Beiträge zur Modellierung von Parallelrechnern

Dissertation
zur Erlangung des Grades
„Doktor der Naturwissenschaften“

am Fachbereich Physik
der Johannes Gutenberg-Universität
in Mainz

Alexander Ebbes
aus Mainz

Mainz, im Februar 2000

Datum der mündlichen Prüfung: 28. Juli 2000

Inhaltsverzeichnis

Kapitel 1

Einführung: Wozu eine Modellierung paralleler Rechner?

Parallelrechner stellen an sich keine neuartige Technologie dar. Bereits Anfang der 1960er Jahre, als die theoretischen und praktischen Grundlagen der heutigen Computerwissenschaften geschaffen wurden, wurden Parallelrechner entwickelt. Die vorliegende Arbeit stellt Beiträge zur Modellierung dieser Parallelrechner vor.

Schon Illiac-IV, dessen Entwicklung 1963 begann, umfasste 64 Prozessoren und 64 Fließkomma-Rechenwerke. Illiac-IV entstand in einem groß angelegten Forschungsprojekt der Universität von Illinois, in dem Hardware-Technologien, Betriebssysteme, Programmiersprachen und Algorithmen entwickelt wurden. Im Laufe der Untersuchungen kristallisierten sich vor allem zwei Problembereiche heraus: Erstens erwies es sich trotz der speziell an Illiac-IV angepassten Programmiersprache als schwierig, den Rechner effizient zu programmieren. Zweitens wuchsen die Hardware-Entwicklungskosten weit über das projektierte Maß hinaus, weshalb das Projekt gleichermaßen als inhaltlich wegweisend und wirtschaftlich fehlgeschlagen bewertet wurde.

Die Schwierigkeiten bei der Hardware-Konstruktion konnten schon bald gelöst werden. Der Übergang von Spezialhardware zu Standardbauteilen erlaubte, Innovationszyklen zu verkürzen und Produktionskosten zu reduzieren. Die Fragestellungen zu Algorithmenentwurf und Programmierung sind jedoch bis heute aktuelle Forschungsgegenstände auf dem Gebiet der Parallelrechner.

Denn obwohl seit den Zeiten von Illiac-IV viele parallele Algorithmen und Programmiersprachen entwickelt wurden, besteht auch heute noch die folgende Diskrepanz: Einerseits werden Parallelrechner in vielen, insbesondere technischen Bereichen wie der Crash-Simulation mittels der Finite-Elemente-Methode erfolgreich in der Praxis eingesetzt. Auf den Rechnern läuft meist hochgradig problemspezifische Software, die in klassischen sequentiellen Programmiersprachen wie C oder Fortran programmiert und durch spezielle Betriebssystemaufrufe oder Kommunikationsschnittstellen nur ansatzweise parallelisiert ist. Andererseits werden in der akademischen Forschung anspruchsvolle parallele Algorithmen für grundlegende mathematische Problemstellungen entwickelt, die sich auf real exisitierender Hardware jedoch nicht effizient implementieren lassen.

Eine der Ursachen dieser offensichtlichen Diskrepanz sind die unterschiedlichen Modellvorstellungen der akademischen Algorithmen-Entwickler und der industriellen Programmierer. Das gängigste parallele Rechnermodell der Informatik, das PRAM-Modell, wurde mit dem Ziel entworfen, effiziente parallele Algorithmen entwerfen zu können. Und in genau diesem Einsatzbereich ist das Modell auch erfolgreich. Der praktische Anwender eines Parallelrechners ist aber nicht an algorithmischer

Effizienz, sondern vielmehr an möglichst großer Ausführungsgeschwindigkeit interessiert. Diese beiden Gütekriterien lassen sich nur schwerlich aufeinander abbilden, da algorithmischer Aufwand nach Anzahlen von Schritten bewertet, Ausführungszeit jedoch in Sekunden gemessen wird. Die für eine Umrechnung erforderlichen Informationen sind im PRAM-Modell nicht enthalten. Als Beispiele für diese Informationen seien die genauen Zugriffsmuster auf den Adressraum oder die verschiedenen Ausführungszeiten der einzelnen Operationen genannt.

Diese Arbeit versteht sich als Beitrag zur Modellierung von Parallelrechnern. Dabei wird ein Parallelrechner als makroskopisches physikalisches dynamisches System mit einer sehr großen Anzahl von Freiheitsgraden, einem diskretem Zustandsraum und diskreter Zeit betrachtet. Derartige Systeme werden von der Nichtlinearen Dynamik untersucht. Wie aktuelle Veröffentlichungen in *Physical Review E* oder *Physica D* zeigen, hat sich in letzter Zeit das Verständnis des Begriffes „physikalisches System“ verbreitert. So werden nicht nur biologische Systeme abgehandelt, sondern sogar auch hybride Systeme wie der Straßenverkehr, bei dem physikalische Gesetzmäßigkeiten mit psychologisch und physiologisch bestimmtem Verhalten zusammenspielen.

Jede modellmäßige Behandlung derartiger Systeme muss sich auf bestimmte, dem Ziel und dem Zweck der Untersuchung angepasste Aspekte beschränken. Die Umsetzung der allgemeinen Vorstellungen und des konkreten Wissens in ein Modell, das sich mathematisch behandeln lässt, erfordert Fachkompetenz und Erfahrung. Diese kann nur gewonnen werden, indem man konkrete Systeme bzw. Systemklassen modelliert und sich der dabei getroffenen Entscheidungen bewusst wird.

Der in dieser Arbeit vorgestellte Beitrag zur Modellierung paralleler Rechner soll so zweierlei Zielen entgegenkommen. Zum einen wird ein Modell kritisch untersucht und weiterentwickelt, das dazu dienen soll, die Ausführungszeit eines konkreten parallelen Programmes auf einem konkreten Parallelrechner brauchbar vorherzusagen. Zum anderen soll die Untersuchung eines konkreten Problems aus dem Bereich von Computerwissenschaft und -technik dazu genutzt werden, neues Licht auf das allgemeine Modellierungsproblem zu werfen, ein tieferes Verständnis für das zu modellierende System zu entwickeln und daraus neue Aspekte für die Modellierung dynamischer Systeme im Allgemeinen zu gewinnen.

Die Untersuchung entspricht einem interdisziplinären Ansatz, in dem einerseits Inhalte der Computerwissenschaften und andererseits Grundmethoden der experimentellen Physik verwendet werden. Die Vorhersagen der abstrakten Modelle werden nämlich mit den experimentell gewonnen Ergebnissen von Messungen an realen Systemen verglichen. Auf dieser Basis wird u. a. gezeigt, dass der hierarchische Aufbau des Speichers Einflüsse von mehreren Größenenordnungen auf die Ausführungsgeschwindigkeit einer Anwendung ausüben kann. Das im Rahmen der vorliegenden Arbeit entwickelte Modell der einzelnen Rechenknoten eines Parallelrechners gibt diese Effekte innerhalb eines relativen Vorhersagefehlers von nur wenigen Prozent korrekt wieder.

Die vorliegende Arbeit gliedert sich wie folgt: In Kapitel 2 wird ein kurzer Überblick über das Spektrum bestehender Parallelrechner-Architkturen gegeben. Ebenso werden die gängigsten Modelle für paralles Rechnen vorgestellt, insbesondere das

PRAM- und das LogP-Modell. In Kapitel 3 wird auf das LogP-Modell näher eingegangen. Es werden diejenigen Aspekte eines Parallelrechners dargestellt, die im LogP-Modell eine Entsprechung finden und diejenigen, die bei der Modellierung ausgeklammert wurden. Nach diesen rein theoretischen Überlegungen werden in Kapitel 4 die Grundannahmen des LogP-Modells einem Praxistest unterzogen und das Verhalten eines Anwendungsprogrammes auf einem konkreten Parallelrechner mit den Modellvorhersagen verglichen. Bei diesen Untersuchungen stellt sich heraus, dass die Dynamik der einzelnen Rechenknoten einen großen Einfluss auf die Gesamtleistung des Systems hat. Da das LogP-Modell keine Darstellung der Abläufe in den einzelnen Rechenknoten enthält, wird in Kapitel 5 die Entwicklung eines Speicherhierarchie-Simulators vorgestellt. Auch die Vorhersagen dieses Simulatos werden in Experimenten mit den Ausführungszeiten einer Testanwendung auf dem Testrechner verglichen. Aus den resultierenden guten Übereinstimmungen zwischen Theorie und Experiment wird in Kapitel 6 ein Fazit über die Effektivität der entwickelten Modelle und Modellerweiterungen gezogen und ein Überblick über plausible Weiterentwicklungen der vorgestellten Methoden und Modelle gegeben. Kapitel 7 schließt die Arbeit mit einer Zusammenfassung der Ergebnisse ab.

Kapitel 2

Ausgangspunkt: Einordnung und theoretische Grundlagen der Modellierung paralleler Rechner

In diesem Kapitel werden einige Grundlagen der Modellierung von Parallelrechnern vorgestellt. Nach einer Vorbemerkung zu den Schwierigkeiten, den Begriff „Parallelrechner“ exakt zu definieren, wird ein Überblick über die prinzipiellen Parallelrechner-Architekturen gegeben, wobei sich die Verbindungsnetze als zentrale Komponenten herausstellen. Nach einem kleinen Beitrag zu den grundlegenden Strukturen dieser Verbindungsnetze und ihrer Behandlung werden mit dem PRAM- und den LogP-Modell die zwei gängigsten Parallelrechner-Modelle vorgestellt.

2.1 Zur Definition des Begriffes „Parallelrechner“

Es erweist sich als schwierig, den Begriff „Parallelrechner“ exakt zu definieren. Der Theoretiker Leighton setzt in seinem Standardwerk „Parallel Algorithms and Architectures“ [Leig92] die Architektur eines Parallelrechners mit seiner graphentheoretischen Struktur gleich. Dabei unterscheidet er insbesondere Gitter, Bäume und Hyperkuben. Das eher auf praktischen Einsatz und technische Realisierung ausgerichtete Lehrbuch „Parallel Computer Architecture“ [Cull99] von Culler, Singh und Gupta enthält sich einer Definition und verweist seinerseits auf die Definition

> A *parallel computer* is a „collection of processing elements that communicate and cooperate to solve large problems fast.“

von Almasi und Gottlieb [Alma94]. In dieser Definition bleiben viele offensichtliche Fragen, bspw. nach der Anzahl der Prozessoren, der Art der Kommunikation oder der inneren Organisation des Parallelrechners, unbeantwortet.

Im Rahmen dieser Arbeit werden unter „Parallelrechnern“ diejenigen Maschinen verstanden, die von anerkannten Herstellern wie Cray, HP, IBM, SGI oder SUN als Parallelrechner verkauft werden. Dennoch ließen sich viele, wenn auch nicht alle, der durchgeführten Untersuchungen und gewonnenen Ergebnisse beispielsweise auf Netzwerke von Workstations oder Linux-PCs übertragen.

2.2 Ebenen der Parallelität

Parallelverarbeitung findet sich in einem modernen Computer auf den verschiedensten Ebenen. Gängige Mikroprozessoren sind in der Lage, mehrere Operationen pro Takt auszuführen, Speicherbusse und Cache-Zwischenspeicher können mehr als eine

Anfrage gleichzeitig bearbeiten und in Multitasking-Betriebssystemen können mehrere Programme zeitlich versetzt, auf Multiprozessoren sogar gleichzeitig, ausgeführt werden.

Für die Definition von drei gängigen Ebenen sei Abschnitt 1.3 des Buches „Prozessorzuteilung in Parallelrechnern“ [Heis94] von Heiss zitiert: „Parallelität ist ein sehr allgemeines Prinzip. Es kann auf unterschiedlichen Ebenen und dadurch in sehr unterschiedlicher Weise in Rechensystemen eingesetzt werden.“:

Programmebene: „Ein komplexer Auftrag an ein Rechensystem, der die Ausführung mehrerer Programme zur Folge hat, wird in sich parallel ausgeführt, z. B. paralleles `make` in UNIX.“

Prozessebene: „Programmstücke, als Prozesses organisiert, arbeiten an einer gemeinsamen Aufgabe parallel, z. B. Parallelisierung von Schleifen.“

Anweisungsebene: „Anweisungen einer höheren Programmiersprache werden in sich parallel abgearbeitet, z. B. die Auswertung eines arithmetischen Ausdruckes.“

Insbesondere die Parallelität auf Anweisungsebene ist ein sehr aktuelles und vielversprechendes Gebiet für den Einsatz von Parallelverarbeitung in PCs und Workstations. In dieser Arbeit werden jedoch lediglich Parallelitäten auf der Programm- und Prozessorebene behandelt, da sich diese zwischen den einzelnen Prozessoren abspielen und damit dem Grundkonzept von Parallelrechnern entsprechen.

2.3 Parallelrechner-Architekturen

Aus technischer Sicht bieten sich mehrere Schemata zur Klassifikation von Parallelrechnern an. Meist wird unterschieden, ob der Rechner über einen von allen Rechenknoten gemeinsam genutzten oder einen verteilten Speicher (*Shared-Memory* versus *Distributed-Memory*) verfügt. Diese Aufteilung wird auch in Abbildung 2.1 dargestellt. Alternativ bietet sich eine Unterscheidung in über Busse oder über Netzwerke verbundene Prozessoren an. Beide Schemata implizieren Auswirkungen des technischen Aufbaus auf die Programmierung.

Distributed-Memory

Ein Distributed-Memory-Parallelrechner besteht aus mehreren Rechenknoten, die über ein Verbindungsnetz gekoppelt sind. Die Rechenknoten sind ihrerseits aus Prozessoren, Caches und Speichermodulen aufgebaut. Heiss bezeichnet einen solchen Parallelrechner als Multirechnersystem.

Da in einem solchen Rechner ein Prozessor nicht direkt auf die in anderen Knoten gespeicherten Daten zugreifen kann, er über *Message-Passing* programmiert werden. Dabei kommunizieren die Knoten, indem sie sich über das Verbindungsnetz Nachrichten zusenden. Aktuelle Beispiele für Distributed-Memory-Rechner sind die IBM SP2 oder jedes Workstation-Cluster.

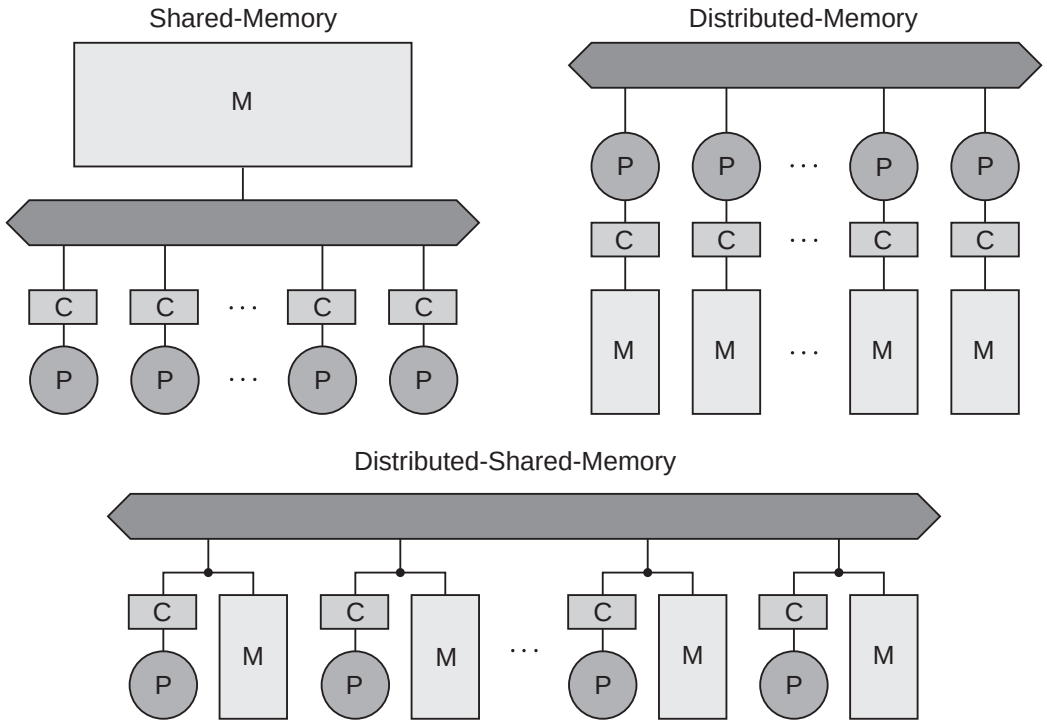

Abbildung 2.1: **Grundlegende Parallelrechner-Architekturen** – In dieser Abbildung sind die gängigsten Parallelrechner-Architekturen dargestellt. Obwohl jede der Architekturen als Elemente Prozessoren (**P**), Caches (**C**), Speichermodule (**M**) und ein Verbindungsnetz enthält, unterscheiden sie sich in der Struktur.

Shared-Memory

In einem Shared-Memory-Parallelrechner sind die verschiedenen Prozessoren über einen Bus oder ein Verbindungsnetz an einen gemeinsamen Speicher angeschlossen. Jeder der Prozessoren kann auf jede der Speicherstellen durch direkte Adressierung zugreifen.

Wenn auch Daten, die von einem Prozessor aus dem Hauptspeicher in seinen Cache kopiert und dort verändert wurden, von allen anderen Prozessoren korrekt gelesen werden können, so bezeichnet man den gemeinsamen Speicher als cache-kohärent (*cache-coherent*). Der SUN Enterprise-Server ist ein Vertreter der Shared-Memory-Parallelrechner.

Distributed-Shared-Memory

Eine Mischform aus Shared- und Distributed-Memory-Parallelrechnern stellen die Distributed-Shared-Memory-Rechner (DSM) dar. Sie bestehen wie Distributed-Memory-Rechner aus Rechenknoten, die ihrerseits Prozessoren, Caches und Hauptspeicher enthalten. Zusätzlich enthalten sie jedoch spezielle Hardware, die es einem

Prozessor erlaubt, auf den Speicher eines anderen Knotens durch direkte Adressierung zuzugreifen.

Parallelrechner mit dieser Fähigkeit bezeichnet man auch als NUMA-Rechner (für *Nonuniform Memory Access*), da die Zugriffszeit auf den Speicher nicht konstant ist, sondern für eine gegebene Adresse von der Entfernung zwischen Prozessor und Speicher abhängt.

Ist in einem NUMA auch der Zugriff auf entfernte Speichermodule cache-kohärent, so bezeichnet man die Architektur als ccNUMA. ccNUMA-Rechner lassen sich sowohl über Message-Passing- als auch über Shared-Memory-Methoden programmieren. Der Cray[1] T3E ist ein NUMA-, die SGI Origin oder Convex Exemplar sind ccNUMA-Parallelrechner.

2.4 Verbindungsnetze als Graphen

Alle in Abbildung 2.1 dargestellten Parallelrechner-Architekturen enthalten als zentrale Komponente ein Verbindungsnetz. Um die Struktur dieser Verbindungsnetze zu beschreiben, bedient man sich üblicherweise graphentheoretischer Begriffe. Einige grundlegende graphentheoretische Definitionen sollen hier kurz aufgeführt werden, für eine ausführliche Darstellung muss auf die Standardliteratur [Tura96], [Knut97.1], [Mesc72] verwiesen werden.

Ein *ungerichteter Graph* G wird aus einer Menge V von *Knoten* und einer Menge E von *Kanten* gebildet: $G = (V, E)$. Eine Kante wird durch ein ungeordnetes Paar von Knoten, ihre Endknoten, definiert. In graphischen Darstellungen werden Knoten üblicherweise durch Punkte und Kanten durch Verbindungslinien dargestellt. Eine solche graphische Repräsentation bestimmt eindeutig einen Graphen, wohingegen sich aus einem gegebenen Graphen keine eindeutige graphische Darstellung ableiten lässt. Es folgen einige gängige Definitionen, die sich bei der Charaktisierung von Verbindungsnetzen als nützlich erweisen:

Nachbarschaft: Zwei Knoten u und v eines ungerichteten Graphen heissen *benachbart*, wenn es eine Kante von u nach v gibt. Ein Knoten u ist *inzident* mit einer Kante e desselben Graphen, wenn u ein Endknoten von e ist.

Vollständigkeit: Ein Graph heisst *vollständig*, wenn jedes Knotenpaar (u, v) durch eine Kante verbunden ist: $e = (u, v)$.

Knotengrad: Der *Knotengrad*, auch kurz *Grad*, eines Knotens v in einem ungerichteten Graphen ist die Anzahl der Kanten, die mit v inzident sind.

Kantenzug: Eine Folge von Kanten $e_1, e_2, \ldots, e_n$ eines Graphen G heisst *Kantenzug*, falls es eine Folge von Knoten $v_0, v_1, \ldots, v_n$ von G gibt, so dass für $j = 1 \ldots n$ gilt $e_j = (v_{j-1}, v_j)$. Ein Kantenzug heisst *Weg*, wenn alle verwendeten Kanten verschieden sind.

[1] Inzwischen haben sowohl die Firmen SGI und Cray als auch HP und Convex fusioniert. Dabei wurden die Produkte im Allgemeinen umbenannt. In dieser Arbeit beziehen sich architektonische Betrachtungen auf die ursprünglichen Bezeichnungen.

Abstand: Der *Abstand* $d(u, v)$ zweier Knoten u und v ist definiert als die minimale Anzahl von Kanten eines Weges mit Anfangsknoten u und Endknoten v. Der *Durchmesser* eines Graphen ist der größte, zwischen zwei beliebigen Knoten vorkommende Abstand.

Bei aktuellen Parallelrechnern sind Kommunikationswege üblicherweise bidirektional ausgelegt. Daher können als Graphen entweder die beschriebenen ungerichteten Graphen oder gerichtete Graphen verwendet werden, bei welchen alle Kanten doppelt, d. h. einmal pro Richtung, ausgelegt sind. In dieser Arbeit wird, wie in der Literatur üblich, mit ungerichteten Graphen argumentiert.

Um die Eignung eines gegebenen Graphen als Verbindungsnetz für Parallelrechner beurteilen zu können, werden üblicherweise folgende Ziele verwendet:

Kleiner Durchmesser: Es ist zu erwarten, dass die Kommunikationsleistung zwischen zwei Knoten sinkt, wenn der Abstand zwischen ihnen wächst. Daher sollte der Graph eines Verbindungsnetzes einen möglichst kleinen Durchmesser aufweisen.

Große Bisektionsweite: Gegeben ein Graph G aus n Knoten, der durch einen Schnitt in zwei gleichgroße Teilgraphen G' und G'' zerlegt wird, die jeweils $n/2$ Knoten enthalten. Durch den Schnitt verläuft eine Anzahl von Kanten. Die Bisektionsweite ist die minmale Anzahl von Kanten, die bei einem solchen Schnitt auftreten kann.

In parallelen Programmen kommuniziert häufig eine Hälfte der Prozessoren mit der anderen Hälfte. Die Bisektionsweite gibt an, wieviele Kanten für diese Kommunikation mindestens zur Verfügung stehen. In technisch orientierten Ausführungen findet man statt der Bisektionsweite häufig die Bisektionsbandbreite, definiert als das Produkt aus Bandbreite der Kommunikationskante und Bisektionsweite.

Langsam wachsender Knotengrad: Um einen Verbindungsgraphen technisch umsetzen zu können, darf der Grad der Knoten nicht zu schnell mit der Knotenanzahl wachsen. Letztlich sind Realisierungen in ihrem Knotengrad üblicherweise beschränkt, so dass einige der Kanten eines Knotens ungenutzt bleiben, solange die Prozessoranzahl kleiner ist als die maximale Knotenanzahl des Verbindungsnetzes für den gegebenen Knotengrad.

Schließlich sollen noch einige populäre Verbindungsgraphen angesprochen und in ihrer Eignung für Parallelrechner bewertet werden. Zum Vergleich stellt Abbildung 2.2 (S. 10) diese Graphen dar:

Vollständiger Graph: Der vollständige Graph stellt bezüglich des Durchmessers und der Bisektionsweite den idealen Graphen dar. Der linear mit der Knotenanzahl wachsende Knotengrad ist jedoch nur für kleine Knotenanzahlen technisch realisierbar.

Crossbar: Der Crossbar ist eine technische Möglichkeit, die Funktionalität eines vollständigen Graphen zu realisieren. Im Gegensatz zum „direkten“ Verbindungsnetz des vollständigen Graphen ist der Crossbar ein „indirektes“ oder „mehrstufiges“ Verbindungsnetz, da die Rechenknoten nur über Zwischenknoten

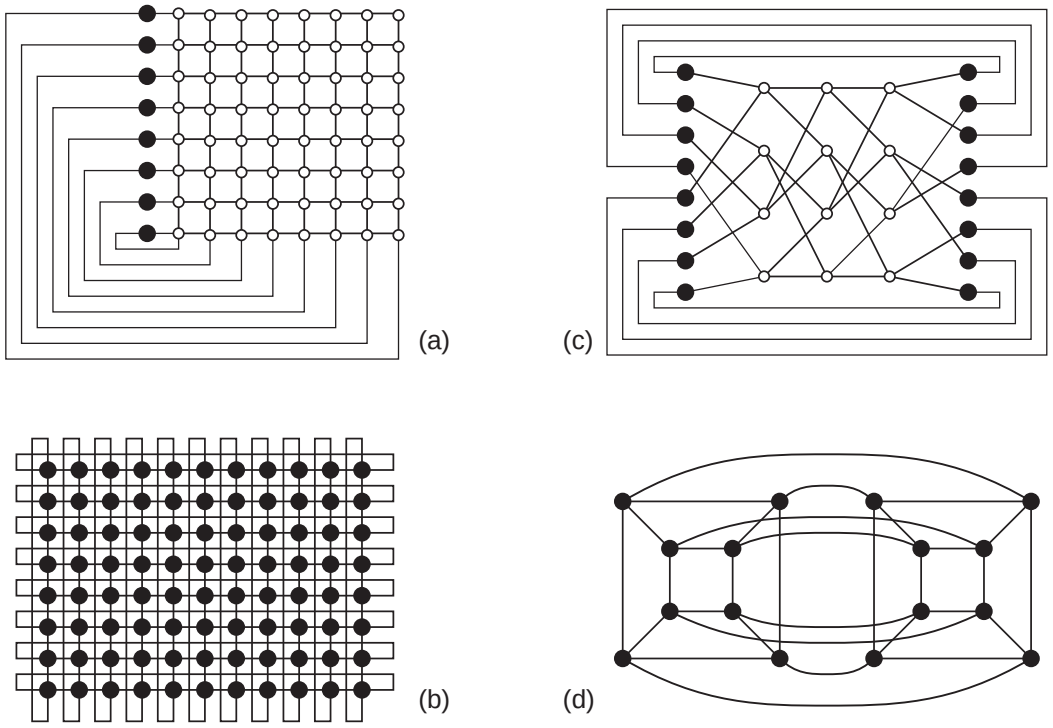

Abbildung 2.2: **Grundlegende Verbindungsgraphen** – Die Abbildung zeigt (a) den Crossbar, (b) den Torus, (c) das Omega-Netzwerk und (d) den Hyperkubus. (Die schwarzen Kreise symbolisieren Rechen-, die weissen Router-Knoten. Bei den mehrstufigen Verbindungsnetzen wurden die Ausgänge mit den Eingängen verbunden.)

verbunden sind. Der Knotengrad dieser, meist Router oder Switch genannten Zwischenknoten ist zwar begrenzt, die Anzahl dieser Zwischenknoten wächst jedoch quadratisch mit der Anzahl der Rechenknoten. Der SUN Enterprise-Server verwendet eine Hierarchie von Crossbars als Verbindungsnetz [Char98].

Gitter und Tori: Gitter werden häufig als Verbindungsgraphen gewählt, da ihr Knotengrad konstant ist. Allerdings wachsen beispielsweise im populären zweidimensionalen (quadratischen) Gitter aus n Knoten sowohl Durchmesser als auch Bisektionsweite mit $O(\sqrt{n})$. Da als Bezugspunkt meist logarithmisches Wachstum gewählt wird, gilt dieser Durchmesser als stark wachsend. Jedoch erhält das Gitter als direktes Verbindungsnetz die Nachbarschaft der Knoten, so dass sich algorithmische Kommunikationsoptimierungen zwischen benachbarten Knoten ausnutzen lassen. Unter einem Torus versteht man ein Gitter, bei dem zusätzlich die ersten und letzten Elemente einer Zeile (und Spalte) miteinander verbunden sind. Die Convex Exemplar ist ein Beispiel für einen zwei-, der Cray T3E für einen dreidimensionalen Torus [HP98a], [SGI99a].

Omega, Banyan oder Butterfly: Mehrstufige Verbindungsnetze werden, wie Gitter oder Tori, eingesetzt, um den Knotengrad gering zu halten. Die P Prozessoren werden dabei im Allgemeinen durch $\log_f P$ Ebenen von Routerkno-

ten mit Knotengrad $2f$ verbunden. Die Routerknoten werden dabei meist als Crossbars mit f Ein- und f Ausgängen realisiert. Typische Werte von f betragen 2 bis 8. Die Bisektionsweite dieser Mehrstufennetze steigt zwar linear bei lediglich logarithmisch wachsendem Durchmesser, jedoch lassen sich Nachbarschaftsoptimierungen eines Algorithmus in einem solchen Netz nicht ausnutzen. Populäre Vertreter dieses Graphentyps sind Omega-, Banyan- oder Butterfly-Netze (vgl. [Leig92]). Der „SP-Switch“ der IBM RS/6000-SP2 implementiert ein Omega-Netzwerk [IBM99a], [Ager95].

Hyperkubus: Der Hyperkubus ist ein besonders geeigneter Verbindungsgraph für Parallelrechner, da es sehr viele effiziente parallele Algorithmen auf Hyperkuben gibt und sich viele andere Graphen unter Erhaltung der Nachbarschaft in Hyperkuben einbetten lassen. Sowohl Abstand als auch Durchmesser des Hyperkubus wachen mit $O(\log_2 n)$, die Bisektionsweite sogar mit $O(n)$. Der Hyperkubus wird beispielsweise in der SGI Origin als Verbindungsgraph verwendet [SGI97a], [Laud97].

2.5 Gängige Parallelrechner-Modelle

Eine Modellierung und damit der Blickwinkel, unter welchem man die zu untersuchenden Systeme betrachtet, ist stark von den Zielen des Betrachters abhängig. Die jeweils verwendeten Modelle sollen sich dadurch auszeichnen, diesen Zielen insofern entgegenzukommen, dass sie die gewünschten Beobachtungen, Erkenntnisse und Aussagen ermöglichen.

Um sowohl die theoretischen Eigenschaften als auch die mit reellen Parallelrechnern gewonnenen Ergebnisse zu verstehen, wurden von verschiedenen Forschern sehr verschiedene Modelle entwickelt. Drei dieser Modelle sollen im Folgenden vorgestellt werden: das PRAM-, das BSP- und das LogP-Modell.

2.5.1 Das PRAM-Modell

Die Informatik interessiert sich insbesondere für Algorithmen, die gegebene Probleme lösen. Um ein Maß für die Güte eines Algorithmus zu erhalten, wird die Abhängigkeit seiner Laufzeit und seines Speicherverbrauches von der Größe der Eingabe analysiert. Wenn mehrere Algorithmen als Kandidaten zur Lösung eines Problems zur Verfügung stehen, so soll diese Analyse denjenigen mit der besten Effizienz identifizieren.

Um einen Algorithmus analysieren zu können, ist ein Modell einer ihn ausführenden Maschine notwendig. Dieses Modell legt die strukturellen und quantitativen Eigenschaften sowie die Grenzen der Maschine fest, so dass es möglich ist, die durchführbaren Operationen zu definieren und sie mit einer Kostenfunktion zu belegen. Erst durch diese Kosten wird es möglich, die Effizienzen der zum Vergleich stehenden Algorithmen zueinander in Beziehung zu setzen.

Die sequentielle Random Access Machine – RAM

Ein sehr erfolgreiches Modell der Informatik ist die *Random Access Machine* oder RAM. Diese Maschine, welche die wesentlichen Aspekte eines realen Computers widerspiegeln soll, besteht aus einem Programm, das auf eine Datenstruktur angewendet wird. Die Datenstruktur einer RAM ist ein Feld von Registern, welche ganze Zahlen enthalten. Dabei kann das Feld unendlich viele unendlich große Einträge enthalten. Das Programm ist eine endliche Folge $\Pi = (\pi_1, \ldots, \pi_n)$ von Instruktionen, welche die Maschinenbefehle von Computerprozessoren abstrahieren (READ, ADD, STORE, JUMP usw.), deren Argumente Zahlen aus dem Registerfeld sind. Zusätzlich besitzt die RAM einen Befehlszähler, welcher es ermöglicht, den Kontrollfluss auszudrücken: In jedem Berechnungsschritt führt die RAM die durch den Befehlszähler referenzierte Instruktion aus, wobei die Argumente so aus den Registern gelesen und in sie geschrieben werden, wie die Instruktion es jeweils erfordert. Die Eingabe einer RAM besteht aus einer endlichen Folge ganzer Zahlen $I = (i_1, \ldots, i_m)$ welche in einem weiteren endlichen Feld, dem Eingabefeld, gespeichert sind. Ein ebensolches Ausgabefeld O nimmt die Ergebnisse der Berechnung auf. Dabei werden alle Berechnungsschritte in einem zeitlichen Einheitstakt abgeschlossen. Für eine detailliertere Beschreibung dieser RAM-Darstellung sowie der damit möglichen Untersuchungen sei beispielsweise auf die Arbeiten von Papadimitriou [Papa94] oder Cormen et. al. [Corm90] verwiesen.

In einer Beschreibung, die sich eher an den Gegebenheiten eines konkreten Computers orientiert, besteht eine RAM aus einem idealisierten Prozessor, welcher mit einem (unendlich großen) Speicher verbunden ist. Die Daten können aus diesem Speicher gelesen und in ihn geschrieben werden; die Instruktionen werden ausgeführt.

Die parallele RAM – PRAM

Die rein sequentiell arbeitende RAM, in welcher alle Instruktionen hintereinander und in deterministischer Reihenfolge ausgeführt werden, lässt sich gradlinig auf mehrere parallel arbeitende Prozessoren erweitern: Es ergibt sich die *Parallel RAM* oder *PRAM* [Fort78]. Ein PRAM-Programm $P = (\Pi_1, \ldots, \Pi_q)$ besteht aus q RAM-Programmen, jeweils eines für die q parallel arbeitenden RAMs. Damit das globale Datenfeld von allen q Prozessoren gelesen und beschrieben werden kann, wird der Befehlssatz um entsprechende Instruktionen erweitert. Die RAMs sind innerhalb der PRAM durchnumeriert; ein weiterer Befehl erlaubt es jeder der RAMs ihre eigene Nummer zu ermitteln, um damit ihre auszuführenden Algorithmen zu parametrisieren. Wie in der sequentiellen RAM benötigen alle Instruktionen aller Prozessoren denselben zeitlichen Einheitstakt. Damit arbeiten die Prozessoren auf Instruktionsebene synchron. Um die Behandlung gleichzeitig stattfindender lesender und schreibender Zugriffe auf dieselbe Speicherzelle zu klassifizieren unterscheidet man PRAMs in *EREW*, *ERCW*, *CREW* und *CRCW*[2].

[2] In diesem Schema stehen die Buchstaben R für Lesen und W für Schreiben, das E bezeichnet exklusiven und das C gemeinsamen Zugriff. Somit steht bspw. *CREW* für „Concurrent Read, Exclusive Write“. Die Kollisionen bei einem gemeinsamen Schreibzugriff werden eine weitere Unterteilung in *common*, *arbitrary*, *priority* und *combining* adressiert [Leig92], [Jaja92].

In der konkreten Darstellung wiederum besteht ein PRAM aus q idealen RAM-Prozessoren, die im Gleichtakt arbeiten und parallel an einen globalen gemeinsamen Speicher anschlossen sind. Optional besitzt jeder der Prozessoren zusätzlich einen lokalen Speicher.

Ein wesentliches Anwendungsgebiet für das PRAM-Modell ergibt sich aus der Möglichkeit, dafür Algorithmen zu schreiben. Diese Algorithmen werden in der üblichen Notation verfasst, wobei insbesondere die Schleifenkonstrukte um eine parallele Variante erweitert sind: Der Programmabschnitt innerhalb des Konstruktes

forall $i \in M$ **pardo** ...

wird zeilenweise synchron von allen Prozessoren parallel ausgeführt. An einem solchen Algorithmus kann nun eine Einteilung in eine parallele Klasse und eine Aufwandsabschätzung vorgenommen werden.

Für den theoretisch interessierten Informatiker ist das PRAM ein erstklassiges Modell eines Parallelrechners, da es seinen Zielen entgegenkommt, indem es von den Widrigkeiten der Umsetzung in eine reale Maschine soweit abstrahiert, dass eine Behandlung auf der Ebene von Algorithmen und asymptotischer Aufwandsabschätzung möglich wird. Leighton [Leig92] bringt dies auf den Punkt:

> „The PRAM model unburdens the parallel algorithm designer from having to worry about wiring and memory organisation issues, thereby allowing him or her to focus on abstract parallelism."

Zur Umsetzung in Hardware

Beim Umsetzen des PRAM-Modells in konkrete Hardware erweist sich jedoch gerade der Punkt „wiring and memory organisation" als höchst problematisch. Das PRAM-Modell verlangt, q Prozessoren gleichberechtigt an einen globalen Speicher anzuschließen. Dieser Speicher ist ein idealer paralleler Speicher, welcher einen Zugriff von jedem Prozessor in jedem Zyklus bearbeiten kann, selbst dann, wenn mehrere Zugriffe auf dieselbe Speicherzelle stattfinden. Alle Speicherzugriffe und alle Synchronisationen werden in einem Zyklus erfüllt. Zusätzlich ist die Prozessoranzahl q eine Funktion $q = q(|x|) : \mathbf{N} \to \mathbf{N}$, die in Abhängigkeit von der Größe der Eingabe x eine Anzahl aktiver Prozessoren bestimmt. Während die Tatsache, dass q im allgemeinen leicht berechenbar ist, meist durch ein Polynom, für einen Informatiker eine gutmütige Klassifikation darstellt, stellt eine von der Größe der Eingabe abhängige Prozessoranzahl, selbst bei lediglich polynomialer Abhängigkeit, ein in der Praxis nicht zu verwirklichendes Konstruktionsziel dar.

2.5.2 Das BSP-Modell

Wie schon in Abschnitt 2.5.1 dargestellt wurde, sind die Synchronizität der Prozessoren und der Einheitstakt, in dem jede mögliche Operation abgeschlossen werden kann zwei wesentliche Eigenschaften des PRAM. Aufgrund dieser formalen Eleganz wurde das PRAM zu einem leistungsfähigen und beliebten Modell für den Entwurf paralleler Algorithmen in der Informatik.

Für den Computer-Architekten ergeben sich jedoch gerade bei der Realisierung dieser eleganten Eigenschaften Probleme. Da die Programmabarbeitung durch mehrere unabhängige Prozessoren an einem globalen Speicher aufgrund der Kollisionen bei gleichzeitigem Lesen und Schreiben im Allgemeinen nicht im Einheitstakt erfolgt, werden die Prozessoren ganz automatisch ihre Synchronizität verlieren.

Diese Synchronizität auf Granularität der einzelnen Instruktionen zu erzwingen, beispielsweise durch einen gemeinsamen globalen Takt, würde die Leistung des Gesamtsystems stark senken – und ist vor allem auch gar nicht notwendig.

Lokale Parallelität – Supersteps

Das *Bulk-Synchronous Parallel Model* [Vali90], oder kurz BSP-Modell, unterteilt das parallele Programm in eine Folge von sogenannten *Supersteps*. In jedem dieser Supersteps führt jeder Prozessor lokale Berechnungen und endlich viele Kommunikationsoperationen mit anderen Prozessoren durch. Von dieser Kommunikation wird implizit angenommen, dass sie über das Versenden und Empfangen von Nachrichtenpaketen realisiert wird. Die Supersteps sind durch globale Barrieren voneinander getrennt und alle Kommunikationsoperationen sind zeitlich bezüglich des Supersteps lokal, müssen also abgeschlossen sein, bevor die globale Barriere zum nächsten Superstep überschritten werden kann.

Da in einem realen parallelen Rechner die Kommunikation über Nachrichten deutlich länger dauert, als eine auf lokalen Daten basierende Rechenoperation, müssen die Supersteps ausreichend lang gewählt werden, um eine effiziente Laufzeit zu erhalten.

Dies forciert die für die Praxis so wesentliche Überlappung von Kommunikation und Berechnung bereits in der Phase des algorithmischen Entwurfs, statt sie in die Implementierungsphase zu verlagern, in welcher die Effizienz dann typischerweise weniger exakt überprüft wird.

Die Parameter des BSP

Ein BSP-Computer besteht aus mehreren Prozessoren mit lokalem Speicher und einem Verbindungsnetzwerk zwischen allen diesen Prozessoren. Das Verbindungsnetzwerk sollte die globale Barriere zur Trennung der Supersteps bereitstellen. Die Leistung dieses BSP-Computers wird durch die folgenden drei Parameter ausgedrückt:

p : Die Anzahl der Prozessoren.

g : Das Verhältnis der Rechenleistung der Prozessoren zur Kommunikationsleistung des Verbindungsnetzwerkes.

L : Die minimale Zeitdifferenz zwischen zwei aufeinanderfolgenden globalen Barrieren.

Dabei wird die Rechenleistung in durchgeführten Operationen pro Zeiteinheit und die Kommunikationsleistung in übertragenen Datenworten pro Zeiteinheit ausgedrückt. Die Zeitdifferenz wird ggf. in Anzahlen von Rechenschritten umgerechnet.

Das BSP-Modell schwächt die starken Restriktionen des PRAMs auf die ausführende Maschine deutlich ab. Insbesondere die Synchronizität der parallelen Prozessoren auf der Ebene der Einzelschritte wurde durch das Konzept der Supersteps abgelöst.

Da die Parameter p, g und L des Modells die Leistungscharakteristik der ausführenden Maschine beschreiben, wurde ein Rahmen geschaffen, in welchem sich bestehende Programme an eine konkrete Hardware anpassen können. Alternativ lässt sich die Leistungsfähigkeit einer gegebenen Hardware in BSP-Parametern erfassen. Damit ist das BSP-Modell deutlich näher an einen realen Computer gerückt, als beispielsweise das PRAM-Modell, ohne sich jedoch in einer Vielfalt von Parametern und Kenngrößen zu verlieren. Dadurch bleibt die Möglichkeit zur Entwicklung und analytischen Bewertung von Algorithmen erhalten.

2.5.3 Das LogP-Modell

Während sowohl die PRAM als auch das BSP Modelle der Informatik sind, wurde das LogP-Modell [Cull93] von Computer-Architekten entworfen. Es ist die ausdrückliche Intention hinter diesem Modell, aktuelle Parallelrechner und darauf laufende Programme darstellen und quantitativ in ihrer Leistung vorhersagen zu können.

Die Parameter des LogP

Das LogP-Modell beschreibt die Leistungscharakteristik eines abstrakten Computers in vier Parametern – L, o, g und P – daher der Name.

L : Der Parameter L ist die obere Schranke der *Latenz*. Die Latenz ist die zeitliche Verzögerung zwischen dem Absenden einer Nachricht und ihrem Eintreffen beim Empfänger.

o : Der *Overhead* o beschreibt die Rechenzeit, welche von einem Prozessor zum Senden oder Empfangen einer Nachricht über das Verbindungsnetzwerk aufgewendet werden muss.

g : Die Bandbreite des Verbindungsnetzwerkes wird definiert über das *Gap* g. Dieses Gap ist die minimale zeitliche Differenz zwischen zwei aufeinanderfolgenden Nachrichtenpaketen.

P : Die Anzahl der Prozessoren wird durch P beschrieben.

Während im BSP die Kommunikation und Synchronisierung noch über den implizierten Austausch von Nachrichten stattfand, wird dies im LogP-Modell explizit betont. Die einzelnen Computer des LogP agieren solange autark und asynchron, bis sie auf das Eintreffen einer Nachricht warten müssen. Selbst das Senden der Nachrichten darf im allgemeinen asynchron stattfinden. Damit erweitert das LogP-Modell den Aspekt der Asynchronizität deutlich über das bisherige Maß hinaus.

Damit stellt das LogP ein Modell für die Kommunikation, nicht jedoch für die eigentliche Berechnung eines parallelen Programmes dar. Die Rechenleistung geht lediglich grob über den Parameter o in die Gesamtleistung des Modells ein.

Abbildung bestehender Hardware und Software

Es wurden bereits zahlreiche Arbeiten veröffentlicht, in welchen die LogP-Parameter für kommerzielle oder akademische Parallelrechner bestimmt wurden. Ebenso wurden parallele Algorithmen verschiedener praktischer Anwendungsfelder wie FFT, LU-Dekomposition sowie verschiedene Sortierverfahren mit diesen Größen parametrisiert [Cull93], [Cull96].

Dabei wurde gezeigt, dass das LogP generell in der Lage ist, das Verhalten eines parallelen Algorithmus auf einer gegebenen parallelen Hardware vorherzusagen.

2.5.4 Zwischenbewertung

In den vorangegangenen Abschnitten wurden drei verschiedene Modellierungsansätze für Parallelrechner betrachtet und charakterisiert. Diese Modelle unterscheiden sich deutlich in ihrem Anwendungsfeld und ihrem Abstraktionsgrad.

Das PRAM-Modell dient dem Entwurf paralleler Algorithmen sowie der Analyse ihrer Korrektheit und Leistungsfähigkeit. Sein hoher Abstraktionsgrad macht es zu einem wertvollen Werkzeug der theoretischen Informatik. Da sich ein PRAM effizient auf Verbindungsnetzwerken, beispielsweise Hyperkuben, simulieren lässt, gilt die geringe Realitätsnähe des zugrundeliegenden Maschinenmodells als irrelevant.

Das BSP-Modell versucht, die Restriktionen des PRAMs bezüglich eben dieses Maschinenmodells zu lockern und führt Asynchronizität in das Zusammenspiel der Einzelprozessoren ein. Zusätzlich versucht das BSP, über die Aussagen asymptotischer Aufwandsabschätzungen hinaus, quantitative Vorhersagen zu ermöglichen. Die Art und Anzahl der Modellparameter lässt dabei lediglich grobe Prognosen zu.

Das LogP-Modell ist ein Modell für das Design und die Bewertung konkreter Maschinen. Es dient weniger dem Entwurf von Algorithmen als der Behandlung konkreter Programme. Dabei ist das Maschinenmodell stark an aktuelle kommerziellen Parallelrechnern angelehnt. Dadurch erlaubt das LogP eine Vorhersage des Verhaltens eines Programms auf einer gegebenen Hardware in Abhängigkeit von den Modellparametern.

Es wäre nicht angemessen, die verschiedenen Modelle in ihrer prinzipiellen Leistungsfähigkeit zueinander in Bezug zu setzen, da sie in sehr verschiedenen Anwendungsfeldern angesiedelt sind. Vielmehr sollten sie als Beispiele dafür angesehen werden, wie das angestrebte Einsatzfeld die Eigenschaften des zugeordneten Modells bestimmt.

Kapitel 3

LogP: Ein spezielles Parallelrechner-Modell

In Kapitel 2 wurde ein Überblick über das Spektrum von Parallelrechner-Architekturen gegeben und das PRAM- dem LogP-Modell gegenüber gestellt.

In den Einleitungen der Orginalarbeiten zum LogP-Modell [Cull93], [Cull96] wird betont, der gängige PRAM-Formalismus ermögliche Zugänge für parallele Algorithmen, die auf kostenfreier Kommunikation, zeitsynchroner Berechnung und paralleler Behandlung einzelner Datenworte aufbauen. Dadurch werde die Entwicklung in der Praxis schlecht einsetzbarer Parallelisierungsverfahren gefördert (vgl. [Cull93], „Introduction“).

Dieses Kapitel diskutiert das LogP-Modell kritisch bezüglich seines eigenen Anspruches als realistisches Parallelrechnermodell. Abschnitt 3.1 beschreibt die Marktsituation von Parallelrechnern zum Zeitpunkt der Modellbildung. Diejenigen Aspekte paralleler Rechner, die die LogP-Autoren in ihr Modell aufgenommen haben, werden in Abschnitt 3.2 aufgeführt, während 3.3 aktuelle Forschungsgegenstände aufzeigt, die in der LogP-Modellierung ausgeklammert wurden. In Folgearbeiten haben sich bestimmte Modellerweiterungen ausgebildet, die in Abschnitt 3.4 dargestellt werden. Abschnitt 3.5 zieht ein Fazit aus der Struktur des LogP-Modells.

3.1 Entwicklung von Technik und Marktsituation als Kontext der Modellbildung

Die Bildung des LogP-Modells war stark von der Entwicklung konkreter, auf dem Markt erhältlicher Parallelrechner beeinflusst. In diesem Abschnitt sollen diejenigen Aspekte von Technik und Marktsituation beschrieben werden, die den Kontext der Modellierung bildeten.

3.1.1 Komplette Computer plus Verbindungsnetze

Die Originalarbeiten zum LogP-Modell [Cull93], [Cull96] beziehen sich auf einen grundlegenden Wandel in der Architektur gängiger Parallelrechner am Anfang der 1990er Jahre. Dieser vollzog sich, ausgehend von großen parallelen Einzelcomputern, bestehend aus einer Vielzahl fein miteinander vermaschter Funktionseinheiten, hin zu Ansammlungen von Rechenknoten, die nur über ein Verbindungsnetz grobgranular miteinander verbunden waren.

Der Aufbau dieser Knoten entspricht dabei weitgehend demjenigen von Workstations, die aus denselben Bauteilen zusammengesetzt sind, indem ein möglichst lei-

stungsfähiger[1] Prozessor über Cache-Zwischenspeicher mit einem großen Hauptspeicher verbunden ist. Ein Netzwerkadapter ermöglich die Ankopplung an das Verbindungsnetz.

In der Terminologie von Heiss [Heis94] entspricht dies dem Übergang von Multiprozessor- zu Multirechner-Systemen. Diese Entwicklung hängt eng mit dem Übergang zu *Off-the-Shelf*-Komponenten zusammen.

3.1.2 Off-the-Shelf-Komponenten statt Spezialbausteinen

Das große Marktvolumen für Computer der Workstation-Klasse hat zu einer beschleunigten Entwicklung ihrer Leistungsfähigkeit geführt. Mikroprozessoren erhöhen im Mittel ihre Rechenleistung um 50 bis 100 Prozent pro Jahr[2]. Hauptspeicher vervierfachen ihre Kapazität etwa alle drei Jahre. Die hierfür erforderlichen Entwicklungsarbeiten können nur über Gewinne durch die großen Stückzahlen bei kleinen bis mittelgroßen Systemen finanziert werden.

Der Markt für kommerzielle Parallelrechner ist sehr viel kleiner als der für Workstations und Server. Daher sind nahezu alle Hersteller solcher Maschinen dazu übergegangen, parallele Computer aus den leistungsfähigsten Standardbauteilen für Einzelprozessor- oder kleine Multiprozessor-Maschinen aufzubauen[3]. Insbesondere die verwendeten Mikroprozessoren und Speicherbausteine dieser Parallelrechner wurden gewöhnlich für deutlich kleinere Server und Arbeitsplatzrechner entwickelt. Daher wurde der Begriff *Off-the-Shelf-Supercomputer* geprägt.

Die verwendeten Verbindungsnetze sind jedoch weiterhin Spezialanfertigungen, da sich die aus dem Bereich der lokalen Netzwerke (LAN) stammenden Technologien nicht oder nur sehr begrenzt als Kommunikationsträger für verteilte Berechnungen eignen. Gründe dafür sind neben den geringen Bandbreiten vor allem die hohen Latenzzeiten von LAN-Netzwerken und ihrer Anbindung an die Rechenknoten. Daher verwendet jeder der genannten Hersteller spezielle Eigenentwicklungen, um seine aus Standardkomponenten aufgebauten Knoten miteinander zu verbinden.

[1] Die Leistungsfähigkeit der Prozessoren und die Größe der Hauptspeicher soll hier besonders betont werden, um den Kontrast zu früheren Parallelrechnern zu verdeutlichen. Dort wurden möglichst viele sehr einfache Prozessoren eingesetzt – die berühmte *Connection-Machine* CM-2 hatte 65536 1-Bit-Prozessoren – und die Größe des Hauptspeichers pro Prozessor war gering.

[2] Hennessy und Patterson [Henn96] nennen eine Verbesserung der CPU-Rechenleistung um einen Faktor 1,55 pro Jahr seit 1987 (davor 1,35) und einen Technologiewechsel bei der Fertigung dynamischer RAM-Bausteine alle drei Jahre. Dabei wird ein Technologiewechsel mit einer Kapazitätszunahme um einen Faktor vier gleichgesetzt. Die Zugriffszeiten (Latenz) von DRAMs verbessert sich gleichzeitig lediglich um einen Faktor 1,07 pro Jahr.

[3] Dies beschreibt die parallelen Rechner von IBM, Silicon Graphics, Hewlett-Packard, SUN, Intel und sogar die massiv parallelen Systeme von Cray (jetzt Silicon Graphics) und Convex (jetzt Hewlett-Packard). Für Vektorrechner trifft diese Beschreibung nicht zu, aber auf Vektorrechner wird im Rahmen dieser Arbeit nicht weiter eingegangen.

3.1.3 Grobgranulare Parallelität

Aufgrund der als kostenfrei[4] angenommenen Kommunikation und der Synchronizität der Einzelprozessoren fördert der PRAM-Formalismus die Entwicklung sehr feingranularer Algorithmen. Im Idealfall steht für die Bearbeitung eines jeden Datenwortes ein eigener Prozessor zur Verfügung. Da keine Maschinen existierten, die diese feine Granularität realisieren konnten, mussten die LogP-Autoren ein Modell entwerfen, in dem die Anzahl parallel arbeitender Prozessoren klein gegenüber der Größe der Eingabe[5] sein konnte. Dies spiegelt die Tatsache wieder, dass Parallelrechner bis heute zwar bis zu mehreren Gigabyte (10^9) oder gar Terabyte (10^{12}) Hauptspeicher, aber lediglich bis zu einigen Tausend (10^3) Prozessoren aufweisen.

3.1.4 Message-Passing als Programmiermethodik

Das LogP-Modell wurde speziell zur Beschreibung großer paralleler Systeme mit bis zu Tausenden von Prozessoren konzipiert (vgl. [Cull93]). Damals waren die bis zu großen Prozessoranzahlen skalierbaren Maschinen durchgehend Multirechner-Systeme mit verteiltem Speicher, deren Kommunikation über Nachrichtenaustausch (*Message-Passing*) stattfand.

Dementsprechend wurde LogP als ein Modell für Message-Passing-Kommunikation entworfen. Da sich noch keine konkrete Schnittstelle oder ein spezielles Protokoll aus der Vielzahl der akademischen und herstellereigenen Lösungen herausgebildet hatte, macht das LogP-Modell über die explizite Programmierung keine Annahmen.

3.2 Bei der Modellierung berücksichtigte Aspekte

Culler, Patterson et. al. haben in den Original-Publikationen zum LogP-Modell [Cull93] und [Cull96] diejenigen Aspekte eines Parallelrechners identifiziert, welche nach ihrer Meinung den wesentlichen Einfluß auf die Leistung des Gesamtsystems ausüben.

3.2.1 Modellvorstellungen von Aufbau und Funktionsweise

Da seine Autoren die Entwicklung von Marktsituation und Technik als Kontext ihrer Modellbildung beachteten, enthält das LogP-Modell bestimmte Grundannahmen über Aufbau und Funktionsweise eines Parallelrechners. Teilweise wurden diese Annahmen bereits in Kapitel 2 angesprochen, um den Vergleich mit dem PRAM- und dem BSP-Modell zu ermöglichen. Dennoch werden werden sie hier erneut angeführt, um eine zusammenhängende Darstellung er ermöglichen.

[4] Im Kontext dieser Untersuchung bezeichnet der Kostenbegriff meist Zeiten, insbesondere Lauf-, Warte- und Ausführungszeiten, sowie Speicherplatzbedarf.

[5] Der Begriff „Größe der Eingabe“ stammt aus der Informatik und bezeichnet die Anzahl der Elemente der Eingabedaten bzw. den durch sie verbrauchten Speicherplatz.

Komplette Computer: Prozessoren und Speicher

Wie bereits erwähnt, besteht ein Parallelrechner im Sinne des LogP-Modells aus Rechenknoten, die über ein Verbindungsnetz gekoppelt sind. Die einzelnen Rechenknoten bestehen aus kompletten Computern nach dem jeweils aktuellen Stand der Technik. Dies bedeutete damals wie heute, dass ein Mikroprozessor über Cache-Zwischenspeicher an einen Hauptspeicher angeschlossen ist.

Dies wird in der Modellbeschreibung [Cull93] und [Cull96] explizit erwähnt. Dennoch wird dieser Aspekt nicht in das Modell aufgenommen. Typ und Funktionsweise der verwendeten Prozessoren werden ebensowenig modelliert wie Kapazität und Bandbreite der verwendeten Speicher und Zwischenspeicher.

Verbindungsnetze: Netzwerkadapter und Netzwerke

Neben Massenspeichern wird insbesondere ein Netzwerkadapter als Teil eines jeden Rechenknotens betrachtet. Durch diesen Adapter wird der Knoten über ein Verbindungsnetz mit den restlichen Knoten verbunden.

Als weitere Rahmenbedingung wird angenommen, dass die Leistungsfähigkeit von Netzwerken deutlich langsamer wächst, als die von Prozessoren. Dies wird mit dem relativ geringeren Marktvolumen für Netzwerktechnik begründet (vgl. [Cull93], „Technological Motivations“).

Auch über die Details des Verbindungsnetzes wie beispielsweise Topologie oder Routing-Verfahren macht das LogP-Modell keine Annahmen. In Abschnitt 3.3 wird auf explitzit und implizit bei der Modellierung ausgesparte Aspekte des Verbindungsnetzes weiter eingegangen.

Nachrichtenaustausch: Paketvermittlung

Da in den betrachteten Parallelrechnern die Netzwerkadapter über den Peripheriebus an die jeweiligen Knoten angeschlossen wurden, gab es keine Möglichkeit, über eine Kopplung der Speicherbusse einen cache-kohärenten gemeinsamen Adressraum aufzubauen. Daher ergab sich als Kommunikationsmethode nur der Nachrichtenaustausch.

3.2.2 Zugang über L, o, g und P

Das LogP-Modell bezieht seinen Namen aus den vier Parametern L, o, g und P, mittels derer es die Leistungscharakteristik eines parallelen Rechners definiert.

Bei den Definitionen dieser Parameter gilt die Grundannahme, dass es eine kleine natürliche Wortbreite des Netzwerkes gibt, welche im Wesentlichen einem Datenwort entspricht.

Die Latenz L : Der Parameter L ist definiert als die obere Schranke der *Latenz*. Die Latenz charakterisiert die zeitliche Verzögerung zwischen dem Absenden einer Nachricht und ihrem Eintreffen beim Empfänger.

Der Overhead o : Der *Overhead o* beschreibt die Rechenzeit, die von einem Prozessor zum Senden oder Empfangen einer Nachricht über das Verbindungsnetzwerk aufgewendet werden muß. Die Overheads für Senden und Empfangen werden als gleich groß angenommen.

Das Gap g : Das *Gap* ist als die minimale zeitliche Differenz zwischen zwei aufeinanderfolgenden Nachrichtenpaketen definiert. Verrechnet mit der Wortbreite des Netzwerks lässt sich aus dieser Zeitdifferenz die inverse Bandbreite bestimmen.

Die Prozessorzahl P : Im LogP-Modell ist die Anzahl P der Prozessoren eine Konstante. Darin unterscheidet sich das Modell von beispielsweise dem PRAM, in dem die Prozessorzahl eine Funktion, meist ein Polynom, der Eingabegröße ist.

Korrekturglieder

Zusätzlich basiert das Modell auf der Annahme, dass das Netzwerk eine begrenzte Kapazität hat, so dass maximal L/g Nachrichten gleichzeitig vermittelt werden können. Versucht ein Prozessor eine Nachricht über ein bereits gesättigtes Netzwerk zu verschicken, so muss er solange warten, bis von einem anderen Prozessor ein Paket entnommen wurde.

Ebenso wird die Annahme gemacht, dass in dem Fall, dass n Prozessoren an ein und denselben Prozessor senden, die Bandbreite des Netzwerkes bei langen Nachrichtensequenzen auf $1/n$ sinkt.

Die Details dieser korrigierenden Annahmen sind in den Originalarbeiten [Cull93], [Cull96] aufgeführt.

Kommunikation über Request und Reply

Zum Zeitpunkt der Formulierung des LogP-Modells hatte sich noch keine Kommunikationsschnittstelle als Standard durchgesetzt. Hinter der Wahl der Modellparameter steht jedoch die implizite Annahme, dass die Kommunikation einer Struktur folgt, die den ebenfalls von Culler stammenden *Active Messages* zugrunde liegt (vgl. [Eick92], [Cull96b]).

Ein wesentliches Konstruktionsmerkmal von Active-Messages besteht darin, dass jede Kommunikation durch das Senden und Empfangen von Nachrichtenpaketen umgesetzt wird. Zum Lesen ist eine Anfrage (*Request*) mit einer Adresse notwendig, auf die eine Antwort (*Reply*) folgt, welche das adressierte Datum enthält. Beim Schreiben enthält der Request das zu schreibende Datum und der Reply lediglich eine Bestätigung (*Acknowledge*).

In der separaten Veröffentlichung [Cull96b] haben Culler und Mitarbeiter eine Messvorschrift für die LogP-Modellparameter vorgestellt. Dort wird auch detaillierter auf die Annahmen über den Ablauf eines Nachrichtenaustausches eingegangen.

3.3 Bei der Modellierung ausgeklammerte Aspekte

Das LogP-Modell wurde mit betont minimalistischem Anspruch entworfen. Dabei wurden bestimmte Aspekte und technische Details explizit bei der Modellierung ausgeklammert. Andere Parameter und Charaktieristika des Systemverhaltens wurden kommentarlos ausgeschlossen.

In diesem Abschnitt wird eine Auswahl von Aspekten dargestellt, welche aktuelle Forschungsgegenstände auf dem Gebiet der Parallelrechner sind, die aber im Rahmen des LogP-Modells keine Beachtung finden. Diese Auswahl erhebt keinen Anspruch auf Vollständigkeit.

Nach Beleuchtung der einzelnen Sachverhalte wird nachstehend als Beleg für ihre Relevanz auf die Originalarbeiten verwiesen, in denen die jeweiligen Aspekte, meist isoliert und selten in Verbindung mit dem LogP-Modell, untersucht werden.

3.3.1 Topologie des Verbindungsnetzwerkes

In der Informatik werden Algorithmen üblicherweise speziell für eine bestimmte Topologie eines Verbindungsnetzes entworfen. Das Standardwerk „Parallel Algorithms and Architectures“ von Leighton [Leig92] besteht beispielsweise aus den drei Kapiteln „Arrays and Trees“, „Meshes and Trees“ und „Hypercubes and related Architectures“, wobei die Kapitelüberschriften gleichermaßen die Strukturen der Algorithmen wie die Topologien der Netze bezeichnen.

Diese enge Kopplung von algorithmischer und Verbindungs-Topologie stammt implizit aus dem PRAM-Formalismus und der damit verbundenen wortweisen Granularität der Parallelität. Die Autoren des LogP halten diese Granularität für nicht realistisch (siehe auch Abschnitt 3.1.3). Nach eigener Aussage hielten sie es zusätzlich für nachteilig, einen Algorithmus an eine spezielle Topologie anzupassen, da die Stabilität seiner Leistung darunter leide (vgl. [Cull93], „Technological Motivations“).

Mathematische Strukturen: Baum, Gitter, Hyperkubus

Außer beim Algorithmen-Entwurf spielt die Topologie auch beim Hardware-Design eine wesentliche Rolle. Kriterien wie die Abhängigkeit der Bandbreite und der Bisektionsbandbreite (vgl. Abschnitt 2.4) von der Prozessoranzahl müssen gegen die Kosten abgewogen werden, die durch die Anzahl und innere Komplexität der zu verwendenden Bauelemente entstehen.

Abschnitt 1.3.1, „Scalable Interconnection Networks“, im Buch von Lenoski [Leno95] enthält eine Darstellung verschiedener Topologien, der aus ihnen resultierenden Leistungscharakteristiken sowie Verweise auf konkrete Maschinen, die die jeweilige Topologie implementieren.

Technische Strukturen: Speicher- oder Peripheriebus?

Das LogP-Modell setzt von der zu modellierenden Maschine nur voraus, dass die Rechenknoten komplette Computer sind, die über einen Netzadapter an ein Verbin-

dungsnetz angeschlossen sind. Wie diese Verbindung technisch realisiert wird, stellt keinen Teil der Modellierung dar.

Zum Zeitpunkt der Formulierung des LogP-Modells (1993) wurde das Verbindungsnetz üblicherweise über einen Peripheriebus angeschlossen. Mittlerweile kommen zunehmend Maschinen auf, bei denen die Anbindung über den Speicherbus geschieht. Dieser Unterschied hat nicht nur Einfluss auf die erreichbaren Bandbreiten und Latenzzeiten, sondern ermöglicht auch die Errichtung eines cache-kohärenten gemeinsamen Adressraumes.

Abbildung von Software- auf Hardware-Topologien

Parallele Algorithmen verwenden implizit oder explizt eine Kommunikationstopologie. Da diese Algorithmen im Allgemeinen nicht immer auf der gleichen Maschinentopologie ausgeführt werden, müssen die jeweiligen Topologien aufeinander abgebildet werden. Beispielsweise basieren viele Algorithmen, bspw. zum Suchen oder Sortieren, auf Baumstrukturen. In modernen Rechnern finden sich jedoch meist keine Bäume, sondern mehrdimensionale Gitter oder Hyperkuben.

Die Arbeit von Loh et. al. [Loh96] untersucht die Auswirkungen der Topologie am Beispiel dynamischer Algorithmen zur Lastbalancierung. Dabei werden für Gitter, Hyperkuben und einen Fibonacci-Kubus verschiedene Effekte untersucht.

3.3.2 Prozessorzuteilung

Unter Prozessorzuteilung versteht man die Zuordnung der vorhandenen oder freien Prozessoren zu den einzelnen parallel ablaufenden Prozessen.

Prozessorzuordnung ist ein umfangreiches eigenständiges Themengebiet innerhalb des Forschungsbereiches paralleler Systeme. Das Spektrum reicht von statischer Partitionierung bis zu Verfahren für dynamische Lastbalancierung. Eine deutschsprachige Einführung in dieses Gebiet findet sich im Buch „Prozessorzuteilung in Parallelrechnern“ von Heiss [Heis94].

Der Artikel „Is an Alligator Better Than an Armadillo?“ [Lisz97] von Liszka et. al. ist eine Abhandlung über die Konsequenzen der Prozessorzuteilung anhand von in modernen Verbindungsnetzen auftretenden Effekten. Dabei stellt er auch Zugänge zu Analysen dar, die aufzeigen, wie gut die Verbindungstopologie zu einem gegebenen Algorithmus passt.

3.3.3 Stauungen im Verbindungsnetz

Bei der Kommunikation eines parallelen Programms werden im Allgemeinen Stauungen und sonstige Wartezustände im Verbindungsnetz auftreten. Diese Stauungen haben verschiedene Ursachen und treten unterschiedlich in Erscheinung.

Hot-Spots

Zustände punktweiser Überbelastung nennt man häufig *Hot-Spots*. Siehe dazu die Arbeiten von Dandamudi [Dand99] „Reducing Hot-Spot Contention in Shared-Memory Multiprocessor Systems“ über gemeinsamen und Liszka [Lisz97] über verteilten Speicher. Hot-Spots treten beispielsweise auf, wenn auf eine für die Berechnung wichtige globale Variable von vielen Prozessoren zeitgleich zugegriffen wird.

Contention und Congestion

Contention bezeichnet allgemein einen Zustand, in dem so viele Prozessoren dieselbe Ressource anfordern, dass diese über ihre maximale Kapazität hinaus belastet wird. Damit bezeichnet Contention einen Wettstreit um diese kritische Ressource. Der Wettstreit verursacht eine Stauung. Diese wiederum bezeichnet man an als *Congestion*[6].

Um das Verbindungsnetz trotz Contention und Congestion funktionsfähig zu halten, werden meist Warteschlangen eingefügt. Die Literatur (vgl. [Hrom97], [Leig92]), enthält ausführliche Analysen für obere Schranken der Warteschlangenlängen bei speziellen Kombinationen von Algorithmen und Topologien.

Moritz et. al. haben in ihrer Arbeit „LoGPC: Modeling Network Contention in Message-Passing Programs“ [Mori98] das LogP-Modell um eine Darstellung von DMA[7]-basierter Übertragung und Stauungen im Verbindungsnetz erweitert und behaupten, dass die Effekte dieser Stauungen wesentlichen Einfluss (bis zu 50%) auf die Gesamtausführungszeit eines parallelen Programmes ausüben können.

Tree Saturation

Eine besondere Form von Contention, die insbesondere in mehrstufigen Verbindungsnetzen wie Butterfly- oder Omega-Netzwerken auftritt, nennt sich *Tree Saturation*.

Zur Definition von Tree-Saturation sei auf das Lehrbuch von Culler [Cull99], S. 760 verwiesen:

> „If the total amount of traffic destined for [an hot spot] exceeds the bandwidth of the output, this traffic will back up within the network. If this condition persists, the backlog will propagate backward through the tree of channels directed at this destination which is called *tree saturation*. Any traffic that crosses the tree will also be delayed.“

Insbesondere in einem mehrstufigen Verbindungsnetzwerk verdient Tree-Saturation besondere Aufmerksamkeit, da sie nicht, wie normale Contention, nur einen einzelnen Knoten oder eine einzelne Kante belastet, sondern umfassende Teile des gesamten Netzes betreffen kann (vgl. [Pfis85]).

[6] Die Begriffe Congestion und Contention werden häufig nicht sauber voneinander getrennt verwendet. Insbesondere praktisch orientierte Arbeiten verwenden den einen oder anderen Begriff für sowohl den Wettstreit als auch die Stauung.

[7] „**direct memory access (DMA)** A mechanism that provides a device controller the ability to transfer data directly to or from the memory without involving the processor“. [Patt98], „Glossary“.

3.3.4 Routing-Verfahren

In der akademischen Literatur zu parallelem Rechnen [Leig92], [Jaja92] finden sich vielfältige Routing-Verfahren, welche jeweils auf bestimmten Verbindungstopologien bei bestimmten Algorithmen besonders geeignet sind.

Technisch orientierte Arbeiten beziehen sich meist auf Unterschiede in der Pufferverwaltung (*Cut Through*- versus *Store and Forward*-Routing) oder der tabellengestützten Routenfindung sowie auf die Möglichkeiten der Implementierung. Die Literatur hierzu ist extrem vielfältig. Daher seien als Überblick nur die Kapitel 10.6, „Routing“, bis 10.8, „Flow Control“, des Buches von Culler [Cull99], sowie das Netzwerk-Buch von Tanenbaum [Tane96] angegeben. Für den technisch orientierten Leser sei noch auf die technischen Ausführungen von Galles [Gall96] zum SPIDER-Router der in Kapitel 4 eingesetzten SGI Origin hingewiesen.

3.3.5 Speicherhierarchien der Rechenknoten

Obwohl es in den Originalarbeiten nicht ausdrücklich betont wird, handelt es sich bei LogP primär um ein Modell des Kommunikationsverhaltens eines Parallelrechners. Das Verhalten der eigentlichen Rechenknoten geht nur als Seiteneffekt über den Overhead-Parameter o ein.

Darin drückt sich die Annahme aus, dass das Modell an die Rechenleistung eines konkreten Computers angepasst werden könne.

In einem modernen Rechner gängiger Bauart hängt die Ausführungszeit eines Programmes jedoch nur sekundär von der reinen Rechenleistung des Prozessors ab. Dieser Abschnitt stellt dar, warum die Leistungsfähigkeit des Speichersubsystems einen wesentlich größeren Einfluß auf die resultierende Gesamtleistung ausübt.

Übergang zu RISC-Prozessoren

Etwa 1980 begann eine deutliche Umstrukturierung der Instruktionssätze von Prozessoren. Fortschritte in der Compiler- und Speichertechnik ermöglichten den Übergang zu einfach strukturierten Befehlsformaten, die sich effizient mit neuen Implementierungsmethoden (insbesondere *Pipelining*) umsetzen ließen. Nach dem Pilotprojekt der Universität von Berkeley fasst man diese bis heute aktuellen Architekturen unter dem Begriff RISC-Prozessoren (*Reduced Instruction Set Computer*) zusammen.

Siehe Kapitel 2, „Instruction Set Principles and Examples“, bei Hennessy [Henn96] oder Kapitel 3, „Instructions: Language of the Machine“, bei Patterson [Patt98] hinsichtlich einer Zusammenfassung des Übergangs zu RISCs, zu den Methoden des Pipelinings und zu den ersten akademischen und kommerziellen CPUs.

RISC-Prozessoren als Load-Store-Architektur

Die im betrachteten Kontext wesentliche Eigenschaft von RISC-Prozessoren ist ihre sogenannte *Load-Store*-Architektur. Der Name erklärt sich dadurch, dass die

Operanden einer auszuführenden Instruktion nicht durch Indizierung direkt im Adressraum angesprochen werden können. Statt dessen müssen die Eingabedaten durch explizite *Load*-Befehle aus dem Speicher in Register geladen und das Ergebnis durch einen *Store*-Befehl aus einem Register zurück in den Speicher geschrieben werden. Die eigentliche Rechenoperation kann nur auf Operanden in Registern angewandt werden.

Die Load-Store-Architektur eines RISC-Prozessors hat, wie in Kapitel 4, „Untersuchung: Das LogP-Modell im Praxistest“, beschrieben, wesentlichen Einfluss auf die Ausführungszeit konkreter Berechnungsprozesse.

Das Processor-Memory-Performance-Gap

Nach Angaben von Hennessy und Patterson [Henn96] verbessert sich die Latenz von Speicherbausteinen um circa 7% pro Jahr. Die Leistung von Mikroprozessoren verbessert sich seit 1987 um circa 55% pro Jahr. Die daraus folgende Leistungsdiskrepanz nennen Hennessy und Patterson das *Processor-Memory-Performance-Gap*.

Aufgrund dieser Diskrepanz wird die Leistung eines Rechners auf der Basis eines modernen RISC-Prozessors maßgeblich von der Leistung des implementierten Speichersubsystems bestimmt. Einer der Chefentwickler des Alpha-Prozessors von DEC, des in vielen Disziplinen leistungsstärksten modernen Prozessors, schreibt in dem Artikel „Architects Look to Processors of Future“ [Site96]:

> „Over the coming decade, memory subsystem design will be the *only* important design issue for microprocessors.“

Aus diesem Grund wurden und werden vor allem die Speichersubsysteme moderner Prozessoren verbessert (vgl. bspw. [Gwen96a], [Gwen98a], [Comp98b] und [Comp99b]).

Untersuchungen von Speicherhierarchien

Wie in den vorangegangenen Abschnitten beschrieben, haben die Eigenschaften der Speicherhierarchie großen Einfluss auf die Gesamtleistung eines Parallelrechners.

In vielen Untersuchungen wurde eine Modellierung der sequentiell arbeitenden Rechenknoten durchgeführt. Hier sollen nur zwei Veröffentlichungen genannt werden, die dies im Rahmen des LogP-Modells tun. In „Fast Parallel Sorting Under LogP: Experience with the CM-5“ [Duss96] von Dusseau et. al. wird festgestellt, dass das LogP-Modell zwar in der Lage ist, die Kommunikationscharakteristik auszudrücken, dass aber für die Vorhersage der Laufzeit eine zusätzliche Modellierung der Rechenknoten benötigt wird.

Die Autoren behaupten sogar, die Modellierung des Knotenverhaltens müsse detaillierter durchgeführt werden als die des Kommunikationsverhaltens, um eine Vorhersage vergleichbarer Genauigkeit zu erhalten[8]. Leider beschreiben die Autoren das „empirische Modell“, das sie verwendet haben, nicht genauer.

[8] [Duss96]: „We discovered that a more elaborate model was needed for the local computation phases than for the communication phases in order to predict the two with comparable accuracy.“

In der Veröffentlichung von Li et. al. „Models and Resource Metrics for Parallel and Distributed Computation“ [Li95] wird das LogP-Modell um ein PRAM-ähnliches Modell zur Darstellung einer Speicherhierarchie zu einem LogP-HMM-Modell erweitert. Mittels dieses HMM lassen sich Lokalitätsaspekte in algorithmischen Referenzierungsfolgen darstellen und auswerten. Das HMM verwendet eine statische Speicheranordnung aus mehreren Ebenen, wobei die k-te Ebene gerade 2^k Datenworte umfasst. Die Speicherzugriffe bilden in diesem Modell eine Markov-Kette, wodurch die Dynamik, die in einem realen Speichersystem durch die Cachekontrolle entsteht, jedoch nicht abgebildet wird (vgl. Abschnitt 5, „Modellierung des Verhaltens der Speicherhierarchie“).

3.4 Gängige Modellerweiterungen

Im Laufe verschiedener Folgeuntersuchungen zur Anwendung des LogP-Modells auf verschiedene konkrete Computer haben sich zwei gängige Modellerweiterungen etabliert.

3.4.1 Unterscheidung von Senden und Empfangen

In einem Multirechnersystem, das mittels Nachrichtenaustausch kommuniziert, bestehen große Unterschiede in der Implementationsweise eines Sende- und eines Empfangsvorganges. Bei der Anwendung des LogP-Modells auf die RS/6000-SP von IBM durch Kort und Trystrom [Kort] wurde daher zwischen Senden und Empfangen unterschieden. Dazu wurde der Overhead-Parameter o in zwei Anteile o_s zum Senden und o_r zum Empfangen aufgespalten. Diese Erweiterung wurde von den meisten nachfolgenden Untersuchungen übernommen.

3.4.2 Unterscheidung von kurzen und langen Nachrichten

Die Hardware der Netzwerkadapter und des Verbindungsnetzwerkes haben gewöhnlich Spezialfunktionalitäten, um die Bandbreite bei großen Paketlängen zu erhöhen. Um dieser Tatsache Rechnung zu tragen, wurde von Alexandrow et. al. [Alex95] eine Erweiterung des LogP- zu einem LogGP-Modell vorgeschlagen. Der neue Parameter G beschreibt das Gap, also die inverse Bandbreite, für große Paketlängen.

3.5 Fazit zur Struktur des Modells

Das LogP-Modell wurde bereits 1993 entworfen, um eine Alternative zum gängigen PRAM-Modell zu schaffen.

Im Gegensatz zum PRAM, dessen Stärken im Entwurf und der Bewertung von Algorithmen liegen, sollte mit LogP ein quantiativer Zugang zur Leistungs-Charakterisierung paralleler Programme auf konkreter Parallelrechner-Hardware ermöglicht werden.

Ein Modell für einen modernen Parallelrechner zu entwerfen, erforderte damals und erfordert noch heute einen schwierigen Kompromiss. Einerseits muss das Modell die wesentlichen strukturellen Merkmale der zu modellierenden Maschine wiedergeben, um ihre Leistung bei verschiedenen Problemklassen vorhersagen zu können. Andererseits dürfen nicht zu viele Konstruktionsdetails fixiert werden, da das Modell ansonsten weder auf die Rechnertypen verschiedener Hersteller noch auf weiterentwickelte Rechner desselben Herstellers anwendbar ist. Gleichzeitig darf der Prozess der eigentlichen Modellbildung nicht zuviel Zeit erfordern, da ansonsten die technische Weiterentwicklung die Modellentwicklung überholt.

Das LogP-Modell tendiert bei dieser Abwägung zur extremen Einfachheit. Nur vier skalare Parameter sollen sowohl die Kommunikations- als auch die Rechenleistung quantiativ erfassen. Dabei wurden die Parameter derart definiert, dass Kommunikations- und Berechnungsphasen nicht als zeitlich separiert angenommen werden müssen, sondern sich überlappen und asynchron ausgeführt werden können.

Zusätzlich bildet das ursprüngliche LogP-Modell eine Grundlage, die andere Forscher zur versuchsweisen Erweiterung einlädt. Das LogGP- [Alex95], das LoGPC- [Mori98] und das LogPQ-Modell [Touy99] sind Beispiele für erfolgreiche Erweiterungen des Grundmodells um in speziellen Anwendungen und Maschinen auftretende Effekte.

Kapitel 4

Untersuchung: Ein Modell im Praxistest

Im letzten Kapitel wurde das LogP-Modell vorgestellt. Dabei wurden sowohl die bei der Modellierung berücksichtigten als auch eine Auswahl der ausgeklammerten Aspekte dargestellt. In diesem Kapitel wird dargestellt, wie sich das LogP-Modell bei einem Praxistest verhält.

Dazu wird zuerst in Abschnitt 4.1 das exemplarische Testsystem aus Anwendungssoftware, Programmiermethodik und ausführender Maschine vorgestellt. Abschnitt 4.2 enthält den Vergleich von LogP-Grundannahmen mit den Ergebnissen entsprechender experimenteller Untersuchungen. Mit dem Testsystem als Ganzes wird in 4.3 das LogP-Modell einem Praxistest unterzogen, wodurch sich die Notwendigkeit ergibt, im folgenden Kapitel 5 die Modellierung der Speicherhierarchie eines Rechenknotens zu diskutieren.

4.1 Das exemplarische Testsystem

In diesem Abschnitt wird das verwendete exemplarische Testsystem aus Anwendung und ausführender Maschine beschrieben. Zusätzlich wird die verwendete Programmiermethodik und die gewählte Kommunikationsschnittstelle vorgestellt.

4.1.1 Matrixmultiplikation

Als Beispielprogramm für den Praxistest des LogP-Modells wird in dieser Untersuchung die Matrixmultiplikation gewählt. Da Verfahren der linearen Algebra in vielfältigen akademischen und kommerziellen Bereichen angewendet werden, bildet die Matrixmultiplikation einen gängigen Benchmark für Parallelrechner. Obwohl sie in ihrer algorithmischen Struktur leicht zu verstehen ist, zeigen sich bei ihrer Ausführung bereits viele interessante Effekte.

Basic Linear Algebra Subprograms (BLAS)

Generell werden in einem leistungsorientierten Umfeld Programmierer ihre Matrixfunktionen nicht selbst schreiben, sondern auf vorgefertigte Bibliotheken wie die oben genannten zurückgreifen.

Die *BLAS* (*Basic Linear Algebra Subroutines*) fassen gängige grundlegende Funktionen der linearen Algebra in einer Bibliothek zusammen. Inhaltlich sind die BLAS eine Zusammenfassung der Pakete LINPACK, LAPACK und EISPACK, die wiederum

aus Algorithmen der *Transactions on Mathematical Software* der *ACM* (*Association for Computing Machinery*) hervorgegangen sind.

Für ein kommerzielles Rechnersystem ist üblichweise eine optimierte Implementierung von BLAS vom Hersteller des Rechners oder einen Drittanbieter erhältlich. In dieser Untersuchung wird auf die Verwendung dieser speziell auf Prozessortyp und Cache-Konfiguration optimierten Pakete verzichtet und statt dessen eine einfache Multiplikationsfunktion selbst entwickelt. Dafür waren zwei Gründe ausschlaggebend:

- **Übertragbarkeit** der Ergebnisse: Die jeweiligen Implementierungen sind auf spezielle Maschinen optimiert. Der sequentielle Anteil der BLAS-Codes ist an die eingesetzen Prozessortypen und Cache-Konfigurationen angepasst, der parallele Anteil schöpft die Möglichkeiten der Hardware des Verbindungsnetzes aus. Damit zeigt sich sehr maschinenabhängiges Verhalten.

  Obwohl in dieser Untersuchung nur mit einem bestimmten Maschinentyp konkret gearbeitet wird, sollen die Aussagen doch, soweit möglich, auf andere Maschinentypen verallgemeinerungsfähig sein.

- **Einfachheit** des verwendeten Codes: Das Ziel dieser Untersuchung besteht nicht darin, möglichst schnell Matrizen zu multiplizieren. Die Multiplikation soll lediglich als exemplarischer Zugang zum Verständnis der bei der parallelen Ausführung auftretenden Effekte verwendet werden.

  Um diese Effekte untersuchen zu können, sollte die Messanordnung aus Programmcode und ausführender Maschine möglichst einfach sein. Daher wurde in dieser Arbeit eine „naive“ Implementierung untersucht, welche nicht speziell für einen Prozessortyp oder eine Cachekonfiguration codiert wurde.

  Trotz der naiven Implementierung des sequentiellen Codes wurde jedoch kein primitives Parallelisierungsverfahren verwendet, sondern der Fox-Algorithmus.

Der Fox-Algorithmus

Der Fox-Algorithmus ist eine Methode zur parallelen Multiplikation zweier Matrizen auf Parallelrechnern mit verteiltem Speicher. Dabei behandelt der Fox-Algorithmus lediglich die Zerlegung der Eingabematrizen in kleinere Untermatrizen und regelt deren Austausch zwischen den einzelnen Rechenknoten. Das eigentliche Ausmultiplizieren der Untermatrizen wird nicht spezifiziert.

Eine Darstellung des Fox-Algorithmus für eine Message-Passing-Umgebung findet sich in Kapitel 7, „Communicators and Topologies“, des Buches von Pacheco [Pach97]. In der Darstellung von Pacheco werden die sogenannten *kollektiven Operatoren* (siehe auch Kapitel 7 der Beschreibung des Interface-Standards „MPI-2: Extensions to the Message-Passing Interface“ [MPI97a]) von MPI verwendet. Da jedoch gerade die Implementierung dieser kollektiven Operatoren dem Hersteller

viel Potential zur Optimierung bietet, wurden diese Funktionen nicht verwendet und durch eine Sequenz von Punkt-zu-Punkt-Verbindungen ersetzt.

4.1.2 Silicon Graphics Origin

Als konkreter Parallelrechner zur Ausführung der Testanwendung wurde die *Origin* des Herstellers Silicon Graphics (SGI) gewählt. Die Origin gehört zur Klasse der ccNUMA-Rechner, die bereits in Abschnitt 2.3 beschrieben wurden. ccNUMA-Parallelrechner stellen einen besonders interessanten Untersuchungsgegenstand dar, da sie Vorzüge der Skalierbarkeit bis zu großen Prozessoranzahlen mit der Verfügbarkeit eines globalen Adressraumes zu verbinden versprechen.

Zur Relevanz der SGI Origin

Die *Top500-Liste* ist eine Liste der 500 leistungsfähigsten Computerinstallationen weltweit, gemessen nach LINPACK-Leistung. Damit stellt die Liste einen Indikator für die Akzeptanz eines Maschinentyps bei kommerziellen und akademischen Anwendern dar. Auf die genauen Platzierungen innerhalb der Liste soll hier kein Bezug genommen werden. Die Top500-Liste wird von den Universitäten von Mannheim und Tennessee geführt. Sie wird vor allem im Internet unter `www.top500.org` und auf den „Supercomputing“-Konferenzen publiziert.

Während dieser Studie war die 13. Top500-Liste vom Juni 1999 aktuell. In dieser belegt der Hersteller SGI mit 182 der 500 größten bzw. 47 der 100 größten Computer den ersten Platz. Die dadurch belegte Relevanz der Produkte dieses Herstellers gab, zusammen mit der Attraktivität der Maschinenklasse und den guten Zugangsmöglichkeiten zu zwei Maschinen dieses Typs, den Ausschlag dafür, die konkreten Messungen auf Origin-Rechnern durchzuführen.

Der Vollständigkeit halber sei erwähnt, dass neben den Origin-Rechner von Silicon Graphics auch die *X-Class* von Hewlett-Packard (ehemals Convex Exemplar) sowie die Sequent *NUMA-Q* die ccNUMA-Architektur umsetzen. Darüber hinaus ist anhand von Beobachtungen der aktuellen Entwicklung sowie technischen Veröffentlichungen abzulesen, dass auch die Hersteller IBM und SUN an der Entwicklung von ccNUMA-Maschinen arbeiten.

Die zugänglichen Testmaschinen

Im Rahmen dieser Arbeit wurden Untersuchungen und Laufzeitmessungen auf zwei unterschiedlich großen Installationen von SGI Origin-Rechnern durchgeführt: Freundlicherweise stellte mir die Firma GIP AG Gesellschaft für Industriephysik aus Mainz ihre Origin-200 mit 4 Prozessoren und 512 MByte Hauptspeicher für das Entwickeln aller Testprogramme zur Verfügung. Die endgültigen Skalierungsuntersuchungen wurden auf der Origin-2000 (48 Prozessoren und 17 GByte Speicher) des Universitätsrechenzentrums der TU-Dresden durchgeführt. Die genauen Konfigurationen und Software-Ausstattungen beider Maschinen sind in Anhang B aufgeführt.

Da die Maschine in Dresden nur während vier Stunden exklusiv für diese Studie reserviert war und sich ansonsten unter starker Benutzerlast befand, stand nur das Mainzer System für Messungen mit langer Laufzeit zur Verfügung. Daher werden, obwohl die Ergebnisse beider Maschinen aufgrund unterschiedlicher Taktfrequenzen sowie Cache- und Hauptspeichergrößen nicht direkt quantitativ vergleichbar sind, im weiteren Verlauf der Ausführungen Ergebnisse von beiden Maschinen angeführt.

4.1.3 Message Passing Interface (MPI)

Das LogP-Modell ist nicht auf eine spezielle Kommunikationsschnittstelle festgelegt. In den Originalarbeiten [Cull93], [Cull96] wird die Programmiersprache Split-C verwendet, welche die nachrichtenbasierte Kommunikation vor dem Programmierer verbirgt. Eine separate Veröffentlichung [Cull96b] behandelt eine Nachrichtenschnittstelle namens Active-Messages.

Da weder Split-C noch Active-Messages im kommerziellen Einsatz gebräuchlich sind, wird in dieser Untersuchung statt ihrer das Message-Passing-Interface (MPI) verwendet.

Vorbemerkungen zur Relevanz von MPI

Zur Zeit dieser Untersuchung ist Message-Passing-Interface (MPI) der de-fakto-Standard für Parallelisierung durch Nachrichtenaustausch. Die aktuelle Version MPI-2 wird durch das MPI-Forum festgelegt ([MPI97a], [MPI97b] und [MPI98a]). Ein Zugang zu MPI aus Benutzersicht findet sich im Lehrbuch von Pacheco [Pach97].

Die Leistungsfähigkeit kommerzieller Parallelrechner wird wesentlich durch die Spezialhardware des Verbindungsnetzes beeinflusst. Daher gibt es auf kommerziellen Parallelrechnern üblicherweise ein speziell durch den Hersteller optimiertes MPI. In dieser Arbeit wurde SGI-MPI verwendet. Genaue Angaben zu Versionsnummern und Produktpaketen findet sich ebenfalls in Anhang B.

Zum Aufbau von MPI

MPI wurde entworfen, um mit den prozeduralen Programmiersprachen C und FORTRAN eingesetzt zu werden. Neben mächtigen Parallelisierungskonstrukten ermöglicht MPI auch eine einfache Parallelisierung über Sende- und Empfangsfunktionen zur Punkt-zu-Punkt-Kommunikation. Um beides zu erreichen ist MPI sehr einfach aufgebaut: Schon sechs[1] Funktionen sind ausreichend, um ein rudimentäres aber voll funktionsfähiges Parallelprogramm zu schreiben.

Der *Communicator*, der Vermittler aller Nachrichtenkommunikation unter MPI, versieht alle teilnehmenden Prozesse mit einer eindeutigien, fortlaufenden Nummer,

[1] Die Funktionen `MPI_Init()` und `MPI_Finalize()` dienen der Initialisierung und der Terminierung. Mittels `MPI_Send()` und `MPI_Recv()` lassen sich Daten versenden und empfangen. Die Aufrufe `MPI_Comm_size()` und `MPI_Comm_rank()` ermöglichen die Interaktion mit dem Kommunikationsmedium, dem Communicator.

dem *Rang*. Über diesen Rang können die Prozesse in einer Kommunikation adressiert werden. Es gibt keine definierte Zuordnung von Prozess-Rängen zu den Prozessoren des ausführenden Rechners.

Neben den einfachen Funktionen zur Punkt-zu-Punkt-Kommunikation gibt es in MPI die sogenannten *kollektiven Operatoren*, in denen ganze Gruppen von Prozessen eine gemeinsame Kommunikation ausführen. Die Gruppenbildung wird dabei über zusätzlich erzeugte Kommunikatoren gesteuert. Gerade diese kollektiven Operationen erlauben umfassende Optimierungen der MPI-Implementierung, indem spezielle Eigenschaften der Hardware des Verbindungsnetzes ausgenutzt werden. Da diese Optimierungen extrem maschinenabhängig und nur schlecht der Analyse zugänglich sind, wurden sie im Rahmen dieser Arbeit nicht verwendet. Es ist jedoch unbestritten, dass die Verwendung kollektiver Operatoren in vielen Fällen ein wesentlicher Schritt ist, um ein paralleles Programm auf einem speziellen Parallelrechner zu optimieren.

Bemerkungen zu Shared-Memory- und Message-Passing-Betrieb

Unter den Parallelrechnern finden sich zunehmend Maschinen, welche neben einem Message-Passing- auch einen Shared-Memory-Betrieb erlauben. Dies gilt, wie in Abschnitt 2.3 beschrieben, insbesondere für die im Rahmen dieser Studie als besonders interessant klassifizierte ccNUMA-Architektur.

Im Laufe der Untersuchungen erwies sich der Shared-Memory-Betrieb der SGI Origin in manchen Operationen als deutlich leistungsfähiger als der Betrieb mit nachrichtenbasierter Kommunikation.

Aus diesem Grund werden an den entsprechenden Stellen neben den Ergebnissen aus dem Message-Passing- auch diejenigen aus dem Shared-Memory-Betrieb aufgeführt und die Praxistauglichkeit des LogP-Modells für jede dieser Betriebsarten separat bewertet.

4.2 Prüfung von Grundannahmen des LogP-Modells

Im letzten Unterkapitel wurde die Kombination aus Rechnersystem, paralleler Programmiermethodik und einem Anwendungsprogramm vorgestellt, mit dem das LogP-Modell in diesem Kapitel einem Praxistest unterzogen wird. Insbesondere die Grundannahmen des Modells werden in diesem Abschnitt überprüft.

Schon bei der Vorstellung des Modellierungskonzeptes hinter LogP wurde auf Aspekte hingewiesen, die aus einem systemischen Verständnis der auftretenden Abläufe heraus als unplausibel eingestuft werden müssen:

- Die Latenz einer Nachricht wird als unahhängig von der Anordnung der Kommunikationsendpunkte im Graphen des Verbindungsnetzes angenommen.
- Die Bandbreiten und Latenzzeiten werden als unabhängig von den auftretenden Kommunikationsmustern vorausgesetzt. Insbesondere werden Congestion- und Contention-Effekte nicht berücksichtigt.

- Die Bandbreite, definiert über den zeitlichen Abstand zwischen zwei Nachrichten, wird als Konstante des Verbindungsnetzes behandelt, so dass sie weder von den Endpunkten noch von der Nachrichtenlänge abhängen kann.

4.2.1 Der vollständige Graph als Grundlage

Die ersten beiden genannten Annahmen sind aus der Modellierung des Verbindungsnetzes als vollständiger Graph abgeleitet. Da sich solche vollständigen Graphen als Verbindungsnetz nur für kleine Prozessoranzahlen in Form von sogenannten *Crossbars* herstellen lassen, werden in real existierenden Parallelrechnern mit größerer Prozessoranzahl andere Netztopologien verwendet.

Zum graphentheoretischen Hintergrund

In einem vollständigen Graphen $G = (V, E)$ ist jedes Knotenpaar $(u, v) \in V$ durch eine Kante $e \in E$ verbunden. Somit ist eine direkte Kommunikation zwischen den Knoten u und v möglich. Wie im Abschnitt 2.4 über Verbindungsnetze dargestellt, ist dieser Graph bezüglich Bisektionsweite und Durchmesser theoretisch optimal, für große Knotenanzahlen ist er jedoch praktisch nicht umsetzbar.

Daher werden in Parallelrechnern häufig vom vollständigen Graphen abweichende Topologien gewählt. Meist werden die Knoten durch verschiedene Varianten von mehrdimensionalen Gittern verbunden. Dadurch verschlechtern sich zwar Bisektionsweite und Durchmesser, die Netze sind jedoch aufgrund der beschränkten Knotengraden technisch realisierbar. Der gestiegene Durchmesser hat jedoch zur Folge, dass der Pfad zwischen einem Knotenpaar $(u, v) \in V$ im Allgemeinen über mehr als eine Kante laufen wird. Dadurch ergibt sich sowohl eine Variation in der Pfadlänge als Funktion des Knotenpaares als auch eine Mehrfachbelegung der Kanten durch mehrere Pfade.

Eine kleine Marktübersicht

Zum Zeitpunkt dieser Studie wurden in den kommerziellen Parallelrechnern der Firmen[2] SGI/Cray, IBM, SUN und HP/Convex verschiedene Netzwerktopologien eingesetzt. Die Maschinen waren durchweg hierarchisch aufgebaut. Die über das Verbindungsnetz miteinander verbundenen Rechenknoten bestanden ihrerseits aus zwei bis 16 Prozessoren, um den Knotengrad des Netzes niedrig zu halten. Während innerhalb der Knoten durchweg Busse oder kleine Crossbars eingesetzt wurden, fanden sich als Verbindungstopologien Hyperkuben und hyperkubische Netze, zwei- und dreidimensionale Tori, Crossbars und Omega-Netzwerke.

Der Verbindungsgraph des hier eingesetzten Maschinentyps, der SGI Origin, ist ein Hyperkubus.

[2]Die Parallelrechner dieser Hersteller decken 87% der 13. Top500-Liste vom Juni 1999 ab. Die restlichen Maschinen sind meist Vektorrechner und damit im Rahmen dieser Untersuchung nicht von Interesse.

Zum Hyperkubus als Verbindungsgraph

Wie bereits in Abschnitt 2.4 über Verbindungsnetze dargestellt wurde, weist der Hyperkubus Eigenschaften auf, die ihn zu einem guten Verbindungsgraphen für Parallelrechner machen.

Knotengrad und Durchmesser betragen $\log_2 P$, d. h. wachsen nur langsam mit der Anzahl der Knoten P. Gleichzeitig wächst die Bisektionsweite linear mit $P/2$. Als Graph erhält der Hyperkubus einen Nachbarschaftsbegriff zwischen den Knoten, so dass eine Optimierung von Kommunikationswegen möglich ist.

Die Knoten eines Hyperkubus lassen sich nach einem bestimmten Schema durch Binärstrings nummerieren, so dass sich benachbarte, also direkt durch eine Kante verbundene, Knoten an nur genau einer Stelle in ihren Binärstrings unterscheiden. Bezeichnet man diese Bitposition mit j, so nennt man die verbindende Kante eine „Kante der Dimension j“. Der Abstand zwischen zwei Knoten lässt sich direkt durch die Anzahl der unterschiedlichen Bits in ihren Knotennummern ablesen. Diese Anzahl und damit die Entfernung bezeichnet man auch als *Hammingdistanz*. Diese und weitere Eigenschaften ermöglichen leistungsfähige und gut zu implementierende Routingverfahren auf dem Hyperkubus.

Zusammen mit der Tatsache, dass sich andere Graphen wie Bäume und mehrdimensionale Gitter effizient in ihn einbetten lassen, macht dies den Hyperkubus zu einer algorithmisch und technisch besonders attraktiven Verbindungstopologie. Eine sehr ausführliche Zusammenfassung der Eigenschaften des Hyperkubus sowie auf ihn abgestimmter Algorithmen und Routing-Verfahren enthält Kapitel 3, „Hypercubes and Related Networks“, des Lehrbuches von Leighton [Leig92]. Abschnitt 10.4, „Interconnection Topologies“ des Buches von Culler, Singh und Gupta [Cull99] betrachtet den Hyperkubus als Verbindungsnetz aus technischer Sicht.

Für den technisch orientierten Leser muss noch bemerkt werden, dass der Abstand zwischen zwei Prozessoren in der SGI Origin durch eine angepasste Hammingdistanz berechnet werden muss. Jeweils zwei Prozessoren sind über einen Hub miteinander verbunden und bilden einen Rechenknoten. Je zwei dieser Rechenknoten sind an einen gemeinsamen Router angeschlossen, der wiederum mit vier weiteren Routern verbunden ist. Die verwendete Hammingdistanz muss um diese Hub- und Router-Effekte korrigiert werden. Zusätzlich wird in dieser Untersuchung eine vom Betriebssystem bereitgestellte *Distanzmatrix* verwendet. Diese findet sich für die verwendeten Maschinen aus Dresden und Mainz in Anhang B.

4.2.2 Die Abhängigkeit der Latenz von der Weglänge

In diesem Abschnitt wird gezeigt, dass die Latenz bereits in einem Parallelrechner gängiger Größe keine Konstante darstellt, sondern durch eine lineare Funktion der Weglänge beschrieben werden kann.

Dazu wird zunächst die sogenannte Round-Trip-Time als direkt messbare Größe für einen experimentellen Zugang gewählt. Danach wird die Annahme gemacht, diese Round-Trip-Time sei linear von der Weglänge abhängig, welche wiederum durch die

Anzahl von Router-Durchgängen beschrieben werden kann. Durch eine Variation der Kommunikationsendpunkte lässt sich auch diese Weglänge variieren, so dass der angenommene Zusammenhang mit einer Messung verglichen werden kann.

Untersuchung über die Round-Trip-Time

Der Latenzparameter L ist in der vom LogP-Modell gewählten Definition [Cull93], [Cull96] einer experimentellen Messung nicht direkt zugänglich, da er sich beispielsweise nur schwer vom Overhead-Parameter o separieren lässt. Daher wird als Zugang zur Erfassung dieser Weglängenabhängigkeit nicht direkt die Latenz L, sondern, wie in der Arbeit von Culler et. al. über Active-Messages [Cull96b], die sogenannte *Round-Trip-Time* gewählt.

Ein Prozess a schickt eine Anfrage, einen *Request*, an einen Prozess b. Dabei kann es sich beispielsweise um die Adresse eines zu lesenden Datums handeln, welches sich im lokalen Speicher von Prozess b befindet. b schickt die zum Request gehörende Antwort, den *Reply*, zurück an a. Die Zeit zwischen dem Absenden der Anfrage und dem Eintreffen der Antwort wird als Round-Trip-Time t_{rtt} bezeichnet:

$$t_{rtt} = 2L + 4o$$

Bei der gängigen Aufteilung des Overhead-Parameters o in separate Parameter für Senden o_s und Empfangen o_r, siehe Abschnitt 3.4, erweitert sich dies auf

$$t_{rtt} = 2L + 2o_s + 2o_r$$

Da wir in dieser Untersuchung davon ausgehen, dass die zeitlichen Overhead-Kosten keine Abhängigkeit von der Weglänge zeigen, kann somit über eine Messung der Round-Trip-Time bei Variation der Kommunikationsendpunkte die Abhängigkeit der Latenz von der Weglänge bestimmt werden.

Zugang über die Anzahl von *Hops*

Wie bereits erläutert, sind die Prozessoren im verwendeten Testrechner über ein Verbindungsnetz aus Routern gekoppelt. Eine Nachricht, die von einem Prozessor a zu einem Prozessor b gesandt werden soll, muss also von Prozessor a über seinen Hub an seinen Router u übergeben werden. Nun muss die Nachricht durch das hyperkubische Verbindungsnetz zum Router v von Prozessor b geroutet werden. Jeden Durchgang der Nachricht durch einen Router bezeichnet man in Netzwerk-Jargon als einen *Hop*. Die Anzahl dieser Hops heißt auch *Hop-Count*.

Sollten sich die Prozessoren a und b an demselben Router befinden, so ist die Anzahl der Hops 1. Sind a und b Prozessoren desselben Nodeboards, so wird die Nachricht ohne Durchlaufen eines Routers zugestellt und der Hop-Count ist 0.

Da das von den Routern der SGI Origin verwendete *Wormhole*-Routing eine Form von Cut-Through-Routing ist, bei welchem das zu routende Paket nicht zwischengespeichert wird, ist die zeitliche Verzögerung pro Hop nicht von der Paketlänge abhängig, sondern kann als Konstante r angenommen werden.

Wenn die Zeit für einen Hop r beträgt, dann beträgt die Zeit für n Hops nr. Da zu erwarten ist, dass es zusätzlich eine Zeit c kostet, aus dem Hub des Quellprozessors in den ersten Router und aus dem letzten Router in den Hub des Zielprozessors zu senden, erwarten wir für die Latenz L als Funktion des Hop-Counts n die einfache Abhängigkeit

$$L(n) = rn + c$$

Variation der Kommunikationsendpunkte

Die 48 Prozessoren der Maschine RAPUNZEL der TU-Dresden sind über einen unvollständig bestückten 4-dimensionalen Hyperkubus aus 12 Routern verbunden. Somit lassen sich Weglängen zwischen 0 und 4 Hops erreichen.

Bei diesem Test wurde der Request-Prozess an die Prozessornummer 47 gebunden, während der Reply-Prozess über alle Prozessornummern von 0 bis 46 variiert wurde. Für jede Position des Repliers wurde die Round-Trip-Time gemessen. Dazu wurde eine 80 Byte lange Nachricht aus 10 Fließkommazahlen doppelter Genauigkeit 1000000-mal als Round-Trip-Request versandt. Die verstrichene Zeit wurde durch den Requester-Prozess über die Echtzeituhr des Systems gemessen.

Ergebnisse bei Message-Passing-Kommunikation

Abbildung 4.1 (S. 38) zeigt die Ergebnisse dieser Messung bei Verwendung von Message-Passing mittels MPI. Die Messwerte zeigen starke Streuung, da die Maschine während der Untersuchung nicht reserviert war, sondern unter Benutzerlast stand. Die Last[3] während der Messungen betrug zwischen 30 und 44.

Die linke Seite von Abbildung 4.1 zeigt die Round-Trip-Time als Funktion der Prozessornummer des Repliers dar. Um die Abhängigkeit der Round-Trip-Time, des Indikators für die Latenz, von der Weglänge ablesen zu können, wird sie auf der rechten Seite von Abbildung 4.1 gegen die Hammingdistanz zwischen Requester und Replier aufgetragen.

Es zeigt sich eine generelle Zunahme der Round-Trip-Time bei Zunahme der Weglänge. Die erwartete Abhängigkeit $L(n) = rn + c$ wird von der Messung aufgrund der starken Streuung nur im Groben bestätigt. Die Mittelwerte der Messwerte bei den einzelnen Distanzen zeigt Tabelle 4.1 auf Seite 40.

Daraus lässt sich ein maximaler Effekt der Weglänge auf die Round-Trip-Time in der Größenordnung von circa 20% ableiten. Genauere Zahlen würden sich nur durch eine leere Maschine ohne Hintergrundlast erzielen lassen.

[3] Die Last wird in diesem Kontext durch den Scheduler des Betriebssystems definiert und entspricht der Anzahl der zum Messzeitpunkt gleichzeitig laufbereiten Prozesse oder Threads.

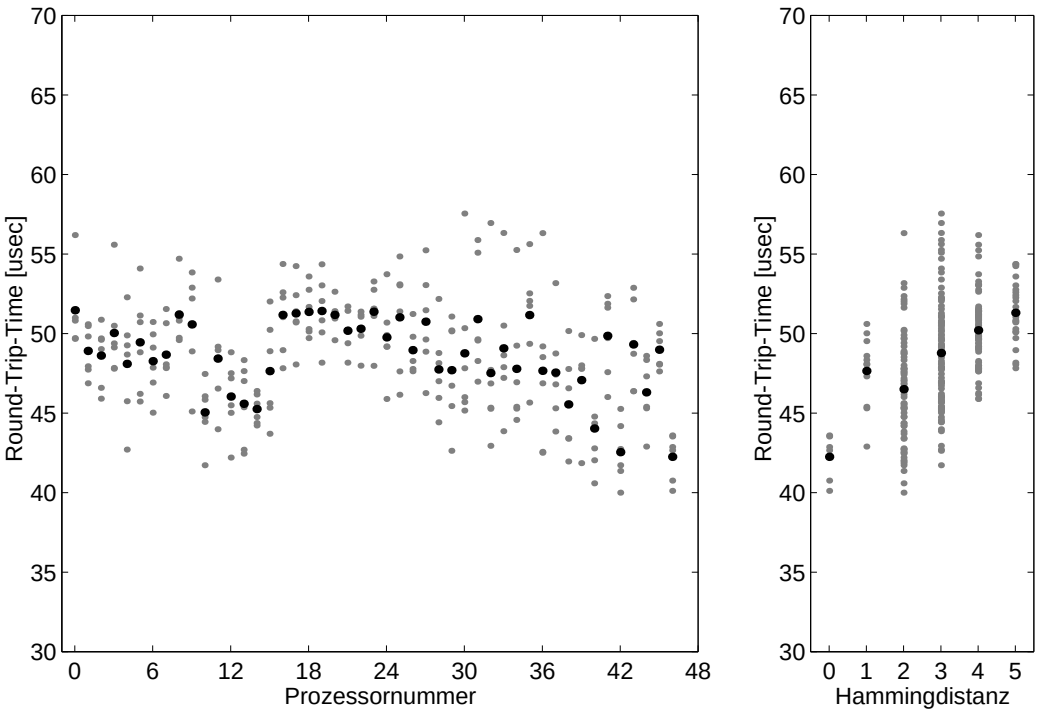

Abbildung 4.1: **Ausführungszeiten für einen Round-Trip-Request bei Message-Passing-Kommunikation** – Die linke Seite zeigt die Ausführungszeit als Funktion der Kommunikationsendpunkte. Der Requester ist fest an Prozessor 47 gebunden, die Position des Repliers wird variiert. Auf der rechten Seite wurde die Position des Repliers auf seine Distanz zum Requester umgerechnet. (Die kleinen grauen Punkte stellen die Rohdaten dar, während die größeren schwarzen Punkte die zugehörigen Mittelwerte markieren. Die Ausführungszeiten wurden über die hochauflösende Echtzeituhr des Systems gemessen.)

Ergebnisse bei Shared-Memory-Kommunikation

Um den Software-Overhead durch die MPI-Zwischenschicht abschätzen zu können, wurde die Messung der Round-Trip-Time auch unter Shared-Memory-Kommunikation durchgeführt.

Für dieses Experiment musste ein Kompromiss gefunden werden. Einerseits wird der Shared-Memory-Betrieb üblicherweise gewählt, um durch hardwarenahe Programmierung die Performance zu maximieren. Andererseits sollte eine Vergleichbarkeit mit dem Message-Passing-Programm erhalten bleiben.

In einem ccNUMA-Rechner wie der hier verwendeten SGI Origin findet die Kommunikation zwischen den einzelnen Prozessoren und Speichermodulen durch Austausch von sogenannten *Cachelines*, den elementaren Adressierungseinheiten der Prozessorcaches, statt. Daher wurden die zu transportierenden Nachrichten in einem Speichermodul durch den Requester-Thread in ganzzahlige Vielfache dieser Cachelines

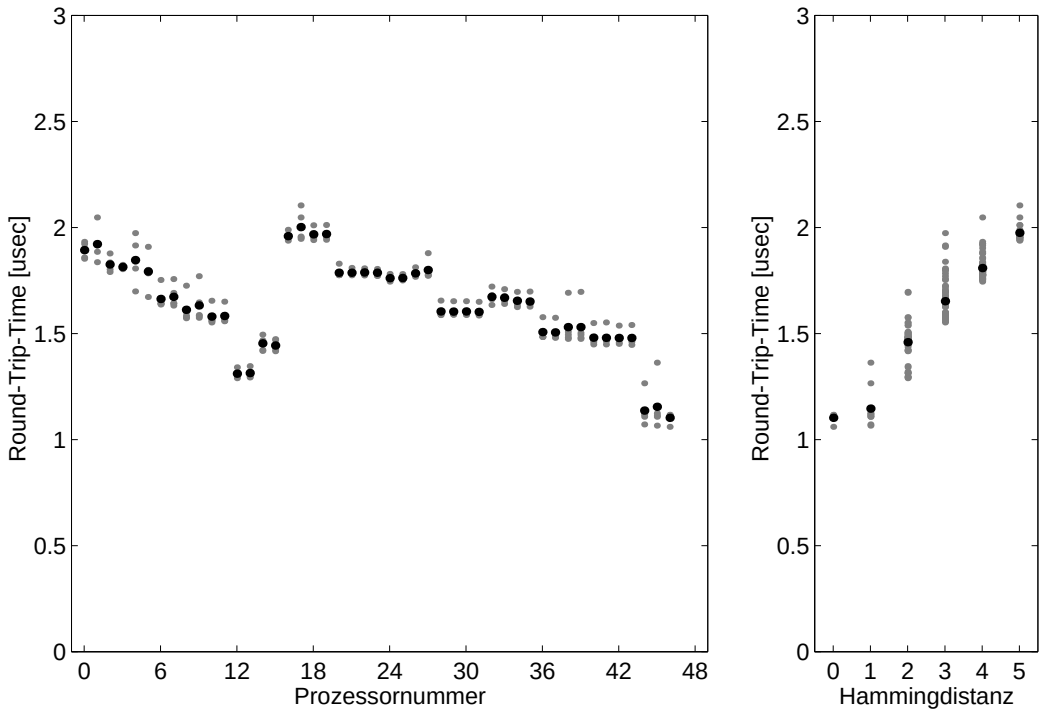

Abbildung 4.2: **Ausführungszeiten für einen Round-Trip-Request bei Shared-Memory-Kommunikation** – Analog zu Abbildung 4.1 stellt die linke Seite die Messwerte in Abhängigkeit von der Prozessornummer des Repliers dar. Auf der rechten Seite wurde diese Prozessornummer in die Hammingdistanz zwischen Requester und Replier umgerechnet.

eingebettet. Der Replier-Thread löste durch einmalige Adressierung jeder dieser Cachelines den Transport der darin enthaltenen Nachrichten aus.

Aus technischen Gründen wurden nicht die mittlerweile gängigen POSIX-Threads, sondern sogenannte *Sprocs* (Share Group Processes) verwendet. Informationen zu den Besonderheiten von Sprocs finden sich in der Programmierreferenz [SGI96a] der Origin und der man-Page zu „sproc“.

Tabelle 4.1 (S. 40) listet die Mittelwerte der Round-Trip-Time zu den jeweiligen Weglängen zwischen Requester und Replier auf. Abbildung 4.2 zeigt die Abhängigkeiten der Ausführungszeit von Prozessornummer und Weglänge.

Auch bei dieser Messung war die Maschine nicht reserviert. Ihre Hintergrundlast lag während der acht Messreihen zwischen 23 und 39. Obwohl auch diese Messdaten deutlich streuen, ist deutlich zu sehen, dass der Effekt der Weglänge auf die Round-Trip-Time bei Shared-Memory-Kommunikation deutlich größer ist als bei Message-Passing-Kommunikation. Tabelle 4.1 zeigt deutlich, dass der Zeitbedarf für 5 Hops circa 80% größer ist als für 0 Hops.

Zusätzlich fällt auf, dass sich ein Datentransfer über Shared-Memory etwa 25- bis 35-mal schneller durchführen lässt als über Message-Passing. Der zusätzliche Overhead

Weglänge [Hops]	Round-Trip-Time Message-Passing	Round-Trip-Time Shared-Memory
0	42.3 μs	1.10 μs
1	47.6 μs	1.15 μs
2	46.5 μs	1.46 μs
3	48.8 μs	1.65 μs
4	50.2 μs	1.81 μs
5	51.3 μs	1.97 μs

Tabelle 4.1: Die Abhängigkeit der Round-Trip-Time von der Weglänge bei Message-Passing- und bei Shared-Memory-Kommunikation.

durch die Verwendung der MPI-Software verschlechtert somit diesen Aspekt der Kommunikationsleistung um mehr als eine Größenordnung.

Der Vollständigkeit wird hier auf die unabhängige Leistungsevaluation der Origin durch Jiang und Singh [Jian98] hingewiesen werden. In dieser Untersuchung mittels einer hardwarenahen messtechnischen Methode die günstigsten und ungünstigsten Latenzen für den Zugriff auf entfernten Speicher in Abhängigkeit von der Entfernung zwischen Prozessor und Speicher bestimmt. Die hier vorgestellten Daten liegen innerhalb der von Jiang und Singh gemessenen Intervallgrenzen.

Zwischenbewertung

Wie erwartet zeigt die Round-Trip-Time, und damit auch die Latenz, eine nahezu lineare Abhängigkeit von der Weglänge.

Die Kommunikation über Message-Passing verursacht einen hohen Software-Overhead, so dass der Einfluss der Weglänge hier lediglich maximal circa 20% ausmacht.

Im dagegen weitgehend von Software-Overhead befreiten Shared-Memory-Betrieb sinkt dagegen die absolute Round-Trip-Time um etwa einen Faktor 30. Dabei steigt der relative Einfluss der Weglänge auf bis zu 80% bei 5 Hops.

Aus Sicht der Modellierung ergibt sich dadurch ein interessanter Perspektivenwechsel. Im Message-Passing-Betrieb stellt die Round-Trip-Time einen praktikablen Zugang zum Modellparameter L dar, da bei Kommunikationszeiten im Bereich von 50 μs zusätzliche Einflüsse durch beispielsweise Prozessorcaches vernachlässigt werden können.

Im Shared-Memory-Betrieb sind die charakteristischen Zeiten jedoch um mehr als eine Größenordnung kleiner, so dass bei bisher vernachlässigten Einflüssen kontrolliert werden muss, ob diese Vernachlässigung auch weiterhin zulässig ist. Für den technisch orientierten Leser seien drei dieser Aspekte genannt: Vor dem Datentransport der Cacheline muss eine Adressierungsanfrage in umgekehrter Richtung gesandt werden. Damit stellt, streng genommen, bereits der einfache Weg einen Round-Trip-Request dar. Weiterhin muss eine Cacheline im Cache des referenzie-

renden Prozessors verworfen oder bei Modifikation (hier nicht der Fall) in den Speicher zurückgeschrieben werden, was zeitlich von derselben Größenordnung wie der eigentliche Datentransport wäre und zusätzlich stark vom Vorhandensein spezieller Hardware (*Victim Buffer*) abhängen würde. Schließlich müssen die adressierten Worte wiederum die Eingabedaten einer nachfolgenden Operation darstellen, damit die Zugriffe im Rahmen der Abhängigkeitsanalyse des Compilers nicht eliminiert werden. Diese und weitere Einflüsse auf maschinennaher Ebene wurden untersucht und als vernachlässigbar eingestuft.

4.2.3 Der Overhead und die Bandbreite als Funktionen der Paketlänge

Das ursprüngliche LogP-Modell definiert die Bandbreite des Verbindungsnetzes über den Parameter g als den zeitlichen Abstand zwischen zwei kleinen Nachrichten. Dabei wird nicht exakt spezifiziert, wie viele Bytes eine kleine Nachricht umfasst.

Während sich diese Sichtweise im Shared-Memory-Betrieb durch die technische Implementierung des cache-kohärenten Adressraumes (vgl. [Leno95], [Cull99]) rechtfertigen ließe, so stellt sie auf Anwendungsebene, gerade bei Message-Passing, eine zu große Vereinfachung dar. Wie zuvor in Abschnitt 4.2.2 dargestellt, ist bei MPI das Senden und Empfangen mit einem großen Overhead verbunden.

Dieser Overhead betrifft das Aufrufen der MPI-Funktion, das Verpacken der Nachricht in einen sogenannten *Envelope*, das eventuelle Zwischenpuffern der Nachricht, das letztendliche Versenden und das Zurückkehren aus der MPI-Funktion. Bis auf das eventuelle Zwischenpuffern, dessen Aufwand durch Kopieren linear von der Länge der Nachrichten abhängen wird, sind all diese Aktivitäten nur einmal pro Nachricht durchzuführen und damit unabhängig von der Länge der Nachricht.

Große Nachrichten

In einem Experiment wurde die zwischen Anwendungsprozessen nutzbare Bandbreite zwischen einem Sender und einem Empfänger gemessen. Um durch längere Messdauer geringere zählstatistische Fehler als in 4.2.2 zu erhalten, wurde eine Langzeit-Messung auf der Origin-200 in Mainz durchgeführt. Dabei wurden die Positionen der Kommunikationsendpunkte nicht variiert, sondern der Sender wurde fest an Prozessor 0 und der Empfänger fest an Prozessor 3 gebunden. Somit mussten die Nachrichten zwischen verschiedenen Nodeboards ausgetauscht werden.

Die Nachrichtenlänge wurde zwischen 64 und 16384 Fließkommazahlen variiert, also zwischen 0,5 und 128 kByte. Es wurden 10^6 Nachrichten versandt. Nach vollständigem Empfang der letzten Nachricht wurde vom Empfänger eine kleine Bestätigungsnachricht zurück zum Sender geschickt. Der Zeitbedarf der Bestätigung war klein gegen die Messdauer und wurde in der Auswertung vernachlässigt. Die Maschine war während der Messungen unter der normalen Grund-Systemlast: Das Betriebssystem, die systemnahen Prozesse und Dämonen liefen, aber neben dem Testprogramm waren keine weiteren Anwenderprogramme aktiv.

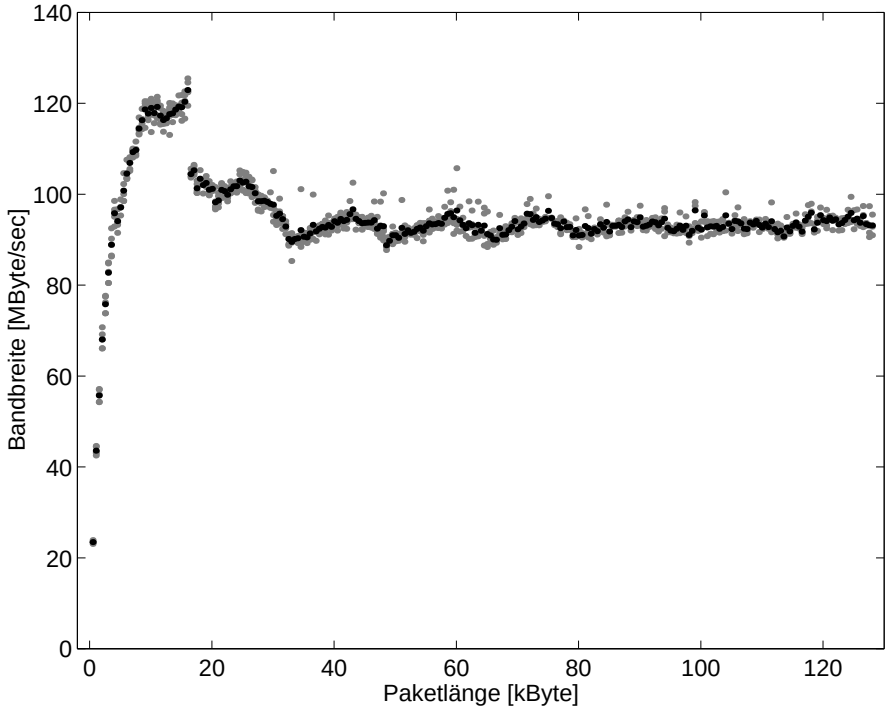

Abbildung 4.3: **Abhängigkeit der Bandbreite von der Paketlänge** – Es zeigt sich ein schneller Anstieg auf über 120 MByte/sec bei einer Paketlänge von 16 kByte. Danach bricht die Bandbreite ein und zeigt Schwankungen mit einer Periode von 16 kByte. (Sender und Empfänger befanden sich auf verschiedenen Rechenknoten einer Origin-200. Die grauen Punkte markieren Rohdaten, die schwarzen Mittelwerte.)

Abbildung 4.3 zeigt das Ergebnis dieser Messung. Bis zu einer Paketlänge von 16 kByte steigt die Bandbreite stark an und erreicht ihren Spitzenwert von circa 123 MByte pro Sekunde. Bei einer Paketlänge von 16 kByte bricht die Bandbreite um circa 15 % auf 104 MB/s ein.

Von dort an zeigen sich periodische Merkmale: Die Bandbreite erreicht bei Nachrichtenlängen, die ganzzahlige Vielfache von 16 kByte sind, jeweils ein lokales Minimum.

Wenn auch periodische Schwankungen in Bandbreiten gute Indizien für Puffer- und Paketierungseffekte darstellen, so lassen sich dennoch nicht alle kleineren Strukturen, wie das lokale Maximum bei circa 10 kByte dadurch erklären.

Puffereffekte

Die periodische Schwankung der Bandbreite bei Variation der Paketlänge lässt sich in der Tat durch Puffereffekte erklären. Im Handbuch zur MPI-Software von SGI findet sich folgender Hinweis: Die Software unterscheidet kleine Nachrichten von bis zu 64 Byte und große Nachrichten von mehr als 64 Byte.

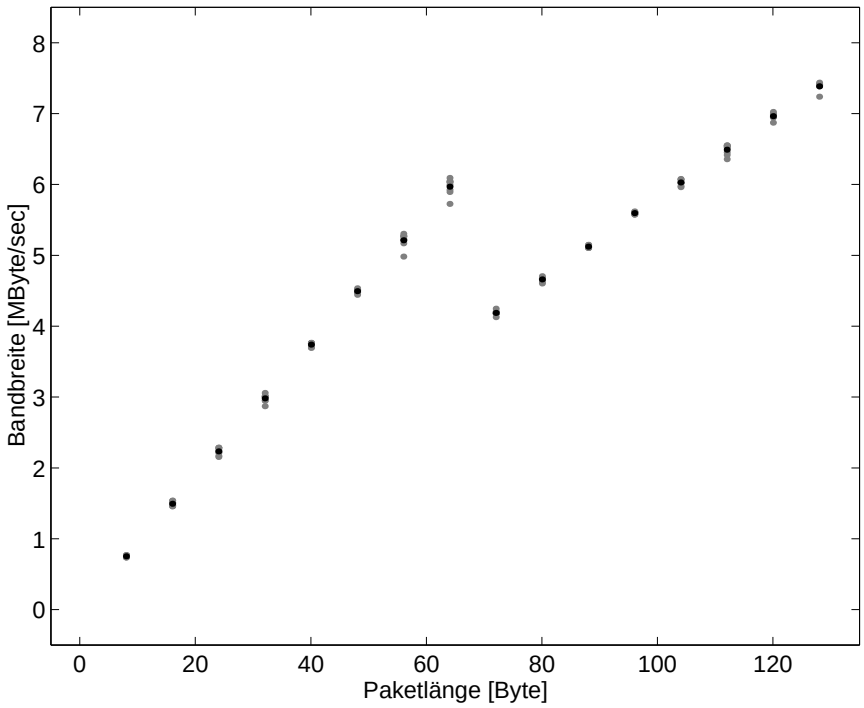

Abbildung 4.4: **Abhängigkeit der Bandbreite von der Paketlänge** – Darstellung des Bereiches kleiner Pakete und mittellanger Pakete um die kritische Länge von 64 Byte, bei welcher sich die Puffer-Strategie ändert.

Das System verwaltet für die langen Nachrichen einen Pool von Pufferblöcken. Die Anzahl der Blöcke ist auf 16 voreingestellt, aber dieses Limit ist durch Umgebungsvariablen modifizierbar. Jeder dieser Blocks ist 16 kByte groß. Dies erklärt die Periode der Schwankung.

Kleine Nachrichten

Um zu überprüfen, ob die Differenzierung zwischen kleinen und großen Nachrichtenlängen zu einem messbaren Unterschied führt, wurde die Bandbreitenmessung im Bereich dieses Überganges wiederholt.

Die Nachrichtenlänge wurde zwischen einer und 16 Fließkommazahlen variiert, also zwischen 8 und 128 Byte. Die Nachricht wurde ebenfalls 10^6-mal versandt. Es wurden 8 Messreihen durchgeführt. Abbildung 4.4 zeigt einen deutlichen 30%igen Einbruch der Bandbreite bei einer Paketlänge von 64 Byte.

Grobstruktur rechtfertigt LogGP

Die Annahme des LogP-Grundmodells, die Bandbreite liesse sich durch Verschicken kleiner Pakete modellieren, trifft nicht zu. Die Bandbreite ist keine Konstante, die

sich ergibt, wenn man lange Nachrichten als Sequenz von Paketen in der natürlichen Wortbreite des Prozessors oder des Netzwerkes betrachtet, denn in diesem Bereich kleiner Pakete liegt die Bandbreite zwischen 1 und 10 MByte/sec, während sie sich für Pakete größer als circa 32 kByte auf einem Wert um 100 MByte/sec einpendelt. Diese Werte unterscheiden sich um eine bis zwei Größenordnungen, also einen nicht vernachlässigbaren Faktor.

Dies wird in dem um große Bandbreiten erweiterte LogGP-Modell [Alex95] (siehe auch 3.4) berücksichtigt. Aber auch dieses Modell setzt die Bandbreite für große Nachrichten als Konstante G an und ist somit nur in der Lage, das Verhalten im Groben, aber nicht die periodische Struktur aufgrund der Paketierung wiederzugeben, welche immerhin Schwankungen von circa 10 % ausmacht.

Feinstruktur erfordert Kenntnisse über die Implementation

Sowohl die periodischen Strukturen als auch die nur vereinzelt auftretenden Effekte, wie der Übergang zwischen Regimes verschiedenen Verhaltens und das lokale Maximum bei 10 kByte, zu erklären, wäre nur mit einem implementierungsspezifischen Modell möglich. Dieses Modell müsste die Pufferungsstrategien der Message-Passing-Software ebenso widerspiegeln, wie diejenigen Einflüsse, die Maschinenzustand, Anwender und Anwendungssoftware auf die Anzahl und Größen der zur Verfügung stehenden Puffer ausüben können.

Zwischenbewertung

Bei der Modellierung der Bandbreite erweist sich das LogP-Modell als zu naiv. Die Darstellung langer Nachrichten als Sequenz kleinster Nachrichten, welche jeweils getrennt voneinander behandelt werden können, ist nicht geeignet, die Bandbreite korrekt zu beschreiben, sondern produziert Fehler von bis zu zwei Größenordnungen (vgl. Abb. 4.3 und 4.4 auf Seiten 42 und 43). Obwohl die nicht zutreffende Grundannahme von technischen Rahmenbedingungen inspiriert worden sein mag, wurde vermutlich zu wenig auf die konkrete Anwendbarkeit geachtet.

Bereits die einfache Erweiterung des LogP-Modells um eine weitere Konstante G zum LogGP-Modell bringt das Modell deutlich näher an die Realität und ermöglicht zumindest die Erfassung der richtigen Größenordnung, wenn auch nicht kleinerer Effekte.

Wollte man auch diese kleinen Effekte modellieren, so würde dies ein sehr implementationsspezifisches Modell erfordern, welches neben der Hardware des Verbindungsnetzes auch die Message-Passing-Software und Teile des Betriebssystems nachbildet. Da insbesondere die letzten beiden Komponenten häufigen technischen Modifikationen unterworfen sind[4], würde dies häufige Modellnachbesserungen erforderlich machen. In solche aufwendigen Projekte zu investieren, könnte nur im Interesse der

[4]Während dieser Messungen waren auch auf den Maschinen in Mainz und Dresden verschiedene Versionen von SGI-MPI installiert. Die geringe Differenz der Versionsnummern schlägt sich durchaus in einem sich in Details unterscheidenden Pufferverhalten wieder. Anhang B führt die Versionsnummern zusammen mit weiteren Konfigurationsinformationen auf.

jeweiligen Hersteller liegen. Dies von einem akademisch orientierten, allgemeinem und herstellerunabhängigem Modell zu fordern, wäre unangemessen.

4.2.4 Die Abhängigkeit der Bandbreite vom Grad der Contention

Wie bereits mehrfach diskutiert, setzt das LogP-Modell als Verbindungsnetz einen vollständigen Graphen voraus, der wirtschaftlich nicht realisierbar ist, da der Knotengrad mit $O(n)$ wächst. Daher werden in konkreten Rechnern Graphentypen eingesetzt, deren Knotengrad entweder beschränkt ist oder zumindest langsamer wächst.

Im hyperkubischen Verbindungsnetz der SGI Origin wächst der Knotengrad der Router nur mit $O(\log_2 n)$ bis zu einem Wert von $n = 6$, danach bleibt er konstant. Diese Abweichung vom vollständigen Graphen und die Reduzierung des Knotengrades hat zur Folge, dass nun nicht mehr jeder Kommunikationsweg nur eine Kante lang ist und somit mehrere Kommunikationspfade eine Kante gemeinsam nutzen müssen.

Wenn aufgrund dieser Mehrfachbelegung zwei Nachrichten an einem Knotenpunkt um die Weiterleitung über ein und dieselbe Kante konkurrieren, so bezeichnet man dies als *Contention*. Dass das Konzept der Contention im LogP-Modell nicht vorkommt, wurde bereits in Abschnitt 3.3.3 angesprochen. An dieser Stelle wird nachgewiesen, dass Contention einen messbaren Effekt auf die resultierende Bandbreite hat.

Contention aufgrund von Netztopologie und Kommunikationsmustern

Als experimentell zu prüfende Hypothese wird angenommen, dass die aus der Contention resultierende Bandbreitenbeschränkung primär vom *Grad* der Contention und nur sekundär von der *Länge* des Kommunikationspfades bei gegebener Contention abhängt. Unter Grad der Contention verstehen wir die Anzahl der Kommunikationspfade, die über dieselbe Kante des Verbindungsgraphen verlaufen.

Zu diesem Zweck wurden, wie in den vorangegangenen Messungen, Paare aus Sende- und Empfangsprozessen fest an bestimmte Prozessoren gebunden. Dabei wurde die Anzahl dieser Paare derart variiert, dass die resultierenden Wege eine gezielte Mehrfachbelegung auf den Kanten des hyperkubischen Verbindungsnetzes verursachten.

Fallunterscheidungen anhand des Verbindungsgraphen

Um gezielt die verschiedenen Grade von Contention auf unterschiedlichen langen Wegen hervorrufen zu können, wurden in einer Fallunterscheidung 15 Prozessorbindungen, durchnummeriert mit A bis O, untersucht. In diesen 15 Fällen wurde sowohl die Weglänge als auch der Grad der Contention variiert. Beispielsweise kommunizierten in Fall A zwei Prozessoren desselben Nodeboards ohne Routerdurchgang direkt miteinander, während in Fall O sich jeweils vier Prozessoren an Endpunkten befanden, die 5 Hops voneinander entfernt waren. Alle Kombinationen zeigt Tabelle 4.2 auf Seite 46.

Fall	Hops	Prozessoranzahl
A	0	2
B	1	2
C	1	4
D	2	2
E	2	4
F	2	8
G	3	2
H	3	4
I	3	8
J	4	2
K	4	4
L	4	8
M	5	2
N	5	4
O	5	8

Tabelle 4.2: Die 15 Fälle der Fallunterscheidung zur Steuerung der Contention.

Durchführung der Messung

Um den Einfluss auch großer Weglängen nachweisen zu können, musste diese Messung auf der Origin-2000 in Dresden durchgeführt werden. Die Maschine war für diese Messung reserviert und stand somit nur unter der Grundaktivität durch Betriebssystem und systemnahe Prozesse.

Dennoch traten während der Durchführung wiederholt kurze Lastspitzen auf. Diese wurden primär dadurch hervorgerufen, dass die starken Lastwechsel während des Messbetriebes große Anzahlen von Systemprozessen des Lastbalancierungssystems zeitgleich aktivierten.

Pro Messdurchgang wurden in jedem der in Tabelle 4.2 dargestellten Fälle 10^4 MPI-Nachrichten mit je 2^{16} Fließkommazahlen verschickt. Während der Zeitspanne, die die Maschine exklusiv zur Verfügung stand, konnten fünf dieser Durchgänge gefahren werden.

Ergebnisse bei Variation der Contention

Insgesamt zeigt die Variation der Contention einen größeren Effekt als die Zunahme der Weglänge. Daher wurden in Abbildung 4.6 (S. 48) die Fälle nach dem Grad der Contention gruppiert.

Obwohl die Maschine reserviert war, zeigen die Daten auch bei dieser Messung starke Streuung. Da der Rechner nicht ausreichend lange reserviert werden konnte, um den zählstatistischen Fehler signifikant zu senken, lassen sich aus den Daten keine exakten quantitativen Aussagen ableiten, sondern nur grundlegende Zusammenhänge ablesen.

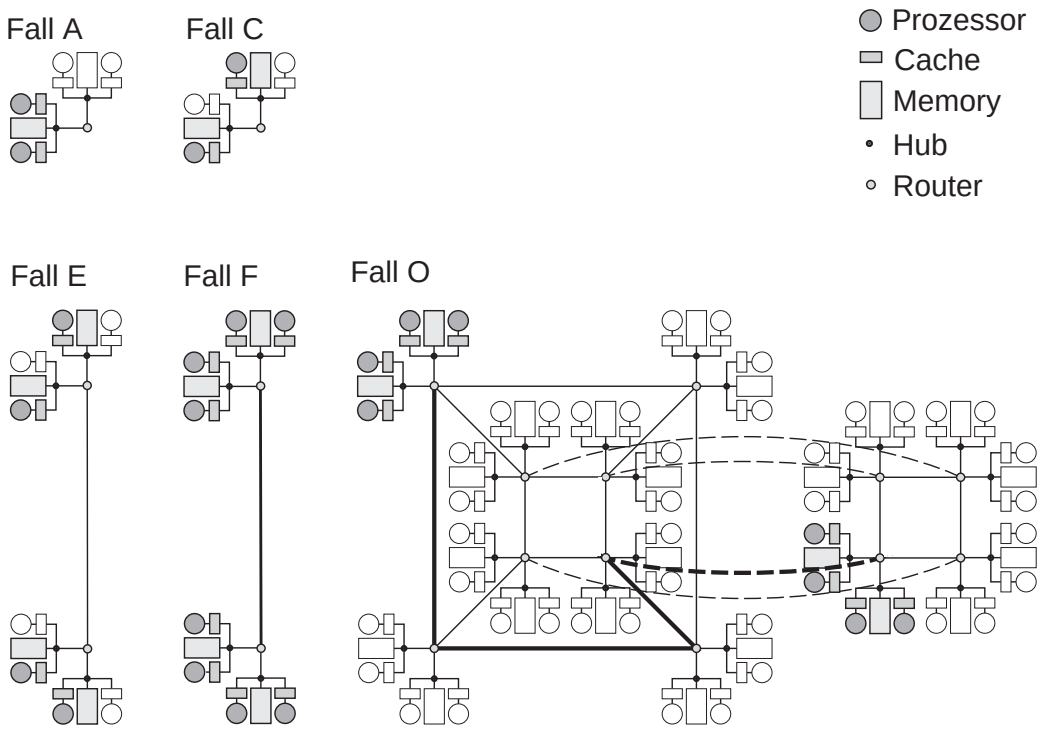

Abbildung 4.5: **Fallunterscheidung zur Erzeugung verschiedener Grade von Contention** – Die Graphik zeigt Beispiele zu verschiedenen Weglängen und Prozessoranzahlen. Für Fall O ist einer der möglichen Wege hervorgehoben. Eine komplette Aufstellung aller untersuchten Fälle enthält Tabelle 4.2.

Die Daten aus den Messungen mit einfacher Contention zeigen Bandbreiten von etwa 120 MByte/s und setzen sich damit deutlich von denjenigen mit zweifacher Contention und circa 80 MByte/s ab. Bei einer Contention von 4 beträgt die Bandbreite noch etwa 70 MByte/s (vgl. Tabelle 4.3). Diese Raten beziehen sich auf die zwischen jedem der Sender-Empfänger-Paare nutzbare Bandbreite.

Trotz der Streuung der Daten lässt sich klar ablesen, dass die resultierenden Mittelwerte der Bandbreiten weder linear noch antiproportional von der Contention abhängen. Eine Verdopplung der Contention von 1 auf 2 verursacht lediglich einen Bandbreitenabfall um 40 MByte/s auf 67%. Noch deutlich schwächer fällt der Abfall bei der Vervierfachung der Contention aus. Statt auf 25% fällt die Bandbreite lediglich auf 58%.

Contention	**Applikations-bandbreite**	**ausgenutzte Routerbandbreite**
1	120 MByte/s	120 MByte/s
2	80 MByte/s	160 MByte/s
4	70 MByte/s	280 MByte/s

Tabelle 4.3: Applikations- und Routerbandbreiten bei Variation des Grades der Contention.

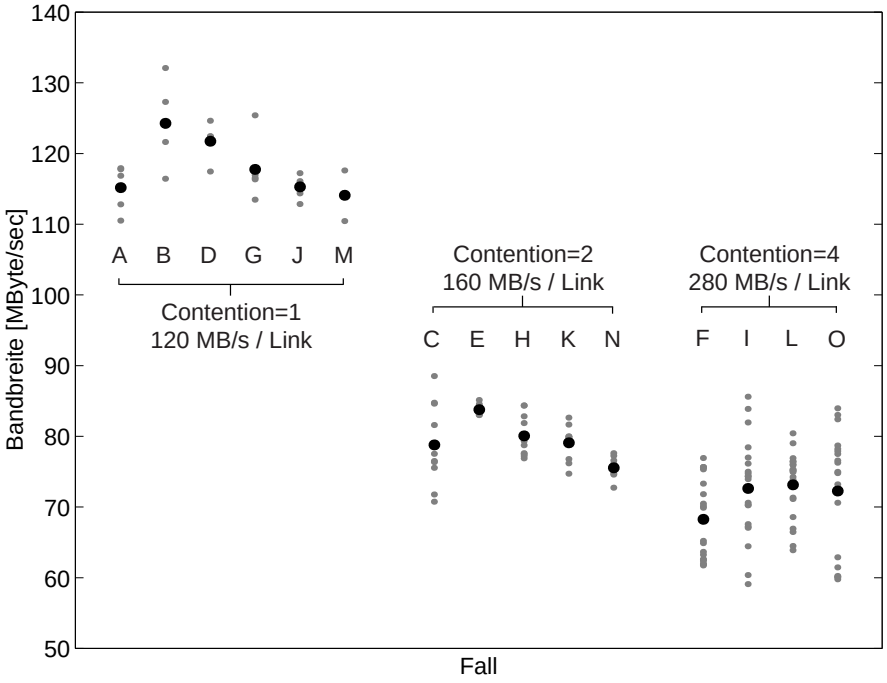

Abbildung 4.6: **Die Messergebnisse der Fallunterscheidung** – Die Fälle wurden nach dem Grad der Contention gruppiert. Es ist deutlich abzulesen, dass die Bandbreite pro Prozesspaar zwar sinkt, pro Kante des Verbindungsnetzes (Link) jedoch mit zunehmender Contention ansteigt. (Die grauen Punkte stellen die Rohdaten, die schwarzen Kreise die Mittelwerte dar.)

Da jeweils alle Sender und alle Empfänger an einen gemeinsamen Router angeschlossen sind, lassen sich die Bandbreiten der Applikationen durch direkte Addition auf die Gesamtbandbreite der Router zurückrechnen. Wie Tabelle 4.3 zeigt, steigt die Gesamtbandbreite der Router mit zunehmender Contention auf bis zu 280 MByte/sec.

Um diesen Anstieg bewerten zu können, muss er mit den technischen Spezifikationen der Maschine verglichen werden. Im Artikel von Galles [Gall96] finden sich detaillierte technische Informationen zum Spider-Router, darunter auch die Tatsache, dass dieser Router pro Kanal und Richtung eine maximale Übertragungsrate von 800 MByte/sec unterstützt. Diese Angaben bestätigt auch das Datenblatt der Origin [SGI99b] und die Leistungsanalyse durch Jiang und Singh [Jian98]. Die Kapazität des Verbindungsnetzes ist somit deutlich höher als die Bandbreiten, die Applikationen mittels Message-Passing über MPI erreichen können. Solange nicht die Übertragungsgrenze einzelner Kanten des Verbindungsnetzes erreicht wird, ist ein linearer Zusammenhang zwischen Grad der Contention und Abfall der Bandbreite nicht zu erwarten.

Ergebnisse bei Variation der Weglänge

Insbesondere bei großer Contention zeigen die Messwerte trotz reservierter Maschine starke Streuung, so dass die Auswirkung der Weglänge nicht quantitativ ausgewertet werden kann. Insgesamt zeigt sich jedoch bei den Daten zu Contention-Graden von 1 und 2 eine Abnahme der Bandbreite bei zunehmender Weglänge. Dies lässt sich durch die Tatsache erklären, dass die Auswirkungen der Konkurrenz mehrerer Pakete um denselben Ausgangskanal eines Routers mit der Anzahl der zu durchlaufenden Router zunehmen.

Der Vollständigkeit halber muss auch erwähnt werden, dass die Festlegung der Kommunikationsendpunkte ab einer Weglänge von drei Hops den Kommunikationspfad nicht mehr eindeutig festlegt. Ursache dafür ist die mit der Prozessorzahl wachsende Bisektionsweite (und damit auch wachsende Bisektionsbandbreite). Gerade diese Eigenschaft macht den Hyperkubus zu einem geeigneten Verbindungsgraphen für Parallelrechner. In einer Variante von *Greedy*-Routing, einem dimensionsbasierten Verfahren, ließe sich durch Festlegung der Endpunkte der Pfad weitgehend festlegen. Noch Parallelrechner der letzten Generation, wie der Intel Paragon, verwendeten zweidimensionales Greedy-Routing. Die Spider-Router der Origin arbeiten jedoch nach einem dynamischen Verfahren, welches den Zielknoten, die Priorität der Nachricht und die von ihr bereits zurückgelegte Wegstrecke ebenso berücksichtigt wie Umgebungsparameter (z. B. die Last auf den anschliessenden Kanten).

Zusätzlich bestätigt sich die schon in Abschnitt 4.2 angesprochene Beobachtung, dass die durch Verwendung von Message-Passing-Software vollzogene Abstraktion von der konkreten Hardware durch drastische Leistungseinbußen erkauft wird. Analog zum Anstieg der Latenzzeiten bleibt bei Verwendung von MPI die durch Applikationen nutzbare Bandbreite deutlich unter dem durch die Hardware gesetzten Limit.

Zwischenbewertung

Einen quantitativen Zugang zu den Auswirkungen von Contention auf die resultierende Bandbreite zu finden, erwies sich als schwierig.

In der gewählten Maschine arbeitet das Routing nicht mehr nach einem der einfachen Schemata früherer Parallelrechner, sondern nach einem komplizierten dynamischen Verfahren, um Nachrichten bei geringen Kollisionsraten mit hohen Bandbreiten weiterzuleiten. Somit lässt sich nur in Sonderfällen ein bestimmter Grad von Contention durch so einfache Methoden wie die Platzierung der Kommunikationsendpunkte gezielt hervorrufen.

Dennoch zeigen bereits die stark streuenden Daten dieser Untersuchung, dass die Auswirkungen schon einfacher Contentioneffekte deutlichen Einfluss auf die Leistungsparameter einer Kommunikation ausüben können. Die Arbeit von Moritz und Frank [Mori98] zeigt, wie das LogP-Modell um ein Contention-Konzept zu einem LoGPC-Modell erweitert werden kann, wodurch es bei hardwarenaher[5] Parametri-

[5] Die in [Mori98] beschriebene Modellbildung wurde sehr konkret für den Forschungsrechner MIT-Alewife vorgenommen. Im Gegensatz zu den vier Parametern des originalen LogP-Modells enthält

sierung eine verbesserte Vorhersage des Laufzeitverhaltens einer Message-Passing-Kommunikation erlaubt.

4.3 Prüfung des LogP-Konzeptes in der Anwendung

In den vorangegangenen Abschnitten wurden einzelne Aspekte des LogP-Modells untersucht und die Gültigkeit gewisser Grundannahmen des Modells isoliert überprüft. Nun soll das Gesamtsystem aus Maschine und einer Anwendung betrachtet werden.

Parallelrechner werden primär betrieben, um durch Verwendung mehrerer Prozessoren eine Verbesserung der Verarbeitungsleistung gegenüber der sequentiellen Verarbeitung zu erzielen. Um zu ermitteln, in welchem Maße die Rechenleistung bei Hinzunahme weiterer Prozessoren steigt, wird üblicherweise eine Skalierungsuntersuchung vorgenommen.

Zuerst behandeln die Abschnitte 4.3.1 und 4.3.2 die Fragen, welche Parameter skaliert und in welcher Weise die einzelnen Prozesse den Prozessoren zugeteilt werden sollen. Anschliessend wird in 4.3.3 die Durchführung der eigentlichen Skalierungsmessung beschrieben. Abschnitt 4.3.4 diskutiert den auftretenden sogenannten *superlinearen Speedup*, der in 4.3.5 durch Effekte in den Prozessor-Caches erklärt wird.

4.3.1 Gängige Skalierungsklassen

In einer Skalierungsuntersuchung sollte im Allgemeinen nur ein einzelner Parameter variiert werden, damit sein Einfluss auf die Messgröße möglichst direkt ermittelt werden kann. In der wissenschaftlichen und technischen Literatur zu Parallelrechnern gibt es keine Einigung, welcher Parameter zu variieren sei. Insbesondere variieren Verkäufer von Parallelrechnern meist andere Parameter als Käufer es tun. Jedoch haben sich Klassen von Skalierungsuntersuchungen herausgebildet. Meist unterscheidet man zwischen *user-oriented* und *resource-oriented properties*.

Culler [Cull99], Kapitel 4.1.4, unterscheidet über den fixen Parameter die drei Klassen PC-, TC- und MC-Skalierung, die im Folgenden kurz dargestellt werden.

PC-Scaling

Problem-Constraint (PC) Scaling geht von dem Szenario aus, dass ein Anwender ein gegebenes Problem mit einem Parallelrechner bearbeitet. Dabei wird die Größe der Eingabe, beispielsweise die Anzahl der Eingabe-Datensätze als konstant angenommen und beobachtet, wie die Hinzunahme zusätzlicher Prozessoren die Lösungsfindung beschleunigt.

Zu dieser Skalierungsklasse ist zu bemerken, dass der relative Overhead durch Parallelverarbeitung bei PC-Scaling im Allgemeinen mit der Anzahl der Prozessoren anwachsen wird, da die Problemgröße pro Prozessor sinkt.

LoGPC mehr als 20 Parameter, um die speziellen Details der Hardware des Verbindungsnetzes von Alewife wiederzugeben.

TC-Scaling

Bei *Time-Constraint (TC) Scaling* ist die Ausführungszeit der Berechnung konstant zu halten. Bei Hinzunahme zusätzlicher Prozessoren wird untersucht, wie die maximale, noch unter der Zeitvorgabe lösbare, Größe der Eingabe wächst.

Culler zählt eine Reihe von praktischen und messtechnischen Problemen auf, die bei der Durchführung einer Messung TC-Scaling auftreten können.

MC-Scaling

Bei *Memory-Constraint (MC) Scaling* wird der Speicherplatz pro Rechenknoten als konstant angenommen. Bei Hinzunahme weiterer Prozessoren wird die Problemgröße linear angepasst. Untersuchungsgegenstand ist die Zunahme der Laufzeit.

Eine Messung von MC-Scaling durchzuführen, wird problematisch, wenn die Ausführungszeit stärker wächst als der Platzbedarf einer Berechnung. Bei der in dieser Arbeit untersuchten Matrixmultiplikation wächst der Speicherplatz von $n \times n$-Matrizen lediglich mit $O(n^2)$, wohingegen die Ausführungszeit in der Ordnung $O(n^3)$ ansteigt. Dieser Unterschied macht MC-Scaling meist unpraktikabel. Ein analoges Problem besteht bei TC-Scaling und schwachem zeitlichen Wachstum.

Die gewählte Skalierungsklasse

Für diese Untersuchung wurde PC-Scaling gewählt. Aufgrund der Aufwände in Zeit und Platz ist MC-Scaling nicht durchführbar und TC-Scaling hätte zu viel Reservierungszeit auf dem Testrechner erfordert.

Generell musste bei der Wahl der Problemgröße ein Kompromiss gefunden werden. Einerseits musste die Ausführungszeit auch bei maximaler Prozessoranzahl noch hinreichend groß gegenüber dem Fehler und der Auflösung der Zeitmessung sein. Andererseits durfte die sequentielle Zeitmessung keine allzu lange Reservierung des Testrechners erfordern. Bei den 48 Prozessoren der SGI Origin in Dresden unterschieden sich die Ausführungszeiten als Funktion der Prozessoranzahl um bis zu zwei Größenordnungen.

Schließlich wurden für die Eingabe $n \times n$-Matrizen mit $n = 2400$ gewählt. Bei dieser Eingabegröße ließen sich die Ausführungszeiten sowohl mit hinreichender Genauigkeit messen als auch innerhalb der reservierten Zeiträume durchführen. Auch die zusätzliche Anforderung, dass sich die Eingabe auf alle untersuchten Prozessoranzahlen aufteilen lässt, wird durch die gewählte Matrixgröße erfüllt.

4.3.2 Prozessorzuteilung

Die Prozessorzuteilung (*Process Placement*), die Zuordnung der Anwendungsprozesse zu den einzelnen Prozessoren, ist einer der wesentlichen Faktoren, mit denen sich die Leistung eines Parallelrechners steuern und optimieren lässt und damit permanenter Forschungsgegenstand.

Als Einführung in dieses umfassende Gebiet sei das Buch „Prozessorzuteilung in Parallelrechnern“ [Heis94] von Heiss genannt. Obwohl manche der Grundannahmen über technische Realisierungen oder den praktischen Betrieb (beispielsweise die Behauptung, Parallelrechner würden vielfach nicht Multitasking-, sondern nur Einprogrammbetrieb zulassen) aktuellen Parallelrechnern nicht mehr entsprechen, ist die Darstellung der grundlegenden Problematiken noch gültig. Heiss unterscheidet dazu das Aufteilungsproblem, das Prozessorverwaltungsproblem, das Einbettungsproblem, das Kontraktionsproblem und das Migrationsproblem. Moderne Arbeiten, die auch auf die konzeptionellen und technischen Raffinessen moderner Parallelrechner eingehen, finden sich beispielsweise in den *Transactions* der IEEE und ACM.

Zuteilung durch MPI

Als plattformübergreifender Standard beschränkt sich Message-Passing-Interface (MPI) auf die gängigsten Funktionalitäten aktueller Betriebssysteme. Da sich deren Konzepte zur Prozess- und Prozessor-Organisation deutlich voneinander unterscheiden[6], wurden bisher keine Systemaufrufe zur Prozessorzuteilung in Interface-Normierungen wie POSIX[7] aufgenommen. Somit bietet auch MPI selbst keinen Zugang zu einer solchen Zuordnung.

Auf der Origin optimiert das SGI-MPI die Kommunikationsleistung insofern für den NUMA-Betrieb, als es die Prozesse bevorzugt auf demjenigen Rechenknoten laufen lässt, in dessen Speicher die Prozesse Daten alloziert haben. Diese Platzierung wird daraufhin über den Programmlauf beibehalten.

Zuteilung durch den Scheduler

Auf einem Shared-Memory-Parallelrechner kann der Scheduler einen Prozess auf jedem der verfügbaren Prozessoren ausführen. Dabei kann sich die Platzierung des Prozesses von Zeitscheibe zu Zeitscheibe aufgrund veränderter Scheduling-Bedingungen ändern. Ein Scheduler wird einen gegebenen Prozess bevorzugt immer wieder auf demselben Prozessor laufen lassen, um eventuell noch vorhandene Kontextinformationen, beispielsweise in den Prozessor-Caches, wiederverwerten zu können. Wenn ein Scheduler diese Funktionalität enthält, so bezeichnet man dies oft als eine Implementierung von *Soft Affinity*. Der Einfluss des Schedulings auf das Caching-Verhalten von Prozessoren ist jedoch ein nur in den Grundzügen erforschtes Gebiet, so dass sich noch keine allgemein gebräuchliche Terminologie ausgebildet hat.

Auf der SGI-Origin wird der Scheduler des Betriebssystems IRIX auf einen Prozessor, auf dem ein MPI-Prozess läuft, keinen weiteren Prozess legen, da SGI-MPI die Empfangsfunktionen über *Polling*, das permanente Abfragen der Kommunikationskanäle, realisiert, um die Latenz zu minimieren.

[6] Als Beispiel sei die HP X-Class (ehemals Convex Exemplar) genannt. Deren Betriebssystem SPP/UX erlaubt keine Bindung von Prozessen an Prozessoren, sondern lediglich an Prozessorgruppen, die sogenannten Hypernodes.

[7] Das Buch von Gallmeister [Gall95] enthält einen Überblick über die Programmierung mittels POSIX.

Explizite Zuteilung

In dieser Untersuchung wurden bereits bei der Prüfung der Grundannahmen des LogP-Modells in Abschnitt 4.2 Prozesse durch Systemaufrufe explizit an bestimmte Prozessoren gebunden.

Dadurch ließen sich sowohl reproduzierbare Verhältnisse schaffen als auch der Einfluss verschiedener Prozessorzuteilungen auf die Ausführungszeit ermitteln.

„Gute“ und „schlechte“ Konfigurationen

Im folgenden Abschnitt 4.3.3 werden Skalierungsmessungen für eine „gute“ und eine „schlechte“ Konfiguration unterschieden. Als Güte wurde die Summe der Abstände zwischen den im Anwendungsprogramm direkt miteinander kommunizierenden Prozessen im Verbindungsgraphen verwendet.

Um für die irregulären Kommunikationsmuster des Fox-Algorithmus eine beste und schlechteste Konfiguration zu finden, wurden in einem Monte-Carlo-Verfahren mittels des Computeralgebra-Systems *Mathematica* [Wolf96] zufällige Permutationen von Prozessorzuteilungen gewürfelt und mit der bisher besten und schlechtesten Konfiguration verglichen. In Anhang A werden alle verwendeten Konfigurationen aufgeführt.

4.3.3 Skalierungsmessung

Als Ausgangspunkt für die Skalierungsmessung soll die Definition des *Speedups* aus Standard-Lehrbüchern für Computerarchitektur übernommen werden. Hennessy und Patterson [Henn96] definieren Speedup in Abschnitt 1.6, „Quantitative Principles of Computer Design“, S. 29, als ein Verhältnis:

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Das beschriebene „Enhancement“ wäre in diesem Fall die Hinzunahme mehrerer Prozessoren. Culler, Singh und Gupta [Cull99] definieren Speedup in Abschnitt 4.1.1, „Basic Measures of Multiprocessor Performance“, S. 203, schlicht als:

$$\text{Speedup} = \frac{\text{Time(1 proc)}}{\text{Time}(p \text{ proc})}$$

Durchführung der Messung

Für diese Messung konnte erneut die Origin-2000 der TU-Dresden reserviert werden. Es zeigten sich während der Durchführung lediglich die bereits beschriebene Grundlast und die sporadischen Lastspitzen durch das Lastbalancierungssystem.

Um die Verkürzung der Ausführungszeit bei Hinzunahme weiterer Prozessoren zu untersuchen, wurde die im Buch von Pacheco [Pach97] genauer beschriebene Implementierung des Fox-Algorithmus auf 1, 4, 9, 16, 25 und 36 Prozessoren ausgeführt.

Prozessoranzahl	„gute“ Konfiguration	„schlechte“ Konfiguration
4	1287.0 ms	2300.1 ms
9	3019.7 ms	3159.7 ms
16	1075.7 ms	1099.1 ms
25	899.1 ms	820.5 ms
36	453.8 ms	435.4 ms

Tabelle 4.4: Zeitbedarf der Kommunikationsphasen zur Verteilung von A.

Die benötigte Zeit wurde vom Prozess mit MPI-Rang 0 über die Echtzeituhr des Systems gemessen.

Zu jeder der Prozessoranzahlen konnten in der zur Verfügung stehenden Zeit drei Messreihen durchgeführt werden. Die zu multiplizierenden Matrizen enthielten jeweils 2400×2400 Fließkommazahlen doppelter Genauigkeit.

Zusätzlich zur Messung der Gesamtausführungszeit wurde die Zeit an vier Messpunkten in der Hauptschleife der Multiplikationsroutinen jedes Ranges gemessen. Durch diese Messpunkte ist es möglich, innerhalb der Matrixmultiplikation

$$C = A \cdot B$$

den Zeitbedarf der Kommunikationsphasen zur Verteilung der Teilmatrizen von A und B separat zu bestimmen.

Einfluss der Prozessorzuteilung

Um den Einfluss der Prozessorzuteilung zu ermitteln, wurden die Messpunkte aus den Protokolldateien extrahiert und separat ausgewertet. Abbildung 4.7 zeigt Ausführungszeiten der Kommunikationsphasen zur Verteilung von A in Millisekunden. Die Streuung der Datenpunkte ist in diesem Fall keine Folge der Systemlast. Die einzelnen Punkt-zu-Punkt-Verbindungen sind teilweise voneinander abhängig und daher müssen die Empfangsprozesse unterschiedlich lange auf das Eintreffen ihrer Daten warten. Dieser systematische Effekt wird dadurch verstärkt, dass das parallele Programm keine explizite Synchronisation durch sogenannte *Barrieren*[8] enthält und somit nicht künstlich „in Phase“ gehalten wird. Aus Abbildung 4.8 (S. 56) lässt sich deutlich erkennen, wie die einzelnen Messpunkte „außer Phase laufen“.

Die Mittelwerte der einzelnen Kommunikationszeiten sind in Tabelle 4.4 für A und in Tabelle 4.5 für B dargestellt.

Bei der Verteilung von A zeigt sich insbesondere bei geringen Prozessoranzahlen ein deutliches Ansteigen der Ausführungszeit beim Übergang von der guten auf die

[8] Das Lehrbuch von Tanenbaum „Distributed Operating Systems“ [Tane95] definiert in Abschnitt 6.3, „Consistency Models“ Barrieren folgendermaßen: „A *barrier* is a synchronization mechanism that prevents any process from starting phase $n + 1$ of a program until all processes have finished phase n. When a process arrives at a barrier, it must wait until all other processes get there too.“

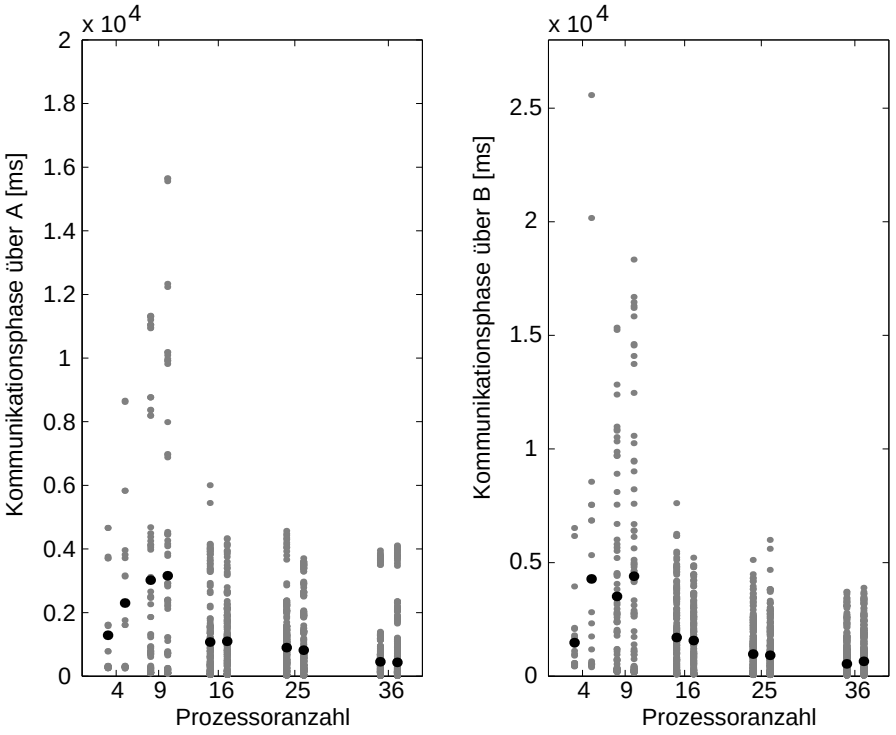

Abbildung 4.7: **Der Einfluss der Prozessorzuteilung auf die Ausführungszeit der Kommunikationsphasen** – Dargestellt sind die Kommunikationsphasen über A (linke Graphik) und B (rechte Graphik). Die Daten der „guten“ Konfiguration sind leicht links, die der „schlechten“ Konfiguration leicht rechts gegenüber der Markierung der Prozessoranzahl versetzt. (Die grauen Datenpunkte kennzeichnen Rohdaten, die schwarzen Punkte deren Mittelwerte.)

schlechte Konfiguration. Jedoch zeigen sich auch gegenteilige Effekte, da die Kommunikation in der schlechten Konfiguration auch schneller ablaufen kann, als in der guten. Ähnliche Effekte wiederholen sich bei der Kommunikation zur Verteilung von B. Hier ist insbesondere zu erwarten, dass der große Unterschied bei vier Prozessoren von einzelnen Ausreißern beeinflusst wurde (vgl. Darstellung der Rohdaten in Abbildung 4.7.)

Prozessoranzahl	**„gute“ Konfiguration**	**„schlechte“ Konfiguration**
4	1476.8 ms	4279.5 ms
9	3512.5 ms	4399.1 ms
16	1703.5 ms	1567.9 ms
25	976.2 ms	921.5 ms
36	546.1 ms	656.1 ms

Tabelle 4.5: Zeitbedarf der Kommunikationsphasen zur Verteilung von B.

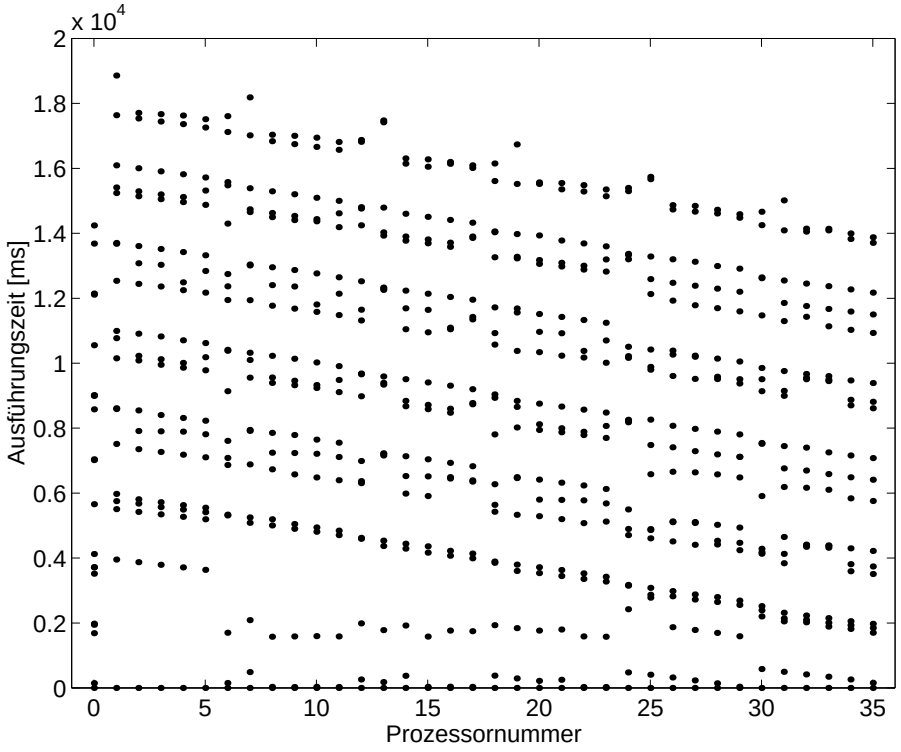

Abbildung 4.8: **Das Fox-Programm läuft „außer Phase“** – Da die Implementierung des Fox-Algorithmus keine Barrieren-Synchronisation enthält, synchronisieren sich immer nur einzelne Prozesspaare. Das Gesamtprogramm läuft daher „außer Phase“. Die Korrektheit der Berechnung wird dadurch nicht eingeschränkt, der Programmablauf jedoch beschleunigt. (Die in diesem Diagramm eingetragenen Punkte bezeichnen das Auftreten bestimmter, lediglich zum Zwecke der Messung ausgezeichneter, Zustände eines der Messdurchläufe mit 36 Prozessoren.)

Aus den in diesem Abschnitt dargestellten Messergebnissen lässt sich der Zusammenhang zwischen Prozessorzuordnung und Ausführungszeit der Kommunikationsphasen nicht qualitativ oder quantiativ ableiten. Um den in Abschnitt 4.2 nachgewiesenen Einfluss der Weglänge auf die Latenz von einzelnen Nachrichten auf eine vollständige Anwendung zu übertragen, müssten Messdaten mit mehr Stützstellen bei besserer Zählstatistik ausgewertet werden.

Einfluss der Prozessoranzahl

Nun soll der eigentliche Speedup untersucht werden. Auf der linken Seite von Abbildung 4.9 sind die Ausführungszeiten für die Matrixmultiplikation mit den verschiedenen Prozessoranzahlen dargestellt. Man erkennt deutlich das starke Abfallen der Ausführungszeit. Auf der rechten Seite ist entsprechend die Arbeit, das Produkt aus Prozessoranzahl und Ausführungszeit, aufgetragen. Trotz des bei wachsender Prozessoranzahl zunehmenden Parallelisierungs-Overheads steigt die Arbeit nicht an, sondern fällt bis zu einer Anzahl von 36 Prozessoren.

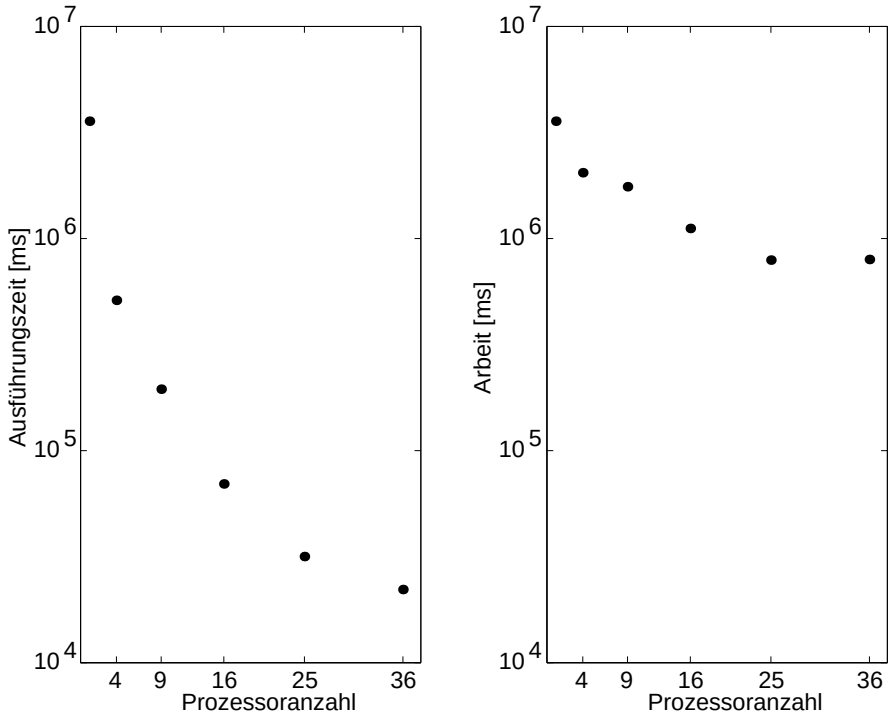

Abbildung 4.9: **Die Skalierungsuntersuchung zeigt superlinearen Speedup** – In dieser Untersuchung wurde die MPI-Implementierung des Fox-Algorithmus auf 1, 4, 9, 16, 25 und 36 Prozessoren auf der Origin-2000 ausgeführt und die benötigte Zeit gemessen. Die linke Teilgraphik zeigt diese Ausführungszeiten, gemessen aus MPI-Rang 0 mit der Echtzeituhr des Systems. In der rechten Teilgraphik ist die Arbeit, das Produkt aus Ausführungszeit und Prozessoranzahl, aufgetragen. Da die Arbeit mit zunehmender Prozessoranzahl fällt handelt es sich um einen sogenannten *superlinearen Speedup*.

4.3.4 Superlinearer Speedup

Nach der graphischen Darstellung in Abbildung 4.9 geht die Ausführungszeit augenscheinlich stärker als antiproportional mit der Prozessoranzahl zurück. Tabelle 4.6 listet die gemittelten Werte für Ausführungszeit und Arbeit bei den einzelnen Prozessoranzahlen auf und berechnet den Speedup.

Prozessoranzahl	**Ausführungszeit**	**Arbeit**	**Speedup**
1	3575916 ms	3575916 ms	1
4	511065 ms	2044260 ms	7.0
9	194993 ms	1754937 ms	18.3
16	69716 ms	1115456 ms	51.3
25	31658 ms	791450 ms	113.0
36	22125 ms	796500 ms	161.6

Tabelle 4.6: Zeit, Arbeit und Speedup als Funktion der Prozessoranzahl.

Offensichtlich ist der Speedup durch eine Prozessoranzahl P größer als P selbst. Diese Relation

$$S(P) > P$$

nennt man *superlinearen Speedup* (vgl. z.B. [Cull99], [Sun95]).

Superlinearer Speedup auf algorithmischer Ebene nicht möglich

Aus Sicht der Informatik ist ein solcher superlinearer Speedup nicht möglich. Um diese Diskrepanz zu beleuchten, müssen wir ein wenig genauer auf die jeweiligen Definitionen des Speedup eingehen. Im Lehrbuch von Leighton [Leig92] findet sich auf S. 8 folgende Ausführung:

> „In general, we would like to develop parallel algorithms that have as much speedup als possible. In particular, given P processors, we would like our parallel algorithm to run P times as fast as the best sequential algorithm. When we can attain such performance (i.e., when $S = P$, or even when $S = \Theta(P)$), we say that the parallel algorithm archives *linear speedup*. Unfortunately, this is often hard to do, particularly for problems such as sorting.“

Ferner wird auf den Punkt gebracht:

> „Note that a parallel algorithm can never attain greater than linear speedup.“

Es folgt eine Argumentation über die Effektivität der Simulation einer parallelen auf einer sequentiellen Maschine.

Die offensichtliche Diskrepanz zu unserem Nachweis eines superlinearen Speedups klärt sich durch die Definition des Begriffes der Zeit. In dieser Untersuchung wird „Zeit“ mit dem in der Physik üblichen Zeitbegriff gleichgesetzt und in Sekunden gemessen. Leighton verwendet Zeit als ein Synonym für die Anzahl ausgeführter Operationen. Dabei wird implizit eine, in einer Art von globalem Gleichtakt laufende, Maschine unterstellt, in der allen Operationen auf allen Operanden die gleiche Ausführungszeit zugewiesen werden kann.

Das praktische Auftreten eines superlinearen Speedups erklärt sich aus der Tatsache, dass in dem hier untersuchten realen Rechner die Ausführungszeit weder für alle Operationen, noch für alle Operanden gleich ist. Damit können superlineare Speedups bei der Ausführung von konkreten parallelen Programmen auftreten, bei parallelen Algorithmen jedoch nicht.

4.3.5 Der Einfluss von Prozessor-Caches

Das Auftreten superlinearer Speedups ist zwar an mehreren Stellen in der Fachliteratur belegt, wird aber dennoch kontrovers diskutiert.

Sun und Zhu nennen in ihrer Untersuchung der Leistungsfähigkeit von Parallelrechnern [Sun95] mehrere mögliche Ursachen für superlinearen Speedup, darunter den

Einfluss von Caches. In ihrer Analyse der Wirtschaftlichkeit des Parallelrechner-Einsatzes unter kommerziellen Gesichtspunkten [Yan99] argumentieren Yan und Zhang explizit über die Ausnutzung superlinearen Leistungswachstums bei lediglich linear wachsenden Kosten.

Systematisch betrachtet, lässt sich superlinearer Speedup als Konsequenz der gewählten Skalierungsklasse identifizieren (vgl. 4.3.1). Für die vorliegenden Untersuchungen wurde aufgrund der begrenzten Reservierungszeit des Testrechners das Problem-Constraint-Scaling gewählt. Bei zunehmender Prozessoranzahl entfällt auf jeden der Prozessoren ein zunehmend kleinerer Anteil der Datenmenge. Da die Größe der Prozessorcaches konstant bleibt, das von ihnen zu puffernde Volumen jedoch sinkt, steigt der Anteil von Daten, auf die schnell zugegriffen werden kann.

Ein vergleichbarer Effekt ließe sich bei sehr viel größeren Datenmengen erneut beobachten: Bei virtuellem Speichermanagement dient der Hauptspeicher quasi als Puffer[9] des Festplattenspeichers. Berechnungen auf Datenmengen, die größer als der Speicher eines Rechenknotens sind, werden bei Hinzunahme weiterer Rechenknoten schneller ausgeführt, falls der Speicher pro Rechenknoten konstant gehalten wird.

Eine solche statische Betrachtung ist jedoch nicht ausreichend, um den Einfluss des Caches auf das Laufzeitverhalten eines Prozesses zu erfassen. Lediglich in dem extremen Fall, dass die gesamten Daten eines Problems in den Cache passen, lässt sich auf die oben dargestellte Weise die Auswirkung des Caches abschätzen. Da die Effektivität von Caches jedoch vor allem von den Zugriffsmustern einer Anwendung abhängt und nicht von ihrem Speicherplatzverbrauch, ist es notwendig, insbesondere die Dynamik des Cache-Verhaltens zu modellieren, um quantitative Aussagen über das Laufzeitverhalten anstellen zu können. Eine solche Modellierung wird im folgenden Kapitel vorgestellt.

[9] Daher nennt man bei UNIX-Betriebssystemen die entsprechende Komponente des Speichermanagers auch *Buffer Cache*. Für virtuelles Speichermanagement sei auf Kapitel 7.4, „Virtual Memory“, des Standardwerkes von Patterson und Hennessy [Patt98] verwiesen.

Kapitel 5

Simulation: Modellierung des Verhaltens der Speicherhierarchie

In diesem Kapitel soll eine Modellierung des Verhaltens einer Speicherhierarchie entwickelt werden. Dazu wird zunächst die Funktionsweise eines Prozessorcaches beschrieben und ein Speicherhierarchie-Simulator vorgestellt, der diese Funktionsweise nachbildet. Mittels des Simulators wird anschließend das Zugriffsverhalten der Matrixmultiplikation simuliert und mit experimentell gemessenen Daten verglichen.

5.1 Funktionsweise eines Prozessorcaches

Um die in Abschnitt 4.3 beobachteten Effekte zu erklären, muss auf die Kontroll-Logik eines Prozessorcaches eingegangen werden. Caches moderner Mikroprozessoren sind komplizierte technische Gebilde, die im Rahmen dieser Arbeit leider nicht umfassend dargestellt werden können. Daher sei auf die Standardliteratur zu diesem Gebiet hingewiesen. Die Bücher von Przybylski [Przy90] sowie Hennessy und Patterson [Henn96], [Patt98] enthalten umfassende Darstellungen über Caching. Lenoski et. al. [Leno95] und Culler et. al. [Cull99] beschreiben Caches insbesondere für den Einsatz in Parallelrechnern. Der Cache des verwendeten MIPS R10000-Prozessors der SGI Origin wird ausgiebig im „User's Manual“ [MIPS96] des Herstellers beschrieben. Eine unabhängige Darstellung findet sich im „Microprocessor Report“ [Gwen94].

5.1.1 Aufteilung: Daten-, Instruktions- und Unified Caches

Moderne Prozessoren haben meist mehrere Ebenen von Caches, genannt *Levels*. Der primäre Cache, oft als *L1-Cache* bezeichnet, ist üblicherweise in Daten- und Instruktionscaches getrennt. Der sekundäre Cache, *L2-Cache*, ist häufig *unified*, d. h. er wird gleichermaßen zum Speichern von Daten wie von Instruktionen verwendet.

Alle Experimente und Simulationsrechnungen, die in diesem Kapitel über Caches durchgeführt werden, beziehen sich auf die Datencaches eines Prozessors. Auf den Testrechnern wurden alle Messungen an dem sekundären Cache vorgenommen. Es wurden weder Instruktionscaches, noch die Einflüsse von Instruktionen auf die Daten in Unified-Caches betrachtet. Da die untersuchten Anwendungen nur aus sehr kleinen Codes, aber großen Datenmengen bestanden, ist diese Vereinfachung gerechtfertigt.

5.1.2 Konzept: Zeitliche und räumliche Lokalität

Wie bereits erwähnt, läßt sich Caching auch dann effektiv einsetzen, wenn die Daten einer Berechnung nicht vollständig in den Cache passen.

Das Grundkonzept hinter der Konstruktion von Caches basiert darauf, Lokalitäten im Zugriffsmuster einer Anwendung auszunutzen. Unter Zugriffsmuster versteht man in diesem Kontext die Sequenz der adressierten Speicherstellen.

Dabei wird üblicherweise zwischen „zeitlicher“ und „räumlicher“ Lokalität unterschieden. Mit *Temporal Locality* bezeichnet man die Beobachtung, dass viele praktisch verwendete Programme während ihres Laufes auf eine kleine Anzahl von Adressen häufig zugreifen. Beispiele hierfür sind die Induktionsvariablen von Schleifen oder Stackindizes. *Spatial Locality* beschreibt die Tatsache, dass häufig Daten aus benachbarten Adressen nacheinander ausgelesen werden. Bei der Verarbeitung von Vektoren und Matrizen wird beispielsweise häufig kontinuierlich auf zusammenhängende Speicherbereiche zugegriffen.

Der Aufbau und die Funktionsweise eines Prozessorcaches sind dafür optimiert, die zeitliche und räumliche Lokalität im Zugriffsmuster auszunutzen, um den Programmablauf zu beschleunigen. Die nachfolgenden Abschnitte stellen diese Konstruktionsprinzipien dar.

5.1.3 Aufbau: Cachelines, Bänke und Assoziativität

Die kleinste adressierbare Einheit eines Caches ist die sogenannte *Cacheline* (auch *Cache Block* genannt). Eine Cacheline enthält typischerweise zwischen 32 und 128 Byte. Der primäre Datencache eines MIPS R10000 umfasst 32 kByte, organisiert in 1024 Cachelines von jeweils 32 Byte. Der sekundäre Cache ist in der Origin-2000 4 MByte und in der Origin-200 1 MByte groß. Die Größe der Cachelines beträgt in Origins 128 Byte.

Diese Cachelines sind ihrerseits zu sogenannten *Bänken* zusammengefasst. Die Anzahl dieser Bänke nennt man die *Assoziativität* des Caches. Die Assoziativität gibt somit die Anzahl von Cachelines an, auf die die Daten einer gegebenen Adresse abgebildet werden können. Gängige Assoziativitäten reichen von 1 (genannt *Direct-Mapped*) bis zu 8 (*Eight-Way Set-Associative*). Einen Sonderfall stellt der *Fully-Associative*-Cache dar, welcher jeder Cacheline eine eigene Bank zuordnet. Der sekundäre Cache eines R10000 ist zweifach set-assoziativ.

5.1.4 Zugriff: Tag, Index und Offset

Um bei einer anliegenden Adresse zu überprüfen, ob sich eine Kopie des zugehörigen Datenwortes im Cache befindet, wird die Adresse in Tag, Index und Offset zerlegt. Der *Offset* bezeichnet die Position des Datenwortes innerhalb einer Cacheline. Der *Index* legt die Nummer der Cacheline fest. Damit legen Offset und Index die Koordinaten eines Datenwortes innerhalb des Caches fest. Das *Tag* hingegen wird, zusammen mit jeder Cacheline, im Cache gespeichert und mit der anliegenden Adresse verglichen. Die hierfür erforderlichen Formeln werden in Abschnitt 5.2.4 dargestellt.

Die Bank wird nicht adressiert. Für eine anliegende Adresse werden die Tags aller Bänke mit dem Tag-Anteil der Adresse verglichen. Diejenige Bank, deren Tag übereinstimmt, enthält eine Kopie des Datenwortes. Ein zusätzliches *Valid Bit* an der Cacheline zeigt an, ob die Kopie gültig ist.

5.1.5 Verwaltung: LRU und die drei Klassen von Misses

Befindet sich das adressierte Datenwort im Cache, so nennt man dies einen *Cache-Hit*. Befindet es sich nicht im Cache, so handelt es sich um einen *Cache-Miss*. Im Falle eines Misses müssen die Daten derjenigen Cacheline, auf die das Datenwort abgebildet wird, aus der nächsten Cache-Hierarchie (ggf. dem Hauptspeicher) in den Cache geladen werden. Dieser Vorgang nennt sich *Cacheline Refill*. Der Index-Anteil der Adresse bezeichnet die Nummer der Cacheline, in die gespeichert wird.

Es gibt mehrere gängige Strategien zur Auswahl einer Bank in einem mehrfach assoziativen Cache, insbesondere *Least-Recently Used* (LRU) und *Random*. Der R10000 verwendet LRU[1]. Wurde der bisherige Inhalt der Cacheline gegenüber dem Hauptspeicher modifiziert, wurde also in die Cacheline geschrieben, so muss die Cacheline zuerst in den Hauptspeicher zurückgeschrieben werden, bevor die neue Cacheline geladen werden kann.

Misses werden nach ihren Ursachen in die drei Kategorien Compulsory-, Capacity- und Conflict-Misses unterschieden: *Compulsory*-Misses treten auf, wenn zum ersten Mal auf ein Datum innerhalb einer Cacheline zugegriffen wird. *Capacity*-Misses entstehen, wenn nicht alle während eines Programmlaufes adressierten Daten gleichzeitig in den Cache passen. *Conflict*-Misses werden verursacht, wenn mehr Cachelines mit derselben Nummer adressiert werden, als es der Assoziativität des Caches entspricht.

5.1.6 Adressierung: Physikalische und virtuelle Adressen

Alle aktuellen Mikroprozessoren enthalten Hardware-Unterstützung für virtuelle Adressierung, so dass im Multitasking-Betrieb aktueller Betriebssysteme jeder Prozess in einem privaten und geschützten virtuellen Adressraum ablaufen kann. Für Realisierungen von Virtualspeicher in gängigen Betriebssystemen sei auf das Standardwerk von Tanenbaum [Tane86], für die dazu notwendigen Abläufe in den Prozessoren auf Hennessy und Patterson [Henn96], [Patt98] verwiesen.

Abhängig davon, an welcher Stelle der Adressumsetzung die Adresse an den Cache gelegt wird, unterscheidet man *physikalisch* oder *virtuell indizierte* Caches. Der R10000 verwendet im L2-Cache physikalische Tags und Indizes.

[1] Dieser LRU-Algorithmus wird zusätzlich vom genauen Verlauf der sogenannten *spekulativen Ausführung* beeinflusst. Hierfür sei auf die Abschnitte 1.7 und 4.2 des „User's Manual“ [MIPS96] verwiesen.

5.2 Der Speicherhierarchie-Simulator (MHS)

Im vorangegangenen Unterkapitel wurden die prinzipiellen Funktionseinheiten eines Prozessorcaches skizziert. Die Dynamik ihrer Funktionsweise durch analytische Modelle zu beschreiben, erweist sich im Allgemeinen als schwierig.

Grundlagen der analytischen Modellierung von Prozessorcaches finden sich bereits in Kapitel 3.4 des Buches von Przybyski [Przy90]. Als aktuelle Arbeiten auf diesem Gebiet seien die Artikel von Fraguella et.al. [Frag98] oder Harper et.al. [Harp99] genannt.

Um auch für irreguläre Zugriffsmuster einen Zugang zu ermöglichen, wird in dieser Untersuchung als Modellierungszugang die Simulation gewählt. Dafür wurde der Speicherhierarchie-Simulator MHS (für *Memory Hierarchy Simulator*) entwickelt, ein allgemeiner Simulator für Caches von RISC-Prozessoren. Die Simulationssoftware ist umfassend konfigurierbar und kann zur Nachbildung von Caches beliebiger Größen, Cachelinelängen und Assoziativitäten eingesetzt werden.

5.2.1 RISC-Prozessoren als Load-Store-Architektur

Die Prozessoren der gängigen kommerziellen Parallelrechner, Bezugspunkt sei wiederum die bereits erwähnte Top500-Liste, sind durchgehend *RISC*-CPUs. Ein wesentliches Konzept bei der Entwicklung von RISC-Prozessoren war der Übergang zu einer reinen *Load-Store-Architektur*. Dabei wurden insbesondere die Adressierungsmodi stark vereinfacht und auf explizite Lade- und Speicher-Operationen zwischen Registersatz und Adressraum beschränkt. Die resultierende Vereinfachung der Lade- und Speicher-Hardware ermöglichte die Einführung weiterer beschleunigender Techniken, insbesondere des Pipelinings. Die historischen und technischen Hintergründe dieser Entwicklung findet man in den detaillierten Arbeiten von Hennessy und Patterson [Henn96], [Patt98].

5.2.2 Hits und Misses als grundlegende Ereignisklassen

Für die Modellierung des Verhaltens der Speicherhierarchie, insbesondere des Zusammenspiels von Anwendungsprogramm, Prozessor, Cache und Hauptspeicher, werden die *Load*- und *Store*-Instruktionen als Ausgangspunkt gewählt. Mit ihnen kopiert das Programm Datenworte aus dem Adressraum in Register und umgekehrt. Auf den in Registern vorliegenden Operanden werden Operationen ausgeführt, deren Ergebnisse wiederum in Registern abgespeichert werden. Daher nennt man Load-Store-Prozessoren auch *Register-Register-Maschinen*.

5.2.3 Überladung von Indizierungsfunktionen der Anwendung

Die Simulation des Cacheverhaltens soll mit den konkreten Zugriffsmustern des Anwendungsprogrammes erfolgen. Um dieses Zugriffsmuster zu erhalten, bieten sich mehrere Möglichkeiten an. In dieser Arbeit wird der Weg gewählt, die Sequenz der Adressen direkt von der laufenden Anwendung zu erhalten. Dazu wird das in C++

geschriebene Matrixmultiplikations-Programm mit einer Bibliothek des Simulators gelinkt. Dabei werden die Indizierungsoperatoren der Anwendung durch die Indizierungsfunktionen des Simulators überladen (vergleiche hierzu Kapitel 11, „Operator Overloading“, in der Sprachreferenz [Stro97] und Abschnitt 3.6 im Entwicklungsbericht [Stro94] von C++), die sowohl die indizierten Daten aus dem Adressraum zurückliefern, als auch die erzeugte Adresse in den Simulator einspeisen. Für jede eingespeiste Adresse bildet der Simulator die Abläufe in der Cachelogik nach.

5.2.4 Nachbildung der Cachelogik

Zur Nachbildung der Cachelogik wird die eingehende Adresse zunächst in Tag, Index und Offset zerlegt. Der Simulator verwendet dabei die virtuellen Adressen des Anwendungsprogramms (vergleiche Abschnitt 5.1.6). Sei t der Tag, i der Index und o der Offset zur Adresse a. Weiterhin sei n_c die Größe des Caches und n_l die Größe einer Cacheline in Byte und n_a die Assoziativität des Caches, so gilt:

$$\begin{aligned} t &= (a \cdot n_a) \text{ div } n_c \\ i &= (a \bmod (n_c \text{ div } n_a)) \text{ div } n_l \\ o &= a \bmod n_l \end{aligned}$$

Damit werden die Nummer der Cacheline, auf die die Adresse abgebildet wird, und die Position innerhalb dieser Cacheline bestimmt. Anschließend werden alle n_a Bänke daraufhin überprüft, ob ihr Tag mit dem soeben berechneten Tag t der Adresse übereinstimmt. Sollte dies bei einer Bank der Fall sein und das zusätzliche Valid-Bit anzeigen, dass die Kopie noch gültig ist, so handelt es sich um einen Cache-Hit.

Im Falle eines Cache-Misses befindet sich keine gültige Kopie im Cache. Über einen LRU-Mechanismus wird der älteste der n_a Einträge zu Cacheline i ermittelt und durch die neuen Informationen überschrieben. Das Tag wird eingetragen und die Daten werden aus dem Adressraum gelesen.

Zu jeder Cacheline jeder Bank hält der Simulator die Anzahlen der aufgetretenen Hits und Misses, so dass sich über statistische Analysen die aufgetretenen Zugriffsmuster den Cacheeffekten zuordnen lassen.

5.3 Bei der Modellierung ausgeklammerte Aspekte

Der Speicherhierarchie-Simulator MHS lässt sich zur Nachbildung vielfältiger Cachekonfigurationen verwenden. Dennoch wurden nicht alle Aspekte eines konkreten Prozessor-Caches nachgebildet.

5.3.1 Kein Timing, nur Logik

Der Simulator arbeitet auf rein logischer Ebene. Für jede der Adressen einer von der Anwendung produzierten Sequenz, unterscheidet der MHS, ob sie aufgrund des bisherigen Zugriffsmusters zu einem Hit oder Miss führt. Der Simulator enthält

jedoch keine Informationen über *Timing*, d.h. die mit einem Hit oder Miss verbundenen Ausführungszeiten, bis die adressierten Daten in den Prozessoren verarbeitet werden können. Diese Ausführungszeiten sind schwierig zu ermitteln und nicht aus den Datenblättern der Prozessoren abzulesen, da sie nicht konstant sind und stark von Kontext und Implementierungsdetails abhängen.

Moderne Mikroprozessoren arbeiten ihre Instruktionen nicht strikt hintereinander oder in der vom Compiler vorgegebenen Reihenfolge ab. Konstruktionseigenschaften wie die Unterstützung von *Pending Loads* oder *Out-of-Order-Execution* erlauben es, nicht auf die Ausführung einer Lade-Instruktion warten zu müssen, oder die Reihenfolge von Instruktionen zu verändern, solange dies die *sequentielle Konsistenz* nicht beeinträchtigt. Das „User's Manual" [MIPS96] des R10000 beschreibt die konkrete Umsetzung dieser Konzepte für den hier verwendeten Prozessor.

5.3.2 Keine Multilevel-Caches

Der MHS bildet nur eine Cache-Ebene nach. Moderne Prozessoren unterstützen jedoch meist mehrere Ebenen von Caches. Der in der SGI Origin verwendete R10000-Prozessor hat zwei Ebenen. Die primäre Ebene besteht aus separaten Daten- und Instruktions-Caches. Der sekundäre Cache ist *unified*, d.h. er enthält sowohl Daten als auch Instruktionen. Alle in diesem Kapitel beschriebenen Experimente und Simulationsrechnungen wurdem mit dem L2-Cache, bzw. seiner Konfiguration, durchgeführt.

5.3.3 Keine Multitasking-Effekte

Im Allgemeinen werden aktuelle Parallelrechner unter einem Multitasking-Betriebssystem betrieben. Damit teilen sich potentiell mehrere Prozesse in einem Zeitscheibenverfahren jeden der vorhandenen Prozessoren. Solange nicht die Anwendungsdaten aller dieser Prozesse gleichzeitig und kollisionsfrei (vergleiche Capacity- und Confict-Misses in Abschnitt 5.1.5) in den Cache dieses gemeinsam genutzten Prozessors passen, wird bei jedem Taskwechsel ein Anteil der Cachelines mit neuen Inhalten besetzt.

Aus dem Multitasking-Betrieb folgen somit Seiteneffekte auf das Cacheverhalten eines Prozesses. Das Verhalten des Speicherhierarchie-Simulators wird jedoch ausschließlich durch das Zugriffsmuster einer einzelnen Anwendung bestimmt. Seiteneffekte durch die Cachezugriffe anderer Prozesse werden nicht simuliert. Diese Effekte in die Simulation aufzunehmen, würde es erforderlich machen, das Zeitscheibenverfahren des verwendeten Betriebssystems (genauer des sogenannten *Schedulers*) nachzubilden. Eine solche Erweiterung liegt jedoch weit außerhalb des Rahmens dieser Arbeit.

5.3.4 Kein cache-kohärenter Multiprozessor

Der Speicherhierarchie-Simulator bildet lediglich das Cacheverhalten eines einzelnen Prozessors nach. Damit ist er geeignet, die einzelnen Prozessoren eines Multirech-

nersystems mit verteiltem Speicher zu simulieren. Um ein Multiprozessorsystem wie einen ccNUMA simulieren zu können, muss auch der Einfluss einer CPU auf den Cache einer anderen CPU in Form von *Coherence*-Misses (zuzüglich zu den in Abschnitt 5.1.5 genannten Klassen, vgl. S. 7) nachgebildet werden. Der in dieser Arbeit verwendete MHS enthält diese Effekte nicht, und obwohl er sich prinzipiell um sie erweitern liesse, würden ähnliche Schwierigkeiten wie beim Multitasking auftreten (vgl. 5.3.3).

5.3.5 Kein Maschinencode, sondern Applikationslogik

Der Cachesimulator benötigt als Eingabe das Zugriffsmuster, d. h. die Sequenz von Adressen, auf welchen der Prozessor Lade- und Speicher-Instruktionen ausführt. Dieses Zugriffsmuster, auch *Address Trace* genannt, ließe sich beispielsweise durch interpretative Ausführung des Maschinenprogramms (*Executable*) erreichen. Dieser Zugang würde es ermöglichen, den Einfluss des Compilers auf die entstehenden Zugriffsmuster zu ermitteln. Allerdings ist das Verfahren aufwendig und muss auf die Maschinensprache jedes zu simulierenden Prozessors angepasst werden. Daher wurde in dieser Arbeit die Methode gewählt, statt dessen das Anwendungsprogramm durch überladene Indizierungsfunktionen den Simulator steuern zu lassen. Dieses Verfahren ist sehr viel einfacher und zusätzlich portabel.

5.4 Test der Cachesimulation

Um die Genauigkeit zu ermitteln, mit der die Speicherhierarchie-Simulation das Hit- und Miss-Verhalten vorhersagt, wurde das folgende Experiment vorgenommen. Eine Beispielanwendung, die Matrixmultiplikation wurde auf dem Testrechner ausgeführt. Dabei wurde die Anzahl der Cache-Misses durch Instrumentierung der Hardware gemessen. Dieses Messergebnis wurde mit Simulationsläufen verglichen, wobei die Konfiguration des simulierten Caches genau der Konfiguration des Prozessors im Testrechner entsprach.

5.4.1 Matrixmultiplikation

Um einen Vergleich mit Abschnitt 4.3 zu ermöglichen, wurde für den Test des Speicherhierarchie-Simulators erneut die Matrixmultiplikation gewählt. Aus technischen Gründen ist der Code dieses Multiplikationsprogrammes nicht mit demjenigen identisch, welches in Abschnitt 4.3 verwendet wurde. Dennoch enthalten beide Implementierungen denselben Algorithmus zur Matrixmultiplikation, d.h. insbesondere dasselbe Speicherlayout der Datentypen und dieselbe Schleifenkonstruktion.

5.4.2 Experiment: Instrumentierung der Hardware

Einige moderne Mikroprozessoren sind instrumentiert, um mittels spezieller Hardware-Zähler zur Laufzeit statistische Größen aufnehmen zu können. Diese

Zähler werden meist *Profiling-* oder *Performance-Counter* genannt. Umfang und Funktionsweise dieser Zähler ist sehr unterschiedlich.

Die Performance-Counter des MIPS R10000-Prozessors ermöglichen vielfältige Messungen, insbesondere der Datenbewegungen zwischen den verschiedenen Ebenen der Speicherhierarchie. Für diese Untersuchungen wurde über die sogenannten *Perfex*-Aufrufe des Betriebssystems die Anzahl der Misses im primären Datencache ausgelesen. Die zurückgelieferten Perfex-Daten sind bereits *virtualisiert*, d.h. sie sind bereits korrekt von den hardwarebezogenen Rohdaten auf den die Aufrufe auslösenden Anwendungsprozess umgerechnet. Allerdings enthält diese Virtualisierung keine Kompensation der in Abschnitt 5.3.3 dargestellten Multitasking-Effekte. Detailierte Hinweise zu Funktionsweise und Programmierung dieser Performance-Counter findet man in Kapitel 10.20 des R10000-Handbuches [MIPS96] und auf der man-Page zu „perfex“.

5.4.3 Simulation: Vorhersage durch den MHS

Für die Simulationsläufe wurde die Software-Konfiguration des MHS-Simulators auf die Hardware-Konfiguration des primären Datencaches der Origin-200 eingestellt. Die Konfiguration umfasste 1024 kByte, zweifach set-assoziativ organisiert in 8192 Cachelines zu jeweils 128 Byte. Dabei wurde der Simulator durch die in Abschnitt 5.4.2 beschriebene Anwendung mit Zugriffsmustern versorgt.

5.4.4 Ausführungszeit als Voruntersuchung

Um die Simulations- mit den Messergebnissen systematisch zu vergleichen, wurden Matrixmultiplikationen mit Matrizen verschiedener Größe durchgeführt. Dabei wurde die benötigte Ausführungszeit über POSIX[2]-Aufrufe der Echtzeituhr und die Anzahl der Cache-Misses über die Perfex-Schnittstelle gemessen. Zusätzlich wurde die Vorhersage des Simulators für die jeweilige Matrixgröße ermittelt.

Als Voruntersuchung wurde lediglich die Ausführungzeit der Matrixmultiplikation gemessen. Um alle während der Skalierungsuntersuchung in Abschnitt 4.3 vorkommenden Matrixgrößen einzuschließen, wurde ein breites Intervall von Matrixgrößen gewählt.

Abbildung 5.1 zeigt die gemessenen Ausführungszeiten. Es zeigt sich deutlich, dass der bekannte Aufwand der Matrixmultiplikation von $O(n^3)$ von vielfachen Überhöhungen überlagert ist. Diese Spitzen überragen die Grundfunktion um mehrere Hundert Prozent. Die folgenden Untersuchungen zeigen, dass es sich bei diesen Überhöhungen nicht um statistische Ausreisser, sondern um reproduzierbare Cache-Effekte handelt.

[2]Die hier verwendeten POSIX-Aufrufe sind im Buch von Gallmeister [Gall95] „POSIX.4 - Programming for the Real World“ dargestellt. (Das POSIX.4-Interface wurde inzwischen in POSIX 1003.1b und POSIX.1d umbenannt.)

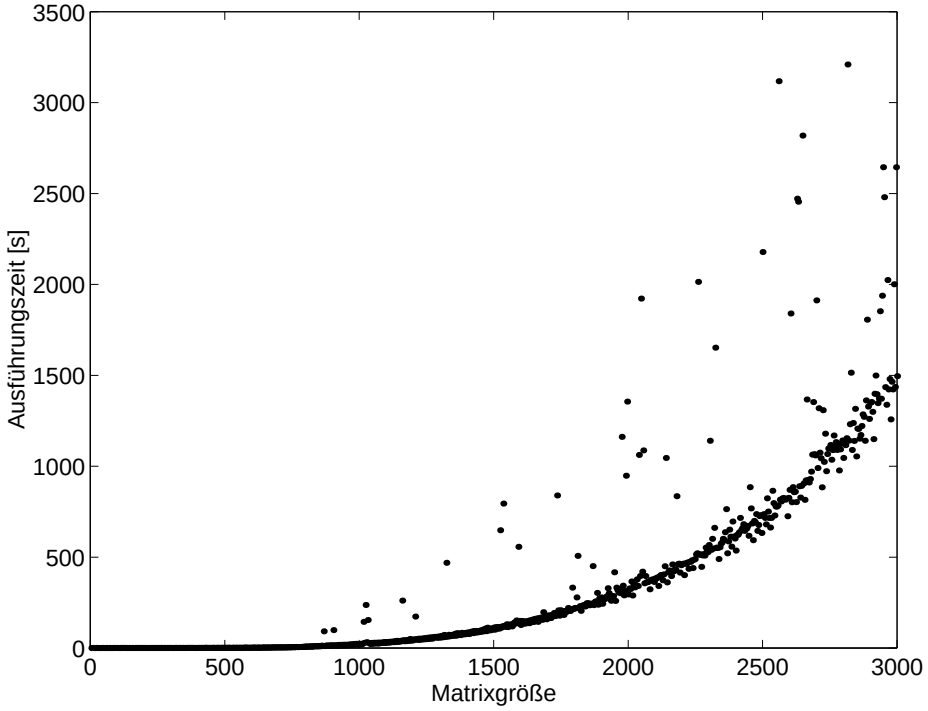

Abbildung 5.1: **Ausführungszeit der Matrixmultiplikation** – Es sind deutliche Überhöhungen über der zeitlichen Komplexität der Matrixmultiplikation von $O(n^3)$ zu erkennen. Diese Überhöhungen können mehrere Hundert Prozent betragen. (Eine Matrixgröße von n bezeichnet Matrizen aus $n \times n$ Fließkommazahlen doppelter Genauigkeit. Die Ausführungszeiten wurden über die Echtzeituhr des Systems gemessen.)

5.4.5 Simulierte und gemessene Cachemisses

Die Simulationsläufe wurden mit experimentell ermittelten Werten der Performance-Counter der Hardware verglichen. Abbildung 5.2 (S. 70) stellt die Simulations- und die Messergebnisse einander gegenüber. Da der Zeitaufwand eines Simulationslaufes um zwei bis drei Größenordnungen höher ist, als der einer Messung, wurde nur eine Stichprobe von $n \times n$-Matrizen untersucht. Im Intervall von $n = 256$ bis $n = 512$ wurden bei einer Schrittweite von 8 jeweils die Simulations- und Messergebnisse ermittelt.

In der linken Teilgraphik von Abbildung 5.2 zeigt sich eine augenscheinlich gute Übereinstimmung zwischen Simulation und Messung. Die durch Punkte dargestellten Messwerte liegen sehr dicht an den Mittelpunkten der Kreise, die die Simulationsergebnisse symbolisieren. Die rechte Teilgraphik von Abbildung 5.2 zeigt, dass der Vorhersagefehler für die gewählte Sichtprobe durchgehend weniger als 4% beträgt.

Es zeigt sich, dass der Speicherhierarchie-Simulator für das aus der gewählten Anwendung vorgegebene Zugriffsmuster eine gute Vorhersage der Ergebnisse liefert. Die

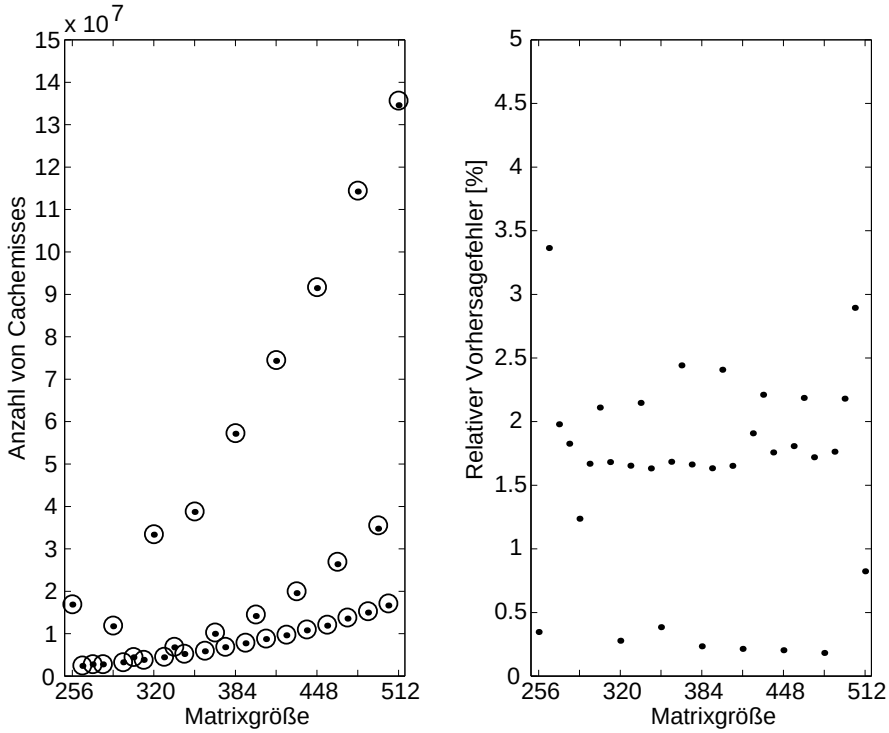

Abbildung 5.2: **Cachemisses bei der Matrixmultiplikation** – In der linken Teilgraphik kennzeichnen die Kreise die Ergebnisse von MH-Simulationen bei der Konfiguration des R10000-Caches. Die Punkte bezeichnen die Hardware-Messergebnisse. (Die durch Punkte symbolisierten Messwerte stammen aus den Performance-Countern des R10000-Prozessors.) Die rechte Teilgraphik stellt den relativen Vorhersagefehler dar. Für die untersuchte Stichprobe lagen die Vorhersagefehler unter 4%.

gemessenen Anzahlen von Cache-Misses liegen ohne Ausnahme über den vorhergesagten. Dies erklärt sich aus den in Abschnitt 5.3.3 beschriebenen Multitasking-Effekten, die durch die Virtualisierung der Performance-Counter nicht kompensiert werden können.

5.4.6 Bestimmen der Kostenkoeffizienten

Im vorhergehenden Abschnitt wurden Anzahlen simulierter und gemessener Cache-Misses miteinander verglichen und eine gute Übereinstimmung festgestellt. Wie aber bereits in 5.3.1 dargestellt wurde, berechnet die Cache-Simulation keine Timing-Informationen. Es werden lediglich die Vorgänge in der Cachelogik nachgebildet, nicht jedoch deren Ausführungszeiten.

Um zusätzlich zu motivieren, dass sich die in Abbildung 5.1 (S. 69) dargestellten Überhöhungen durch eine Modellierung des Cacheverhaltens erklären lassen, wurden zeitliche Kostenkoeffizienten zu den Simulationsdaten ermittelt. Dazu wurden aus den Daten der Messungen der Ausführungszeiten nach Abbildung 5.1 und den

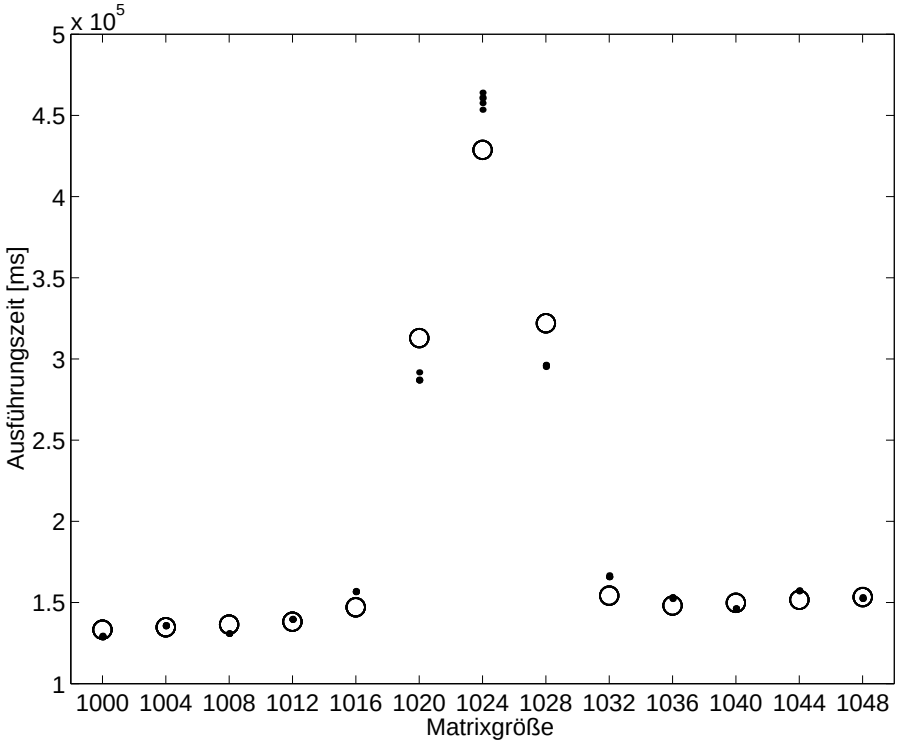

Abbildung 5.3: **Simulation der Ausführungszeit** – In einem *Least Squares*-Fit wurden aus den Ergebnissen des Speicherhierarchie-Simulators die zeitlichen Kosten für Cache-Hits und -Misses bestimmt. Mittels dieser Kostenkoeffizienten wurde aus der Simulation die Ausführungszeit vorhergesagt. Der relative Fehler dieser Vorhersage bleibt unter 7%. (Die Punkte stellen Messwerte, die Kreise Simulationsergebnisse dar.)

Cachesimulationen nach Abbildung 5.2 folgende Tripel gebildet:

$$(n_h, n_m, t)$$

Dabei bezeichnet n_h die Anzahl der Hits und n_m die Anzahl der Misses. Die Summe $n_h + n_m$ ergibt die Anzahl der insgesamt ausgeführten Load-Operationen. Da Store-Operationen asynchron ausgeführt werden können, wurden sie zwar bei der Cachesimulation berücksichtigt, bei der Aufstellung der Kostenfunktion jedoch vernachlässigt. Aus den Datentripeln und der einfachen Funktion

$$t = c_h \cdot n_h + c_m \cdot n_m$$

wurden die Kostenkoeffizienten c_h für einen Hit und c_m für einen Miss in *Mathematica* mit einem Least-Squares-Fit berechnet. Mit dieser Methode ergaben sich Ausführungszeiten von

$$c_h = 58 \text{ ns}$$

für einen Hit und

$$c_m = 344 \text{ ns}$$

für einen Miss. Verrechnet man diese Koeffizienten mit den Hit- und Miss-Angaben aus der Simulationsrechnung so ergeben sich die in Abbildung 5.3 dargestellten simulierten Ausführungszeiten.

Für diese Gegenüberstellung wurde speziell eine der Überhöhungen gewählt, um die anfängliche Hypothese zu bestätigen, dass diese Überhöhungen keine statistischen Ausreisser darstellen, sondern einer reproduzierbaren Systematik folgen. Die Punkte stellen erneut die Messdaten, die Kreise die Simulationsergebnisse dar. Qualitativ gibt die Simulation sowohl die Lage als auch die Größenordnung der Überhöhung wieder. Quantitativ bleibt der relative Vorhersagefehler auch am Maximum der Überhöhung unter 7%.

Dies ist eine sehr gute Übereinstimmung zwischen Theorie und Experiment, insbesondere angesichts der extremen Einfachheit der gewählten Methode zur Ableitung der Ausführungszeiten aus den Simulationsergebnissen.

5.5 Zwischenbewertung

An die verwendeten Datentripel und die Fit-Funktion wurde nicht der Anspruch gestellt, die genaue zeitliche Dynamik eines modernen Mikroprozessors wiedergeben zu können. Statt dessen wurde für Datenstrukturen und Verarbeitungsfunktion der einfachste Ansatz gewählt, der mit den prinzipiellen Abläufen im zu modellierenden System verträglich war.

Um einen Maßstab für die Beurteilung der Güte der gewählten Modellierung zu erhalten, sollen die berechneten Koeffizienten c_h und c_m mit den Ergebnissen einer Arbeitsgruppe aus Princeton verglichen werden.

Jiang und Singh haben in ihrer Veröffentlichung „A Methodology and an Evaluation of the SGI Origin-2000“ [Jian98] die Ergebnisse detaillierter Benchmark-Messungen vorgestellt. Dafür wurden die Cachemisses in verschiedene Kategorien eingeteilt, wodurch sich systematisch die kleinsten und größten Ausführungszeiten ermitteln ließen. Jiang und Singh fanden als Kosten für die Hits und Misses im sekundären Cache

$$c'_h = 56.9 \text{ ns} \quad c'_m = 329 \text{ ns} \quad c''_m = 472 \text{ ns}$$

Diese Messdaten sind mit den Konstanten

$$c_h = 58 \text{ ns} \quad c_m = 344 \text{ ns}$$

die sich bei der Speicherhierarchie-Simulation für den Zusammenhang von Hits, Misses und Ausführungszeit ergaben, perfekt verträglich. Insbesondere gilt

$$c'_m < c_m < c''_m$$

Die Übereinstimmung zwischen den im Rahmen dieser Arbeit durch Simulation gewonnenen Daten und den von Jiang und Singh messtechnisch ermittelten Werten darf als sehr gut bezeichnet werden.

Kapitel 6

Fazit: Was haben wir gelernt?

6.1 Bewertung der Effektivität des LogP-Modells

Im Rahmen dieser Arbeit erwies es sich als notwendig, das LogP-Modell für Parallelrechner um Gegebenheiten realer Hardware zu erweitern. Dies ist kein ungewöhnliches Vorgehen. Schon in frühen Folgeuntersuchungen wurde das LogP-Modell erweitert, um das Verhalten konkreter Rechner beschreiben zu können. Das LogGP-, das LoGPC- und das LogPQ-Modell sind nur einige Beispiele für solche Weiterentwicklungen des ursprünglichen LogP-Modells. Hinzu kommen viele Untersuchungen, in denen dem modifizierten Modell kein neuer Name gegeben wurde.

„Wie gut funktioniert das LogP-Modell?“ ist eine Frage, die sich angesichts der vielfältigen Modifikationen anbietet. Die Antwort ist abhängig von der Perspektive.

Aus der Sicht des Algorithmen-Entwurfes für den praktischen Einsatz ist das LogP-Modell ein großer Fortschritt gegenüber dem PRAM. Durch das LogP-Modell wird es beispielsweise möglich, Berechnungs- und Kommunikationsphasen algorithmisch voneinander zu separieren und im implementierten Programm zeitlich überlagert auszuführen. Dadurch wird die in der Praxis unvermeidbare Kommunikations-Latenz verdeckt und der Programmablauf beschleunigt. Dieses Grundkonzept forciert den Übergang zum Entwurf asynchroner paralleler Algorithmen und damit den Zugang zu einer wesentlichen Leistungssteigerung für die resultierende Implementation.

Für einen Programmierer, der einen bestehenden Algorithmus optimal an einen bestehenden Parallelrechner anpassen muss, um eine minimale Ausführungszeit zu erreichen, lässt das LogP-Modell jedoch zu viele Fragen offen. Es bietet beispielsweise keine Möglichkeit, verschiedene Puffer-Strategien für die Kommunikation miteinander zu vergleichen. Zur Einordnung sei bemerkt, dass MPI über acht verschiedene Puffer-Strategien zum Senden im Punkt-zu-Punkt-Betrieb verfügt. Die Wahl der richtigen Strategie für ein gegebenes Kommunikationsmuster stellt einen wesentlichen Anteil der Optimierung eines MPI-Programmes dar. Desweiteren gibt es keine Möglichkeiten, die Konsequenzen einer Prozessorzuordnung zu bewerten. Daher muss das LogP-Modell in der Optimierungssphase durch ein hardwarenahes Modell ergänzt werden.

Auch die Betriebsart des Parallelrechners hat großen Einfluss auf die Anwendbarkeit des LogP-Modells. Im Message-Passing-Betrieb wurden lediglich Abweichungen von bis zu 20% vom vorhergesagten Verhalten nachgewiesen. Im Shared-Memory-Betrieb waren es dagegen bis zu 80%. Um diese Zahlen einschätzen zu können, muss beachtet werden, dass der Testrechner aus Dresden mit 48 Prozessoren deutlich unter der maximalen Prozessoranzahl liegt. SGI bietet die Origin-2000 mit bis zu 512 Prozessoren an. Die eigentliche Hardware der Cache-Kohärenz ermöglicht

bis zu 1024 Prozessoren an einem virtuellen Adressraum. Da die effiziente Programmierung eines Parallelrechners mit wachsender Prozessoranzahl immer schwieriger wird, ist das LogP-Modell zur Modellierung großer Parallelrechner nur sehr begrenzt geeignet.

6.2 Bewertung der Effektivität der Speicherhierarchie-Simulation

In einem Praxistest des LogP-Modells mit einer Anwendung zur Matrixmultiplikation zeigte sich, dass die vom LogP behandelten Kommunikationseffekte von Vorgängen in den Rechenknoten verdeckt wurden. Effekte in den Prozessorcaches erzeugten Schwankungen in der Ausführungszeit der Teilberechnungen und verursachten einen unerwarteten superlinearen Speedup.

Um einen Zugang zu diesen dominanten Beiträgen zur Ausführungszeit zu erhalten, wurde im Rahmen dieser Arbeit der Speicherhierarchie-Simulator MHS entwickelt. Dieser Simulator bildet für das konkrete Zugriffsmuster einer Anwendung die Vorgänge in der Kontroll-Logik des Prozessorcaches nach und ermöglicht für den betrachteten Knoten eine Vorhersage der Ausführungszeit. Bei der gewählten Stichprobe lag der relative Vorhersagefehler unter 7%. Zusätzlich lassen sich auf diese Weise auch die Auswirkungen von Änderungen im Zugriffsmuster systematisch analysieren. Neben den Einflüssen der Prozessoranzahl lassen sich somit auch Änderungen in den Daten- oder Kontrollstrukturen untersuchen. Darüber hinaus ist es möglich, Cachekonfigurationen zu evaluieren, die nicht als Hardware zur Verfügung stehen.

Für sich betrachtet ist der MHS kein Parallelrechner-Simulator, da er lediglich die Effekte im Cache eines einzelnen Rechenknotens nachbildet. In einem Multirechnersystem im Message-Passing-Betrieb lässt sich der MHS zusammen mit dem LogP-Modell einsetzen um sowohl die Kommunikations- als auch die Berechnungseffekte einer Modellierung zugänglich zu machen. In einem Multiprozessor-System im Shared-Memory-Betrieb zeigte das LogP-Modell jedoch nur geringe Anwendbarkeit und auch mit dem Speicherhierarchie-Simulator ließen sich ohne Erweiterung seiner Funktionalität die Auswirkungen einer cache-basierten Kommunikation nicht wiedergeben.

6.3 Bemerkungen zur Effektivität der Modellierung von Parallelrechnern

Parallelrechner sind sehr komplizierte technische Systeme. Sie definieren den *State of the Art* der Computertechnologie. Im Gegensatz zu den meisten anderen technischen Systemen werden sie nicht auf Beherrschbarkeit oder Alltagstauglichkeit, sondern auf maximale Leistungsfähigkeit optimiert. Für diese permanente Optimierung ist eine permanente Weiterentwicklung erforderlich. Aus diesen Eigenschaften des zu modellierenden Systems ergeben sich gewisse Randbedingungen für seine Modellierung.

Das Modell muss „schlank“ formuliert sein und darf nur die wesentlichen Konstruktionsmerkmale enthalten. Ansonsten wird die hohe technische Änderungsrate eines Systems, dessen Weiterentwicklung von großen Marktkräften getrieben wird, dieses permanent aus dem Gültigkeitsbereich des Modells entfernen. An Parallelrechnern werden häufig sogar innerhalb eines Rechner-Typs in kurzen Zeitabständen Änderungen[1] vorgenommen, welche Einfluss auf seine Leistungsfähigkeit und damit auf sein dynamisches Verhalten ausüben.

Das Modell muss sich kontinuierlich an die technische Weiterentwicklung der Parallelrechner anpassen. Dieser Vorgang kann unterschiedlich große Änderungen am Modellierungsansatz erfordern. Beispielsweise verursacht der Übergang von Peripheriebus- zu Speicherbus-gekoppelten Verbindungsnetzen primär nur bessere Übertragungsleistungen. Letztlich ist dieser Übergang jedoch der Grundstein zum Aufbau eines virtuellen Shared-Memory, dessen Leistungs-Charakteristik sich von der eines Message-Passing-Systems grundlegend unterscheidet.

Schließlich muss ein erfolgreiches Modell für ein technisches System, speziell für einen Parallelrechner, einen interdisziplinären Ansatz aus Wissenschaft und Technik verfolgen. Die Trennung zwischen wesentlichen Konstruktionseigenschaften und unwesentlichen Realisierungsdetails ist nicht trivial zu vollziehen. Da sowohl wissenschaftliche Erkenntnisse als auch technische Fertigkeiten erforderlich waren, um einen Parallelrechner und seine Anwendungen für das obere Ende des Leistungsspektrum zu konstruieren, sind auch beide Disziplinen notwendig, um sein dynamisches Verhalten durch Modellierung verstehen und optimieren zu können.

6.4 Parallelrechnermodellierung im Lichte der nichtlinearen Modellierungsaufgabe

Diese Untersuchung ist auf der einen Seite im Bereich der angewandten und technischen Informatik angesiedelt. Dort liegt auch der Schwerpunkt der methodischen Details. Auf der anderen Seite geht es um die Modellierung eines konkreten nichtlinearen dynamischen Systems. Die Hauptschwierigkeit der Modellierung nichtlinearer dynamischer Systeme besteht darin, dass Modifikationen bzw. Einflüsse, die bei linearer Betrachtung als klein angesehen werden, hier zu großen und auch zu qualitativ bedeutsamen Effekten führen können.

Der Parallelrechner kann als ein nichtlineares System mit diskretem Zustandsraum und diskreter Zeit aufgefasst werden. Die einzelnen Verarbeitungsschritte entsprechen zustandsgesteuerten Abbildungen des Zustandsraumes. Dies ist eine Standardsituation der Nichtlinearen Dynamik. Um die Rechenzeit zur Lösung einer bestimmten Aufgabe vorherzusagen, kann man im Extremfall die Rechnung tatsächlich durchführen und dadurch Vorhersagen für spätere Durchführungen gewinnen. Dieses Verfahren ist aber relativ aufwendig. Daher versucht man vereinfachte Modellklassen aufzustellen. Eine derartige Modellklasse soll Maschinen und Aufgaben darstellen, die im Hinblick auf solche Details äquivalent sind, die als unerheblich

[1] An dieser Stelle sei auf die „Konfiguration der Testmaschinen“ im Anhang B hingewiesen: Viele der zentralen Komponenten geben im Protokoll des Hardware-Inventars sogar ihre Revisionsnummer an.

erachtet werden. Die notwendigen Einschätzungen und Entscheidungen setzen viel Erfahrungswissen über die allgemeinen Verhaltensweisen nichtlinearer Systeme und die spezielle Arbeitsweise des zu modellierenden Systems voraus.

Die hiermit vorgelegte Untersuchung zur Modellierung von Parallelrechnern zeigt deutlich, wie die konkrete, oft eingeschränkte Realisierung allgemeiner Konstruktionsprinzipien und weitere technische Details das Systemverhalten wesentlich beeinflussen und damit eine vereinfachende Modellierung erschweren.

Ein zweites, wesentliches Ergebnis besteht darin, dass eine Grenze des Modellierungsvorganges deutlich wird, die üblicherweise nicht beachtet und auch nicht diskutiert wird: der Zeitbedarf für die Modellierung in Relation zur technischen Entwicklung des Objektbereiches. Wird es notwendig, viele technische Details in das Modell zu integrieren, so wächst der Zeitaufwand, der erforderlich ist, um das Modell zu formulieren, zu testen und zu validieren. Bei einer raschen technischen Entwicklung des Objektbereiches, z. B. durch neue Herstellungstechniken oder neue Konstruktionsprinzipien, kann leicht die Situation eintreten, dass beim Abschluss der Modellierung der Objektbereich in seiner Ausgangsform gar nicht mehr existiert oder zumindest nicht mehr relevant ist. Daher stellt sich eine neue Forderung: die Entwicklung von Modellierungsverfahren, die eine rasche Anpassung an die sich ändernden Situtionen im Objektbereich ermöglichen.

Kapitel 7

Zusammenfassung

In der vorliegenden Arbeit wurden Methoden zur Modellierung von Parallelrechnern vorgestellt. Dazu wurden sowohl Erweiterungen an bestehenden Rechnermodellen vorgenommen als auch ein eigenständiges neues Modell entwickelt.

Um den geeignetesten Ausgangspunkt weiterer Untersuchungen zu finden, wurden die gängigsten parallelen Rechnermodelle der Informatik analysiert. Als Gütekriterium wurde die Eignung des Modells gewählt, die Effizienz eines als Programm implementierten parallelen Algorithmus auf einem real existierenden Parallelrechner vorhersagen zu können.

Ausgangspunkt dieser Arbeit war das LogP-Modell von Culler, Patterson et. al. Dieses Modell wurde mit dem Ziel formuliert, die Ausführungszeiten eines konkreten Programmes auf einem konkreten Parallelrechner in Abhängigkeit von mehreren Parametern, darunter der Prozessoranzahl, quantitativ vorhersagen zu können. Daher schien es in Hinblick auf die Ziele dieser Arbeit besonders vielversprechend zu sein.

Die Definition der Leistungsparameter von LogP basiert auf einer Reihe von Grundannahmen. Einige dieser Grundannahmen mussten durch einen Vergleich mit den Konstruktionsprinzipien aktueller Parallelrechner von vornherein als unplausibel eingestuft und nach experimentellen Vergleichsmessungen an einem modernen Parallelrechner verworfen werden.

Nachdem das LogP-Modell durch Einbeziehung einiger Hardware-Gegebenheiten korrigiert worden war, konnte die Vorhersagegenauigkeit des Modells in einer Skalierungsuntersuchung überprüft werden. Dazu wurde die Abhängigkeit der Ausführungszeit von der Prozessoranzahl anhand einer Beispielanwendung untersucht. Das vorgenommene Experiment identifizierte für die gewählte Beispielanwendung einen Effekt als dominant, der im LogP-Modell nicht vorkommt. Die Auswertung des Experiments ergab, dass der hierarchische Aufbau des Speichers der einzelnen Rechenknoten – in Abhänigigkeit von den im Programmablauf auftretenden Zugriffsmustern – einen Einfluss von mehreren Größenordnungen auf die Ausführungszeit ausüben kann. Der innere Aufbau der Rechenknoten ist jedoch nicht Teil der LogP-Modellierung, so dass im Rahmen des LogP-Modells die aufgetretenen Effekte nicht modelliert und vorhergesagt werden können.

Um die Effekte der Cachehierarchie auf die Ausführungszeit vorhersagen zu können, wurde im Rahmen dieser Arbeit ein Speicherhierarchie-Simulator entwickelt. Dieser diskrete, ereignisbasierte Simulator verwendet als Eingabe das Zugriffsmuster eines Anwendungsprogrammes und bildet die Vorgänge der Kontroll-Logik eines Prozessorcaches nach. Durch diese funktionell sehr allgemeine, quantitativ jedoch exakt konfigurierbare Simulation sollten sich sehr akurate Vorsagen machen lassen.

Die Vorhersagen des Speicherhierarchie-Simulators wurden in verschiedenen Messungen mit dem konkreten Verhalten realer Hardware und den rein messtechnisch erlangten Ergebnissen einer Arbeitsgruppe aus Princeton verglichen. Dadurch wurde der sehr geringe Vorhersagefehler mehrfach bestätigt.

Um diese Arbeit erfolgreich durchführen zu können, mussten vielfältige Methoden aus verschiedenen wissenschaftlichen und technischen Bereichen eingesetzt werden. Es wurden analytische Modelle und asymptotische Aufwandsmaße der Informatik mit messtechnischen Verfahren und der Auslese hardwarespezifischer Performance-Register kombiniert. Jeweils mehrere Verfahren zur Parallelisierung mittels Shared-Memory und Message-Passing wurden verwendet und standardisierte portable mit meist leistungsfähigeren systemspezifischen Verfahren verglichen. Um eine korrekte Messung erhalten zu können, mussten die Experimente unter Beachtung der Konstruktionsdetails der verwendeten Silicon Graphics Origin sowie ihres Betriebssystems konzipiert werden. Insbesondere waren Aufbau und Funktionsweise von virtuellem Speichermanagement und parallelem Scheduling zu beachten. Im Wechselspiel wurden theoretische Vorhersagen und experimentelle Messungen gegenüber gestellt und miteinander verglichen und beides anhand von unabhängigen Veröffentlichungen von Ergebnissen, die durch gänzlich andere Methoden anderer Arbeitsgruppen entstanden, überprüft.

Die an mehreren Stellen in dieser Arbeit beschriebene Diskrepanz zwischen akademischem Algorithmenentwurf und konkreter Hardware- und Software-Entwicklung basiert nach Meinung des Autors dieser Arbeit im Wesentlichen auf zu geringer Kommunikation zwischen den beteiligten Parteien. Ein Parallelrechner ist ein hochentwickeltes und komplexes technisches System. Das Verständnis und die Optimierung seines dynamischen Verhaltens erfordert die Einbeziehung vieler sehr verschiedener Disziplinen aus Wissenschaft und Technik. Der im Rahmen dieser Arbeit verfolgte interdisziplinäre Ansatz soll helfen, eine Brücke zwischen diesen verschiedenen Aspekten zu schlagen und damit einen Beitrag zur Modellierung von Parallelrechnern zu leisten.

Anhang A

Die für den Fox-Algorithmus verwendeten Prozessorzuteilungen

Dieser Anhang enthält Zusatzinformationen zu der Skalierungsuntersuchung in Kapitel 4. Dort wurde anhand der vorgenommenen Prozessorzuteilungen zwischen „guten“ und „schlechten“ Zuteilungen unterschieden. Da die Zuteilungen durch ein Monte-Carlo-Verfahren erzeugt wurden, werden sie zum Zwecke der Reproduzierbarkeit der durchgeführten Untersuchungen hier aufgeführt.

Um Übertragungsfehler auszuschließen, wurden die mittels *Mathematica* [Wolf96] generierten Zuteilungen direkt in den Text übernommen. Dies erklärt die verwendete Syntax. Die aufgeführten Zuteilungen bestehen aus Listen von Tupeln der Form `{n,p}`, wobei `n` den MPI-Rang eines Prozesses und `p` seine Prozessornummer enthält.

A.1 Die „guten“ Konfigurationen

4 Prozessoren

```
{{0,2},{1,19},{2,3},{3,18}}
```

Entfernungssumme: 4

9 Prozessoren

```
{{0,5},{1,2},{2,23},{3,37},{4,3},{5,7},{6,4},{7,10},{8,6}}
```

Entfernungssumme: 54

16 Prozessoren

```
{{0,19},{1,32},{2,24},{3,25},{4,2},{5,41},{6,8},{7,10},
{8,15},{9,44},{10,12},{11,11},{12,14},{13,42},{14,45},
{15,27}}
```

Entfernungssumme: 182

25 Prozessoren

```
{{0,20},{1,0},{2,4},{3,3},{4,1},{5,5},{6,26},{7,7},{8,16},
{9,25},{10,46},{11,30},{12,6},{13,33},{14,41},{15,12},
{16,13},{17,18},{18,10},{19,40},{20,8},{21,9},{22,14},
{23,11},{24,24}}
```

Entfernungssumme: 418

36 Prozessoren

```
{{0,7},{1,15},{2,36},{3,44},{4,40},{5,21},{6,38},{7,5},
{8,47},{9,42},{10,43},{11,31},{12,22},{13,30},{14,45},
{15,14},{16,8},{17,6},{18,16},{19,9},{20,3},{21,12},
{22,26},{23,17},{24,18},{25,10},{26,35},{27,29},{28,27},
{29,2},{30,19},{31,39},{32,33},{33,1},{34,34},{35,20}}
```

Entfernungssumme: 800

A.2 Die „schlechten“ Konfigurationen

4 Prozessoren

```
{{0,23},{1,40},{2,41},{3,22}}
```

Entfernungssumme: 60

9 Prozessoren

```
{{0,29},{1,6},{2,35},{3,46},{4,17},{5,11},{6,1},{7,42},{8,21}}
```

Entfernungssumme: 150

16 Prozessoren

```
{{0,31},{1,44},{2,39},{3,33},{4,36},{5,28},{6,16},{7,15},
{8,25},{9,40},{10,47},{11,23},{12,32},{13,26},{14,17},{15,45}}
```

Entfernungssumme: 374

25 Prozessoren

```
{{0,22},{1,12},{2,17},{3,42},{4,21},{5,40},{6,16},{7,43},
{8,37},{9,31},{10,34},{11,45},{12,23},{13,27},{14,39},{15,25},
{16,18},{17,5},{18,44},{19,13},{20,32},{21,14},{22,38},
{23,20},{24,15}}
```

Entfernungssumme: 686

36 Prozessoren

```
{{0,12},{1,17},{2,45},{3,30},{4,44},{5,39},{6,43},{7,4},{8,23},
{9,41},{10,11},{11,27},{12,29},{13,1},{14,36},{15,28},{16,32},
{17,21},{18,34},{19,9},{20,31},{21,2},{22,24},{23,42},{24,25},
```

```
{25,6},{26,37},{27,26},{28,47},{29,20},{30,7},{31,16},{32,18},
{33,33},{34,13},{35,40}}
```

Entfernungssumme: 1170

Anhang B

Die Konfigurationen der Testmaschinen

In diesem Anhang werden die Konfigurationen der verwendeten Testmaschinen aufgeführt. Diese Informationen sollen dazu dienen, die vorgenommenen Untersuchungen mit Messungen an gleichen oder unterschiedlichen Maschinen vergleichen zu können. Dafür führt das angegebene Hardware-Inventar sowohl die Anzahlen und Typen der enthaltenen Prozessoren und Speichermodule als auch die eingesetzten Router und ihre Verschaltung auf.

Das Hardware-Inventar wurde mittels des IRIX-Kommandos `hinv -vm` generiert. Um Übertragungsfehler zu verhindern, wurde die Ausgabe nicht gekürzt. Die angegebenen Prozessor- und Router-Bezeichnungen entsprechen den in Kapitel 4 verwendeten Nummern.

B.1 Die SGI Origin-2000 der TU Dresden

Das Universitätsrechenzentrum der TU Dresden betreibt eine Origin-2000 namens RAPUNZEL. Der Rechner enthält 48 Prozessoren und 17 GByte Hauptspeicher. Die Router des Verbindungsnetzes bilden einen unvollständig besetzten vierdimensionalen Hyperkubus. Siehe Abbildung B.1 auf Seite 84.

B.1.1 Hardware-Inventar

Protokoll der Ausgabe des Kommandos `hinv -vm` auf der Maschine RAPUNZEL des Universitätsrechenzentrums der TU Dresden, ausgeführt am 20.12.1999. Während der Durchführung dieser Studie wurde die Hardware-Ausstattung nicht geändert.

```
Location: /hw/module/1/slot/n1/node
        MODULEID Board: barcode K0002200   part              rev
       8P12_MPLN Board: barcode DAP962     part 013-1547-003 rev  B
            IP27 Board: barcode DBA393     part 030-1266-001 rev  C
Location: /hw/module/1/slot/n2/node
            IP27 Board: barcode DGR084     part 030-1266-001 rev  C
Location: /hw/module/1/slot/n3/node
            IP27 Board: barcode DDR952     part 030-1266-001 rev  C
Location: /hw/module/1/slot/n4/node
            IP27 Board: barcode DKH844     part 030-1266-001 rev  C
Location: /hw/module/1/slot/r1/router
      ROUTER-IR1 Board: barcode CTA282     part 030-0841-002 rev  D
Location: /hw/module/1/slot/r2/router
      ROUTER-IR1 Board: barcode CTA293     part 030-0841-002 rev  D
Location: /hw/module/1/slot/io2/pci_xio
         PCI-XIO Board: barcode DPZ876     part 030-1062-002 rev  C
Location: /hw/module/1/slot/io1/baseio
          BASEIO Board: barcode CSH980     part 030-1124-001 rev  B
Location: /hw/module/2/slot/n1/node
```

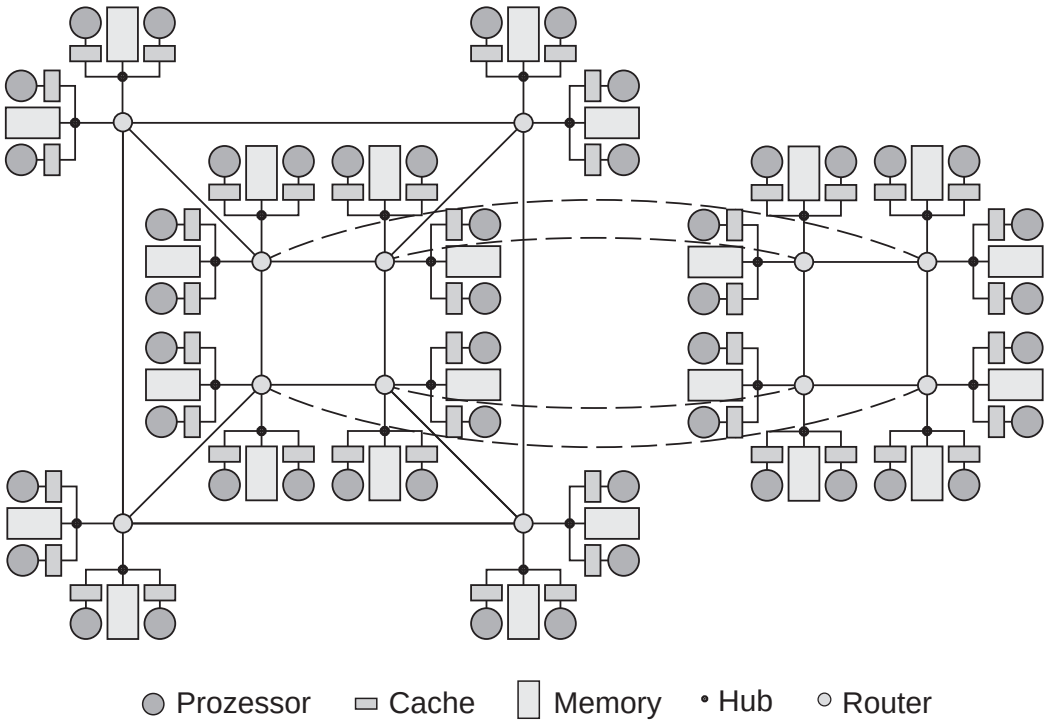

Abbildung B.1: **Die SGI Origin der TU Dresden** – Die Router des Verbindungsnetzes bilden einen unvollständig besetzten vierdimensionalen Hyperkubus. Jeder Router ist an zwei Rechenknoten und drei oder vier weitere Router angeschlossen. Die Rechenknoten enthalten ihrerseits jeweils zwei Prozessoren und ein Speichermodul. Die Verbindung zwischen dieser graphischen Darstellung und den in der Arbeit explizit angegebenen Prozessornummern ist durch das abgedruckte Hardware-Inventar gegeben.

```
        MODULEID Board: barcode K0002748   part              rev
            IP27 Board: barcode DKH965     part 030-1266-001 rev  C
       8P12-MPLN Board: barcode CEM902     part 013-1547-003 rev  B
Location: /hw/module/2/slot/n2/node
            IP27 Board: barcode DGL236     part 030-1266-001 rev  E
Location: /hw/module/2/slot/n3/node
            IP27 Board: barcode DNR446     part 030-1266-001 rev  C
Location: /hw/module/2/slot/n4/node
            IP27 Board: barcode DWJ564     part 030-1266-001 rev  C
Location: /hw/module/2/slot/r1/router
      ROUTER-IR1 Board: barcode CYM207     part 030-0841-002 rev  E
Location: /hw/module/2/slot/r2/router
      ROUTER-IR1 Board: barcode CYM198     part 030-0841-002 rev  E
Location: /hw/module/2/slot/io1/baseio
          BASEIO Board: barcode DCH881     part 030-1124-001 rev  C
Location: /hw/module/3/slot/n1/node
        MODULEID Board: barcode K0002875   part              rev
       8P12_MPLN Board: barcode DAP794     part 013-1547-003 rev  B
            IP27 Board: barcode DKH084     part 030-1266-001 rev  A
Location: /hw/module/3/slot/n2/node
            IP27 Board: barcode DGR032     part 030-1266-001 rev  C
Location: /hw/module/3/slot/n3/node
            IP27 Board: barcode DWK471     part 030-1266-001 rev  C
Location: /hw/module/3/slot/n4/node
```

```
           IP27 Board: barcode DGR269     part 030-1266-001 rev  C
Location: /hw/module/3/slot/r1/router
      ROUTER-IR1 Board: barcode CYM016     part 030-0841-002 rev  E
Location: /hw/module/3/slot/r2/router
      ROUTER-IR1 Board: barcode DPC526     part 030-0841-002 rev  G
Location: /hw/module/3/slot/io11/hippi_serial
    HIPPI_SERIAL Board: barcode FHA339     part 030-0968-004 rev  B
Location: /hw/module/3/slot/io1/baseio
          BASEIO Board: barcode DCH893     part 030-1124-001 rev  C
Location: /hw/module/4/slot/n1/node
        MODULEID Board: barcode K0002953   part              rev
           IP27 Board: barcode DKH500     part 030-1266-001 rev  C
       8P12_MPLN Board: barcode DAP823     part 013-1547-003 rev  B
Location: /hw/module/4/slot/n2/node
           IP27 Board: barcode DGR411     part 030-1266-001 rev  C
Location: /hw/module/4/slot/n3/node
           IP27 Board: barcode DKH849     part 030-1266-001 rev  C
Location: /hw/module/4/slot/n4/node
           IP27 Board: barcode DKH796     part 030-1266-001 rev  C
Location: /hw/module/4/slot/r1/router
      ROUTER-IR1 Board: barcode DPC984     part 030-0841-002 rev  G
Location: /hw/module/4/slot/r2/router
      ROUTER-IR1 Board: barcode DDB619     part 030-0841-002 rev  F
Location: /hw/module/4/slot/io1/baseio
          BASEIO Board: barcode DFM453     part 030-1124-001 rev  C
Location: /hw/module/5/slot/n1/node
        MODULEID Board: barcode K0002734   part              rev
           IP27 Board: barcode DGR581     part 030-1266-001 rev  C
       8P12-MPLN Board: barcode CEM881     part 013-1547-003 rev  B
Location: /hw/module/5/slot/n2/node
           IP27 Board: barcode DXA116     part 030-1266-001 rev  C
Location: /hw/module/5/slot/n3/node
           IP27 Board: barcode DAY083     part 030-1266-001 rev  C
Location: /hw/module/5/slot/n4/node
           IP27 Board: barcode DXA940     part 030-1266-001 rev  C
Location: /hw/module/5/slot/r1/router
        8PROUTER Board: barcode CYM094     part 030-0841-002 rev  E
Location: /hw/module/5/slot/r2/router
      ROUTER-IR1 Board: barcode CYM033     part 030-0841-002 rev  E
Location: /hw/module/5/slot/io1/baseio
          BASEIO Board: barcode DJS845     part 030-1124-002 rev  G
Location: /hw/module/6/slot/n1/node
        MODULEID Board: barcode K0002672   part              rev
       8P12-MPLN Board: barcode CEM743     part 013-1547-003 rev  A
           IP27 Board: barcode DGR840     part 030-1266-001 rev  C
Location: /hw/module/6/slot/n2/node
           IP27 Board: barcode DKH794     part 030-1266-001 rev  C
Location: /hw/module/6/slot/n3/node
           IP27 Board: barcode DKH845     part 030-1266-001 rev  C
Location: /hw/module/6/slot/n4/node
           IP27 Board: barcode DKH906     part 030-1266-001 rev  C
Location: /hw/module/6/slot/r1/router
      ROUTER-IR1 Board: barcode CYM229     part 030-0841-002 rev  E
Location: /hw/module/6/slot/r2/router
      ROUTER-IR1 Board: barcode CYM079     part 030-0841-002 rev  E
Location: /hw/module/6/slot/io1/baseio
          BASEIO Board: barcode DFM435     part 030-1124-001 rev  C
Location: /hw/module/6/slot/io3/mscsi
           MSCSI Board: barcode EGS178     part 030-0872-003 rev  A
48 195 MHZ IP27 Processors
CPU: MIPS R10000 Processor Chip Revision: 2.6
FPU: MIPS R10010 Floating Point Chip Revision: 0.0
CPU 0 at Module 1/Slot 1/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 1 at Module 1/Slot 1/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 2 at Module 1/Slot 2/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
```

```
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 3 at Module 1/Slot 2/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 4 at Module 1/Slot 3/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 5 at Module 1/Slot 3/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 6 at Module 1/Slot 4/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 7 at Module 1/Slot 4/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 8 at Module 2/Slot 1/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 9 at Module 2/Slot 1/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 10 at Module 2/Slot 2/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 11 at Module 2/Slot 2/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 12 at Module 2/Slot 3/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 13 at Module 2/Slot 3/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 14 at Module 2/Slot 4/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 15 at Module 2/Slot 4/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 16 at Module 3/Slot 1/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 17 at Module 3/Slot 1/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 18 at Module 3/Slot 2/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 19 at Module 3/Slot 2/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 20 at Module 3/Slot 3/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 21 at Module 3/Slot 3/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 22 at Module 3/Slot 4/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 23 at Module 3/Slot 4/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 24 at Module 4/Slot 1/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 25 at Module 4/Slot 1/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 26 at Module 4/Slot 2/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 27 at Module 4/Slot 2/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 28 at Module 4/Slot 3/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 29 at Module 4/Slot 3/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 30 at Module 4/Slot 4/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 31 at Module 4/Slot 4/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 32 at Module 5/Slot 1/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 33 at Module 5/Slot 1/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 34 at Module 5/Slot 2/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 35 at Module 5/Slot 2/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
```

```
CPU 36 at Module 5/Slot 3/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 37 at Module 5/Slot 3/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 38 at Module 5/Slot 4/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 39 at Module 5/Slot 4/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 40 at Module 6/Slot 1/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 41 at Module 6/Slot 1/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 42 at Module 6/Slot 2/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 43 at Module 6/Slot 2/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 44 at Module 6/Slot 3/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 45 at Module 6/Slot 3/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 46 at Module 6/Slot 4/Slice A: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
CPU 47 at Module 6/Slot 4/Slice B: 195 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 4 MB Speed 130 Mhz
Main memory size: 17408 Mbytes
Instruction cache size: 32 Kbytes
Data cache size: 32 Kbytes
Secondary unified instruction/data cache size: 4 Mbytes
Memory at Module 1/Slot 1: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 1/Slot 2: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 1/Slot 3: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 1/Slot 4: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 2/Slot 1: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 2/Slot 2: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 2/Slot 3: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 2/Slot 4: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
```

```
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 3/Slot 1: 1024 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
  Bank 6 contains 128 MB (Premium) DIMMS (enabled)
  Bank 7 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 3/Slot 2: 1024 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
  Bank 6 contains 128 MB (Premium) DIMMS (enabled)
  Bank 7 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 3/Slot 3: 1024 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
  Bank 6 contains 128 MB (Premium) DIMMS (enabled)
  Bank 7 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 3/Slot 4: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 4/Slot 1: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 4/Slot 2: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 4/Slot 3: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 4/Slot 4: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 5/Slot 1: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 5/Slot 2: 4096 MB (enabled)
  Bank 0 contains 512 MB (Premium) DIMMS (enabled)
  Bank 1 contains 512 MB (Premium) DIMMS (enabled)
  Bank 2 contains 512 MB (Premium) DIMMS (enabled)
  Bank 3 contains 512 MB (Premium) DIMMS (enabled)
  Bank 4 contains 512 MB (Premium) DIMMS (enabled)
  Bank 5 contains 512 MB (Premium) DIMMS (enabled)
  Bank 6 contains 512 MB (Premium) DIMMS (enabled)
  Bank 7 contains 512 MB (Premium) DIMMS (enabled)
```

```
Memory at Module 5/Slot 3: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 5/Slot 4: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 2 contains 128 MB (Premium) DIMMS (enabled)
  Bank 3 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 6/Slot 1: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 6/Slot 2: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 6/Slot 3: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
Memory at Module 6/Slot 4: 512 MB (enabled)
  Bank 0 contains 128 MB (Premium) DIMMS (enabled)
  Bank 1 contains 128 MB (Premium) DIMMS (enabled)
  Bank 4 contains 128 MB (Premium) DIMMS (enabled)
  Bank 5 contains 128 MB (Premium) DIMMS (enabled)
ROUTER in Module 1/Slot 2: Revision 2: Active Ports [1,2,3,4,5,6] (enabled)
ROUTER in Module 1/Slot 4: Revision 2: Active Ports [1,2,3,4,5,6] (enabled)
ROUTER in Module 2/Slot 2: Revision 2: Active Ports [1,2,3,4,5,6] (enabled)
ROUTER in Module 2/Slot 4: Revision 2: Active Ports [1,2,3,4,5,6] (enabled)
ROUTER in Module 3/Slot 2: Revision 2: Active Ports [1,2,4,5,6] (enabled)
ROUTER in Module 3/Slot 4: Revision 2: Active Ports [1,2,4,5,6] (enabled)
ROUTER in Module 4/Slot 2: Revision 2: Active Ports [1,2,4,5,6] (enabled)
ROUTER in Module 4/Slot 4: Revision 2: Active Ports [1,2,4,5,6] (enabled)
ROUTER in Module 5/Slot 2: Revision 2: Active Ports [1,3,4,5,6] (enabled)
ROUTER in Module 5/Slot 4: Revision 2: Active Ports [1,3,4,5,6] (enabled)
ROUTER in Module 6/Slot 2: Revision 2: Active Ports [1,3,4,5,6] (enabled)
ROUTER in Module 6/Slot 4: Revision 2: Active Ports [1,3,4,5,6] (enabled)
Integral SCSI controller 20: Version QL1040B, single ended
  Disk drive: unit 1 on SCSI controller 20 (unit 1)
  Disk drive: unit 2 on SCSI controller 20 (unit 2)
  Disk drive: unit 3 on SCSI controller 20 (unit 3)
  Disk drive: unit 4 on SCSI controller 20 (unit 4)
  Disk drive: unit 5 on SCSI controller 20 (unit 5)
Integral SCSI controller 0: Version QL1040B, single ended
  Disk drive: unit 1 on SCSI controller 0 (unit 1)
  Disk drive: unit 2 on SCSI controller 0 (unit 2)
  Disk drive: unit 3 on SCSI controller 0 (unit 3)
  Disk drive: unit 4 on SCSI controller 0 (unit 4)
  Disk drive: unit 5 on SCSI controller 0 (unit 5)
  CDROM: unit 6 on SCSI controller 0
Integral SCSI controller 30: Version QL1040B, single ended
  Disk drive: unit 1 on SCSI controller 30 (unit 1)
  Disk drive: unit 2 on SCSI controller 30 (unit 2)
  Disk drive: unit 3 on SCSI controller 30 (unit 3)
  Disk drive: unit 4 on SCSI controller 30 (unit 4)
Integral SCSI controller 40: Version QL1040B, single ended
  Disk drive: unit 2 on SCSI controller 40 (unit 2)
  Disk drive: unit 3 on SCSI controller 40 (unit 3)
  Disk drive: unit 4 on SCSI controller 40 (unit 4)
Integral SCSI controller 50: Version QL1040B (rev. 2), single ended
  Disk drive: unit 1 on SCSI controller 50 (unit 1)
  Disk drive: unit 2 on SCSI controller 50 (unit 2)
```

```
  Disk drive: unit 3 on SCSI controller 50 (unit 3)
Integral SCSI controller 60: Version QL1040B, single ended
  Disk drive: unit 1 on SCSI controller 60 (unit 1)
  Disk drive: unit 2 on SCSI controller 60 (unit 2)
  Disk drive: unit 3 on SCSI controller 60 (unit 3)
Integral SCSI controller 51: Version QL1040B (rev. 2), single ended
Integral SCSI controller 21: Version QL1040B, single ended
Integral SCSI controller 41: Version QL1040B, single ended
Integral SCSI controller 1: Version QL1040B, single ended
Integral SCSI controller 31: Version QL1040B, single ended
Integral SCSI controller 61: Version QL1040B, single ended
  Disk drive: unit 1 on SCSI controller 61 (unit 1)
  Disk drive: unit 2 on SCSI controller 61 (unit 2)
  Disk drive: unit 6 on SCSI controller 61 (unit 6)
Integral SCSI controller 62: Version QL1040B (rev. 2), single ended
  Tape drive: unit 1 on SCSI controller 62: DAT
Integral SCSI controller 63: Version QL1040B (rev. 2), differential
  Disk drive: unit 3 on SCSI controller 63 (unit 3)
  Disk drive: unit 4 on SCSI controller 63 (unit 4)
  Disk drive: unit 5 on SCSI controller 63 (unit 5)
Integral SCSI controller 64: Version QL1040B (rev. 2), differential
  Disk drive: unit 3 on SCSI controller 64 (unit 3)
  Disk drive: unit 4 on SCSI controller 64 (unit 4)
  Disk drive: unit 5 on SCSI controller 64 (unit 5)
  Disk drive: unit 6 on SCSI controller 64 (unit 6)
Integral SCSI controller 65: Version QL1040B (rev. 2), differential
  Disk drive: unit 7 on SCSI controller 65 (unit 7)
  Disk drive: unit 8 on SCSI controller 65 (unit 8)
IOC3 serial port: tty3
IOC3 serial port: tty4
IOC3 serial port: tty1
IOC3 serial port: tty2
IOC3 serial port: tty12
IOC3 serial port: tty8
IOC3 serial port: tty5
IOC3 serial port: tty11
IOC3 serial port: tty6
IOC3 serial port: tty7
IOC3 serial port: tty9
IOC3 serial port: tty10
RNS 2200 PCI/FDDI controller: rns0, module 1, PCI slot 0, version 48
HIPPI-Serial adapter: unit 2, in module 3 I/O slot 11
Integral Fast Ethernet: ef0, version 1, module 1, slot io1, pci 2
Fast Ethernet: ef1, version 1, module 2, slot io1, pci 2
Fast Ethernet: ef5, version 1, module 6, slot io1, pci 2
Fast Ethernet: ef2, version 1, module 3, slot io1, pci 2
Fast Ethernet: ef3, version 1, module 4, slot io1, pci 2
Fast Ethernet: ef4, version 1, module 5, slot io1, pci 2
Origin PCI XIO board, module 1 slot 2: Revision 3
  PCI Adapter ID (vendor 4370, device 8704) pci slot 0
Origin BASEIO board, module 1 slot 1: Revision 2
Origin BASEIO board, module 2 slot 1: Revision 2
Origin BASEIO board, module 3 slot 1: Revision 2
Origin BASEIO board, module 4 slot 1: Revision 2
  PCI Adapter ID (vendor 4265, device 2) pci slot 0
  PCI Adapter ID (vendor 4265, device 2) pci slot 1
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
Origin BASEIO board, module 5 slot 1: Revision 3
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
```

```
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
Origin BASEIO board, module 6 slot 1: Revision 2
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
Origin MSCSI board, module 6 slot 3: Revision 3
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
  PCI Adapter ID (vendor 4215, device 4128) pci slot 2
  PCI Adapter ID (vendor 4215, device 4128) pci slot 3
IOC3 external interrupts: 1
IOC3 external interrupts: 2
IOC3 external interrupts: 6
IOC3 external interrupts: 3
IOC3 external interrupts: 4
IOC3 external interrupts: 5
HUB in Module 1/Slot 1: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 1/Slot 2: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 1/Slot 3: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 1/Slot 4: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 2/Slot 1: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 2/Slot 2: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 2/Slot 3: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 2/Slot 4: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 3/Slot 1: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 3/Slot 2: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 3/Slot 3: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 3/Slot 4: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 4/Slot 1: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 4/Slot 2: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 4/Slot 3: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 4/Slot 4: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 5/Slot 1: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 5/Slot 2: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 5/Slot 3: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 5/Slot 4: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 6/Slot 1: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 6/Slot 2: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 6/Slot 3: Revision 5 Speed 97.50 Mhz (enabled)
HUB in Module 6/Slot 4: Revision 5 Speed 97.50 Mhz (enabled)
IP27prom in Module 1/Slot n1: Revision 6.31
IP27prom in Module 1/Slot n2: Revision 6.31
IP27prom in Module 1/Slot n3: Revision 6.31
IP27prom in Module 1/Slot n4: Revision 6.31
IO6prom on Global Master Baseio in Module 1/Slot io1: Revision 6.31
IP27prom in Module 2/Slot n1: Revision 6.31
IP27prom in Module 2/Slot n2: Revision 6.31
IP27prom in Module 2/Slot n3: Revision 6.31
IP27prom in Module 2/Slot n4: Revision 6.31
IP27prom in Module 3/Slot n1: Revision 6.31
IP27prom in Module 3/Slot n2: Revision 6.31
IP27prom in Module 3/Slot n3: Revision 6.31
IP27prom in Module 3/Slot n4: Revision 6.31
IP27prom in Module 4/Slot n1: Revision 6.31
IP27prom in Module 4/Slot n2: Revision 6.31
IP27prom in Module 4/Slot n3: Revision 6.31
IP27prom in Module 4/Slot n4: Revision 6.31
IP27prom in Module 5/Slot n1: Revision 6.31
IP27prom in Module 5/Slot n2: Revision 6.31
IP27prom in Module 5/Slot n3: Revision 6.31
IP27prom in Module 5/Slot n4: Revision 6.31
IP27prom in Module 6/Slot n1: Revision 6.31
IP27prom in Module 6/Slot n2: Revision 6.31
```

```
IP27prom in Module 6/Slot n3: Revision 6.31
IP27prom in Module 6/Slot n4: Revision 6.31
```

B.1.2 Die Distanzmatrix

Die vom Betriebssystem IRIX bereitgestellte Distanzmatrix wurde über den Opcode `MP_NUMA_GETDISTMATRIX` des `sysmp()`-Systemaufrufes ausgelesen. Das i, j-te Element dieser Matrix gibt die Distanz zwischen den Rechenknoten i und j an. Die größte in RAPUNZEL auftretende Distanz beträgt 5 Hops.

```
00 01 02 02 02 02 03 03 02 02 03 03 03 03 04 04 02 02 03 03 03 03 04 04
01 00 02 02 02 02 03 03 02 02 03 03 03 03 04 04 02 02 03 03 03 03 04 04
02 02 00 01 03 03 02 02 03 03 02 02 04 04 03 03 03 03 02 02 04 04 03 03
02 02 01 00 03 03 02 02 03 03 02 02 04 04 03 03 03 03 02 02 04 04 03 03
02 02 03 03 00 01 02 02 03 03 04 04 02 02 03 03 03 03 04 04 02 02 03 03
02 02 03 03 01 00 02 02 03 03 04 04 02 02 03 03 03 03 04 04 02 02 03 03
03 03 02 02 02 02 00 01 04 04 03 03 03 03 02 02 04 04 03 03 03 03 02 02
03 03 02 02 02 02 01 00 04 04 03 03 03 03 02 02 04 04 03 03 03 03 02 02
02 02 03 03 03 03 04 04 00 01 02 02 02 02 03 03 03 03 04 04 04 04 05 05
02 02 03 03 03 03 04 04 01 00 02 02 02 02 03 03 03 03 04 04 04 04 05 05
03 03 02 02 04 04 03 03 02 02 00 01 03 03 02 02 04 04 03 03 05 05 04 04
03 03 02 02 04 04 03 03 02 02 01 00 03 03 02 02 04 04 03 03 05 05 04 04
03 03 04 04 02 02 03 03 02 02 03 03 00 01 02 02 04 04 05 05 03 03 04 04
03 03 04 04 02 02 03 03 02 02 03 03 01 00 02 02 04 04 05 05 03 03 04 04
04 04 03 03 03 03 02 02 03 03 02 02 02 02 00 01 05 05 04 04 04 04 03 03
04 04 03 03 03 03 02 02 03 03 02 02 02 02 01 00 05 05 04 04 04 04 03 03
02 02 03 03 03 03 04 04 03 03 04 04 04 04 05 05 00 01 02 02 02 02 03 03
02 02 03 03 03 03 04 04 03 03 04 04 04 04 05 05 01 00 02 02 02 02 03 03
03 03 02 02 04 04 03 03 04 04 03 03 05 05 04 04 02 02 00 01 03 03 02 02
03 03 02 02 04 04 03 03 04 04 03 03 05 05 04 04 02 02 01 00 03 03 02 02
03 03 04 04 02 02 03 03 04 04 05 05 03 03 04 04 02 02 03 03 00 01 02 02
03 03 04 04 02 02 03 03 04 04 05 05 03 03 04 04 02 02 03 03 01 00 02 02
04 04 03 03 03 03 02 02 05 05 04 04 04 04 03 03 03 03 02 02 02 02 00 01
04 04 03 03 03 03 02 02 05 05 04 04 04 04 03 03 03 03 02 02 02 02 01 00
```

B.1.3 System- und systemnahe Software

Betriebssystem

- IRIX Execution Environment 6.5.5m
- IRIX Development Headers 6.5.5m

Entwicklungsumgebung

- Development System 7.2.1
- MIPSpro Compiler 7.30
- C Frontend, Headers and Libraries 7.3
- C++ Frontend, Headers and Libraries 7.3

Message Passing Interface

- SGI-MPI Version 3.1.1.3
- MPT (Message Passing Toolkit) Version 1.2.1.3 (Implementiert den Interface-Standard MPI-1.2).

B.2 Die SGI Origin-200 der GIP AG, Mainz

Die Firma GIP AG Gesellschaft für Industriephysik in Mainz betreibt eine Origin-200 namens PALU. Der Rechner enthält 4 Prozessoren und 512 MByte Hauptspeicher.

B.2.1 Hardware-Inventar

Protokoll der Ausgabe des Kommandos `hinv -vm` auf der Maschine PALU der Firma GIP AG in Mainz, ausgeführt am 20.12.1999. Während der Durchführung dieser Studie wurde die Hardware-Ausstattung nicht geändert.

```
Location: /hw/module/1/slot/MotherBoard/node
   PIMM_2XT5_1MB Board: barcode DWK722     part 013-1896-001 rev  D
Location: /hw/module/1/slot/MotherBoard/node/xtalk/8
            IP29 Board: barcode DWT421     part 030-1025-002 rev  J
Location: /hw/module/1/slot/MotherBoard/node/xtalk/8/pci/2
Location: /hw/module/2/slot/MotherBoard/node
   PIMM_2XT5_1MB Board: barcode DWT716     part 013-1896-001 rev  D
Location: /hw/module/2/slot/MotherBoard/node/xtalk/8
            IP29 Board: barcode DWT213     part 030-1025-002 rev  J
Location: /hw/module/2/slot/MotherBoard/node/xtalk/8/pci/2
4 180 MHZ IP27 Processors
CPU: MIPS R10000 Processor Chip Revision: 2.6
FPU: MIPS R10010 Floating Point Chip Revision: 0.0
CPU 0 at Module 1/Slot 1/Slice A: 180 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 1 MB Speed 120 Mhz
CPU 1 at Module 1/Slot 1/Slice B: 180 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 1 MB Speed 120 Mhz
CPU 2 at Module 2/Slot 2/Slice A: 180 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 1 MB Speed 120 Mhz
CPU 3 at Module 2/Slot 2/Slice B: 180 Mhz MIPS R10000 Processor Chip (enabled)
  Processor revision: 2.6. Secondary cache: Size 1 MB Speed 120 Mhz
Main memory size: 512 Mbytes
Instruction cache size: 32 Kbytes
Data cache size: 32 Kbytes
Secondary unified instruction/data cache size: 1 Mbyte
Memory at Module 1/Slot 1: 256 MB (enabled)
  Bank 0 contains 64 MB (Standard) DIMMS (enabled)
  Bank 1 contains 64 MB (Standard) DIMMS (enabled)
  Bank 2 contains 64 MB (Standard) DIMMS (enabled)
  Bank 3 contains 64 MB (Standard) DIMMS (enabled)
Memory at Module 2/Slot 2: 256 MB (enabled)
  Bank 0 contains 64 MB (Standard) DIMMS (enabled)
  Bank 1 contains 64 MB (Standard) DIMMS (enabled)
  Bank 2 contains 64 MB (Standard) DIMMS (enabled)
  Bank 3 contains 64 MB (Standard) DIMMS (enabled)
```

```
Integral SCSI controller 0: Version QL1040B (rev. 2), single ended
  Disk drive: unit 1 on SCSI controller 0
  Disk drive: unit 2 on SCSI controller 0
  Disk drive: unit 3 on SCSI controller 0
  Disk drive: unit 4 on SCSI controller 0
  Disk drive: unit 5 on SCSI controller 0
Integral SCSI controller 3: Version QL1040B (rev. 2), single ended
Integral SCSI controller 4: Version QL1040B (rev. 2), single ended
  CDROM: unit 3 on SCSI controller 4
Integral SCSI controller 1: Version QL1040B (rev. 2), single ended
  Disk drive: unit 5 on SCSI controller 1
Integral SCSI controller 2: Version QL1040B, differential
IOC3 serial port: tty1
IOC3 serial port: tty2
IOC3 serial port: tty3
IOC3 serial port: tty4
IOC3 parallel port: plp1
IOC3 parallel port: plp2
Integral Fast Ethernet: ef0, version 1, module 1, slot MotherBoard, pci 2
Fast Ethernet: ef1, version 1, module 2, slot MotherBoard, pci 2
Origin 200 base I/O, module 1 slot 1
Origin 200 base I/O, module 2 slot 2
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
  PCI Adapter ID (vendor 4265, device 3) pci slot 2
  PCI Adapter ID (vendor 4215, device 4128) pci slot 5
  PCI Adapter ID (vendor 4215, device 4128) pci slot 0
  PCI Adapter ID (vendor 4215, device 4128) pci slot 1
IOC3 external interrupts: 1
IOC3 external interrupts: 2
HUB in Module 1/Slot 1: Revision 3 Speed 90.00 Mhz (enabled)
HUB in Module 2/Slot 2: Revision 3 Speed 90.00 Mhz (enabled)
IP27prom in Module 1/Slot n1: Revision 6.11
IP27prom in Module 2/Slot n2: Revision 6.11
```

B.2.2 Die Distanzmatrix

```
00 01
01 00
```

B.2.3 System- und systemnahe Software

Betriebssystem

- IRIX Execution Environment 6.5
- IRIX Development Headers 6.5

Entwicklungsumgebung

- Development System 7.2.1
- MIPSpro Compiler 7.2.1

- C Frontend, Headers and Libraries 7.2.1
- C++ Frontend, Headers and Libraries 7.2.1

Message Passing Interface

- SGI-MPI Version 3.2.0.0
- MPT (Message Passing Toolkit) Version 1.3.0.0 (Implementiert den Interface-Standard MPI-1.2).

Literaturverzeichnis

[Ager95] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. *SP2 System Architecture.* IBM Systems Journal, 34(2), 1995.

[Alex95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. *LogGP - Incorporating Long Messages into the LogP Model.* Proc. Seventh Ann. ACM Symp. Parallel Algorithms and Architectures, pages 95–105, July 1995.

[Alma94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing (Second Edition).* Benjamin/Cummings Publishing, 1994.

[Char98] Alan Charlesworth. *STARFIRE: Extending the SMP Envelope.* IEEE Micro, pages 39–49, January/February 1998.

[Comp98b] Compaq Computer Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual.* Compaq Computer Corporation. Order Number EC-QP99CTE, 1998.

[Comp99b] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual.* Compaq Computer Corporation. Order Number Order Number EC-RJRZA-TE, 1999.

[Corm90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Cull93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. *LogP: Towards a Practical Model of Parallel Computation.* Proceedings of the Fourth ACM SIGPLAN Symposium on Priciples and Practice of Parallel Programming, May 1993, 1993.

[Cull96] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. *LogP - A Practical Model of Parallel Computation.* Communications of the ACM, 39(11), 1996.

[Cull96b] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. *Assessing Fast Network Interfaces.* IEEE Micro, February 1996.

[Cull99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1999.

[Dand99] Sivarama P. Dandamudi. *Reducing Hot-Spot Contention in Shared-Memory Mulitprocessor Systems.* IEEE Concurrency, pages 48–59, January-March 1999.

[Duss96] Andrea C. Dusseau, David E. Culler, Klaus Erik Schauser, and Richard P. Martin. *Fast Parallel Sorting Under LogP: Experience with the CM-5.* IEEE Transactions on Parallel and Distributed Systems, 7(8):791–805, August 1996.

[Eick92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. *Active Messages: A Mechanism for Integrated Communication and Computation. In* Proc. 19th Annual International Symposium on Computer Architecture, pages 256–266, 1992.

[Fort78] S. Fortune and J. Wyllie. *Parallelism in Random Access Machines. In* Proc. 10th Symp. on the Theory of Computing, pages 114–118, 1987.

[Frag98] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. *Modeling Set Associative Caches Behavior for Irregular Computations. In* Proc. SIGMETRICS '98/PERFORMANCE '98. Joint International Conference on Measurement and Modeling of Computer Systems. Madison, Wisconsin, USA., pages 192–201, June 1998.

[Gall95] Bill O. Gallmeister. *POSIX.4. Programming for the Real World.* O'Reilly & Associates, Inc., 1995.

[Gall96] Mike Galles. *The SGI SPIDER Chip – Scalable Pipelined Interconnect for Distributed Endpoint Routing.* Hot Interconnects Symposium IV, 1996.

[Gwen94] Linley Gwennap. *MIPS R10000 Uses Decoupled Architecture.* Microprocessor Report, 8(14), October 24, 1994.

[Gwen96a] Linley Gwennap. *Digital 21264 Sets New Standard. Clock Speed, Complexity, Performance Surpass Records, But Still a Year Away.* Microprocessor Report, 10(14), October 28, 1996.

[Gwen98a] Linley Gwennap. *Alpha 21364 to Ease Memory Bottleneck. Compaq Will Add Direct RDRAM to 21264 Core for Late 2000 Shipments.* Microprocessor Report, 12(14), October 26, 1998.

[HP98a] Hewlett-Packard. *Exemplar Architecture, S-Class and X-Class Servers, Second Edition.* Customer Order Number: A4716-90001, February 1998.

[Harp99] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. *Analytical Modeling of Set-Associative Cache Behavior.* IEEE Transactions of Computers, 12(14), October 26, 1998.

[Heis94] Hans Ulrich Heiss. *Prozessorzuteilung in Parallelrechnern.* BI-Wissenschaftsverlag, 1994.

[Henn96] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach.* Morgan Kaufmann Publishers, 1996.

[Hrom97] Juraj Hromkovič. *Communication Complexity and Parallel Computing.* Springer Texts in Theoretical Computer Science, 1997.

[IBM99a] IBM. *IBM POWERparallel Technology Briefing. Interconnection Technologies for High-Performance Computing RS/6000 SP.* IBM Resources, 1999.

[Jaja92] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing, 1992.

[Jian98] Dongming Jiang and Jaswinder Pal Singh. *A Methodology and an Evaluation of the SGI Origin-2000. In* Proc. SIGMETRICS '98/PERFORMANCE '98. Joint International Conference on Measurement and Modeling of Computer Systems. Madison, Wisconsin, USA., pages 171–181, June 1998.

[Knut97.1] Donald Ervin Knuth. *The Art of Computer Programming. Volume 1. Fundamental Algorithms. Third Edition.* Addison-Wesley Publishing, 1997.

[Kort] Iskander Kort and Denis Trystram. *Assessing LogP Model Performance for the IBM-SP.* LMC-IMAG, 1996.

[Laud97] James Laudon and Daniel Lenoski. *The SGI Origin: A ccNUMA Highly Scalable Server.* Silicon Graphics, 1997.

[Leig92] F. Thomson Leighton. *Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[Leno95] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publishers, 1995.

[Li95] Zhiyong Li, Peter H. Mills, and John H. Reif. *Models and Resource Metrics for Parallel and Distributed Computation. In* Proc. 28th Annual Hawaii International Conference on System Sciences. IEEE Press., 1995.

[Lisz97] Kathy J. Liszka, John K. Antonio, and Howard Jay Siegel. *Is an Alligator Better Than an Armadillo?* IEEE Micro, October-December 1997.

[Loh96] Peter Kok Keong Loh, Wen Hing Hsu, Cai Wentong, and Nadarajah Sriskanthan. *Now Network Topology Affects Dynamic Load Balancing*. IEEE Parallel & Distributed Technology, pages 25–35, Fall 1996.

[MIPS96] MIPS. *R10000 Microprocessor User's Manual. Version 2.0*. MIPS Technologies, 1996.

[MPI97a] Message Passing Interface Forum. *MPI-2: Extensions for the Message-Passing Interface*. July 18, 1997.

[MPI97b] Message Passing Interface Forum. *MPI-2: Journal of Development*. July 18, 1997.

[MPI98a] Message Passing Interface Forum. *Errata for MPI-2*. May 20, 1998.

[Mesc72] Herbert Meschkowski. *Meyers Handbuch über die Mathematik*. Bibliographisches Institut. Lexikonverlag, 1972.

[Mori98] Csaba Andras Moritz and Matthew I. Frank. *LoGPC: Modeling Network Contention in Message-Passing Programs. In* Proc. SIGMETRICS '98/PERFORMANCE '98. Joint International Conference on Measurement and Modeling of Computer Systems. Madison, Wisconsin, USA., pages 254–263, June 1998.

[Pach97] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[Papa94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing, 1994.

[Patt98] David A. Patterson and John L. Hennessy. *Computer Organisation & Design – The Hardware/Software Interface*. Morgan Kaufmann Publishers, 1998.

[Pfis85] G. E. Pfister and V. A. Norton. *Hot Spot Contention and Combining Multistage Interconnection Networks*. IEEE Transactions on Computers, 34(10), 1985.

[Przy90] Steven A. Przybylski. *Cache and Memory Hierarchy Design*. Morgan Kaufmann Publishers, 1990.

[SGI96a] Silicon Graphics. *Origin and Onyx2 Programmer's Reference Manual*. Document Number 007-3410-001, 1996.

[SGI97a] Silicon Graphics. *Origin and Onyx2 Theory of Operations Manual*. Document Number 007-3439-002, 1997.

[SGI99a] Silicon Graphics. *Performance of the Cray T3E Multiprocessor*. Cray T3E Whitepapers. 1999.

[SGI99b] Silicon Graphics. *Origin 2000 Datasheet*. 1999.

[Site96] Richard Sites. *It's the Memory, Stupid! Architects Look to Processors of Future. Applications, Instruction Sets, Memory Bandwidth Are Key Issues*. Microprocessor Report, 10(10), 1996.

[Stro94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing, 1994.

[Stro97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing, 1997.

[Sun95] Xian-He Sun and Jianping Zhu. *Performance Considerations of Shared Virtual Memory Machines*. IEEE Transactions on Parallel and Distributed Systems, 6(11), November 1995.

[Tane86] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1986.

[Tane95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[Tane96] Andrew S. Tanenbaum. *Computernetzwerke*. Prentice Hall, 1996.

[Touy99] Takayoshi Touyama and Susumu Horiguchi. *Performance Evaluation of Practical Parallel Computation Model LogPQ*. *In* Fourth International Symposium on Parallel Architectures, Algorithms, and Networks. 23-25 June, 1999. Fremantle, Australia, 1999.

[Tura96] Volker Turau. *Algorithmische Graphentheorie*. Addison-Wesley Publishing, 1996.

[Vali90] L. G. Valiant. *A Bridging Model for Parallel Computation*. Communications of the ACM, 33(8):103–111, August 1990.

[Wolf96] Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, 1996.

[Yan99] Yong Yan and Xiaodong Zhang. *Profit-Effective Parallel Computing*. IEEE Concurrency, April-June 1999.