigt, daß sich skelettbasierte Programmierkonzepte im Umfeld des Schedulings paralleler Applikationen sinnvoll einsetzen lassen, um eine effizientere Nutzung von Parallelrechnern zu erreichen. Es wurde dargestellt, wie Schedulingverfahren die Eigenschaften skelettbasierter paralleler Applikationen nutzen können, um mittels Optimierung von Ressourcenanforderungen effizientere Schedules zu generieren oder den Ablauf schon berechneter Schedules zu optimieren. Die weitgehende Skalierbarkeit der hier eingesetzten algorithmischen Skelette ermöglichte es, allein mittels statischer Optimierung die Turnaround-Zeit der Schedules signifikant zu verringern. Durch die Verlagerung der Optimierung vom eigentlichen Schedule auf die Ressourcenanforderungen war es möglich, das statische Optimierungsverfahren so zu gestalten, daß es mit all denjenigen Schedulingverfahren einsetzbar ist, die als Input die Prozessorzahl und die Laufzeit einer parallelen Applikation benötigen.

Eine weitere Verbesserung bei der effizienten Abarbeitung der Schedules wurde durch die Erweiterung der Skelette um Funktionen zum dynamischen Remapping erreicht. Der Einsatz dieses Konzepts erfordert jedoch für optimale Ergebnisse einen Eingriff ins eingesetzte Schedulingverfahren, da geeignete Partner für das Remapping zeitlich direkt nacheinander im Schedule angeordnet sein müssen. Das Verfahren bietet dafür den Vorteil, daß keine Information über die Laufzeit der parallelen Applikation vorhanden sein muß, und dennoch akzeptable Laufzeitverbesserungen erzielt werden können. Es läßt sich unabhängig von der statischen Optimierung einsetzen, bietet aber in Kombination mit dieser eine zusätzliche Steigerung der Effizienz. Die hier vorgestellten Konzepte stehen nicht in direkter Konkurrenz zu anderen Sche-

dulingverfahren, sondern bieten sich vielmehr als Ergänzung für bestehende Verfahren an. Insbesondere die statische Optimierung durch Reskalierung ist hierzu gut geeignet. Sie läßt sich, wie in Abschnitt 5.7 erwähnt, ideal mit dem Schedulingverfahren des Back-Fillings kombinieren oder beim Scheduling auf Netzwerken heterogener Parallelrechner zum Slotting benutzen. Die Vorteile von algorithmischen Skeletten im Scheduling können nur genutzt werden, wenn ein Anwender bereit ist, die angebotenen parallelen Programmierrahmen zu nutzen. Die im Rahmen der Arbeit vorgestellten Skelette reichen sicher nicht aus, um beliebige parallele Applikationen effizient zu implementieren. Ein Anwender wird ein nicht optimales Skelett nur einsetzen, wenn für ihn die Vorteile, wie z.B. eine bevorzugte Behandlung der Applikation oder geringere Nutzungskosten für den Parallelrechner, überwiegen.

7.2 Ausblick

Mit Abschluß dieser Arbeit ist die Thematik Scheduling in Kombination mit algorithmischen Skeletten bei weitem noch nicht ausgeschöpft. Aktuelle Forschungsergebnisse verdeutlichen dies.

Algorithmische Skelette:

Im Bereich algorithmischer Skelette gibt es Bestrebungen, die Akzeptanz skelettbasierter Modellierung zu erhöhen. Ein Trend, weg von speziellen Programmiersprachen und Programmierumgebungen und hin zu etablierten Programmiersprachen und Programmierbibliotheken (Kuchen [Kuc02]) ist zu erkennen. Mit der Definition mehrstufiger Skelette wird versucht, das Einsatzgebiet skelettbasierter Programmierung zu vergrößern (Cole, Kuchen [KC02]). Setzen sich diese Trends durch, so ist mit einer deutlich höheren Prozentzahl an skelettbasierten Anwendungen und damit einer gesteigerten Bedeutung dieses Programmierparadigmas für das Scheduling paralleler Applikationen zu rechnen.

Scheduling von Parallelrechnern:

Seit dem Beginn dieser Arbeit hat sich das Bild des Parallelrechners deutlich gewandelt. Die Leistungsfähigkeit der Prozessorknoten und des Kommunikationsnetzwerks hat sich stark vergrößert. Dafür besitzen heutige Systeme meist eine deutlich geringere Prozessoranzahl als früher. Einzelne Parallelrechner werden nicht mehr als autonome Systeme betrachtet, sondern schon oft im Grid-Computing, (Foster [FK99]) als Teil von weltweiten Metacomputern eingesetzt, um komplexe Berechnungsprobleme zu lösen. Hierbei werden

sie von aktuellen Versionen der Managingsoftware (Reinefeld, Keller [KR01]) unterstützt. Insgesamt gesehen hat die parallele Datenverarbeitung auf lose gekoppelten Rechnernetzen im Laufe der letzten Jahre einen starken Aufschwung bekommen. Konkurrieren mehrere parallele Applikationen um die Rechnerknoten eines solchen Netzwerks, so ist auch hier der Einsatz eines Schedulingverfahrens notwendig (Vadhiyar, Dongarra [VD02]). Da ein zentrales Scheduling bei großen Rechnernetzwerken nicht praktikabel ist, bietet es sich an, das Konzept des dynamischen Remappings auf dieses Themengebiet zu übertragen. Im Bereich des Grid-Computings können algorithmische Skelette zudem eingesetzt werden, um Applikationen auf virtuellen homogenen Parallelrechnern ablaufen zu lassen. Dazu müssen die Skelette so konzipiert und implementiert werden, das Teilberechnungen auf beliebige Parallelrechner des Grids gemappt werden können. Die Kommunikation zwischen den Teilberechnungen und die Synchronisation kann dabei durch Botschaften erfolgen, die als Basis MPI [mpi94] benutzen. Der Vorteil eines solchen Ansatzes besteht darin, daß alle notwendigen Mechanismen zur Ablaufsteuerung einer parallelen Applikation im algorithmischen Skelett integriert sind. Auf jedem Parallelrechner müssen nur einheitliche Strukturen zur Kommunikation und Prozeßplazierung zur Verfügung stehen. Die Homogenisierung der Rechenleistung pro Prozessor wird durch den Einsatz virtueller Prozessoren erreicht. Innerhalb der Skelettimplementationen können die virtuellen Prozessoren so auf die realen Prozessoren abgebildet werden, daß sie auf allen Parallelrechnern vergleichbare Rechenleistungen aufweisen. Mit einer solchen Technik ist es möglich, beliebige Rechnernetzwerke als einen virtuellen, quasi-homogenen Parallelrechner zu nutzen.

Anhang A

A.1 Preemptive Erweiterung von Benutzerfunktionen

Als Beispiel für eine sequentielle Benutzerfunktion sei die Funktion *userfunction* gegeben. Sie besteht aus den 6 sequentiellen Funktionen *func1* bis *func6*. Die Funktionsparameter dieser Funktionen sind für das Beispiel nicht von Bedeutung. Es soll hier gezeigt werden, wie es prinzipiell möglich ist, die Abarbeitung der Berechnung vor der Ausführung der nächsten Teilfunktion zu unterbrechen.

```
userfunction(input,output)
             {;
              func1(input,out1),
              func2(out1,out2),
              func3(out,out1),
              func4(out,out1),
              func5(out,out1),
              func6(out,output)
             }
```

Um die Abarbeitung unterbrechen zu können, werden vor dem Aufruf einer Teilfunktion Zeitwächter eingefügt. Jeder Zeitwächter vergleicht bei seinem Aufruf die aktuelle Zeit mit den Zeitmarken der zugeordneten Schedulingintervalle. Er erhält diese Informationen durch eine Zeitliste, aus der die Systemzeit entnommen werden kann und einer Schedulingliste, die für jede Zeitscheibe die Startzeit und die Endzeit enthält. Ist die Zeitscheibe noch nicht überschritten, so beendet sich der Zeitwächter und gibt damit die Ausführung der Teilfunktion frei. Die von ihm nicht mehr benötigten Zeitlisten werden an den nächsten Zeitwächter übergeben. Ist die aktuelle Zeitscheibe überschritten, so wartet der Zeitwächter solange, bis die nächste

Zeitscheibe beginnt. Wird das Ende der letzten Zeitscheibe überschritten, so terminiert der Zeitwächter die Berechnung der Benutzerfunktion.

```
userfunction(time, timerest, t_i, t_irest, wait,input,output)
            {;
            timeguard(time,  time1 t_i, t_i1, TRUE),
            func1(input,out1),
            timeguard(time1, time2, t_i1, t_i2, TRUE),
            func2(out1,out2),
            timeguard(time2, time3,t_i2, t_i3, TRUE),
            func3(out,out1),
            timeguard(time3, time4, t_i3, t_i4, TRUE),
            func4(out,out1),
            timeguard(time4, time5, t_i4, t_i5, TRUE),
            func5(out,out1),
            timeguard(time5, timerest, t_i5, t_irest, TRUE),
            func6(out,output)
            }
```

Das folgende PCN-Fragment zeigt den detailierten Aufbau des Zeitwächters

```
timeguard(time, out_time, t_i, out_t_i, wait)

   {?
     wait = =TRUE,
      /* Falls Startaufruf oder Zeitwächter in Warterunde
     time = [t|new_time],
      /*  hole neue Zeitmarke aus Zeitliste
     t_i  = [t_start|new_t_i] ->
      /* hole neue Startmarke aus Schedulingliste
               {?
                 t  < t_start ->
                  /* neue Zeitscheibe noch nicht erreicht
                  /* Zeitwächter startet neue Warterunde
                            timeguard(new_time, t_i, TRUE),

                 t >= t_start ->
                 /*  Neue Zeitscheibe ist erreicht
                          {; out_time = new_time,
                 /* Übergabe der Zeitliste
                            out_ti   = new_t_i}
                 /* Übergabe der Schedulingliste
```

```
              /* Zeitwächter beendet sich
            },

 wait == FALSE,
   /* Falls Zeitscheibe noch nicht abgelaufen
time = [t|new_time],
   /* hole neue Zeitmarke aus Zeitliste
t_i  = [t_end|new_t_i] ->
   /* hole neue Schedulingzeitmarke aus Schedulingliste
          {?
            t  < t_end ->
             /* Falls  Ende der Zeitscheibe noch nicht erreicht
                  {; out_time = new_time,
             /* Übergib Zeitliste an nächsten Zeitwächter
                     out_ti = t_i},
             /* Übergib Schedulingliste an Zeitwächter
             /*  Zeitwächter beendet sich

              t >= t_end ,
             /* Zeitscheibe ist überschritten
               new_t_i == [] -> terminate(),
             /* keine weiteren Schedulingzeitmarken vorhanden
             /* Terminierung der Anwenderfunktion

               t >= t_end ,
             /* Zeitscheibe ist überschritten
               new_t_i != [] ->
             /* Schedulingliste enthält noch weitere Zeitmarken
             /* Zeitwächter startet Warterunde

          timeguard(new_time, out_time, t_i, out_t_i, TRUE)
              },
 }
```

A.2 Berechnung der H-Baum Einbettung

Wird dem Gitter eine 2-dimensionale Skalierung zugrunde gelegt, bei dem Nullpunkt (0,0) im Mittelpunkt des Gitters liegt, läßt sich die Abbildung eines Binärbaums mit v Knoten rekursiv wie folgt bestimmen:

- die Wurzel wird dem Nullpunkt (0,0) zugeordnet
- jeder Knoten mit Tiefe d : $(0 \leq d \leq log_2(\frac{v+1}{2}) - 1)$, der sich im Gitter an der Position (a, b) befindet, plaziert seine beiden Söhne an die Positionen

  1. $(a, b + 2^{\frac{1}{2}(log_2(\frac{v+1}{2})-d-2)})$ und $(a, b - 2^{\frac{1}{2}(log_2(\frac{v+1}{2})-d-2)})$,

     falls $log_2(\frac{v+1}{2})$ und d beide gerade oder beide ungerade sind,

     oder
  2. $(a + 2^{\frac{1}{2}(log_2(\frac{v+1}{2})-d-2)}, b)$ und $(a - 2^{\frac{1}{2}(log_2(\frac{v+1}{2})-d-2)}, b)$,

     sonst.

Die Größe des benötigten Gitters zur Einbettung eines vollständigen Binärbaums der Höhe h beträgt $(2^{\lceil \frac{h+1}{2} \rceil} - 1)x(2^{\lceil \frac{h+2}{2} \rceil} - 1)$.

Literaturverzeichnis

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[BD+93] B. Bacci, M. Danelutto, et al. p^3l: A Stuctured High-Level Parallel Language and its Structured Support. Technical Report PHL-PSC-93-55, Pisa Science Center, Hewlett Packard Laboratories, 1993.

[Bra94a] T. A. Bratvold. Parallelising a Functional Program Using a List-Homomorphism Skeleton. In *Proceedings of the First International Symposium on Parallel Symbolic Computation*, pages 44–53, 1994.

[Bra94b] T. A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, 1994.

[CG96] B. Kalthoff C. Geiger. Adaptive Parallelization of Strategies in Agent Based Systems. In *ANZIIS 96*, 1996.

[CGW96] G. Lehrenfeld C. Geiger, B. Kalthoff and A. Weber. Advanced Modeling of Complex Behaviour in Concurrent Systems. In *MPCS'96*, 1996.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley, 1988.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.

[DDM+92] M. Danelutto, R. Di Melio, et al. A Methodology for the Development and the Support of Massively Parallel Programs. *Future Generation Computer Systems*, 8:205–220, 1992.

[DGBL96] X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive Scheduling of parallel jobs on multiprocessors. In *7th SIAM Symp. Discrete Algorithm*, January 1996.

[DGTY95] J. Darlington, Y. K. Guo, H. W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In *Proceedings of Europar*, 1995.

[DT93] J. Darlington and H. W. To. Building Parallel Applications Without Programming. In *Proceedings of the Leeds' Workshop on Abstract Parallel Machine Models 93*, 1993.

[FJ96] D.G. Feitelson and M.A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. Technical Report UCRL-JC-125614, Lawrence Livermore National Laboratory, 1996.

[FK94] I. Foster and C. Kesselman. Language Constructs and Runtime Systems for Computational Parallel Programming. In *FN*, volume LNCS 854, pages 444–555, 1994.

[FK99] I. Foster and C. Kesselman. *The Grid:Blueprint for a New Computing Infrastructure*. Morgan Kaufman Publ., 1999.

[Fos92] I. Foster. Information Hiding in Parallel Programs. Technical Report MCS-P290-0292, Mathematics and Computer Science Division, Argonne National Laboratories, 1992.

[Fos94] I. Foster. Language Constructs for Modular Parallel Programs. Technical Report MCS-P391-1093, Mathematics and Computer Science Division, Argonne National Laboratories, 1994.

[Fos96] I. Foster. Compositional Parallel Programming Languages. *ACM Transactions on Programming Languages and Systems*, 8(1):111–134, 1996.

[FOT92] I. Foster, R. Olson, and S. Tuecke. Productive Programming: The PCN Approach. *Scientific Programming*, 1(1):51–66, 1992.

[FR96] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer Verlag, 1996.

[FRS+96] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–25. Springer Verlag, 1996.

[FT89] I. Foster and S. Tuecke. *Strand – New Concepts in Parallel Programming.* Prentice Hall, 1989.

[FT91] I. Foster and S. Tuecke. Parallel Programming with PCN. Technical report, Argonne National Laboratories, 1991.

[Gib96] R. Gibbons. A Historical Application Profiler for Use by Parallel Schedulers. Technical Report CSRI-TR354, Dept. Computer Science, University of Toronto, 1996.

[GL95] S. Gorlatch and C. Lengauer. $\mathcal{N}$-Graphs: A Topology for Parallel Divide-and-Conquer on Transputer Networks. In B. M. Cook, M. R. Jane, P. Nixon, and P. H. Welch, editors, *Transputer Applications and Systems 95*, pages 396–409. IOS Press, 1995.

[GR96] J. Gehring and F. Ramme. Architecture-independent request-scheduling with tight waiting-time estimations. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science.* Springer Verlag, 1996.

[KC02] H. Kuchen and M. Cole. The Integration of Task and Data Parallel Skeletons. In *Proceedings of CMPP 2002, TU Berlin, Forschungsberichte der Fakultät IV, No. 2002/07, ISSN 1436-9915, pp. 3-16.*, 2002.

[KR01] A. Keller and A. Reinefeld. Anatomy of a Resource Management System for HPC-Clusters. In *Annual Review of Scalable Computing, Vol. 3, Singapore.* University Press, 2001.

[Kuc02] H. Kuchen. *A Skeleton Library.* Technical Report 6/02-I, Angewandte Mathematik und Informatik, University of Münster, 2002.

[mpi94] Mpi: A Message-Passing Interface Standard, 1994.

[Pel93] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, University Pisa, 1993.

[PS95] E.W. Parsons and K.C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 127–145. Springer Verlag, 1995.

[RSD+94] E. Rosti, E. Schmirni, L. W. Dowdy, G. Serazzi, and B.M. Carlson. Robust partitioning schemes of multiprocessor systems. *Performance Evaluation*, 19(2-3):141–165, March 1994.

[SCZL96] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY - LoadLeveler api project. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer Verlag, 1996.

[Sev94] K.C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, March 1994.

[Ste96] A. Steffens. Implementierung und Analyse algorithmischer Skelette für dynamische Scheduling-Verfahren. Technical report, Universität - Gesamthochschule Paderborn, 1996.

[Sto94] H. Stoltze. *Implementierung einer parallelen funktionalen Sprache mit algorithmischen Skeletten zur Lösung mathematisch-technischer Probleme*. PhD thesis, RWTH Aachen, 1994.

[STW95] U. Schwiegelshohn, J. Turek, and J. L. Wolf. Preemptive Scheduling of Parallel Tasks. *IBM Research Reports RC 20104 (88932)*, June 1995.

[UST94] J. L. Wolf U. Schwiegelshohn, W. Ludwig and J. Turek. Smart SMART Bounds for Weighted Response Time Scheduling. *IBM Research Reports RC 19789 (87176)*, July 1994.

[VD02] S. VADHIYAR and J. DONGARRA. A metascheduler for the Grid. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing. To appear.*, 2002.

[Wal93] G.Dueck; T.Scheuer; H.-M. Wallmeier. Toleranzschwelle und Sintflut: neue Ideen zur Optimierung. *Spektrum der Wissenschaft*, 3:42–51, 1993.

[Wei92] M. A. Weiss. *Data Structures and Algorithm Analysis*. The Benjamin/Cummings Publishing Company, 1992.