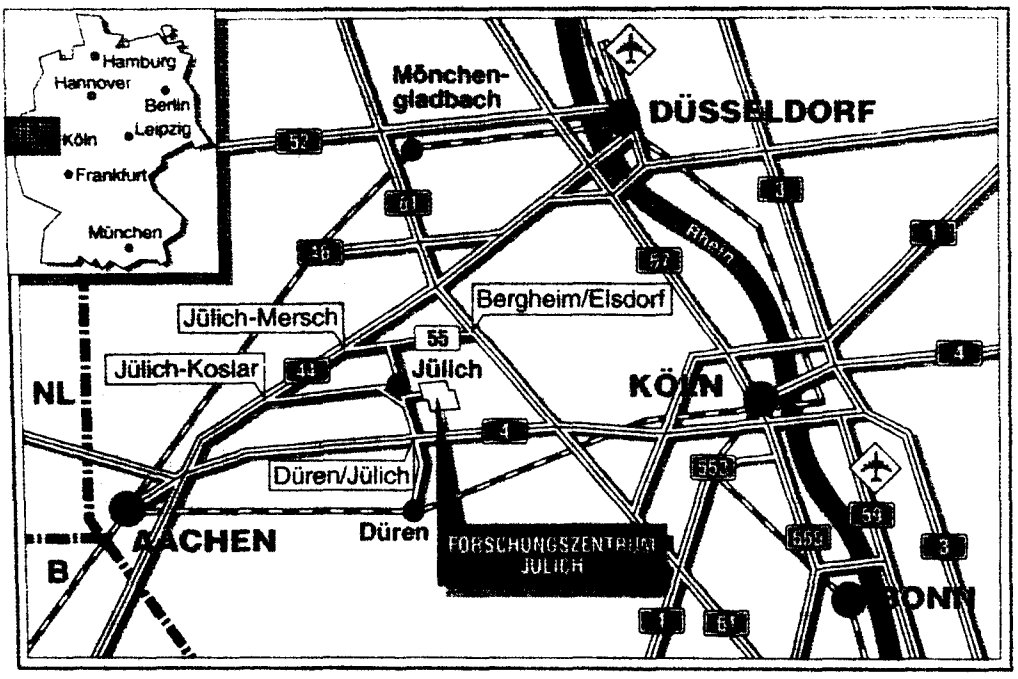
FORSCHUNGSZENTRUM JÜLICH GmbH

Zentralinstitut für Angewandte Mathematik

Graphenalgorithmen auf massiv-parallelen Rechnern

Frank Friedrichs



Berichte des Forschungszentrums Jülich ; 2885
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-2885

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

Graphenalgorithmen auf massiv-parallelen Rechnern

Frank Friedrichs

Kurzfassung

In diesem Bericht werden drei Graphenalgorithmen hinsichtlich ihrer Parallelisierbarkeit für massiv-parallele Rechner untersucht. Der Unterteilungsalgorithmus von Karp ermittelt gemäß einer Divide-and-Conquer-Strategie eine Näherungslösung für das Traveling-Salesman-Problem. Durch den Algorithmus von Prim-Dijkstra werden minimale spannende Bäume berechnet. Der dritte Algorithmus baut darauf auf und untersucht Graphen bezüglich ihrer Zusammenhangskomponenten. Nach einer ausführlichen Beschreibung der beiden massiv-parallelen Rechner Intel iPSC/860 und Intel Paragon XP/S 5 werden für jeden Algorithmus unterschiedliche parallele Kommunikationsvarianten implementiert. Die dabei erzielten Ergebnisse werden in Schaubildern und Tabellen dargestellt.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die Graphentheorie gewinnt mit der Verfügbarkeit immer leistungsstärkerer Rechnersysteme für praxisnahe Anwendungen immer mehr an Bedeutung. Viele reale Probleme lassen sich leicht auf Probleme aus der Graphentheorie abbilden. Für die meisten graphentheoretischen Probleme existieren schon exakte Lösungsmethoden oder auch Näherungsverfahren, mit denen Lösungen auf Rechnersystemen erstellt werden können.
Die Lösung von praxisorientierten Anwendungen, die mit graphentheoretischen Verfahren modelliert werden, erfordert aber neben einer großen Speicherkapazität des Rechners auch hohe Rechenzeiten. Diese können nun mit Hilfe von parallelen Rechnersystemen drastisch verkürzt werden.
Daneben bietet der größere Hauptspeicher der Parallelrechner die Möglichkeit, größere Probleme zu bearbeiten und bei der Berechnung weitgehend auf externe Speicher zu verzichten.

Die Entwicklung neuer leistungsstarker Rechner begann mit der Einführung von Vektor-Supercomputern, wie zum Beispiel 1976 der CRAY-1 oder 1981 der Cyber 205. Sie besitzen wenige sehr leistungsfähige Prozessoren und einen gemeinsamen Hauptspeicher.
Die massiv-parallelen Rechner zeichnen sich durch eine sehr große Anzahl von Prozessoren aus, die weniger leistungsstark sind. Zu den ersten Projekten im Bereich der massiv-parallelen Rechner gehören der C.mmp (Computer with multiple mini-processors), dessen Entwicklung 1971 an der Carnegie-Mellon University begann, sowie der ILLIAC IV. 1980 wurde von ICL der Distributed Array Processor (DAP) eingeführt.
Die Firma Intel stellte 1986 ihren ersten Hypercube, den iPSC/1, vor, der 32 bis 128 Prozessoren enthielt. 1988 kam der Nachfolger iPSC/2 auf den Markt. Der in dieser Arbeit untersuchte iPSC/860 setzte die Entwicklung der Hypercube-Rechner 1990 fort. Im Gegensatz dazu ist der Paragon XP/S, das neueste Produkt der Firma Intel (1992), ein massiv-paralleler Rechner, dessen Verbindungsnetzwerk aus einem zweidimensionalen Gitter besteht.

Dadurch, daß jeder Prozessor nur direkten Zugriff auf die Daten in seinem lokalen Speicher hat, müssen neue Programmierkonzepte entworfen werden. Daneben bieten die neuen Rechnerarchitekturen die Möglichkeit, direkt parallele Algorithmen zu entwickeln anstatt sequentielle Algorithmen zu parallelisieren.

Für Rechnersysteme mit gemeinsamem Hauptspeicher sind schon in vielen Arbeiten Graphenalgorithmen untersucht worden. Neben theoretischen Betrachtungen sind zum Beispiel in [Gur86], [Kar90] und [Tho90] unterschiedliche Algorithmen auf realen Parallelrechnern mit gemeinsamem Speicher implementiert worden.
Für massiv-parallele Rechnersysteme existieren derzeit hauptsächlich nur theoretische Arbeiten, wie etwa in [DDP90].

Ziel dieser Arbeit ist es, ausgewählte Graphenalgorithmen auf ihre Parallelisierbarkeit hin zu untersuchen, zu implementieren und die Ergebnisse mit denen zu vergleichen, die auf anderen Rechnersystemen erzielt wurden.

Dazu werden in Kapitel 2 nach einer allgemeinen Einführung in die Thematik der parallelen Rechnerarchitekturen die in dieser Arbeit verwendeten massiv-parallelen Rechner der Firma Intel, der iPSC/860 und der Paragon XP/S 5 vorgestellt. Neben den Charakteristiken der Hardware-Komponenten wird insbesondere das Programmiermodell Message Passing erläutert und ausführlich die Benutzung der Rechner beschrieben. Nach einer kurzen Einführung in die Grundbegriffe der Parallelverarbeitung wird die Kommunikationsleistung der beiden Rechner untersucht. Dafür werden verschiedene Situationen simuliert, die während eines Mehrbenutzterbetriebs auf den Rechnern eintreten können. Die Geschwindigkeit, in der Nachrichten zwischen Prozessoren ausgetauscht werden können, spielt eine große Rolle bei Rechnern mit verteiltem Speicher, denn die Beschleunigung, die durch die Parallelisierung von Algorithmen erreicht werden kann, hängt stark von den Übertragungsleistungen der Verbindungsnetzwerke ab.
Kapitel 3 stellt die für das Verständnis der untersuchten Graphenalgorithmen notwendigen Grundlagen der Graphentheorie bereit.
In den folgenden drei Kapiteln werden drei Graphenalgorithmen hinsichtlich ihrer Eignung zur Parallelisierbarkeit für massiv-parallele Rechner untersucht. Der erste Algorithmus bestimmt mit Hilfe einer Divide-and-Conquer-Struktur eine Näherungslösung für das Traveling-Salesman-Problem nach dem Unterteilungsalgorithmus von Karp. Der zweite implementierte Algorithmus berechnet einen minimalen spannenden Baum in ungerichteten Graphen. Darauf aufbauend wird ein dritter Algorithmus entwickelt, der Zusammenhangskomponenten und minimale spannende Wälder liefert. Nach jeder Beschreibung eines Algorithmus folgt eine Analyse und Bewertung der vorgenommenen Implementationen.
Die erreichten Ergebnisse verschiedener Testläufe mit unterschiedlichen Problemgrößen werden anhand von Tabellen und Graphiken dokumentiert.

Kapitel 2

Massiv-parallele Rechnerarchitekturen

In diesem Kapitel soll ein Einblick in parallele Rechnerarchitekturen gegeben werden. Nach einer Einführung in verschiedene Programmiermodelle und Speicherstrukturen werden die in dieser Arbeit benutzten massiv-parallelen Rechner der Firma Intel, der **iPSC/860** und der **Paragon XP/S 5**, ausführlich beschrieben.

Ein grundsätzliches Unterscheidungsmerkmal verschiedener Rechnerarchitekturen ist die Behandlung der Befehls- und Datenströme, also der Folge auszuführender Befehle und der Folge der von diesen zu verarbeitenden Daten. Parallelismus beschreibt hier die gleichzeitige Behandlung mehrerer Objekte (Befehle und Daten) durch den zugrundeliegenden Rechner. Die gebräuchlichste Klassifikation von Rechnerarchitekturen geht auf Flynn zurück [Fly66]. Er unterscheidet die Architekturen gemäß ihrer Fähigkeit, Datenströme in einer oder in parallelen Verarbeitungseinheiten bzw. Befehlsströme in einer oder in multiplen Steuereinheiten zu bearbeiten. Nach Flynn gibt es vier Kategorien:

- **SISD**: **S**ingle **I**nstruction Stream **S**ingle **D**ata Stream
- **SIMD**: **S**ingle **I**nstruction Stream **M**ultiple **D**ata Streams
- **MISD**: **M**ultiple **I**nstruction Streams **S**ingle **D**ata Stream
- **MIMD**: **M**ultiple **I**nstruction Streams **M**ultiple **D**ata Streams

Mit dem Aufkommen der massiv-parallelen Rechner läßt sich die Klassifikation von Flynn um ein weiteres Modell erweitern:

- **SPMD**: **S**ingle **P**rogram **M**ultiple **D**ata Streams

Beim SPMD-Modell werden die verschiedenen Datensätze auf den Rechenknoten nicht nur von einzelnen gleichlautenden Instruktionen bearbeitet, sondern direkt von vollständigen Programmen [AlGo89]. Jeder Rechenknoten erhält eine Kopie des auszuführenden Programmes, welches er auf den nur ihm zugewiesenen Daten ausführt. Die unabhängig voneinander operierenden Rechenknoten können als eigenständige Rechner betrachtet werden. Das SPMD-Modell stellt damit eine Erweiterung des SIMD-Modells dar.

Die Klasse der SISD-Rechner umfaßt alle die Rechner, die auf der klassischen von-Neumann-Architektur beruhen. Die große Gruppe der Personal-Computer gehört dieser Klasse an.
Die Kategorie der MISD-Rechner ist von eher untergeordneter Bedeutung. Derzeitig existiert kein bekannter Rechner, der zu dieser Modellklasse gerechnet werden kann. Für die Parallelverarbeitung von Interesse sind daher nur die Rechner, die zu der Klasse der SIMD- bzw. SPMD- und zu der Klasse der MIMD-Rechner gerechnet werden können.

Die Rechner der SIMD-Klasse zeichnen sich dadurch aus, daß sie eine Steuer- und mehrere Verarbeitungseinheiten besitzen. Die zwei wichtigsten Untergruppen dieser Klasse sind die *Pipeline-* oder *Vektorrechner* und die *Prozessor-Arrays*.

Beim Pipelining werden Operationen oder Instruktionen in Segmente unterteilt, die durch eine entsprechende Hardware-Komponente sukzessiv abgearbeitet werden [Gil93]. Insbesondere die Verarbeitung von Vektoren wird durch eine Pipeline unterstützt. Daher werden diese Rechner auch als Vektorrechner bezeichnet.
Vertreter dieser Rechnergruppe sind die CRAY-Rechner wie die CRAY X-MP, die CRAY Y-MP und die CRAY C 90, die Rechner der Fujitsu VP-S-Serie sowie andere Systeme wie CONVEX, Alliant oder Cyber 205 [HwBr84].

In einem Prozessor-Array sind die Prozessoren in einem Feld mit gitterähnlicher Verbindung angeordnet. Sie werden durch eine zentrale Einheit gesteuert und synchronisiert. Die einzelnen Prozessoren können dabei mit einer begrenzten Anzahl anderer Prozessoren kommunizieren. (Sinngemäß fallen in diese Kategorie auch die *systolischen Arrays* [HwBr84], [AlGo89]). Die ILLIAC IV, die ICL DAP und die CM-2 der Thinking Machines Corporation sollen als Beispiele genannt werden [HoJe88].

Bei den MIMD-Rechnern führen die Prozessoren in der Regel unterschiedliche Befehlssätze für verschiedene Datensätzen aus.

Man unterscheidet zwei Kategorien von Speicheranordnungen (Abbildung 2.1):

- Parallelrechner mit gemeinsamem Hauptspeicher (**Shared Memory**)
- Parallelrechner mit verteiltem Hauptspeicher (**Distributed Memory**)

Die Datenkommunikation bei den Shared-Memory-Rechnern erfolgt über den gemeinsamen Hauptspeicher. Alle Daten und Informationen im Speicher sind für alle Prozessoren in gleicher Weise verfügbar.

Die Rechnerprototypen C.mmp und Cm* der Carnegie-Mellon University, der Parallelrechner Denelcor HEP und das Ultracomputer-Konzept der New York University (NYU) sind Rechner mit gemeinsamem Hauptspeicher [HoJe88].

Im Gegensatz zu den Shared-Memory-Rechnern müssen bei Rechnern mit verteiltem Hauptspeicher die Daten und Informationen durch die Prozessoren untereinander ausgetauscht werden. Dieser Nachrichtenaustausch wird durch **Message Passing** realisiert. Der NCUBE/ten, der Cosmic Cube vom California Institute of Technology und die Intel-Rechner der iPSC-Serie sowie deren Nachfolger Paragon sind Rechner mit verteiltem Hauptspeicher [AlGo89].

Der Paragon bietet, wie andere, grundsätzlich auch die Möglichkeit, das neue Speicherkonzept des Shared Virtual Memory anzuwenden, wie es im Rechner KSR-1 von Kendall Square Research erstmalig verwendet wurde [BEK93]. Das Konzept des Shared Virtual Memory bedeutet, daß trotz des physikalisch verteilten Speichers die Rechenknoten über einen gemeinsamen Speicheradreßraum verfügen und so das Konzept des Shared Memory simuliert wird.

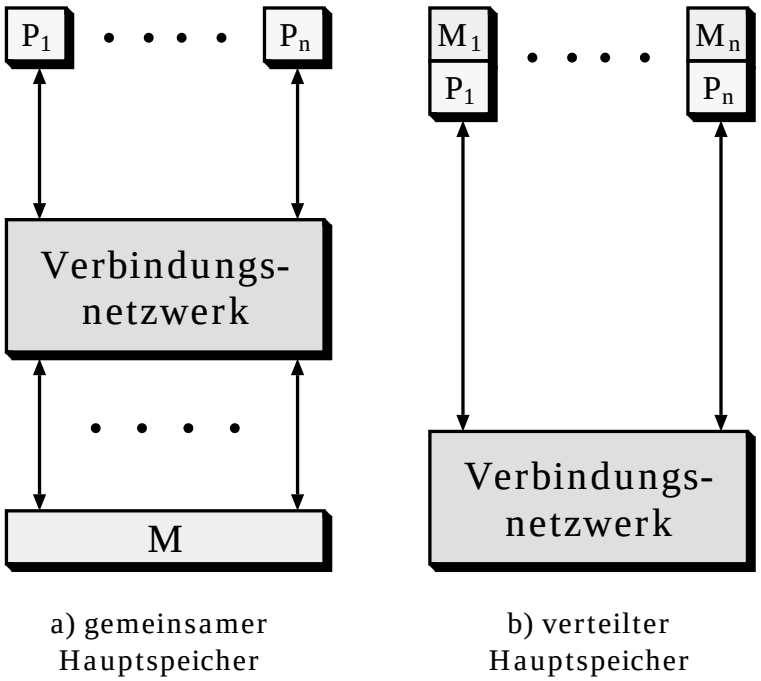


Abbildung 2.1: Verschiedene MIMD-Strukturen

2.1 Der Rechner Intel iPSC/860

Mit dem iPSC/860 stellte die Firma Intel 1990 ihr neuestes Produkt aus der Familie der MIMD-Rechner vor, das eine Hypercube-Netzwerktopologie mit verteiltem Speicher aufweist und die Interprozessorkommunikation durch Nachrichtenaustausch realisiert. Anders als beim Vorgängermodell, dem auf den Mikroprozessortypen 80386/80387 basierenden iPSC/2, wurde mit dem i860-Prozessor ein leistungsfähigerer Prozessor mit neuer Architektur verwendet. Die übrigen Systemkomponenten, wie etwa die Kommunikationseinheiten, blieben unverändert [BeHe90].

2.1.1 Systemarchitektur

Das im Januar 1992 im Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich installierte System Intel iPSC/860 Modell 32 verfügt über 32 Rechenknoten. Ein iPSC/860-Rechenknoten ist in Abbildung 2.2 dargestellt. Jeder Rechenknoten kann als eigenständige, unabhängige Einheit betrachtet werden und enthält einen Prozessor des Typs i860 XR, einen lokalen Speicher und für die Kommunikation das **D**irect **C**onnect **M**odul (DCM). Dem Netzwerk, mit dem die Rechenknoten miteinander verbunden sind, liegt eine Hypercube-Topologie zugrunde.

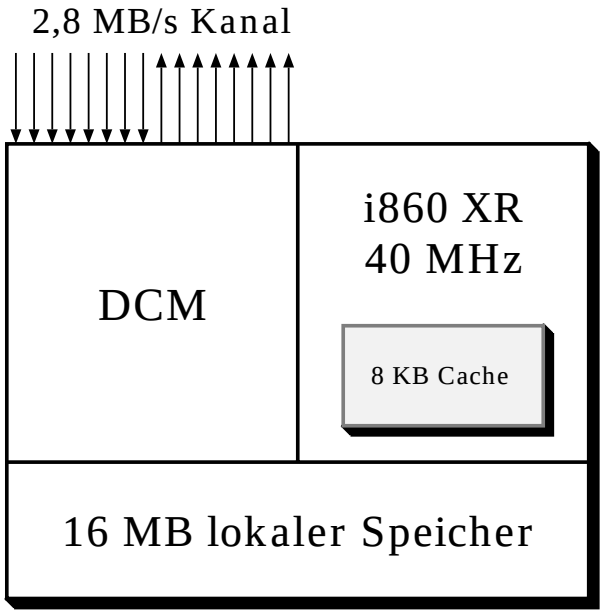


Abbildung 2.2: Darstellung eines iPSC/860-Rechenknotens

Der Mikroprozessor i860 XR [Int90] besitzt eine RISC-Architektur (**R**educed **I**nstruction **S**et **C**omputer [Bod90]) und enthält jeweils eine Funktionseinheit für Fließkommaaddition und -multiplikation. Zusätzlich ist der Prozessor i860 XR mit einer unabhängigen Funktionseinheit ausgestattet, die in der Sekunde 40 Millionen INTEGER-Operationen ausführen kann (Abbildung 2.3).

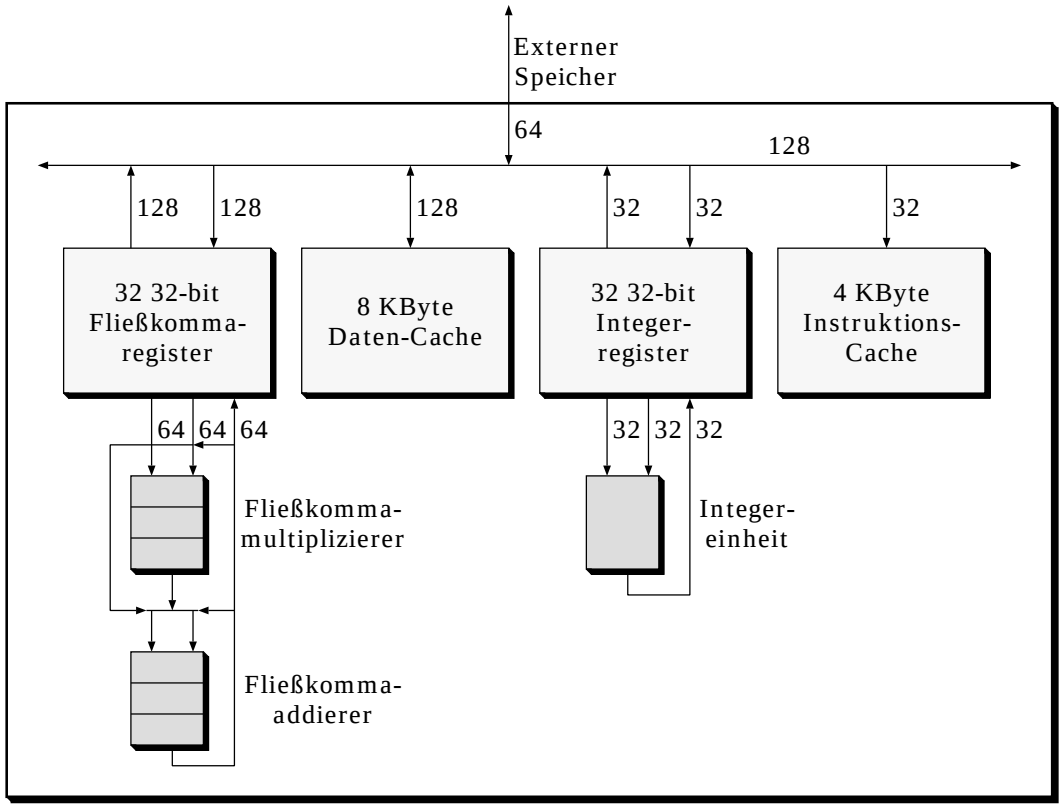


Abbildung 2.3: Vereinfachte Architektur des Intel i860-Chips

Bei einer Taktzeit von 25 ns (Taktfrequenz 40 MHz) beträgt die theoretisch mögliche Rechenleistung pro Knoten 80 MFLOPS bei 32-Bit- und 60 MFLOPS bei 64-Bit-Arithmetik. Daraus resultiert eine theoretische Maximalleistung des hier betrachteten Systems von 2.65 GFLOPS bzw. 1.92 GFLOPS. Der als LIW-Hypercube (LIW = **L**ong **I**nstruction **W**ord) charakterisierbare iPSC/860 kann mit bis zu 128 Prozessoren ausgerüstet werden, woraus sich eine maximale theoretische Rechenleistung von 10.24 GFLOPS bei einer Wortlänge von 32 Bit und 7.6 GFLOPS bei 64 Bit ergibt [HeBe90]. Die Knoten sind mit dem NX/2-Betriebssystem ausgestattet. Im Gegensatz zum iPSC/2, wo 20 Prozesse pro Knoten erlaubt sind, kann auf dem iPSC/860 nur ein Prozeß pro Knoten gestartet werden.

Der lokale Speicher eines Rechenknotens verfügt maximal über eine Kapazität von 16 MByte bei einer Zykluszeit von 70 ns. Die Gesamtkapazität der lokalen Speicher des iPSC/860 Modell 32 beträgt somit 512 MByte; maximal lassen sich bei 128 Knoten 2048 MByte Speicherkapazität erreichen.

Die Kommunikation der Rechenknoten untereinander steuert das **D**irect **C**onnect **M**odul (DCM) (Abbildung 2.4). Jedes der Kommunikationsmodule besitzt 7 Kanäle (von denen jedoch beim hier betrachteten System nur 5 belegt sind; daraus resultiert die Größe des Hypercube von $2^5 = 32$ bzw. die maximale Größe eines Hypercube von $2^7 = 128$ Knoten), über die der Nachrichtenaustausch eines jeden Rechenknotens mit seinem Nachbarknoten stattfindet. Für die externe Ein-/Ausgabe ist in jedem DCM ein achter Kanal reserviert. Die Kanäle sind bidirektional und bitseriell ausgelegt und verfügen dabei in beiden Richtungen zur gleichen Zeit über eine Bandbreite von 2.8 MByte/s [BDG90].

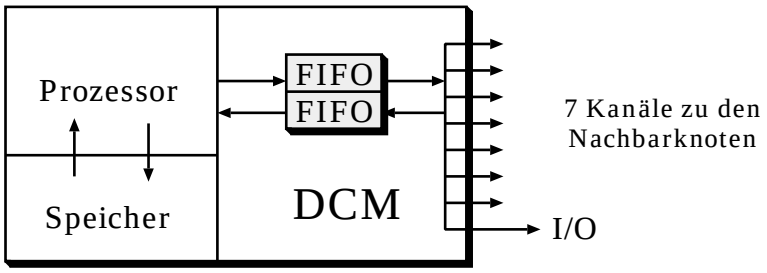


Abbildung 2.4: Kommunikationsmodul DCM

Das beschriebene System besitzt, je nach Länge der Nachricht, verschiedene Übertragungsleistungen. Bei kurzen Nachrichten bis zu 100 Byte wird das mit der geringen Startup-Zeit von 69 μs arbeitende **short message protocol** aufgerufen. Längere Nachrichten werden mit Hilfe des **long message protocol** verschickt. Die Startup-Zeit ist höher und die Übertragungsraten liegen bei 1 MByte/s für bis zu 300 Byte lange und 2 MByte/s für 1500 Byte lange Nachrichten. Unter Startup-Zeit ist die von der Nachrichtenlänge unabhängige Zeit zu verstehen, die das System zur Initiierung einer Nachrichtenübertragung benötigt. Beim short message protocol wird die bis zu 100 Byte lange Nachricht einfach ohne Rückfragen vom Sender an den Empfänger verschickt. Im Gegensatz dazu ist das long message protocol ein 3-Wege-Protokoll. Der Sender überprüft durch eine Anfrage, ob der Empfänger genügend Speicherplatz für die zu empfangende Nachricht besitzt. Nach Erhalt einer Bestätigung wird dann die Nachricht gesendet. Dadurch erhöht sich die Startup-Zeit des long message protocol auf 195 μs [BeHe90].

Soll ein Nachrichtenaustausch zwischen zwei benachbarten Rechenknoten stattfinden, so bauen die jeweiligen DCM dynamisch die benötigten Kommunikationskanäle auf. Erst wenn die Nachrichtenübertragung vollständig beendet wurde, werden die dafür aufgebauten Kanäle wieder freigegeben. Die zu versendenden oder empfangenden Nachrichten werden in den beiden je 4 KByte großen FIFO-Puffern eines DCM zwischengespeichert. Kommunikation findet auch zwischen nicht benachbarten Rechenknoten statt. Eine Unterbrechung der auf den Zwischenknoten ablaufenden Prozesse oder eine dortige Nachrichtenpufferung ist nicht nötig. Die gleichzeitige Kommunikation beschränkt sich nicht nur auf zwei Knoten. Es können zur gleichen Zeit eine Vielzahl von Kommunikationskanälen aufgebaut werden, in denen

paralleler Nachrichtenaustausch zwischen mehreren Prozessen möglich ist. Obwohl die Rechenknoten mittels des DCM in eine Hypercube-Topologie eingebunden sind und die Interprozessorkommunikation gegebenenfalls über einen oder mehrere Zwischenknoten stattfindet, erscheint dem Programmierer das Verbindungsnetzwerk als vollständig verbundener Graph, in dem jeder Rechenknoten mit jedem anderen direkt verbunden ist.

2.1.2 Die Hypercube-Topologie des iPSC/860

Die Hypercube-Topologie, ein gebräuchliches und effizientes Verbindungsnetzwerk in zur Zeit verfügbaren kommerziellen MIMD-Rechnern, bietet eine Reihe von Vorteilen. Andere, ähnlich aufgebaute und ebenfalls weit verbreitete Verbindungsstrukturen, wie etwa der Binärbaum, der Ring, das Gitter oder der Mesh of Tree, lassen sich mit geringfügigem Aufwand in einem Hypercube simulieren. Speziell auf die oben genannten Strukturen ausgerichtete und implementierte Algorithmen können mit einem nur geringen Verlust an Effizienz auf einen Hypercube-Rechner portiert werden. Der Hypercube mit $N = 2^d$ Rechenknoten (d bezeichnet die Dimension des Hypercube) ist ein in einem *statischen* Verbindungsnetzwerk *lose gekoppeltes* und *rekursiv* aufgebautes System (Abbildung 2.5).

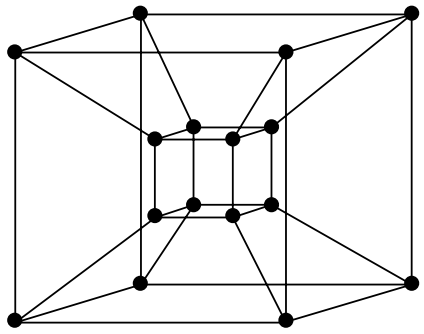


Abbildung 2.5: Vierdimensionale Hypercube-Struktur mit $2^4 = 16$ Knoten

Ein gegebenes Hypercube-Netzwerk kann also relativ leicht in Subcubes, das heißt in Hypercube-Netzwerke geringerer Dimension, aufgespalten werden. Umgekehrt läßt sich ein gegebenes Netzwerk nur durch aufwendige Verdopplung der Verbindungen aller Knoten erweitern. Die Konsequenz dessen ist, daß die Anzahl der Verbindungen eines Netzwerkes logarithmisch mit seiner Größe wächst. In der Struktur des d-dimensionalen Würfels ist jeder Knoten mit d, also in jeder Dimension mit einem, Nachbarknoten verbunden [GoMr89].
Das Verbindungsnetzwerk mit Hypercube-Topologie stellt somit eine ausgezeichnete Wahl für einen parallelen *Multipurpose*-Rechner dar [Lei92]. Eine detaillierte Darstellung der Eigenschaften der hier vorgestellten Hypercube-Topologie und artverwandter Strukturen sowie der diesen Systemen zugrundeliegenden Kommunikationskonzepte bietet [Röd91].

2.1.3 Benutzung des iPSC/860

Der iPSC/860 läßt sich nicht direkt ansprechen, stattdessen wird der Zugang zu den Rechenknoten oder einem *Cube* mit Hilfe eines *Front-End-Rechners*, auch *Host* genannt, realisiert. Mit einem Cube wird im folgenden ein Rechensystem von 2^k ($0 \leq k \leq 5$) Rechenknoten bezeichnet. Als Front-End-Rechner steht dem System ein PC i386 mit einer Taktfrequenz von 16 MHz und einer Speicherkapazität von 8 MByte bzw. einer Platte mit 380 MByte zur Verfügung. Der PC als Front-End-Rechner wird auch als **S**ystem **R**esource **M**anager (SRM) bezeichnet. Durch ein Ethernet-Interface und ein TCP/IP-Kommunikationsprotokoll ist der SRM mit externen Rechensystemen verbunden. Als Betriebssystem wird UNIX System V 3.2 verwendet.

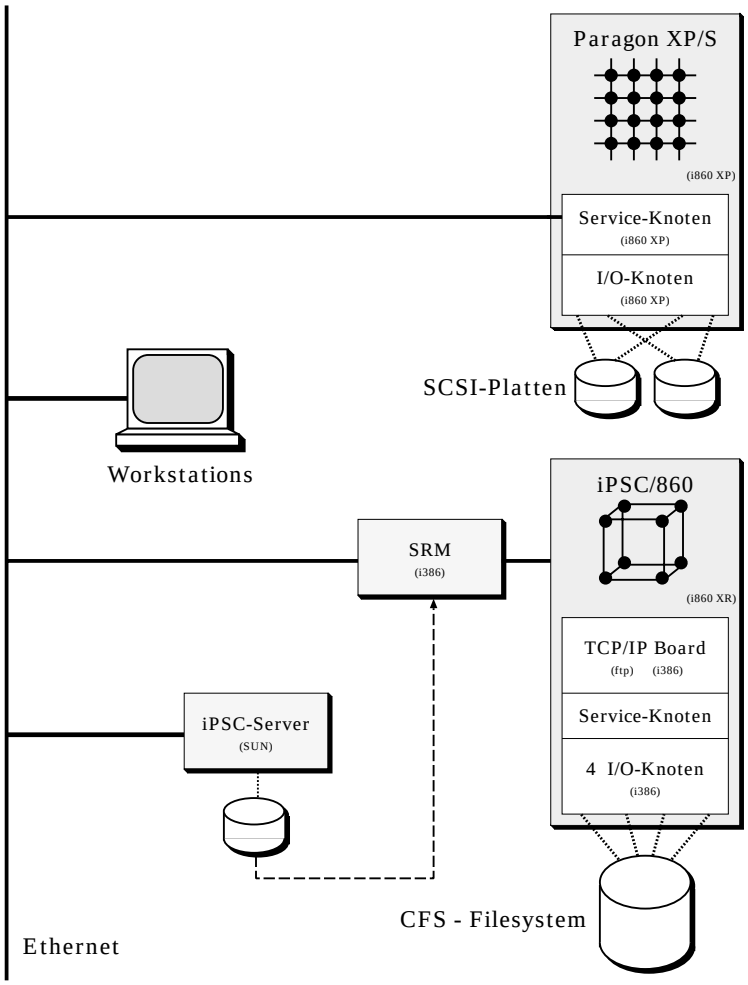


Abbildung 2.6: Konfiguration der massiv-parallelen Rechner im Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich

Neben dem **SRM** als Front-End-Rechner steht in der hier betrachteten Konfiguration eine SUN-Workstation als **iPSC-Server** für die Programmentwicklung bereit, die eine höhere Leistungsfähigkeit und größeren Speicherplatz aufweist. Die auch als *Remote Host* bezeichnete Workstation läuft unter dem Betriebssystem SUN OS. Neben den hier, sowie auf dem SRM, verfügbaren Cross Compilern für FORTRAN- und C-Programme ist die SUN-Workstation mit einer speziellen Remote-Host-Software ausgestattet. Es besteht eine ständige Verbindung zum iPSC (Abbildung 2.6). Das **C**oncurrent **F**ile **S**ystem (CFS) bietet dem iPSC/860 zusätzlichen externen Plattenplatz.

2.1.4 Programmierung des iPSC/860

Neben den Prozessen, die auf den Rechenknoten, also in einem Cube, abgearbeitet werden, den *Knotenprogrammen*, bietet das iPSC-System die Möglichkeit, auch auf dem Host-Rechner ein *Hostprogramm* zu starten (Prozeß und Prozessor werden im weiteren synonym verwendet). Ein Knotenprogramm wird im SPMD-Modell auf alle in einem Cube zur Verfügung stehenden Rechenknoten geladen. Jeder Knoten führt dann, im Gegensatz zum MIMD-Modell, wo jeder Knoten ein anderes Programm bearbeiten kann, das gleiche aus. Ein Hostprogramm kann sowohl auf einen Host als auch auf einen Rechenknoten gebracht werden. Im weiteren werden jedoch nur *Knotenprogramme* im SPMD-Modell betrachtet.
Programme, die auf dem iPSC/860 bearbeitet werden sollen, werden auf den Front-End-Rechnern entwickelt. Die FORTRAN- und C-Cross Compiler werden mit

if77 [*options*] *name.f* bzw. **icc** [*options*] *name.c*

aufgerufen. Im weiteren sollen nur FORTRAN-Programme betrachtet werden.

Bei dem verwendeten Compiler existieren fünf Optimierungsstufen, die mit der Option **-O** [*level*] gesetzt werden können [Int93a]. Jede Stufe enthält den Optimierungsgrad der vorherigen. In Stufe *0* wird für jede FORTRAN-Anweisung ein eigener Block generiert. Es findet keine globale Optimierung statt. Stufe *1* verbessert die Registerallokation. Der globale Optimierer der Stufe *2* verbessert im Vergleich zur Stufe *1* die skalare Optimierung. In Stufe *3* wird Software-Pipelining durchgeführt, das in Stufe *4* durch weiterführende Registerallokation verbessert wird. Zusätzlich wird hier der Programmcode von Schleifen, die mittels Pipelining ausgeführt werden, optimiert. Die vorab eingestellte Optimierungsstufe ist Stufe *1*. Weiterführende Untersuchungen über den verwendeten Compiler enthält [Mly93].
Für die Bestimmung einer Dateigröße steht der Befehl

size860 *file*

zur Verfügung. Bei dessen Ausführung wird der Speicherumfang von *file* in Bytes angegeben.

Bei der Ausführung eventuell auftretende Fehler können mit dem kommandogesteuerten **i**nteraktiven **p**arallelen **D**ebugger (IPD) auf dem SRM gefunden werden. Weiter kann der IPD behilflich sein bei der Datenreduktion, der Kontrolle von Prozeßkommunikation und der Analyse von Prozessen.
Performance **A**nalysis **T**ools (PAT) dienen der Analyse und damit der Optimierung der ausführbaren Programme. Es werden Performance-Daten zur Laufzeit gesammelt, und diese lassen sich nach dem Programmlauf graphisch oder tabellarisch darstellen.

- Das Execution Tool (**xtool**) vermerkt, an welcher Stelle im Programm wieviel Rechenzeit verbraucht wurde (vgl. hierzu auch Abbildung 6.7 in Abschnitt 6.3);
- die für die Kommunikation und die Ein-/Ausgabe relevanten Daten werden durch das Communikation Tool (**ctool**) aufgelistet (vgl. hierzu Abbildung 4.17 in Abschnitt 4.5 und Abbildung 5.7 in Abschnitt 5.3);
- wann und wo benutzerspezifizierte Ereignisse stattfanden, wird durch das Event Tool (**etool**) angegeben.

Über die Host-Rechner kann der Programmierer mit dem iPSC/860 kommunizieren. Die Administration eines Cube wird vom SRM oder vom Remote Host aus gesteuert. Die ausführbaren Programme werden auf den iPSC geladen und dann gestartet. Für diese Aufgaben stehen eine Reihe von Befehlen zur Verfügung [Int92a]. Soll mit einem oder mehreren Cubes gearbeitet werden, so werden eine Reihe von Informationen benötigt. Wieviele Rechenknoten mit welchen Nummern noch frei sind, der größte Cube, der angefordert werden kann und welche Cubes bereits von welchem Benutzer auf welchem Front-End-Rechner angefordert wurden, seien hier als Beispiele genannt. Diese Informationen erhält der Programmierer durch Ausführen des Befehls

cubeinfo.

Er erhält z.B. folgende Informationen:

CUBENAME	USER	SRM	HOST	TYPE	TTYS
iocube	root	srm-name	srm-name	0	null
defaultname	user	srm-name	srm-name	8m16rxn0	ttyp02

Der Benutzer **user** besitzt auf dem SRM **srm-name** einen Cube mit Namen **defaultname** und 8 Rechenknoten, dessen Knoten die **physikalischen** Nummern 0 bis 7 haben und 16 MByte Speicherplatz besitzen. Es sind demnach noch 24 Knoten frei. Als größter kann ein vierdimensionaler Cube, beginnend bei der Knotennummer acht, angefordert werden.

Ein optional mit *name* benannter Cube wird mit

getcube [**-c** *name*] [**-t** *num*]

angefordert, wobei die Option [**-t** *num*] die Anzahl der Rechenknoten angibt, das heißt die Größe des Cube.

Ein ausführbares Programm *filename* wird mit

load [**-c** *name*] *filename*

auf den Cube *name* geladen.

Werden mehrere Prozesse auf den verschiedenen Knoten abgearbeitet und sind diese zu unterschiedlichen Zeitpunkten beendet, so warten alle bereits fertigen Knoten bei Eingabe von

waitcube [**-c** *name*]

auf das Ende der noch arbeitenden Knoten, bevor neue Prozesse gestartet werden können. Mit

killcube [**-c** *name*]

können Prozesse auf einem Cube vorzeitig abgebrochen und noch im Netzwerk befindliche Nachrichten gelöscht werden. Bei Beendigung des interaktiven Betriebs müssen die Cubes mit

relcube [**-c** *name*] [**-a**]

explizit wieder freigegeben werden.

Neben der interaktiven Nutzung ist der iPSC/860 auch für einen Batch-Betrieb ausgelegt. Mit dem **N**etwork **Q**ueueing **S**ystem (NQS) können Jobs aus unterschiedlichen Queues ausgeführt werden. Die verschiedenen Queues zeichnen sich durch eine unterschiedliche Anzahl von Rechenknoten und durch bestimmte Time Limits aus. Auszuführende Jobs werden mit dem Befehl

qsub [**-q** *queuename*] [*jobname*]

submittiert und mit

qdel -k *requestname*

abgebrochen. Der Batch-Job *jobname* ist in einem Shell-Skript genauer zu spezifizieren. In dem Skript können auch die benötigten Time Limits und Ein- bzw. Ausgabedateien explizit angegeben werden.
Die für die Programmausführung notwendigen Dateien können sowohl interaktiv über ein Terminal oder jedes andere über das **N**etwork **F**ile **S**ystem (NFS) erreichbare File-System bereitgestellt werden als auch parallel über das **C**oncurrent **F**ile **S**ystem (CFS) eingelesen werden (Abbildung 2.8). Das CFS bietet zusätzlich die Möglichkeit des parallelen Schreibens auf externe Dateien (siehe zum CFS auch Unterabschnitt 2.1.6).

2.1.5 Paralleles Programmiermodell des iPSC/860

Das dem parallelen Programmiermodell zugrundeliegende Konzept zeichnet sich durch die Existenz einer Menge von Prozessen, die völlig unabhängig voneinander und asynchron auf den einzelnen Rechenknoten ausgeführt werden, aus. Die Prozesse können nur auf ihre eigenen Daten im lokalen Speicher des Rechenknotens zugreifen. Werden von einem Prozeß Informationen eines anderen benötigt, so müssen diese über das Verbindungsnetzwerk von Knoten zu Knoten verschickt werden. Das diesem Nachrichtenaustausch zugrundeliegende Programmiermodell ist das **Message Passing**, bei dem Kopien von Speicherinhalten eines Sendeprozesses zu einem Empfangsprozeß verschickt werden. Die Nachrichten leitet das Betriebssystem gegebenenfalls über Zwischenknoten weiter (siehe Abbildung 2.7). Dabei erscheinen dem Programmierer Sende- und Empfangsprozeß jedoch direkt verbunden.

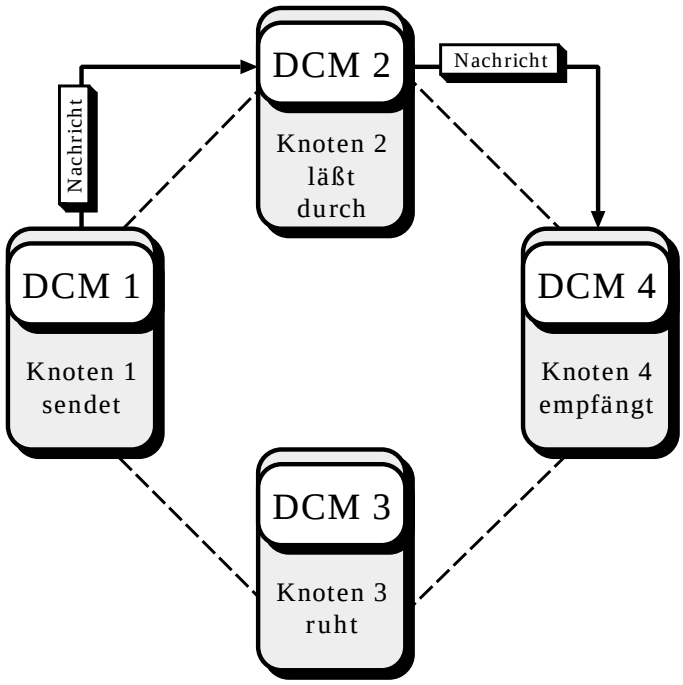


Abbildung 2.7: Konzept des Message Passing

Jeder Prozeß kann mit jedem anderen kommunizieren und mehrere Datentransfers werden ohne gegenseitige Beeinflussung gleichzeitig durchgeführt. Man unterscheidet die Datenübermittlung je nach Anzahl der Empfänger in *Unicast*, *Broadcast* oder *Multicast*. Bei einem Unicast sendet ein Knoten eine Nachricht an *genau einen* anderen Knoten. Eine als Broadcast verschickte Nachricht ist eine an *alle* Knoten eines Cube adressierte Sendung. Im allgemeinen Fall spricht man von einem *Multicast*. Die Anzahl der Empfänger bei einem Multicast liegt zwischen einem und allen Knoten eines Cube.

Das hier beschriebene Message-Passing-Konzept kann in den zwei nachfolgenden Modi betrieben werden:

- Im Rahmen des **synchronen** oder auch **blockierenden** Sendens einer Nachricht wird der auf dem Sendeknoten zu bearbeitende Prozeß beim Aufruf des Befehls **csend** solange blockiert, bis die Nachricht vollständig an das Verbindungsnetzwerk übergeben wurde. Es besteht keine Sicherheit darüber, ob die Nachricht den Empfänger auch erreicht hat. Nach Übergabe der zu versendenden Nachricht an das Netzwerk wird der unterbrochene Prozeß wieder aufgenommen. Der Empfängerknoten unterbricht beim Aufruf von **crecv** ebenfalls den auf ihm ablaufenden Prozeß. Erst wenn die Nachricht vollständig in den dafür vorgesehenen Empfangspuffer eingetroffen ist, setzt der blockierte Prozeß wieder ein. Die Kommunikationsroutine für das blockierende Senden einer Nachricht hat die Form

  call csend (*type, buf, len, node, pid*)

  und die für blockierenden Empfang

  call crecv (*type, buf, len*).

  Neben der Nachrichtenkennung *type* und dem Speicherplatz *buf* wird die Länge *len* der Nachricht in Byte übergeben. *node* und *pid* geben die **logische** Knoten- bzw. Prozeßnummer des Zielprozesses an ($node = -1$ bedeutet Broadcast) [Int92b].
  Beim iPSC/860 ist die Prozeßkennung *pid* des Zielknotens immer gleich Null. Auf jedem Knoten eines Cube ist nur ein Prozeß möglich. Daß trotzdem der Parameter *pid* bei jeder Nachrichtenübertragung mit übergeben wird, hat historische Gründe. Das Betriebssystem NX/2 auf dem iPSC/860 läßt im Gegensatz zum iPSC/2 nur einen Prozeß pro Knoten zu. So bleibt mit Übergabe von *pid* die Kompatibilität von Programmen für die Rechner der iPSC-Serie erhalten.

- Im Gegensatz zur synchronen Nachrichtenübertragung werden beim **asynchronen** oder **nicht blockierenden** Message Passing die auf den sendenden bzw. empfangenden Rechenknoten ablaufenden Prozesse nicht unterbrochen. Stattdessen wird mit der nächsten Anweisung des Programms fortgefahren, obwohl eine Nachricht an das Netzwerk übergeben wird oder noch nicht eingetroffen ist. Aber eine Nachrichtenkennung, *msgid*, wird zurückgegeben, mit der das Ende der Übertragung festgestellt werden kann.

Im asynchronen Modus lauten die Kommunikationsroutinen

msgid = **isend** (*type, buf, len, node, pid*)

bzw.

msgid = **irecv** (*type, buf, len*).

Die Parameter besitzen die gleiche Bedeutung wie oben. Eine asynchrone Sende- oder Empfangsoperation kann mit

call msgcancel (*msgid*)

abgebrochen werden. Nach Aufruf dieser System-Routine ist die asynchrone Operation inaktiv, *msgid* wird freigegeben und der Nachrichtenpuffer gelöscht. Mit der System-Funktion

msgdone (*msgid*)

kann festgestellt werden, ob die mit *msgid* gekennzeichnete Nachrichtenübertragung stattgefunden hat. **msgdone** gibt den Wert 0 bei andauernder Übertragung zurück und die Rückgabe einer 1 indiziert die erfolgreiche Beendigung der Operation. Der Nachrichtenpuffer beinhaltet eine gültige Nachricht, die Nachricht verbleibt im Puffer. *msgid* wird für nachfolgende Sendungen wieder freigegeben. Eine asynchrone Operation wird durch die System-Routine **msgwait** synchronisiert. Der Aufruf

call msgwait (*msgid*)

veranlaßt den aufrufenden Knotenprozeß, auf das Ende der durch *msgid* gekennzeichneten **isend**- oder **irecv**-Operation zu warten. Die Nachricht bleibt im Systempuffer liegen. *msgid* wird wieder freigegeben.
Da die Anzahl der beim iPSC/860 verfügbaren Nachrichtenkennungen begrenzt ist, empfiehlt sich zur Vermeidung von Programmfehlern die rasche Freigabe von *msgid* durch die oben aufgelisteten Kontrollroutinen.

Zur Realisierung einer fehlerfreien Kommunikation werden von einem Knoten mehr Informationen über sich selber oder den Cube benötigt. Die Selbstidentifikation wird mit der System-Funktion

mynode ()

gewährleistet. Sie liefert dem aufrufenden Prozeß seine eigene **logische** Knotennummer.

Die Anzahl der in seinem Cube spezifizierten Rechenknoten erhält ein Prozeß durch Ausführen des Befehls

numnodes (),

und

nodedim ()

gibt ihm die Dimension des aktuellen Cube an, zu dem er gehört. Durch die System-Funktion

gsync ()

wird eine globale Synchronisation aller zu einem Cube gehörenden Rechenknoten ausgeführt. Alle Knoten, die **gsync()** aufgerufen haben, besitzen dann den gleichen Systemstatus und fahren zur gleichen Zeit an derselben Stelle des abzuarbeitenden Programmes fort. Die Zeitmessung wird mit Hilfe der Funktion

dclock ()

realisiert, die dem aufrufenden Prozessor die Zeit in Sekunden angibt, die vergangen ist, seitdem er das letzte mal neu gestartet wurde. Neben den hier aufgeführten Funktionen und Routinen existiert eine Vielzahl von globalen Operationen, wie etwa die globale Summe oder die Berechnung des globalen Minimums, die Daten von allen Rechenknoten benötigen. Die Kommunikation der Knoten untereinander wird bei Aufruf eines dieser Unterprogramme selbständig realisiert.

2.1.6 Das Concurrent File System

Neben den File-Systemen der Host-Rechner steht dem iPSC/860 mit dem **C**oncurrent **F**ile **S**ystem (CFS) ein externer Plattenspeicher zur Verfügung. Das CFS erlaubt den Knotenprogrammen, sehr schnell auf eine große Menge von Speicherplatten zuzugreifen. Die Kapazität des für gleichzeitigen Zugriff durch mehrere Knoten optimierten Systems beträgt bei dem im Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich installierten iPSC/860 4.6 GByte, die auf 7 Platten mit jeweils 670 MByte verteilt sind. Jede Platte besitzt eine eindeutige *Volume*-Nummer. Dem Programmierer erscheint das File-System als ein einheitliches System auf einer Platte, obwohl die Dateien in Blöcken von jeweils 4 KByte nach dem *Round-Robin*-Verfahren in Reihenfolge der Volume-Nummer auf die Platten verteilt sind. So wird zum Beispiel eine 32 KByte große Datei in acht 4 KByte große Blöcke unterteilt. Der erste Block wird auf die Platte mit Volume-Nummer 0 gespeichert, der zweite auf Nummer 1 usw. Der letzte Block wird wieder auf Platte Nummer 0 abgelegt. Dateien, die im CFS abgespeichert werden sollen, sind mit dem Präfix */cfs* zu kennzeichnen. Anders benannte Dateien werden im File-System der Host-Rechner abgelegt.

Der Datenaustausch zwischen Rechenknoten und CFS wird, wie in Abbildung 2.8 gezeigt, in dem im ZAM installierten System über vier zusätzliche unabhängige I/O-Knoten und jeweils einen SCSI-Bus (SCSI = **S**mall **C**omputer **S**ystem **I**nterface) realisiert. Jeder der nur für Kommunikationsaufgaben reservierten I/O-Knoten basiert auf der i386-Mikroprozessortechnologie und besitzt eine Speicherkapazität von 4 MByte. Die Bandbreite der Übertragung zwischen Rechen- und I/O-Knoten beträgt 2.8 MByte/s und beim SCSI-Bus 4 MByte/s. Die Speicherplatten besitzen eine Transferrate von 1 MByte/s.

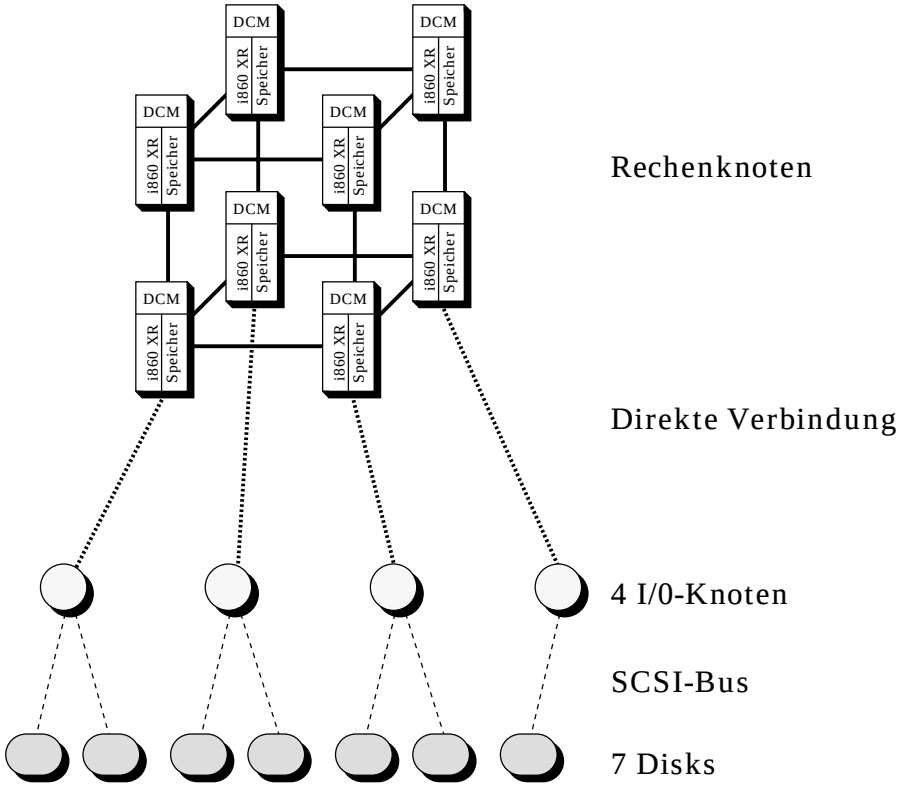


Abbildung 2.8: Concurrent File System

Der Zugriff auf im CFS abgelegte Dateien erfolgt von einem iPSC/860-FORTRAN-Knotenprogramm aus durch Angabe von */cfs/userid/name* im OPEN-Statement. Eingeschränkt wird der Gebrauch des CFS dadurch, daß FORTRAN-Dateien unformatiert geöffnet werden müssen. Mit dem **F**ile **T**ransfer **P**rotocol (FTP) können Daten über das TCP/IP-Interface auf einen Host-Rechner übertragen werden.
Über eine **node shell** erhält man interaktiven Zugang zum CFS. Die zur **C shell (csh)** des UNIX System V kompatible **node shell** wird mit dem Befehl

nsh [**-s**] [*csh_arguments*]

aufgerufen. Die Option **nsh -s** bedeutet, daß die *node shell* auf dem aktuellen Cube, ansonsten auf einem I/O-Knoten, aufgerufen wird.

Ähnlich wie beim Message Passing kann auf das CFS sowohl synchron als auch asynchron zugegriffen werden:

- Die synchrone Leseanweisung lautet

  call cread (*fileID, buf, len*)

  und die Schreibanweisung

  call cwrite (*fileID, buf, len*).

  Der Parameter *fileID* bezeichnet die Unit-Nummer der Datei, *buf* den Datenpuffer und *len* dessen Größe. Die Ausführung des laufenden Programms wird solange unterbrochen, bis die synchrone Lese- oder Schreiboperation beendet ist.

- Die entsprechenden asynchronen Befehle lauten

  id = **iread** (*fileID, buf, len*)

  bzw.

  id = **iwrite** (*fileID, buf, len*).

  Die asynchronen Befehle bewirken das Fortsetzen der auf den Rechenknoten laufenden Programme, ohne auf das Ende der Datenübertragung zu warten. Diese Routinen geben wie die Message-Passing-Routinen eine I/O-Kennung zurück. Mit Hilfe der Kontrollfunktion

  iodone (*id*)

  kann festgestellt werden, ob eine Ein-/Ausgabe-Operation beendet wurde. Die Rückgabe einer 1 zeigt das Ende der Operation an. Ansonsten wird eine 0 zurückgegeben. Wie beim asynchronen Message Passing kann auch hier eine asynchrone Operation synchronisiert werden. Dazu ist die System-Routine

  call iowait (*id*)

  aufzurufen. Der aufrufende Prozeß wartet mit der Fortsetzung des laufenden Programmes bis zum Ende der Ein-/Ausgabe. Die Anzahl der I/O-Kennungen ist auch hier begrenzt, so daß deren baldmögliche Freigabe zwingend erforderlich ist.

Aus Performance-Gründen ist es sinnvoll, die Größe einer Datei bei deren Bekanntheit zu Beginn eines CFS-Zugriffs festzulegen. Dafür steht dem Programmierer der Befehl

size = **lsize** (*fileID, offset, whence*)

zur Verfügung. *fileID* bezeichnet die I/O-Unit und *whence* gibt an, wie die durch *offset* angegebene Anzahl von Bytes die Dateigröße beeinflußt. In *size* steht nach

der Ausführung des Befehls die neue tatsächliche Größe der Datei. Der Befehl

loc = **lseek** (*fileID, offset, whence*)

erlaubt das Setzen von File-Pointern. *fileID* bezeichnet wieder die File-Unit, *offset* die relative File-Pointer-Position und *whence* gibt an, wohin der File-Pointer positioniert werden soll. In *loc* wird die neue Pointer-Position vom Dateianfang in Byte geschrieben.
Mit den oben aufgeführten Routinen ist paralleles synchrones oder paralleles asynchrones Arbeiten mit dem CFS möglich. Der Zugriff kann in einem von vier Modi geschehen, die wie folgt festgelegt werden:

call setiomode (*fileID, mode*)

mit:

mode = 0
Jeder Knoten besitzt seinen eigenen File-Pointer. Schreiben zwei Knoten an derselben Stelle einer Datei, so überschreibt der zuletzt schreibende Knoten die Daten des zuerst schreibenden. Der Benutzer muß, um die korrekte Reihenfolge der Daten zu gewährleisten, mit dem Kommando lseek den File-Pointer der einzelnen Knoten an die richtige Stelle positionieren. Da bei mode 0 keine Kommunikation zwischen den Rechenknoten stattfindet und sie bei Bedarf direkt auf ihre Stelle in der Datei zugreifen können, ist mode 0 der Modus mit der höchsten Geschwindigkeit. Verwendung findet dieser als default eingestellte Modus oft dann, wenn die Rechenknoten nur aus einer Datei lesen oder für ihren eigenen spezifischen Speicherraum verantwortlich sind.

mode = 1
Alle Knoten besitzen einen gemeinsamen File-Pointer. Der Daten-Zugriff erfolgt unsynchronisiert nach dem first-come-first-served-Prinzip. Weil Dateizugriffe der Knoten in beliebiger Reihenfolge ausgeführt werden, kann der Inhalt einer Datei bei mehrmaligen Programmläufen variieren. mode 1 ist deshalb sinnvoll bei gemeinsamen *log*-Dateien einzusetzen. Da zu einem Zeitpunkt jeweils nur ein Knoten auf die Datei zugreifen kann, ist mode 1 langsamer als mode 0.

mode = 2
Der Zugriff auf die Datei erfolgt synchronisiert nach dem Round-Robin-Verfahren. Wie in mode 1 besitzen alle in einem Cube befindlichen Knoten einen gemeinsamen File-Pointer. Sie müssen die Datei öffnen und die gleichen Operationen in der gleichen Reihenfolge ausführen, um nochmals zugreifen zu können. Lesen und Schreiben in eine Datei erfolgt in Reihenfolge der Knotennummern, die Größe der Daten ist variabel. Wird das oben beschriebene lseek ausgeführt, so können alle Knoten nur an der gleichen Position in der Datei ihre Pointer setzen, ansonsten erfolgt eine Fehlermeldung. Alle Knoten müssen auf die Datei gemäß ihrer Knotennummer zugreifen; mode 2 besitzt deshalb die langsamste Geschwindigkeit aller Modi.

mode = 3

mode 3 liefert ähnliche Ergebnisse wie mode 2, arbeitet aber effizienter. Alle Knoten besitzen einen gemeinsamen File-Pointer wie in mode 2, müssen hier jedoch Daten gleicher Länge schreiben. Dies erlaubt eine parallele Ausgabe, denn der File-Pointer jedes Knotens kann im voraus bestimmt werden. Die Operationen der Rechenknoten in einem Cube müssen die gleichen sein und in der gleichen Reihenfolge ausgeführt werden (siehe mode 2). Gemeinsames Lesen oder Schreiben ist nur bei Daten gleicher Größe möglich und wird in der Reihenfolge der Knotennummern durchgeführt. Das Schließen einer Datei und die Behandlung des Befehls lseek ist wie in mode 2. Da die in mode 3 zu bearbeitenden Daten im Gegensatz zu mode 2 von gleicher Größe sein müssen und die Rechenknoten damit auf die Datei bei Bedarf zugreifen können, ist mode 3 schneller als mode 2.

2.2 Der Rechner Paragon XP/S 5

Mit dem Paragon-Rechner setzt Intel die Serie der auf lokalen Speichern arbeitenden MIMD-Rechner fort. Auf der Grundlage des iPSC/860 wurde der Paragon mit einem leicht verbesserten i860 XP-Prozessor und neuer Kommunikationseinheit ausgestattet. Das Verbindungsnetzwerk des Paragon basiert auf einer Gitter-Topologie. Die einzelnen Rechenknoten kommunizieren wie beim iPSC/860 über Message Passing miteinander. In Zukunft soll das Konzept des *Shared Virtual Memory* auf dem Paragon eingesetzt werden [Int91].

2.2.1 Systemarchitektur

Im Dezember 1992 wurde im Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich ein Paragon XP/S 5 mit 72 Rechenknoten installiert. In Abbildung 2.9 ist ein Paragon-Rechenknoten dargestellt. Zwei Prozessoren vom Typ i860 XP, im folgenden *Kommunikations-* und *Applikationsprozessor* genannt, bilden zusammen mit dem lokalen Speicher und einem **N**etwork **I**nterface **C**ontroller (NIC) einen Rechenknoten. Über eine zweidimensionale Gitter-Struktur sind die Rechenknoten miteinander verbunden.

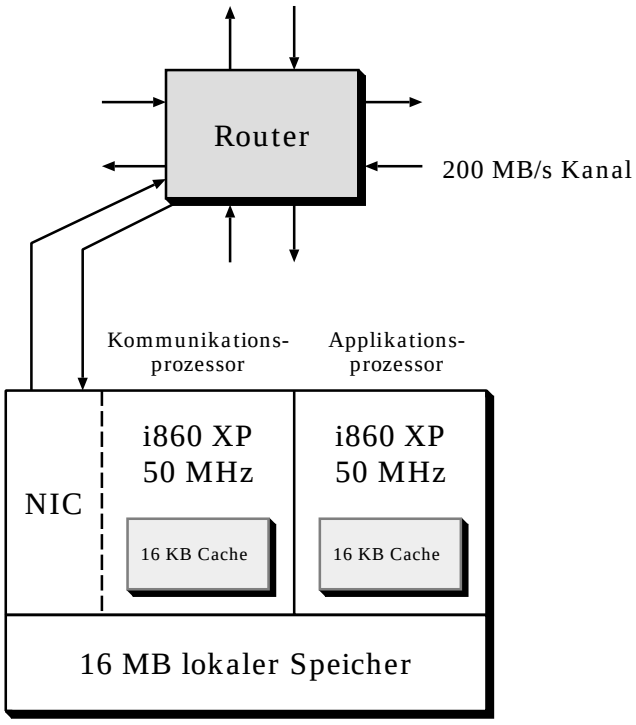


Abbildung 2.9: Darstellung eines Paragon-Rechenknotens

Der Mikroprozessor i860 XP [Int92] ist eine Weiterentwicklung des im iPSC/860 eingesetzten Prozessors i860 XR und besitzt im Vergleich dazu eine um 25% höhere Rechenleistung. Die RISC-Architektur des i860 XR wurde beibehalten, die Taktfrequenz des i860 XP aber auf 50 MHz erhöht, was einer Taktzeit von 20 ns entspricht. Daraus resultiert eine theoretische maximale Rechenleistung von 75 MFLOPS bei 64-Bit-Fließkommaoperationen. Die unabhängige INTEGER-Einheit führt 42 Millionen Operationen in der Sekunde aus. Mit den 72 Prozessoren beträgt die theoretische maximale Rechenleistung des Paragon XP/S 5 5.4 GFLOPS.
Auf den Paragon-Rechenknoten wurde im Gegensatz zum iPSC/860, dessen Knoten mit dem NX/2-Betriebssystem arbeiten, Paragon OSF/1 installiert, welches auf OSF/1 AD basiert, der Multiprozessorversion des OSF/1-Betriebssystems der Open Software Foundation [OSF92]. Wie OSF/1 basiert Paragon OSF/1 auf dem an der Carnegie-Mellon University entwickelten Mach 3 *microkernel*. Da der Paragon jedoch ein Rechner mit verteiltem Speicher ist, wird die NORMA-Version (**NO** **R**emote **M**emory **A**ccess) benutzt. Darauf aufgesetzt sind eine *emulation library* und ein OSF/1-*server*, der auf jedem Rechenknoten OSF/1-Dienste wie Prozeß-Management, File-Systeme oder den Zugang zum Netzwerk unterstützt. Das zum Paragon-Betriebssystem gehörende NX-Interface besitzt alle Befehle des NX/2-Message-Passing-Interface und realisiert die Kommunikation der Rechenknoten untereinander.
Paragon OSF/1 unterstützt das *Shared-Virtual-Memory*-Konzept, das heißt einen virtuellen gemeinsamen Speicher auf allen Knoten des Systems. Auf einem Rechenknoten des Paragon können, wie auf dem iPSC/2, mehrere Prozesse gestartet werden.

Daten- sowie Instruktions-Cache eines jeden Knotens umfassen 16 KByte, wurden somit im Vergleich zum iPSC/860 vergrößert, wo sie 8 KByte bzw. 4 KByte betrugen (vgl. Abbildung 2.3). Die Bandbreite des internen Datenbusses von der Instruktionseinheit zum Cache wurde um 160 MByte/s auf 800 MByte/s erhöht, die des externen Datenbusses vom Cache zum lokalen Speicher um 240 MByte/s auf 400 MByte/s.
Der lokale Speicher eines Rechenknotens basiert auf der DRAM-Technologie und besitzt im ZAM eine Kapazität von 16 MByte. Die Zugriffszeit auf die dort gespeicherten Daten beträgt 60 ns. Jedes DRAM ist mit einer Einheit für die Erkennung von Fehlern und deren Korrektur ausgestattet.

2.2.2 Die Gitter-Struktur des Paragon

Die Kommunikation von Rechenknoten zu Rechenknoten oder zu externen Adressaten wird über das NIC geführt. Der zweite i860 XP-Prozessor, der Kommunikationsprozessor, ist Teil des NIC und übernimmt die Aufgaben der Kommunikation, die beim iPSC/860 vom DCM übernommen wurden. Er paketiert und protokolliert die Nachrichtenübertragung und hält damit den Applikationsprozessor frei von der zeitaufwendigen Kommunikation. Dadurch werden eventuell auftretende Cache-Turbulenzen in beiden Prozessoren reduziert und die beim iPSC/860 noch relativ hohe Startup-Zeit verringert (sie beträgt beim Paragon unabhängig von der Größe des Systems 30 μs). Ankommende Nachrichten werden vom Kommunikationsprozessor empfangen und zum weiteren Gebrauch für den Applikationsprozessor aufbereitet. Jeder Kommunikationsprozessor kann in alle vier Netzwerkrichtungen gleichzeitig Nachrichten verschicken. In 40 ns entscheidet er, in welche Richtung eine angekommene Nachricht weitergeleitet werden muß, um ein optimales Routing zu garantieren. Eine Nachricht wird immer zuerst in x-Richtung weitergeleitet, dann in y-Richtung. Damit ist der verwendete Routing-Algorithmus deadlock-frei. Die Kommunikation wird durch **Wormhole Routing** [DaSe86] realisiert, bei dem eine Nachricht in eine Sequenz von *flits* (flow control digits) unterteilt wird, die dann von Router zu Router weitergeleitet werden [NiMc93].

Im Gegensatz zum iPSC/860 werden beim Paragon die Kommunikationskanäle erst dann aufgebaut, wenn sie explizit durch eine im Programmcode stehende Sendeanweisung benötigt werden. Die Rechenknoten halten die Größe der Nachrichtenspeicher ihrer Kommunikationspartner dynamisch nach. Beim Aufbau eines Kommunikationskanals erfragt der Sender die freie Speicherkapazität des Nachrichtenpuffers des Empfängers. Der Nachrichtenpuffer eines Rechenknotens wird auf alle zu einer Partition gehörenden Rechenknoten aufgeteilt. Für jeden wird die gleiche Größe reserviert. Diese wird dem Sender mittels einer zweiten Nachricht zurückgeliefert. Erst wenn die benötigten Informationen beim Sender liegen, findet die eigentliche Nachrichtenübertragung statt. Der Sender verwaltet nun für sich im weiteren Verlauf einer Applikation die Kapazität der Nachrichtenpuffer der Empfänger, indem er bei jeder Nachrichtenübermittlung deren Größe von der freien Pufferkapazität abzieht. Diese erste Nachrichtenübertragung ist ähnlich wie das long message protocol des iPSC/860 eine dreistufige Kommunikation. Die Startup-Zeit einer Kommunikation liegt zur Zeit (Betriebssystem Release 1.0) bei 135 μs, soll aber in Zukunft 30 μs betragen. (Beim iPSC/860 betrugen die Startup-Zeiten 69 μs für das short message protocol und 195 μs für das long message protocol.)

Interner paralleler Nachrichtenaustausch findet beim Paragon mit einer Übertragungsrate von 200 MByte/s statt (beim iPSC/860 beträgt sie 2.8 MByte/s). Der Kommunikationsprozessor kann diese hohe Übertragungsgeschwindigkeit für lange Nachrichten jedoch nicht aufrecht erhalten. Deshalb wurden in jedem Rechenknoten zwei sogenannte *Block-Transfermaschinen* integriert. Diese sind Bestandteil des NIC und können gleichzeitig 4096 Byte verschicken. Dadurch unterstützen sie die Interprozessorkommunikation.

Durch die Auslagerung der Kommunikation auf einen eigenständigen i860 XP-Prozessor und das automatische Routing einer Nachricht erscheint dem Benutzer das Verbindungsnetzwerk des Paragon wie das des iPSC/860 als ein System, in dem jeder Knoten mit jedem anderen vollständig verbunden ist. Aufgebaut ist das zweidimensionale Gitter des Paragon durch **M**esh **R**outing **C**hips (iMRC), die jeweils fünf Eingangs- und fünf Ausgangskanäle besitzen. Je ein Ein- und ein Ausgangskanal stellen die Verbindung mit einem der vier Nachbarknoten sicher. Das fünfte Paar ist verbunden mit dem NIC des eigenen Rechenknotens (Abbildung 2.12.
Für Systeme mit bis zu 256 Rechenknoten beträgt die Bandbreite der Kommunikation 6 GByte/s. Bei 1000 Knoten erreicht das Paragon-System Bandbreiten von bis zu 12 GByte/s.

Die Gitter-Struktur zählt, wie die Hypercube-Topologie des iPSC/860, zu den *lose gekoppelten, statischen* Netzwerken. In dieser einfachen homogenen Netzwerktopologie werden $N = m \cdot n$ Rechenknoten, wie in Abbildung 2.10 gezeigt, miteinander verbunden. Beim Paragon XP/S 5 sind dies $N = 6 \cdot 12 = 72$ Knoten.

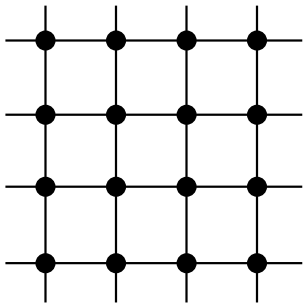


Abbildung 2.10: Zweidimensionale Gitter-Struktur mit $4 \times 4 = 16$ Knoten

Die maximale Entfernung zweier Knoten ist gleich $\max(n, m)$ und der mittlere Abstand gleich $\frac{1}{2} \cdot \max(n, m)$. Aufgrund der Gitter-Struktur sind die Routingverfahren zur Nachrichtenübertragung einfach, da alle zu einer Übertragung notwendigen Wege, die die Differenz zwischen Quell- und Zielkoordinate um eins verringern, offenstehen [GoMr89].
Die Gitter-Struktur besitzt den Vorteil, daß andere Topologien, wie z.B. Bäume oder Hypercubes, simuliert und für diese Strukturen entwickelte Applikationen leicht auf einen Paragon portiert werden können.

2.2.3 Benutzung des Paragon

Im Gegensatz zum iPSC/860 wird der Paragon nicht über einen Front-End-Rechner angesprochen, sondern der Benutzerzugang erfolgt direkt auf das System. Gemeinsam ist ihnen jedoch, daß Applikationen sowohl interaktiv als auch mittels Batch-Betrieb abgearbeitet werden können.
Das Knotengitter des Paragon gliedert sich, wie in Abbildung 2.11 gezeigt, in eine **Berechnungspartition** und eine **Servicepartition**.

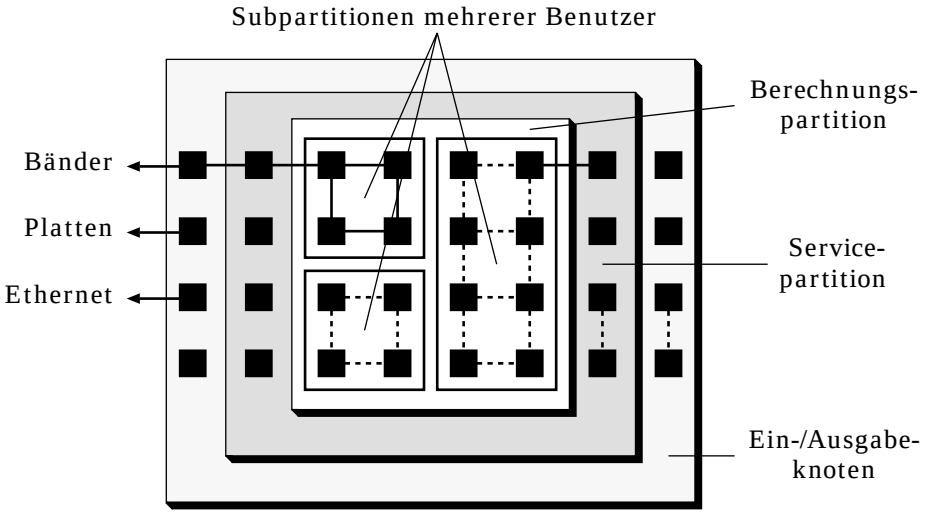


Abbildung 2.11: Partitioniertes Paragon-System

In die Berechnungspartition gehen die meisten Rechenknoten eines Paragon-Systems ein. Hier werden die vom Benutzer spezifizierten parallelen Anwendungen ausgeführt. Die Berechnungspartition kann wiederum in hierarchisch aufgebaute Subpartitionen unterteilt werden, in denen verschiedene Applikationen gleichzeitig sowohl im interaktiven als auch im Batch-Betrieb ausgeführt werden können.
Über die Knoten der Servicepartition erfolgt der Zugang der Benutzer auf das Paragon-System. Diese Partition stellt den Benutzern generelle Arbeitsmittel wie Compiler, Shells oder Editoren zur Verfügung. Die Anzahl der Serviceknoten kann vom System-Administrator dynamisch den Erfordernissen der Benutzer und des Betriebssystems angepaßt werden. Die Kapazität der lokalen Speicher der Berechnungs- und Servicepartition läßt sich von 16 bis 64 MByte skalieren. Neben diesen Rechen- und Serviceknoten gibt es noch Ein-/Ausgabeknoten. Diese sind spezielle Knoten, die in einer Ein-/Ausgabepartition zusammengefaßt werden können. Sie können gleichzeitig aber auch Serviceknoten sein. Ein-/Ausgabeknoten ermöglichen dem Paragon Zugang zu externen File- und anderen Rechensystemen. Als mögliche File-Systeme kommen Bänder und Platten in Frage. Ethernet, HIPPI oder andere Netzwerke verbinden den Paragon mit weiteren Rechen- oder Ein-/Ausgabesystemen. Für weitere Details über das Paragon XP/S-System sei auf [EsKn93] verwiesen.

2.2.4 Programmierung des Paragon

Wie oben bereits erwähnt, läßt sich mit der Gitter-Struktur des Paragon ein Hypercube, wie der des iPSC/860, simulieren. Ebenso können auf dem iPSC/860 entwickelte Applikationen auf den Paragon portiert werden, so daß der iPSC/860 auch als Entwicklungsumgebung für diesen benutzt werden kann.

Dem Benutzer steht die Möglichkeit des direkten Zugangs zu einem Paragon-Subsystem offen. Die Entwicklung einer Anwendung wird in der Servicepartition des Paragon vorgenommen, im Gegensatz zum iPSC/860, wo die Programmentwicklung auf den Front-End-Rechnern stattfindet. Ein Benutzer kann auch auf Workstations mittels Cross-Entwicklungs-Tools Programme für den Paragon entwickeln und diese dann darauf zur Anwendung bringen.
Neben den Programmiersprachen FORTRAN 77, C und Ada für SPMD- und MIMD-Applikationen steht dem Benutzer in Zukunft auch das Konzept des *Data Parallel Programming* für SPMD-Programme zur Verfügung.

Programme, die auf dem Paragon bearbeitet werden sollen, werden im wesentlichen mit den gleichen Compileraufrufen wie Programme für den iPSC/860 übersetzt. Zu beachten ist jedoch, daß für parallele Programme die Option **-nx** mit angegeben wird. **-nx** bewirkt, daß das fertig übersetzte Programm in der Berechnungspartition ausgeführt wird. Bei Weglassen dieser Option bearbeitet ein Knoten der Servicepartition die Applikation. Der Aufruf der Compiler hat folgende Gestalt:

if77 -nx [*options*] *name.f* bzw. **icc -nx** [*options*] *name.c*.

Wie auf dem iPSC/860 ist auch auf dem Paragon der **i**nteraktive **p**arallele **D**ebugger (IPD) zur Fehlerfindung und Programmanalyse installiert. Mit Hilfe des IPD können Trace-Files eines Programmlaufes erzeugt werden, die dann anschließend mit dem Profiling Tool **prof860** analysiert werden können.

Ähnlich wie beim iPSC/860 mit dem **cubeinfo**-Kommando kann der Benutzer auch auf dem Paragon Informationen über Partitionen anfordern. Der Befehl

lspart [**-r**] [*partition*]

gibt eine Liste aller Partitionen und deren Subpartitionen an. So bewirkt zum Beispiel der Aufruf von **lspart** die Ausgabe von

USER	GROUP	ACCESS	SIZE	RQ	EPL	PARTITION
me	users	777	16	15m	3	mypart
other	extern	755	21	10m	10	otherpart

Die Charakteristiken einer Partition *partition* werden dem Benutzer durch Angabe des Kommandos

showpart [*partition*]

bildlich dargestellt.

showpart *mypart* erzeugt zum Beispiel die Bildschirmausgabe

```
USER    GROUP    ACCESS    SIZE    RQ    EPL    PARTITION
me      users    777       16      15m   3      mypart
    +-----------------+
   0| . . . . . . . . |
   8| . . . . . . . . |
  16| . . . . . . . . |
  24| . . . . . . . . |
  32| . * * * * * * . |
  40| . * * * * * * . |
  48| . * * * * . . . |
  56| . . . . . . . . |
  64| . . . . . . . . |
  72| . . . . . . . . |
  80| . . . . . . . . |
  88| . . . . . . . . |
  96| . . . . . . . . |
 104| . . . . . . . . |
 112| . . . . . . . . |
 120| . . . . . . . . |
    +-----------------+
```

Die Darstellung enthält die Informationen, welche Rechenknoten zur Partition *mypart* gehören und wo sie physikalisch auf dem zweidimensionalen Gitter des Paragon liegen. Diese Rechenknoten sind durch einen Stern gekennzeichnet. Die Punkte in der obigen Abbildung bezeichnen die möglichen Plätze für Rechenknoten. Im Zentralinstitut für Angewandte Mathematik sind 72 Rechenknoten zu einem Gitter der Größe 6×12, beginnend mit Nummer 33, angeordnet.
Durch den Befehl

pspart [*partition*]

werden die Applikationen in einer Partition aufgeführt. Im Gegensatz zum iPSC/860 braucht man keine Partition explizit anzufordern. Es genügt, wie in einem UNIX-System, die Eingabe eines ausführbaren Programmes, und das OSF/1-Betriebssystem des Paragon erstellt automatisch die gewünschte Partition. Die einzige erforderliche Angabe ist die Größe der Partition, das heißt die Anzahl *num* der benötigten Rechenknoten, durch die Option **-sz** *num*. Es existieren noch eine Vielzahl von Optionen. In dieser Arbeit sollen jedoch nur die benötigten **-pkt**, **-mbf** und **-plk** beschrieben werden. Mit **-pkt** läßt sich die Größe eines Nachrichtenpaketes festlegen. Der Default-Wert beträgt 1792 Byte. Dieser Wert wurde bei allen Berechnungen gewählt. Durch die Option **-mbf** kann man die Größe eines Nachrichtenpuffers variieren. Der Default-Wert hier beträgt 1 MByte, und nur bei der Messung der Kommunikation wurde die Größe der Nachrichtenpuffer auf 4 MByte erhöht. Die Angabe von **-plk** bewirkt, daß das vollständige Programm direkt in

den lokalen Speicher eines jeden Rechenknotens der Partition geladen wird. Beim Fehlen dieser Option werden die Daten erst bei Bedarf auf die Knoten geladen, was die Ausführungszeit der Applikation beträchtlich verlängert und bei gleichen Messungen zu stark unterschiedlichen Werten führt.
Der Paragon bietet aber auch die Möglichkeit, eine Partition explizit anzugeben. Dazu steht der Befehl

mkpart [*options*] *partition*

zur Verfügung. **mkpart** kreiert eine Partition mit Namen *partition*, die auch nach Ende einer darauf laufenden Applikation erhalten bleibt. Die Option **-sz** *num* gibt die Anzahl *num* der zu der Partition *partition* gehörenden Rechenknoten an; mit **-sz** *h***X***w* lassen sich die Höhe *h* und die Breite *w* der angeforderten Partition festlegen, und durch **-nd** *nodspec* können die Knoten, die zu der Partition gehören sollen, explizit angegeben werden. Will man die mit **mkpart** *partition* angeforderte Partition wieder freigeben, so muß der Befehl

rmpart *partition*

ausgeführt werden. Über das **N**etwork **Q**ueing **S**ystem (NQS) können auf dem Paragon, wie auf dem iPSC/860, Jobs aus verschiedenen Queues im Batch-Betrieb ausgeführt werden. Die in einem Shell-Skript genauer zu formulierenden Jobs werden mit

qsub [**-p** *queuename*] [*jobname*]

in die sich durch unterschiedliche Anzahl von Rechenknoten auszeichnenden Queues submittiert und können mit

qdel -k *requestname*

wieder aus einer Queue entfernt werden.

2.2.5 Parallele Programmiermodelle des Paragon

Das Paragon-System unterstützt eine Vielzahl von parallelen Programmiermodellen. Das Hauptmodell ist das *Message Passing* (MP). Daneben werden die Modelle *Data Parallel Programming* (DPP) und *Shared Virtual Memory* (SVM) zur Verfügung stehen. Zur Realisation der SPMD-, SIMD- und MIMD-Modelle werden die Konzepte Message Passing und Data Parallel Programming benutzt. Das *Shared-Memory*-Konzept (SM) soll durch SVM simuliert werden.

- **Message Passing**
  Das Konzept des auch beim Paragon verwendeten Message Passing wurde bereits in Unterabschnitt 2.1.5 ausführlich beschrieben. Deshalb wird an dieser Stelle auf eine Beschreibung von Message Passing verzichtet. Die vom iPSC/860 her bekannten Message-Passing-Routinen existieren hier neben einigen Erweiterungen.

- **Data Parallel Programming**
  Für das Konzept des Data Parallel Programming favorisiert Intel *High Performance FORTRAN* (HPF). High Performance FORTRAN ist eine Erweiterung von FORTRAN 90. Die Parallelität wird durch Feldoperationen ausgedrückt. Direktiven geben an, wie die Felder auf die parallelen Prozesse verteilt werden. Daneben spezifiziert der Programmierer ein Gitter von Prozessen, welches dann auf die existierende Hardware abgebildet wird. Eine ausführliche Beschreibung von High Performance FORTRAN wird in [HPFF92] geboten.

- **Shared Virtual Memory**
  Das Shared-Memory-Modell wird mit Hilfe des noch in der Entwicklung stehenden Shared-Virtual-Memory-Konzeptes realisiert. Eine parallele Applikation, deren Daten allen beteiligten Prozessen gemeinsam sind, wird auf einem oder mehreren Rechenknoten ausgeführt. Der physikalisch verteilte Speicher bildet einen globalen Adreßraum über alle beteiligten Rechenknoten. Das heißt, es ist kein expliziter Nachrichtenaustausch nötig. Das Konzept des Shared Virtual Memory vereinfacht die Portierung großer Applikationen, da die Message-Passing-Routinen nicht mehr direkt aufgerufen werden. In [Li86] wird SVM näher vorgestellt.

2.2.6 Die File-Systeme

Das File-System des Paragon basiert auf dem standardisierten und nichtparallelen UNIX-File-System und dem Network File System (NFS). Für den parallelen Zugriff auf Dateien wird vom Paragon OSF/1-Betriebssystem das **P**arallel **F**ile **S**ystem (PFS) unterstützt.
Der Massenspeicher des Paragon besteht aus mehreren Speicherplatten, die im Paragon-Gehäuse integriert sind. Die Speicherplatten oder Disks basieren auf der RAID-Technologie (**R**edundant **A**rrays of **I**nexpensive **D**isks). Jedes RAID besitzt eine Speicherkapazität von 4.8 GByte und ist durch einen SCSI-Bus mit einem ihm zugewiesenen Ein-/Ausgabeknoten verbunden. In seiner größten Konfiguration stellt das Paragon-System mehr als ein TeraByte Speicherplatz mit einer Übertragungsrate von 6.4 GByte/s zur Verfügung. Die Controller des RAID organisieren mehrere Disk-Drives und einen Parity-Drive. Im Falle eines Drive-Fehlers während der Ausführung einer Applikation kann der RAID-Controller verlorene Daten wiederfinden und neu zusammensetzen.
Der im Zentralinstitut für Angewandte Mathematik installierte Paragon XP/S 5 besitzt zwei RAID-Systeme mit jeweils fünf SCSI-Speicherplatten. Die gesamte Speicherkapazität beträgt 9.6 GByte. Zusätzlich besteht eine Verbindung zu einem Bandlaufwerk und es sind zwei Ethernet-Anschlüsse installiert. Die derzeitige Konfiguration des Paragon XP/S 5 im ZAM ist in Abbildung 2.12 dargestellt.

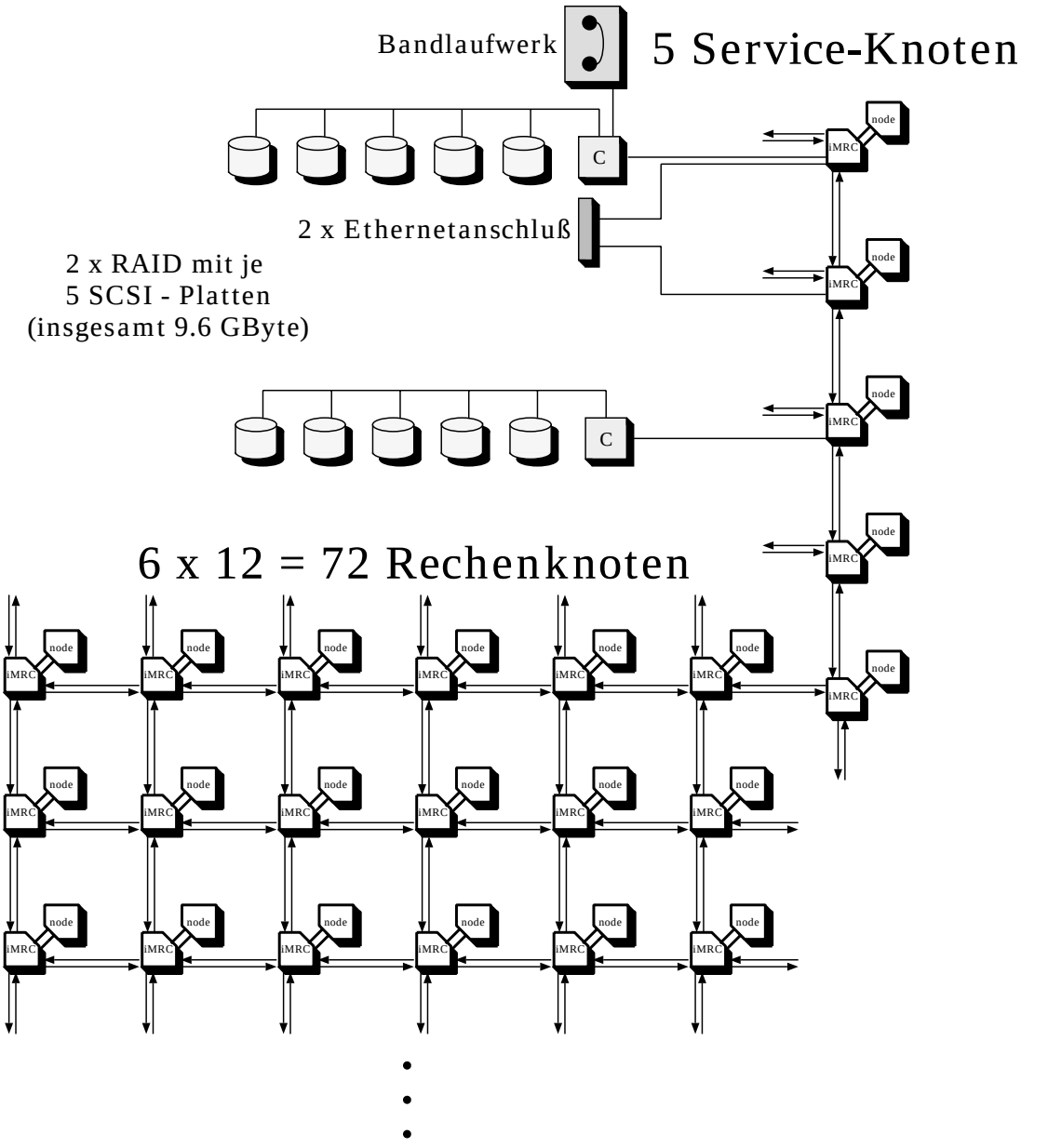


Abbildung 2.12: Konfiguration des Paragon XP/S 5 im Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich

2.3 Grundbegriffe der Parallelverarbeitung

Für die Bewertung von parallelen Algorithmen und deren Eignung für verschiedene Parallelrechner sind u. a. von [Hoß92], [KrSm88] und [Qui87] mehrere Begriffe eingeführt worden, die im folgenden erläutert werden sollen.

Die parallel abzuarbeitenden Programme werden durch unterschiedliche Prozesse gekennzeichnet. Im Falle der hier betrachteten massiv-parallelen Rechner werden diese auf den einzelnen Rechenknoten oder Prozessoren bearbeitet.
Der Parallelismus eines Algorithmus kann sich dabei in zwei verschiedenen Arten ausdrücken:

- Beim **Datenparallelismus** werden die Daten auf alle zur Verfügung stehenden Prozessoren aufgeteilt, die dann die gleichen Operationen auf diesen ausführen. Die beim Datenparallelismus verwendete Rechnerarchitektur ist der SIMD- bzw. der SPMD-Rechner.

- Die zweite Art des Parallelismus ist der **Prozeßparallelismus**. Dabei wird der Algorithmus in verschiedene Segmente aufgeteilt, die auf den einzelnen Prozessoren bearbeitet werden. Gleichzeitig zu den verschiedenen Anweisungen werden auf den Prozessoren auch verschiedene Datensätze bearbeitet. Die hier verwendete Architektur ist das MIMD-Modell.

Ein weiteres Maß für die Güte oder den Grad der Parallelisierungsmöglichkeit bietet der Begriff der **Granularität**. Er wird in der Literatur vielfältig verwendet, soll hier aber die Größe parallel abzuarbeitender Prozesse beschreiben. Es wird zwischen **grober** und **feiner Granularität** unterschieden. Bei feiner Granularität ist die Anzahl parallel abzuarbeitender Anweisungen gering. Sie werden dafür aber häufiger aufgerufen. Die feine Granularität besitzt hohe Kommunikationsanforderungen und es ist oftmalige Synchronisation erforderlich. Parallel abzuarbeitende Schleifen sind durch eine feine Granularität gekennzeichnet. Im Gegensatz dazu ist die grobe Granularität auf der Ebene von Unterprogrammen zu finden. Hier ist nur eine geringe Kommunikation und kaum Synchronisation erforderlich.

Die Ausführung eines parallelen Algorithmus im Vergleich zu dem entsprechenden sequentiellen erfordert oft eine Reihe von zusätzlichen Operationen. So muß zur Sicherstellung der korrekten Verteilung der Daten und der Kommunikation ein hoher Verwaltungsaufwand betrieben werden. Es müssen eine neue Indexberechnung, eine Selbstidentifikation der Prozesse und Synchronisationsschritte durchgeführt werden. Dieser Mehraufwand des parallelen gegenüber dem sequentiellen Algorithmus wird als **Overhead** bezeichnet.

Die mögliche Vergrößerung der Anzahl der Prozesse und damit die Steigerung der vorhandenen Leistungsfähigkeit wird durch den Begriff **Skalierbarkeit** ausgedrückt.

Ein Maß für die Güte der Ausführung eines parallelen Algorithmus auf einem Parallelrechner ist der **Speedup**. Der Speedup eines Algorithmus, der auf p Prozessoren parallel ausgeführt wird, ist gegeben durch

$$S_p = \frac{T_s}{T_p}.$$

T_s ist die Ausführungszeit des besten sequentiellen Algorithmus für das Problem. T_p bezeichnet die Ausführungszeit des zu bewertenden parallelen Algorithmus auf p Prozessoren.
Im Idealfall ist der erreichbare Speedup gleich der Anzahl p der verwendeten Prozessoren. Der maximal erzielbare Speedup wird jedoch durch den Anteil der nur sequentiell ausführbaren Operationen beschränkt. Dies wird durch das **Gesetz von Amdahl** ausgedrückt. f_p sei der Anteil der parallel ausführbaren Anweisungen, entsprechend $1 - f_p$ der Anteil der sequentiell auszuführenden Anweisungen des parallelen Algorithmus. Dann berechnet sich der Speedup zu

$$S_p = \frac{T_s}{(1 - f_p) \cdot T_s + \frac{f_p \cdot T_s}{p}} = \frac{1}{1 - f_p + \frac{f_p}{p}}.$$

Um den erreichten Speedup eines parallelen Algorithmus einschätzen zu können, ist der Begriff der **Effizienz** eingeführt worden. Dabei wird der Speedup S_p relativ zur Anzahl p der verwendeten Prozessoren betrachtet:

$$E_p = \frac{S_p}{p}.$$

Im Idealfall ist die Effizienz gleich eins.

2.4 Messung der Kommunikationsleistung

Die Ausführungszeit einer parallelen Applikation gegenüber ihrer sequentiellen Version ist einerseits stark abhängig vom Problem und andererseits vom Ausmaß der benötigten Kommunikation. Das heißt, es sollten so wenig Nachrichten wie möglich ausgetauscht werden. Kommunikation läßt sich jedoch nicht gänzlich aus einer parallelen Anwendung, die auf einem Rechner mit verteiltem Speicher ausgeführt wird, herausnehmen. Deshalb wurde die Kommunikationsleistung der beiden in den Abschnitten 2.1 und 2.2 beschriebenen massiv-parallelen Rechner untersucht.

Die Zeitdauer einer Nachrichtenübertragung berechnet sich aus der Startup-Zeit und der Summe der Übertragungszeiten der einzelnen Nachrichtenpakete, also

$$T_{comm}(k) = t_{startup} + k \cdot t_{send},$$

wobei k die Anzahl der zu versendenden Nachrichtenpakete darstellt. Unter der *Startup*-Zeit ist die Zeit zu verstehen, die das System braucht, um eine Nachrichtenübertragung zu initiieren. Kann eine Nachricht aufgrund ihrer Länge nicht auf einmal versendet werden, so wird sie in kleinere Pakete aufgeteilt, die dann hintereinander verschickt werden.

Zur Ermittlung der Kommunikationsleistung wurde ein kleines Programm (**TALK**) geschrieben und auf beide Rechner implementiert. Sender- und Empfängerknoten tauschen mittels einer in Abbildung 2.13 dargestellten Ping-Pong-Strategie Nachrichten verschiedener Längen aus. Die Kommunikation verläuft im blockierenden Modus.

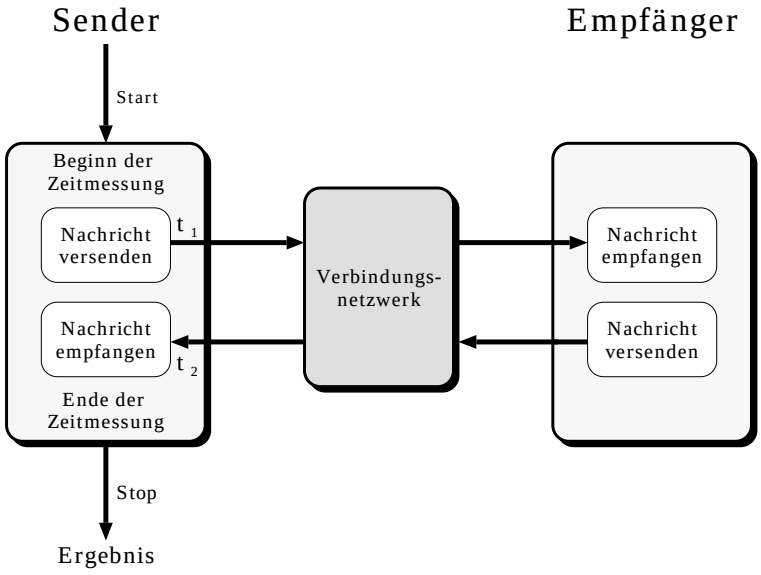


Abbildung 2.13: Kommunikationszeitmessung mit Ping-Pong-Strategie

Das synchrone Kommunikationsprogramm **TALK** lautet:

```
      PROGRAM TALK
      . . .
      DO 100 msglen = 0, 5000, 10
         IF (me .EQ. sender) begin = DCLOCK()
         DO 110 i = 1, 1000, 1
            IF (me .EQ. sender) THEN
               CALL CSEND (10 * i, x, msglen, empfaenger, 0)
               CALL CRECV (11 * i, x, msglen)
            ELSEIF (me .EQ. empfaenger) THEN
               CALL CRECV (10 * i, x, msglen)
               CALL CSEND (11 * i, x, msglen, sender, 0)
            ENDIF
 110     CONTINUE
         IF (me. EQ. sender) end = DCLOCK()
 100  CONTINUE

      END
```

Die Übertragung wird nicht direkt gemessen. Beide Knoten verfügen zwar über eigene Uhren, diese können aber nicht ausreichend synchronisiert werden. Deshalb wurde die Dauer einer Übertragung nur von einem Knoten gemessen. Da diese für eine genaue Messung jedoch zu gering ist, wurde der Durchschnittswert eines 1000-maligen Austauschs einer Nachricht einer bestimmten Länge ermittelt.
Der Sender initiiert die Nachricht, bestimmt den Zeitpunkt t_1 des Sendens und schickt die Nachricht zum Empfänger. Der sendet diese sofort zurück. Der Sender empfängt die von ihm initiierte Nachricht wieder und bestimmt den Zeitpunkt t_2 des Empfangs. Die Dauer der Nachrichtenübermittlung vom Sender zum Empfänger berechnet sich dann durch

$$T_{comm} = \frac{t_2 - t_1}{2}.$$

Diese Prozedur wird entsprechend oft wiederholt und der Durchschnittswert bestimmt. Die Nachrichtenlänge variiert zwischen 0 und 5000 Bytes und wurde für den nächsten Durchlauf immer um 10 Bytes erhöht. Gemessen wurden die Zeitpunkte des Sendens und Empfangens mittels der System-Funktion **dclock()**, die die vergangene Zeit in Sekunden zurückgibt.

Eine Frage bleibt nun noch zu klären: Wie groß ist der durch den Aufruf von **dclock()** mitgemessene Overhead?
Um eine Aussage darüber zu erhalten, wurde der Durchschnittswert von einhundert **dclock()**-Aufrufen bestimmt. Es zeigte sich, daß der durch die **dclock()**-Funktion bei der Zeitmessung verursachte Overhead vernachlässigbar gering ist. Ein Aufruf benötigt durchschnittlich 1.63 μs. Diese minimale Zeit fällt, sowohl bei der Messung der Kommunikationsleistung, als auch bei der Ausführungszeit einer Applikation, nicht ins Gewicht und führt so nicht zu verfälschten Ergebnissen.

In der Legende der in den folgenden Unterabschnitten aufgeführten Graphiken tritt die Bezeichnung **Hop** auf. Damit ist die Anzahl der bei einer Kommunikation übersprungenen Rechenknoten gemeint. Wurde zum Beispiel eine Messung mit der Bezeichnung 2 Hop durchgeführt, so lagen zwischen Sender und Empfänger zwei Rechenknoten. Angegeben wird die absolute Dauer einer Nachrichtenübermittlung in Millisekunden.

2.4.1 Kommunikationsleistung des iPSC/860

Die Kommunikationsmessungen wurden beim iPSC/860 im normalen Benutzerbetrieb durchgeführt. Das heißt, neben dem oben beschriebenen Meßprogramm wurden von weiteren Benutzern Applikationen auf Cubes ausgeführt, die das Netzwerk mit den dafür nötigen Nachrichtenübermittlungen belasteten. Die einzige Ausnahme bildete die 4-Hop-Messung. Dabei wurden 32 Rechenknoten benötigt, die im Batch-Betrieb angefordert wurden. An der 2-Hop-Messung waren 8 Rechenknoten beteiligt, die einen dreidimensionalen Cube bilden. Bei der einfachsten Kommunikationsmessung, dem 0 Hop, lagen Sender und Empfänger direkt nebeneinander. Die Ergebnisse der ausgewählten drei Messungen zeigt Abbildung 2.14.

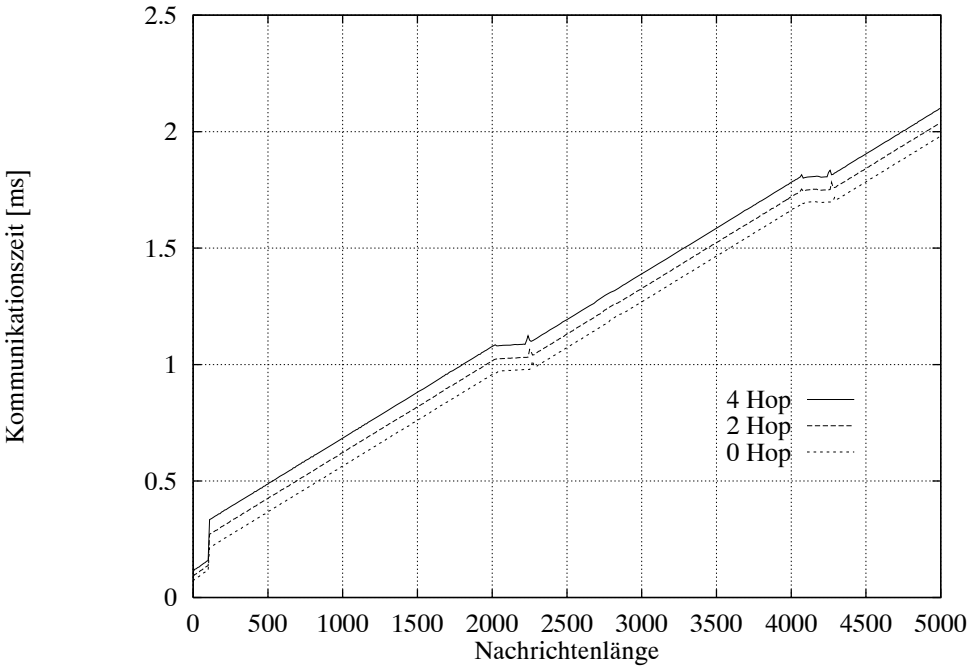


Abbildung 2.14: Kommunikationszeit in Millisekunden in Abhängigkeit von der Nachrichtenlänge bei unterschiedlichem Abstand von Sender und Empfänger auf dem iPSC/860

Man erkennt deutlich, daß sich die Entfernung von Sender und Empfänger durch eine längere Übertragungsdauer bemerkbar macht. Die Kommunikationszeit wächst mit dem Abstand der Kommunikationspartner.

Beim Übergang der Nachrichtenlänge von 100 Bytes auf 110 Bytes findet der Wechsel vom short message protocol zum long message protocol statt. Die Übertragungszeit von 100 Bytes liegt bei der 2-Hop-Messung bei 135 μs, die von 110 Bytes bei 272 μs. Das heißt, der Protokollwechsel verursacht aufgrund der erhöhten Startup-Zeit einen Anstieg der Kommunikationszeit um 126 μs und für die um 10 Bytes längere Nachricht erhöht sich die reine Übertragungszeit um 11 μs.

2.4.2 Kommunikationsleistung des Paragon XP/S 5

Es muß zu Anfang erwähnt werden, daß der in Unterabschnitt 2.2.2 beschriebene Kommunikationsprozessor zur Zeit der hier vorgestellten Kommunikationsuntersuchung noch nicht vom Betriebssystem unterstützt wurde. Die Kommunikation des Paragon wurde stattdessen von einem DMA-Chip (DMA = **D**irect **M**emory **A**ccess) realisiert. Der Nachrichtenpuffer der Rechenknoten wurde mit der Option **-mbf** von 1 MByte auf 4 MByte erhöht.

Die Kommunikationszeiten beim Paragon wurden wie die beim iPSC/860 im normalen Benutzerbetrieb gemessen. Die Entfernung der Kommunikationspartner wirkt sich beim Paragon auf die Übertragungsdauer einer Nachricht nicht wesentlich aus. Das Ergebnis einer diesbezüglichen Untersuchung zeigt Abbildung 2.15. Aus diesem Grund wird im weiteren nur der eine Fall betrachtet, in dem sich zwischen Sender und Empfänger vier weitere Rechenknoten befinden.

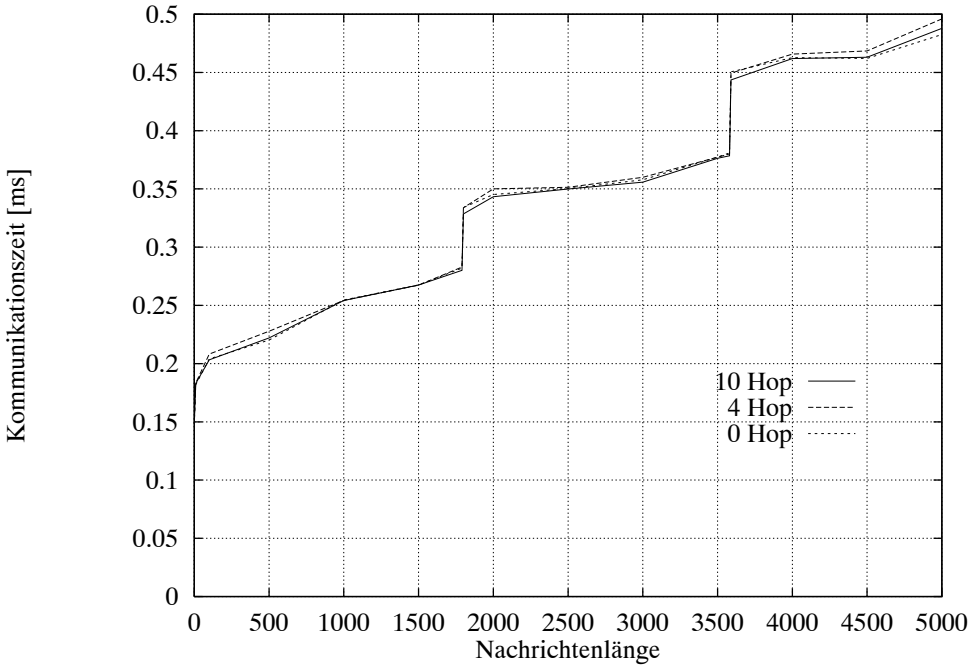


Abbildung 2.15: Kommunikationszeit in Millisekunden in Abhängigkeit von der Nachrichtenlänge bei unterschiedlichem Abstand von Sender und Empfänger auf dem Paragon

Da auf dem Paragon Partitionen frei allokiert werden können, lassen sich mehrere Kommunikationssituationen untersuchen. Es werden für die Untersuchungen zwei Partitionen verwendet, eine mit zwei (Partition **X**) und eine mit vier Rechenknoten (Partition **Y**), die folgendermaßen angelegt wurden:

```
X  Y  Y  Y  Y  X
```

Die Rechenknoten der Partition **Y** werden im weiteren als Zwischenknoten bezeichnet. Grundsätzlich tauschten die beiden Knoten der Partition **X** Nachrichten untereinander aus. Folgende Situationen auf den Zwischenknoten werden dabei untersucht:

1. Die beiden äußersten Zwischenknoten kommunizieren nur miteinander. Die ausgetauschten Nachrichten sind dieselben, die von den beiden Rechenknoten der Partition X versendet werden.

2. Alle vier Rechenknoten bearbeiten eine sequentielle Applikation.

3. Auf den Zwischenknoten wird nichts ausgeführt.

Die Ergebnisse der Untersuchung der drei oben aufgeführten Situationen sind in Abbildung 2.16 dargestellt.

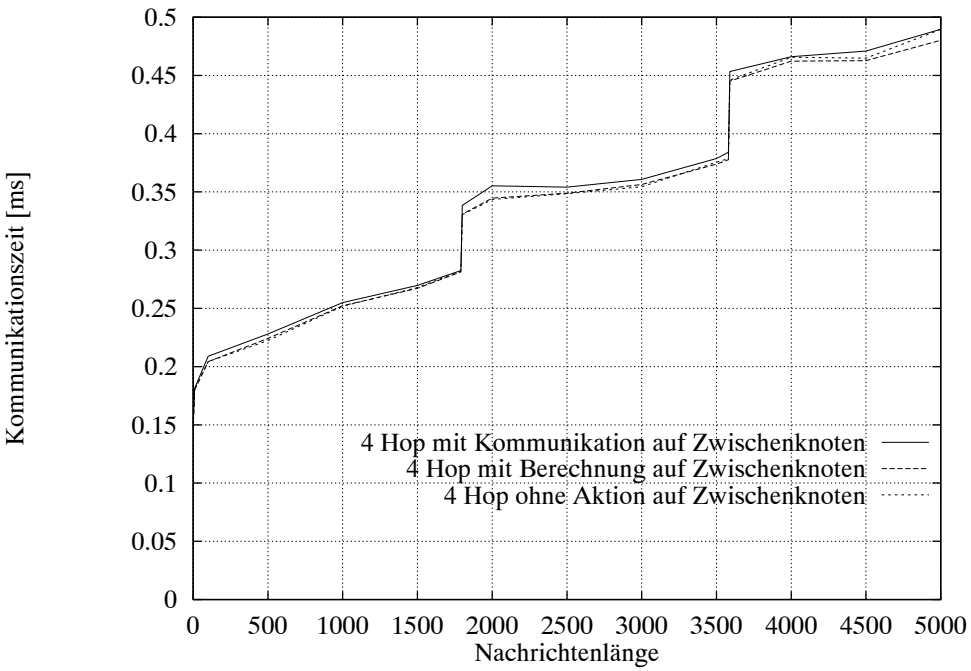


Abbildung 2.16: Kommunikationszeit in Millisekunden in Abhängigkeit von der Nachrichtenlänge bei unterschiedlicher Aktivität der Zwischenknoten auf dem Paragon

Deutlich ist der hohe Anfangswert, bedingt durch die derzeitige hohe Startup-Zeit von 135 μs, zu sehen. Die Sprünge beim Übergang von 1790 Bytes auf 1800 Bytes und von 3580 Bytes auf 3590 Bytes sind durch die Paketierung der Nachrichten zu erklären. Wie bereits im Unterabschnitt 2.2.4 erwähnt, läßt sich die Paketgröße einer Nachricht mit der Option **-pkt** variieren. Die maximale Größe eines Paketes beträgt aber 1792 Bytes. Der FIFO-Puffer des NIC besitzt eine Kapazität von 2 KByte. Die Differenz von maximaler Paketgröße zur Größe des FIFO-Puffers beträgt 256 Bytes. Diese Differenz geht durch den nötigen Verwaltungsaufwand bei der Übermittlung einer Nachricht verloren. Die maximale Paketgröße von 1792 Bytes spiegelt sich in den Zeitverläufen der Kommunikation wieder. An den Stellen, wo ein Wechsel von ein auf zwei und von zwei auf drei Pakete erfolgt, erhöht sich die Kommunikationszeit zum Beispiel in der Situation 1 um 55.9 μs bzw. 69.2 μs.
Abbildung 2.16 zeigt, daß sich die Kommunikationszeiten in den drei Situationen ein wenig voneinander unterscheiden. Kommunizieren die Zwischenknoten miteinander, so braucht eine Nachrichtenübertragung vom Sender zum Empfänger der Partition **X** etwas länger als in den anderen zwei Fällen. Generell läßt sich jedoch sagen, daß die Verzögerung der Kommunikationszeit durch Aktionen auf Zwischenknoten so gering ist, daß sie für die Performance einer Applikation keine Bedeutung besitzt.

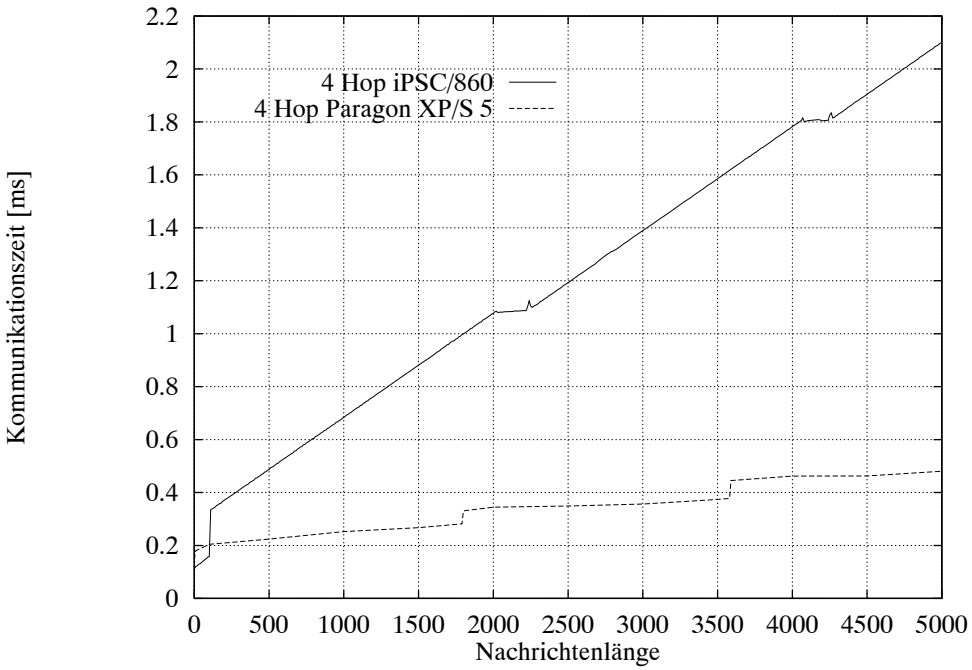


Abbildung 2.17: Vergleich der Kommunikationszeit in Abhängigkeit von der Nachrichtenlänge bei iPSC/860 und Paragon

In Abbildung 2.17 werden die Zeiten einer 4-Hop-Kommunikation mit Zwischenknoten ohne Aktion auf Paragon und iPSC/860 miteinander verglichen. Die Übertragungsdauer einer Nachricht beim iPSC/860 ist bei Anwendung des long message protocol sehr viel länger als beim Paragon. Dafür überträgt der iPSC/860 kurze Nachrichten bis 100 Bytes schneller als dieser. Die Gründe für dieses unterschiedliche Zeitverhalten bei der Kommunikation sind einerseits in den verschiedenen

Message-Passing-Protokollen zu suchen. Andererseits besitzt der Paragon ein sehr viel schnelleres Verbindungsnetzwerk als der iPSC/860.

Die Kommunikation ist also, wie gezeigt wurde, ein nicht zu unterschätzender Faktor in der Ausführungszeit einer parallelen Applikation [HeBe90]. Es sollte daher so selten wie möglich kommuniziert werden. Die Kommunikationszeit gliedert sich in Startup-Zeit und reine Übertragungszeit. Der Anteil dieser Zeit ist abhängig von der Länge einer Nachricht, kann somit kaum verringert werden. Um jedoch den Anteil der Startup-Zeit an der Kommunikationszeit so gering wie möglich zu halten, muß weitgehend auf ein Versenden kleiner Nachrichten zugunsten großer verzichtet werden.

Kapitel 3

Grundlagen der Graphentheorie

In diesem Kapitel sollen die Grundlagen und Begriffe der Graphentheorie bereitgestellt werden, die im weiteren Verlauf dieser Arbeit benötigt werden.
Graphen dienen als mathematische Hilfsmittel zur Modellierung von Problemen. Durch einen Graphen werden zweistellige Relationen dargestellt. Diese können sowohl symmetrischer als auch asymmetrischer Natur sein. Daraus ergibt sich eine Unterteilung in gerichtete und ungerichtete Graphen.
Die Bezeichnungen der folgenden Definitionen und Beispiele sind angelehnt an die am Ende dieser Arbeit aufgeführte Literatur zur Graphentheorie, wie [DöMü73], [Eve79] und [Har69].

3.1 Ungerichtete Graphen

Symmetrische Relationen werden durch ungerichtete Graphen dargestellt. Graphen können auf unterschiedliche Art und Weise definiert werden. Im weiteren werden Kanten als Paare von Knoten betrachtet.

Definition 1 : Ungerichteter Graph

Ein *ungerichteter Graph* ist ein Paar $G = (V, E)$ aus einer endlichen nichtleeren Menge V von n *Knoten* (vertices) und einer Menge E von m ungeordneten Paaren $e = \{v_i, v_j\}$, $i, j, m, n \in \mathbb{N}$, verschiedener Elemente aus V, den *Kanten* (edges).
Für eine Kante $e = \{v_i, v_j\}$ heißen v_i und v_j *Endknoten* von e.
Man sagt auch, daß v_i und v_j *inzident* mit e und v_i und v_j *adjazent* zueinander sind.
Zwei Kanten heißen *adjazent* zueinander, wenn sie einen gemeinsamen Endknoten besitzen.
Ein Graph mit n Knoten und m Kanten heißt (n, m)-*Graph*.

Abbildung 3.1 zeigt einen ungerichteten Graphen G:

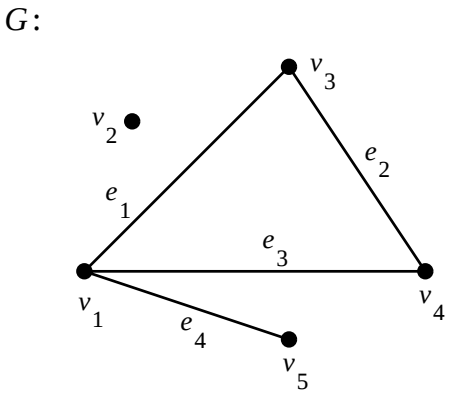


Abbildung 3.1: Ungerichteter Graph G

Im folgenden werden weitere Begriffe vorgestellt und anhand von Abbildung 3.1 erläutert:

- Der *Grad* $d(v)$ des Knotens v ist die Anzahl der mit ihm inzidenten Kanten ($d(v_1) = 3$).
- Hat jeder Knoten eines Graphen denselben Grad, so heißt der Graph *regulär*.
- Ein Knoten v heißt *isolierter Knoten*, falls er mit keiner Kante inzident ist, also $d(v) = 0$ ($d(v_2) = 0$).

Definition 2 : Bewerteter Graph

Ein Graph $G = (V, E)$ heißt *bewertet*, falls jeder Kante $e = \{v_i, v_j\} \in E$ eine Zahl $w(e) = w_{ij} \in \mathbb{R}$ zugeordnet ist, $i, j \in \mathbb{N}$. Diese Zahl heißt das *Gewicht* oder die *Kosten* der Kante $e = \{v_i, v_j\}$.

Definition 3 : Teilgraph

Ein Graph $G' = (V', E')$ heißt *Teilgraph* von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$ ist.
Ein Teilgraph G' heißt *spannender Teilgraph* von G, falls $V' = V$ und $E' \subseteq E$.
Enthält E' alle Kanten aus E, deren Endpunkte in V' liegen, so heißt G' ein *gesättigter Teilgraph* von G. Man sagt auch, G' *wird von* $V' \subseteq V$ *in* G *aufgespannt.*

In Abbildung 3.2 wird ein Teilgraph G_t, ein spannender Teilgraph G_s und ein gesättigter Teilgraph G_g des Graphen G aus Abbildung 3.1 dargestellt.

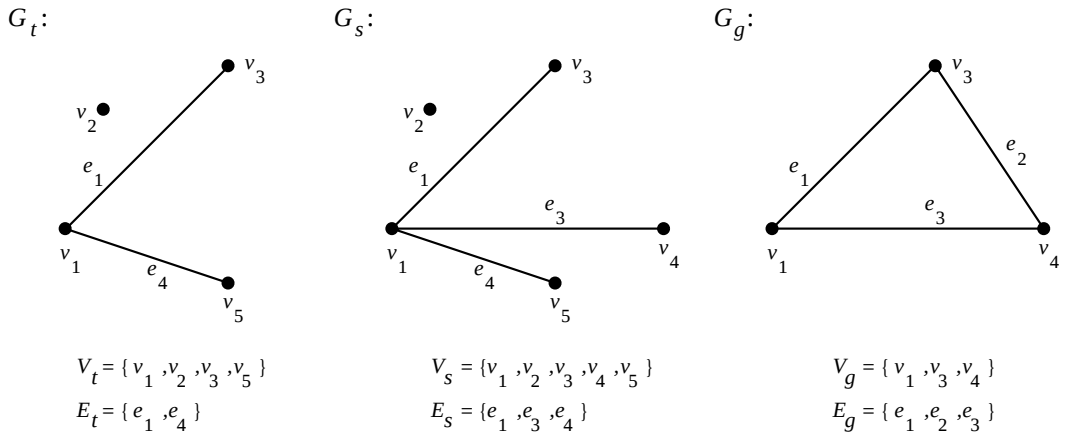


Abbildung 3.2: Teilgraph G_t, spannender Teilgraph G_s und gesättigter Teilgraph G_g des Graphen G

Definition 4 : Kantenfolge

Eine *Kantenfolge* von Knoten v_1 nach Knoten v_n im Graphen G ist eine endliche Folge von Kanten $\{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_{n-1}, v_n\}$ mit $n \in \mathbb{N}$. Je zwei aufeinanderfolgende Kanten besitzen also einen gemeinsamen Endpunkt.
Ist $v_1 \neq v_n$, heißt die Kantenfolge *offen*.
Eine Kantenfolge heißt *geschlossen*, falls $v_1 = v_n$.

Definition 5 : Weg

Ein *Weg* ist eine offene Kantenfolge, in der alle Knoten v_1, ..., v_n, $n \in \mathbb{N}$ verschieden sind.

Definition 6 : Kantenzug

Ein *Kantenzug* ist eine Kantenfolge, in der alle Kanten voneinander verschieden sind.

Definition 7 : Kreis

Sind in einem Kantenzug die Knoten v_1, ..., v_{n-1} voneinander verschieden und gilt $v_1 = v_n$, $n \in \mathbb{N}$, so heißt dieser Kantenzug *Kreis* oder *Zykel*.
Ein Graph heißt *kreisfrei*, falls er keinen Kreis enthält.

Definition 8 : Hamiltonkreis

Gibt es in einem Graphen $G = (V, E)$ einen Kreis K, der jeden Knoten $v \in V$ des Graphen G genau einmal trifft, so heißt K *Hamiltonkreis*.

Definition 9 : Eulerkreis

Gibt es in einem Graphen $G = (V, E)$ einen geschlossenen Kantenzug Z, der jede Kante $e \in E$ des Graphen G genau einmal enthält, so heißt Z *Eulerkreis*.

In Abbildung 3.1 bilden die Kanten (e_4, e_1, e_2, e_3) einen Kantenzug und die Kanten (e_2, e_1, e_4) einen Weg von v_4 nach v_5. Die Kanten (e_1, e_2, e_3) bilden einen Kreis. In G gibt es keinen Hamilton- und keinen Eulerkreis.

Definition 10 : Länge

Die *Länge einer Kantenfolge* eines unbewerteten Graphen ist definiert durch die Anzahl der Kanten dieser Kantenfolge.
Entsprechend wird die *Länge eines Kantenzuges*, *eines Weges* und *eines Kreises* in einem unbewerteten Graphen definiert.
Unter dem *Abstand* $d(v_i, v_j)$ zweier Knoten v_i und v_j, $i, j \in \mathbb{N}$, in einem unbewerteten Graphen versteht man die Länge des kürzesten Weges von Knoten v_i nach Knoten v_j.

Definition 11 : Zusammenhängender Graph

Ein Graph $G = (V, E)$ heißt *zusammenhängend*, falls zu je zwei verschiedenen Knoten $v_i \in V$ und $v_j \in V$ mit $v_i \neq v_j$, $i, j \in \mathbb{N}$, ein Weg existiert, der v_i und v_j verbindet (Abbildung 3.3).

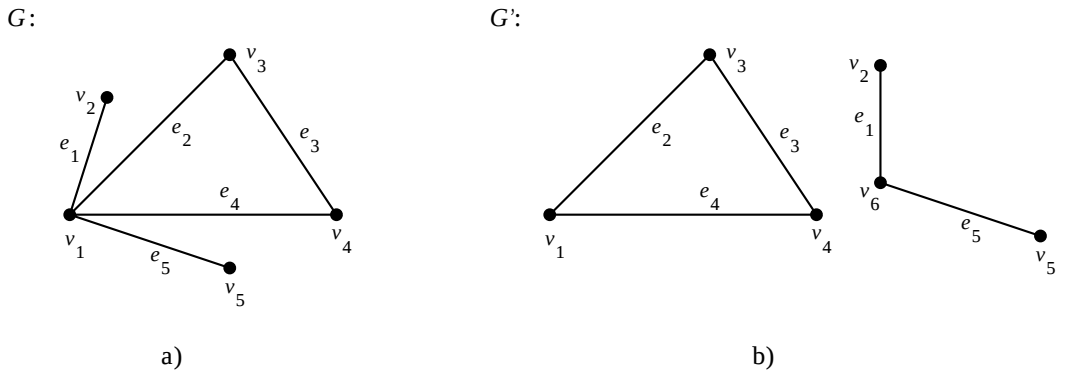


Abbildung 3.3: a) Zusammenhängender Graph G und b) nichtzusammenhängender Graph G'

Definition 12 : Zusammenhangskomponente

Eine *Zusammenhangskomponente* eines ungerichteten Graphen $G = (V, E)$ ist ein maximaler zusammenhängender gesättigter Teilgraph von G.

Definition 13 : Matching

Gegeben sei ein Graph $G = (V, E)$. Eine Teilmenge $M \subseteq E$ heißt ein *Matching von* G, falls keine zwei Kanten aus M adjazent sind. Ein Matching M *berührt einen Knoten* $v \in V$, wenn eine Kante aus M mit v inzident ist, sonst berührt M den Knoten v nicht. Berührt ein Matching alle Knoten von G, so heißt M *perfekt*. Ein Matching M heißt *maximal*, wenn es kein Matching M' gibt mit $|M'| > |M|$.

In Abbildung 3.4 ist ein Matching $M = \{e_2, e_4\}$ des Graphen G aus Abbildung 3.1 dargestellt. Das Matching berührt den Knoten v_2 nicht, ist also nicht perfekt.

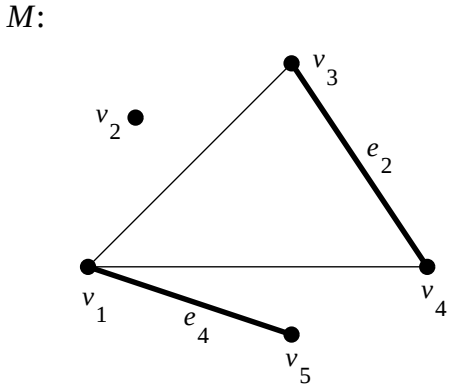


Abbildung 3.4: Matching M des Graphen G

3.2 Gerichtete Graphen

In Praxisanwendungen besitzen Relationen zwischen zwei Objekten oft einen asymmetrischen Charakter. Werden solche Relationen durch Graphen dargestellt, so muß den Kanten eine Richtung zugewiesen werden. Diese Relationen werden mittels gerichteter Graphen dargestellt.

Definition 14 : Gerichteter Graph

Ein *gerichteter Graph* oder *Digraph* ist ein Paar $D = (V, E)$ aus einer endlichen nichtleeren Menge V von n *Knoten* und einer Menge E von m geordneten Paaren $e = (v_i, v_j)$, $i, j, m, n \in \mathbb{N}$, verschiedener Elemente aus V, den *Kanten* von D. Den Kanten ist ein *Richtungssinn* gegeben.
Für eine Kante $e = (v_i, v_j)$ heißt v_i *Anfangspunkt* und v_j *Endpunkt* von e.
Die Knoten v_i und v_j heißen *adjazent* zueinander, und der Knoten v_i bzw. v_j und die Kante e heißen *inzident*.

Abbildung 3.5 zeigt einen gerichteten Graphen.

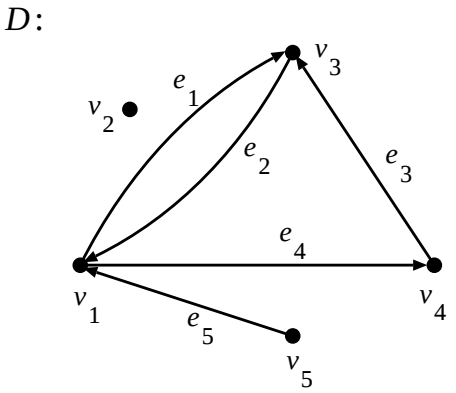


Abbildung 3.5: Gerichteter Graph D

Analog zu den in Abschnitt 3.1 eingeführten Begriffen werden hier weitere erklärt und an dem Beispiel des gerichteten Graphen D in Abbildung 3.5 erläutert.

- *Antiparallele Kanten* sind parallele Kanten, die einen entgegengesetzten Richtungssinn aufweisen (e_1, e_2).
- Der *Innengrad* $d_{in}(v)$ eines Knotens v ist gleich der Anzahl der eingehenden Kanten $(d_{in}(v_3) = 2)$. Entsprechend ist der *Außengrad* $d_{out}(v)$ eines Knotens v gleich der Anzahl der von ihm ausgehenden Kanten $(d_{out}(v_3) = 1)$.
- Der *Grad* $d(v)$ eines Knotens v ist die Summe seines Innen- und Außengrads $(d(v_3) = 3)$.

Die Definitionen für den *isolierten Knoten*, den *gerichteten bewerteten Graph* mit *Kantengewicht* w_{ij}, den *gerichteten Teilgraph*, den *gerichteten spannenden Teilgraph* und den *gerichteten gesättigten Teilgraph* erfolgen analog zu denen in Abschnitt 3.1.

Definition 15 : Symmetrischer gerichteter Graph

Ein Graph $D = (V, E)$ heißt *symmetrischer gerichteter Graph*, falls für je zwei seiner Knoten $v_i \in V$ und $v_j \in V$ mit $(v_i, v_j) \in E$ stets auch $(v_j, v_i) \in E$ gilt, $i, j \in \mathbb{N}$.

Die Definitionen für *gerichtete Kantenfolge*, *gerichteter Weg*, *gerichteter Kantenzug*, *gerichteter Kreis*, *gerichteter Hamiltonkreis* und *gerichteter Eulerkreis* sind wieder analog zu denen aus Abschnitt 3.1.
Enthält ein Digraph $D = (V, E)$ keinen gerichteten Kreis, so heißt er *azyklisch*.

Definition 16 : Länge

Sei $D = (V, E)$ ein gerichteter Graph, dessen Kanten $e = (v_i, v_j)$, $i, j \in \mathbb{N}$, die Gewichte $w(e) = w_{ij}$ zugeordnet sind. Ist $W(v_1, v_n) = \{(v_1, v_2), ..., (v_{n-1}, v_n)\}$, $n \in \mathbb{N}$, ein Weg von $v_1 \in V$ nach $v_n \in V$, dann heißt $\sum_{e \in W} w(e)$ die *Länge des Weges* W.
Ist $M(v_1, v_n) = \{W_i | W_i = W_i(v_1, v_n)\}$, so heißt $W_i \in M(v_1, v_n)$ *kürzester Weg* von v_1 nach v_n, falls für alle Wege $W_j \in M(v_1, v_n)$ mit $i \neq j$ gilt:

$$\sum_{e \in W_i} w(e) \leq \sum_{e \in W_j} w(e).$$

Analog werden die *Länge eines Kreises* und der *Kreis minimaler Länge* definiert.

Ein wesentlicher Unterschied zwischen ungerichteten und gerichteten Graphen ergibt sich bei dem Begriff des Zusammenhangs.
Werden bei einem gerichteten Graphen die Richtungen der Kanten fortgelassen, so entsteht der diesem Digraphen *zugeordnete ungerichtete Graph*.

Definition 17 : Schwacher Zusammenhang

Ein gerichteter Graph heißt (*schwach*) *zusammenhängend*, falls der zugeordnete ungerichtete Graph zusammenhängend ist.

Definition 18 : Starker Zusammenhang

Ein gerichteter Graph heißt *stark zusammenhängend*, falls es zu je zwei Knoten $v_i \in V$ und $v_j \in V$, $i, j \in \mathbb{N}$, einen gerichteten Weg von v_i nach v_j und einen gerichteten Weg von v_j nach v_i gibt.

Definition 19 : Starke Zusammenhangskomponente

Eine *starke Zusammenhangskomponente* eines gerichteten Graphen $D = (V, E)$ ist eine maximale Menge von Knoten, in der ein gerichteter Weg von jedem Knoten zu einem anderen Knoten aus dieser Menge existiert.

3.3 Bäume

Die in diesem Unterabschnitt behandelten Bäume bilden eine Klasse von einfach strukturierten Graphen, denn sie enthalten keine Kreise. Bäume sind nicht nur in der Graphentheorie, sondern auch in vielen praktischen Anwendungsgebieten von großer Bedeutung.

Definition 20 : Baum, Wald

Gegeben sei ein ungerichteter Graph $G = (V, E)$. G heißt *Baum*, falls G zusammenhängend und kreisfrei ist.
Ist jede Zusammenhangskomponente eines Graphen G ein Baum, so heißt G ein *Wald*.

Definition 21 : Wurzel, gerichteter Baum

Sei $D = (V, E)$ ein gerichteter Graph. Ein Knoten v heißt *Wurzel*, falls jeder Knoten von D von v aus auf einem gerichteten Weg erreichbar ist.
Ist der D zugeordnete ungerichtete Graph ein Baum und besitzt D eine Wurzel, so heißt D ein *gerichteter Baum*.

Definition 22 : Wurzelbaum

Ein *Wurzelbaum* ist ein gerichteter Baum, bei dem ein Knoten (die Wurzel) den Innengrad 0 und alle anderen Knoten den Innengrad 1 aufweisen.

Definition 23 : Blatt

Alle Knoten eines gerichteten Baumes, die den Außengrad 0 besitzen, heißen *Blätter* oder *Endknoten*. Die anderen Knoten werden als *Innenknoten* bezeichnet.

Definition 24 : Vater, Sohn

Gegeben sei ein gerichteter Baum mit $v_i, v_j, v_k \in V$, $i, j, k, \in \mathbb{N}$. Ist $(v_i, v_j) \in E$, dann heißt v_i *Vater* von v_j und v_j *Sohn* von v_i. Existiert ein Weg von v_i nach v_k, dann heißt v_i ein *Vorgänger* von v_k und v_k ein *Nachfolger* von v_i. Ein Knoten v_i mit allen seinen Nachfolgern heißt *Teilbaum* und v_i heißt *Wurzel des Teilbaums*.

Definition 25 : Niveau, Höhe

Das *Niveau* (die *Stufe*) *eines Knotens* v in einem Baum ist die Länge des Weges von der Wurzel zu v. Die *Höhe eines Knotens* v in einem Baum ist die Länge des längsten Weges von v zu einem Blatt. Die *Höhe des Baumes* ist die Höhe der Wurzel.

Eine im Zusammenhang mit Graphenalgorithmen und Datenstrukturen oft verwendete Klasse von Bäumen ist die der Binärbäume [AHU83].

Definition 26 : Binärbaum

Ein *Binärbaum* ist ein gerichteter Baum, in dem jeder Knoten v höchstens zwei Söhne besitzt. Jeder Sohn des Knotens v wird entweder als linker oder als rechter Sohn von v bezeichnet.
Ein Binärbaum heißt *vollständig*, falls für eine ganze Zahl k gilt:
Jeder Knoten auf einem Niveau des Binärbaumes kleiner als k hat genau zwei Söhne. Jeder Knoten des Niveaus k ist ein Blatt. Ein vollständiger Binärbaum der Höhe k hat genau $(2^{k+1} - 1)$ Knoten.

Definition 27 : Spannender Baum

Es seien ein ungerichteter Graph $G = (V, E)$, ein gerichteter Graph $D = (V', E')$ und der zugeordnete ungerichtete Graph D_u von D gegeben.

Ist T ein spannender Teilgraph des ungerichteten Graphen G und zugleich ein Baum, dann heißt T *spannender Baum von* G.
Ist T_s ein spannender Teilgraph des zugeordneten ungerichteten Graphen D_u von D und zugleich ein Baum, dann heißt T_s *spannender Baum von* D.
Ist T_g ein gerichteter spannender Teilgraph des gerichteten Graphen D und zugleich ein gerichteter Baum, dann heißt T_g *gerichteter spannender Baum von* D.

In Abbildung 3.6 ist ein gerichteter spannender Baum T_g eines gerichteten Graphen D dargestellt.

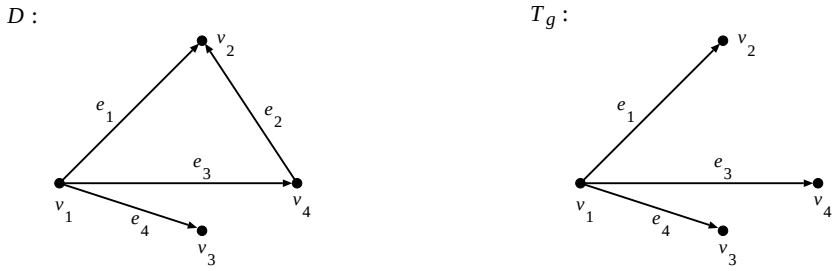


Abbildung 3.6: Gerichteter Graph D mit gerichtetem spannenden Baum T_g

Satz 1 :

Gegeben sei ein Graph $G = (V, E)$. G besitzt einen spannenden Baum genau dann, wenn G zusammenhängend ist.

Der Beweis dieses Satzes kann in [DöMü73] nachgelesen werden.

Definition 28 : Spannender Wald

Sei $G = (V, E)$ ein nichtzusammenhängender Graph mit k Zusammenhangskomponenten. W_s heißt *spannender Wald von* G, falls W_s aus k spannenden Bäumen besteht und jeder davon ist spannender Baum einer anderen Zusammenhangskomponente.

In der Graphentheorie werden bei vielen Problemen spannende Bäume untersucht. Insbesondere die Frage nach minimalen spannenden Bäumen ist von großem Interesse.

Definition 29 : Minimaler spannender Baum

$G = (V, E)$ sei ein endlicher zusammenhängender ungerichteter Graph. Jede Kante $e = \{v_i, v_j\} \in E$ sei bewertet mit dem Gewicht $w(e) = w_{ij}$, $i, j \in \mathbb{N}$.
$T = (V, E')$ heißt *minimaler spannender Baum von* G, falls

1. T ein spannender Baum von G ist und
2. $\sum_{e \in E'} w(e) \leq \sum_{e \in E''} w(e)$

für jeden anderen spannenden Baum $T' = (V, E'')$ mit $E', E'' \subseteq E$.

Definition 30 : Minimaler spannender Wald

Sei $G = (V, E)$ ein endlicher nichtzusammenhängender ungerichteter Graph. Jede Kante $e = \{v_i, v_j\} \in E$ sei bewertet mit dem Gewicht $w(e) = w_{ij}$, $i, j \in \mathbb{N}$.
$W = (V, E')$ heißt *minimaler spannender Wald von* G, falls

1. W ein spannender Wald von G ist und
2. $\sum_{e \in E'} w(e) \leq \sum_{e \in E''} w(e)$

für jeden anderen spannenden Wald $W' = (V, E'')$ mit $E', E'' \subseteq E$.

3.4 Darstellung von Graphen

Für die Darstellung eines Graphen gibt es zwei Möglichkeiten, die gleichfalls geeignete Formen für die Abspeicherung in Rechnern liefern: Matrizen und Listen.

3.4.1 Matrizen

Unterschieden werden kann hier die Inzidenz- von der Adjazenzmatrix. Die Inzidenzmatrix stellt die Inzidenz von Knoten und Kanten dar, wohingegen die Adjazenzmatrix die Adjazenz von Knoten zueinander wiedergibt.

Definition 31 : Inzidenzmatrix

Gegeben sei ein ungerichteter Graph $G = (V, E)$. Die Knoten und Kanten von G seien fest numeriert, also $V = \{v_1, ..., v_n\}$ und $E = \{e_1, ..., e_m\}$, $n, m \in \mathbb{N}$.
Die *Inzidenzmatrix* $I(G) = (x_{i,j})$ *des ungerichteten Graphen* G zur gegebenen Numerierung von V und E ist eine $n \times m$-Matrix, wobei die $x_{i,j}$, $i, j \in \mathbb{N}$, definiert sind durch

$$x_{i,j} = \begin{cases} 1\,, & \text{falls } v_i \text{ und } e_j \text{ inzident sind} \\ 0\,, & \text{sonst.} \end{cases}$$

Damit entsprechen die Spalten von $I(G)$ den Kanten und die Zeilen den Knoten. Jede Spalte enthält genau zwei Einsen, sonst Nullen. Die Summe der Einsen pro Zeile gibt den Grad des Knotens an.

Gegeben sei nun ein gerichteter Graph $D = (V, E)$ mit numerierter Knoten- und Kantenmenge wie oben. Die *Inzidenzmatrix* $I(D) = (x_{i,j})$ *des gerichteten Graphen* D ist definiert durch

$$x_{i,j} = \begin{cases} 1\,, & \text{falls } e_j = (v_i, y) \in E \\ -1\,, & \text{falls } e_j = (y, v_i) \in E \\ 0\,, & \text{sonst} \end{cases}$$

für $v_i, y \in V$, $i, j \in \mathbb{N}$.

Der Speicherplatzbedarf für die Darstellung eines Graphen $G = (V, E)$ durch eine Inzidenzmatrix beträgt $O(|V| \cdot |E|)$.

Definition 32 : Adjazenzmatrix

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit der numerierten Knotenmenge $V = \{v_1, ..., v_n\}$, $n \in \mathbb{N}$. Die *Adjazenzmatrix* $A(G) = (x_{i,j})$, $i, j \in \mathbb{N}$, *des ungerichteten Graphen* G ist eine $n \times n$-Matrix, wobei die $(x_{i,j})$ definiert sind durch

$$x_{i,j} = \begin{cases} 1\ , & \text{falls } v_i \text{ und } v_j \text{ adjazent sind} \\ 0\ , & \text{sonst.} \end{cases}$$

Der Speicherplatzbedarf einer Adjazenzmatrix beträgt $O(|V|^2)$. Die Adjazenzmatrix eines ungerichteten Graphen ist immer symmetrisch. Es genügt daher, nur die Elemente über der Hauptdiagonalen zu betrachten, was zu einer Verringerung des Speicherplatzes führt. Die Elemente der Hauptdiagonalen sind gleich Null. Die Definition einer Adjazenzmatrix eines gerichteten Graphen ist analog. Auch hier ist die Hauptdiagonale mit Nullen besetzt. Die Matrix ist in diesem Fall jedoch nicht symmetrisch. Es wird an der Stelle (i, j) nur dann eine 1 eingetragen, falls v_i Anfangs- und v_j Endknoten einer Kante ist. Existiert keine antiparallele Kante dazu, so steht an der Stelle (j, i) eine 0. Die Anzahl der Einsen in einer Adjazenzmatrix eines gerichteten Graphen ist somit gleich der Anzahl der Kanten des Graphen.

Definition 33 : Gewichtsmatrix

Gegeben sei ein bewerteter ungerichteter Graph $G = (V, E)$. Jede Kante $e = \{v_i, v_j\}$, $e \in E$, $i, j \in \mathbb{N}$, sei bewertet mit dem Gewicht $w(e) = w_{ij}$. Die *Gewichtsmatrix* $W(G) = (w_{i,j})$ ist definiert durch

$$w_{i,j} = \begin{cases} w_{ij}\ , & \text{falls } v_i \text{ und } v_j \text{ adjazent sind} \\ 0\ , & \text{sonst.} \end{cases}$$

Die Definition der Gewichtsmatrix eines gerichteten Graphen läuft analog.

Die Speicherung eines Graphen mittels einer Adjazenzmatrix bietet den Vorteil des direkten Zugriffs auf die Elemente (i, j) in der Matrix. Für Algorithmen, die Kanten einfügen, entfernen oder deren Existenz in einem Graphen oft nachprüfen müssen, ist diese Darstellungsform die günstigste.

3.4.2 Listen

Eine andere Darstellungsform für Graphen ist die Liste. Insbesondere bei Graphen, die im Vergleich zur Anzahl der Knoten nur relativ wenige Kanten besitzen, ist eine Liste vorzuziehen. Denn der Speicherplatzbedarf einer Adjazenzmatrix ist im Fall dieser *dünnbesetzten Matrix* verhältnismäßig groß.

Die Kanten des Graphen werden mit Hilfe zweier Felder, g und h, als Paare von Knoten aufgelistet. Jedes Feldelement g_i und h_i, $1 \leq i \leq m$, $i, m \in \mathbb{N}$, steht für einen Knoten. Die i-te Kante eines Graphen besitzt folglich als Endpunkte die in g_i und h_i abgespeicherten Knoten [RND77].

Eine andere Darstellungsart mittels Listen ist die durch Adjazenzlisten.

Definition 34 : Adjazenzliste

Eine *Adjazenzliste A_i für einen Knoten $v_i \in V$* ist eine Liste aller Knoten $v_j \in V$, $i, j \in \mathbb{N}$, die adjazent zu v_i sind.

Ein durch Adjazenzlisten dargestellter Graph $G = (V, E)$ mit $V = \{v_1, ..., v_n\}$ wird demnach spezifiziert durch die Angabe der Zahl n und die Angabe von n Adjazenzlisten, die für jeden Knoten seine zu ihm adjazenten Knoten enthalten.
Der Speicherplatzbedarf bei der Darstellung eines Graphen durch Adjazenzlisten beträgt $O(|V| + |E|)$. Die Form der Abspeicherung ist für alle Operationen von Vorteil, die alle von einem Knoten ausgehenden Kanten untersuchen.
In Abbildung 3.7 sind für den zusammenhängenden Graphen G aus Abbildung 3.3 a) die Inzidenzmatrix $I(G)$, die Adjazenzmatrix $A(G)$ und die Adjazenzlisten A_i für $i = 1, ..., 5$ dargestellt.

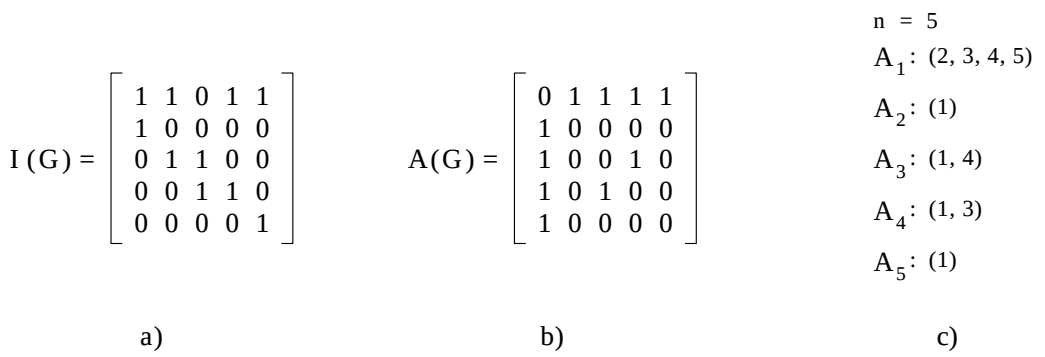


Abbildung 3.7: a) Inzidenzmatrix $I(G)$, b) Adjazenzmatrix $A(G)$ und c) Adjazenzlisten A_i des Graphen G

Kapitel 4

Traveling-Salesman-Problem

Das **T**raveling-**S**alesman-**P**roblem (**TSP**) oder Problem des Handlungsreisenden ist das bekannteste Problem der kombinatorischen Optimierung. Diese beschäftigt sich mit der Frage: Welche aus einer großen Menge von Kombinationen ist die beste, das heißt die kürzeste, die billigste oder die effizienteste? Die Thematik des Traveling-Salesman-Problems wird in [LLRS86] ausführlich behandelt.

Gegeben sei eine feste Anzahl von Städten oder Punkten, die paarweise direkt miteinander verbunden sind. Für jede Verbindung entstehen Kosten, wie sie z.B. durch die natürliche Entfernung von Städten vorgegeben ist. Diese Kosten werden in einer Matrix, der sogenannten *Kostenmatrix*, aufgeführt. Aufgabe des TSP ist es, eine Tour minimaler Länge bzw. Kosten zu finden, die alle Punkte genau einmal berührt und zum Ausgangspunkt zurückführt.
Das TSP ist in verschiedenen Varianten formuliert worden [CMTS79]. Man unterscheidet zwischen symmetrischen und asymmetrischen Kostenmatrizen. Bei symmetrischer Kostenmatrix fallen für Hin- und Rückweg von einem Punkt zu einem anderen gleiche, bei asymmetrischer Kostenmatrix unterschiedliche Kosten an. Weiter unterscheidet man das TSP für euklidische und nicht-euklidische Entfernungen. In dieser Arbeit soll nur das **e**uklidische **T**raveling-**S**alesman-**P**roblem (**ETSP**) mit symmetrischer Kostenmatrix betrachtet werden.

Probleme wie das ETSP werden mit Hilfe von *Graphen* modelliert. Den Punkten oder Städten entsprechen intuitiv die *Knoten* des Graphen, den Verbindungen die *Kanten*. Graphen werden u. a. durch *Adjazenzmatrizen* dargestellt, deren Einträge an den Stellen (i, j) anzeigen, ob die Punkte i und j durch eine Kante miteinander verbunden sind. Jede dieser Kanten besitzt im ETSP einen Kostenfaktor. Dieser Kostenfaktor fließt in die Adjazenzmatrix als Kantenbewertung mit ein. Solch ein kantenbewerteter Graph wird dann durch eine bewertete Adjazenzmatrix, der *Gewichtsmatrix* W, dargestellt. An der Stelle (i, j) der Gewichtsmatrix W stehen dann die *Gewichte* oder *Kosten* w_{ij} der Kante $\{i, j\}$, im vorliegenden Fall also die Entfernung vom Punkt i nach Punkt j.

Das ETSP läßt sich mathematisch wie folgt formulieren: In einem zusammenhängenden, kantenbewerteten Graphen G ist ein Kreis K mit minimalem Gewicht $W(K)$ gesucht, der jeden Knoten des Graphen genau einmal trifft, also ein *Hamiltonkreis*. Die Beschränkung auf euklidische Graphen verlangt, daß alle Knoten im euklidischen Raum eingebettet liegen und die *Dreiecksungleichung*

$$w_{ij} \leq w_{ik} + w_{kj}$$

für die Verbindungen zwischen je drei Knoten i, j und k erfüllt ist.

Neben der *Aufzählung* als "brutale" Methode und verschiedene *Branch-and-Bound*-Verfahren [LMSK63] (parallelisiert durch Mohan [Moh82] und [Moh83]) kann zur exakten Bestimmung der optimalen Tour ein *Lineares Programm* [Sed89] aufgestellt werden.
Die Aufzählung zählt einfach alle existierenden Möglichkeiten auf ($(n-1)!$ Kombinationen bei n Städten).
Branch-and-Bound-Verfahren beginnen mit allen möglichen Touren, die jedoch im einzelnen unbekannt sind. Für diese Menge werden die minimalen Kosten bestimmt. Die Lösungsmenge wird nun in zwei disjunkte Teilmengen aufgespalten. Eine Lösung bedeutet, daß eine Tour durch alle Knoten ohne Berücksichtigung der Kosten gebildet wird. Eine Teilmenge ist dadurch charakterisiert, daß alle darin zusammengefaßten Lösungen eine Kante enthalten, die die Lösungen der anderen Teilmenge nicht enthalten. Die Teilmengen sind Knoten eines Zustandsbaumes. Für sie wird wiederum eine untere Schranke bestimmt. Es wird solange weiterverzweigt, bis eine Tour gefunden ist, deren Kosten kleiner oder gleich den unteren Schranken aller anderen Teilmengen ist [Gur86].
Die zu minimierende Zielfunktion des Linearen Programms ist die Summe der Bewertungen der zur Lösungstour gehörenden Kanten. Neben der Dreiecksungleichung sind in weiteren Nebenbedingungen die Beziehungen der Knoten und Kanten zueinander aufgeführt. Eine wichtige Restriktion ist, daß alle Knoten nur genau einmal in der Lösungstour enthalten sein dürfen. Das Lineare Programm kann mittels des *Simplex-Algorithmus* [Zim87] oder der *Kamarkar-Methode* [GrTe93] gelöst werden. Der Simplex-Algorithmus sucht die Ecken des Lösungsraumes nach der optimalen Lösung ab. Im Gegensatz dazu nähert sich die Kamarkar-Methode aus dem Innern des Lösungsraumes durch Angabe einer Richtung und einer Schrittweite dem auf einer Ecke liegenden Optimum an.
Die Zahl und der Inhalt der Nebenbedingungen verursachen die Zugehörigkeit des ETSP in die Klasse der NP-vollständigen Probleme. Das heißt, es ist kein Algorithmus bekannt, der das Problem in polynomialer Zeit bezüglich der Problemgröße löst. Schon bei kleinen Problemgrößen steigt der exponentiell wachsende Rechenaufwand so stark an, daß kein heute verfügbarer Rechner das Problem in vernünftiger Zeit lösen kann. Deshalb sind eine Reihe von Heuristiken entwickelt worden, die in einem vertretbaren, das heißt polynomialen Zeitaufwand sehr nahe an das Optimum heranreichende Lösungen für das ETSP liefern.

Man unterscheidet drei Gruppen von Heuristiken:

- **Tour-Konstruktionsalgorithmen**
  Tour-Konstruktionsalgorithmen bauen eine Tour Knoten für Knoten auf und verwenden dabei eine lokale Optimierungsstrategie. Zu dieser Gruppe zählen u.a. *Nearest-Neighbour*-Heuristik [ObSc92], *Farthest-Insertion*-Heuristik (Heuristik des weitesten Einfügens) [Jun87] und *Christofides*-Algorithmus [PaSt82]. Bei der Nearest-Neighbour-Heuristik wird der Knoten in die Tour eingefügt, der zu dieser die geringste Entfernung besitzt. Im Gegensatz dazu wird bei der Farthest-Insertion-Heuristik derjenige Knoten hinzugenommen, der am weitesten von der bisher gebildeten Tour entfernt ist (siehe Abschnitt 4.2). Beide Verfahren besitzten eine Komplexität von $O(n^2)$ bei n Knoten.
  Eine vollkommen andere Vorgehensweise besitzt der Christofides-Algorithmus. Zuerst wird ein minimaler spannender Baum des Graphen gesucht, mit dem das ETSP modelliert wurde. Zu diesem minimalen spannenden Baum muß ein perfektes Matching minimalen Gewichtes gefunden werden, welches nur aus den Knoten ungeraden Grades gebildet wird. Die Kanten des Matchings werden mit denen des minimalen spannenden Baumes vereinigt. In dem so entstandenen Graphen wird ein Eulerkreis gebildet. Der Christofides-Algorithmus besitzt bei n Knoten eine Komplexität von $O(n^3)$ (wegen der Berechnung des Matchings) und garantiert eine Lösung, die maximal um den Faktor 1.5 schlechter ist als die optimale.

- **Tour-Verbesserungsalgorithmen**
  Algorithmen dieser Gruppe beginnen mit einer beliebigen Tour und minimieren deren Länge durch wiederholtes Austauschen einer bestimmten Anzahl von Kanten. Dabei werden auch schlechtere Zustände zugelassen. Wird durch den Austausch der Kanten die Tourlänge nach einer bestimmten Zeit nicht mehr verbessert, stoppt der Algorithmus. Mit dieser Vorgehensweise wird verhindert, daß der Algorithmus in lokalen Optima endet. Beispiele für Tour-Verbesserungsalgorithmen sind *Simulated Annealing* [ObSc92], *Sintflut-Algorithmus* [DSW93] und *Threshold Accepting* [DSW93]. Beim Simulated Annealing wird mit einer zufällig ausgewählten Schranke entschieden, ob eine Verschlechterung akzeptabel ist oder nicht. In [MGPO89] ist Simulated Annealing für das TSP parallelisiert worden. Der Sintflut-Algorithmus erlaubt eine Verschlechterung bis zu einem bestimmten Schwellenwert, wohingegen beim Threshold Accepting eine Verschlechterung in einem festen Wertebereich toleriert wird.

- **Partitionsalgorithmen**
  Die Partitionsalgorithmen unterteilen ein gegebenes Problem in mehrere Teilprobleme und lösen diese mit Hilfe einer anderen Heuristik. Sind die verbliebenen Probleme klein, so kann ein exakter Algorithmus benutzt werden. Die Gesamttour wird schließlich durch Kombination der Ergebnisse der Subprobleme berechnet. In die Gruppe der Partitionsalgorithmen fällt der *Unterteilungsalgorithmus von Karp* [Kar77].

4.1 Unterteilungsalgorithmus von Karp

Der Unterteilungsalgorithmus von Karp ist eine rekursive Methode zur Bestimmung einer Näherungslösung des ETSP. Er gehört zur Gruppe der Partitionsalgorithmen, bei denen das Originalproblem in Teilprobleme unterteilt wird, die getrennt voneinander gelöst und deren Ergebnisse zu einer Gesamttour kombiniert werden. Im folgenden werden die Unterteilung auch als Partitionierung und die Teilprobleme als Partitionen bzw. Subpartitionen bezeichnet.

Die Vorgehensweise des Unterteilungsalgorithmus von Karp basiert auf der Annahme, daß jeder Nachfolger eines Knotens räumlich nahe bei diesem liegt. Das zu lösende ETSP kann somit "geographisch" in zwei Teilprobleme unterteilt werden, die unabhängig voneinander, aber in gleicher Weise, weiterbehandelt und schließlich gelöst werden. Eine Menge von Knoten, die sortiert nach x- oder y-Koordinaten vorliegt, wird in zwei Teilmengen unterteilt. Der einzige Knoten, der in beiden Teilmengen enthalten ist, ist der Knoten mit dem mittleren x- oder y-Wert. Dieser Unterteilungsprozeß wird rekursiv weitergeführt, bis ein Teilproblem entstanden ist, dessen Größe auf ein beherrschbares Niveau gesunken ist. Die Zahl der Knoten, die maximal in einer Partition enthalten sein dürfen, wird zu Anfang festgelegt. Haben die Teilprobleme die geforderte Größe erreicht, werden sie mit Hilfe eines anderen Algorithmus, z.B. der Heuristik des weitesten Einfügens, in polynomialer Zeit näherungsweise gelöst. Man erhält eine Menge von Teillösungen, die entsprechend der Unterteilung des Originalproblems- in Teilprobleme, miteinander kombiniert werden. Die Touren zweier Teilprobleme werden miteinander vereinigt, indem zwei Kanten entfernt werden, die mit dem gemeinsamen mittleren Knoten inzident sind. Geschlossen wird die Tour dann mit einer Kante, die mit den Knoten inzident ist, an denen die beiden Kanten entfernt wurden. So erhält man eine Näherungslösung für das aufwendige Originalproblem [Gur86].
Der Unterteilungsalgorithmus von Karp stellt eine gute Wahl dar für die Bestimmung einer suboptimalen Lösung eines sehr großen euklidischen Traveling-Salesman-Problems. Der Grund dafür scheint klar. Der Aufwand für das Unterteilen der Probleme und das anschließende Kombinieren der Teillösungen ist wesentlich geringer als der für das Lösen des einen großen Originalproblems anstelle der vielen kleinen Teilprobleme. Es ist aber zu erwarten, daß durch das Aufteilen des Originalproblems die resultierende Tourlänge in dem Maße größer wird, je feiner die Unterteilung war. Dies ist intuitiv klar, denn bei feinerer Aufteilung wird die Auswahl der Knoten für die zu bildende Lösungstour einer Partition geringer. Zwangsläufig vergrößert sich dadurch die Länge einer Teil- und damit auch der Gesamttour. Der deutliche Gewinn an Rechengeschwindigkeit überwiegt jedoch gerade bei großen Problemen den Nachteil der steigenden Tourlänge.

Die Strategie, der Karps Unterteilungsalgorithmus folgt, ist die des *Divide-and-Conquer* [Sed89]. Gerade die Möglichkeit der unabhängigen Behandlung der Teilprobleme macht diesen Unterteilungsalgorithmus attraktiv zur Lösung eines ETSP mit Hilfe eines massiv-parallelen Rechners. Die voneinander unabhängigen Prozessoren können auf allen Ebenen der Unterteilung die einzelnen Teilprobleme bearbeiten.

Das folgende Beispiel zeigt ein in der Fachwelt bekanntes Traveling-Salesman-Problem. Es wurde 1984 von Grötschel publiziert und wird im weiteren als *Grötschel-Problem* bezeichnet [DSW93].
In eine Leiterplatine sollen an 442 fest vorgegebenen Punkten Löcher gebohrt werden. Der Gesamtweg des Bohrkopfes von Loch zu Loch soll möglichst kurz sein. Der Start- und Endpunkt hat die Koordinaten $x = 0$ und $y = 0$. Die Entfernungen der Löcher zueinander und damit auch die Länge der Ergebnistour wird in inch angegeben.
Ausgehend von der Verteilung der Bohrpunkte in der euklidischen Ebene der Leiterplatine (Abbildung 4.1) wird das Originalproblem nach der *Divide-and-Conquer*-Strategie rekursiv absteigend in acht etwa gleichgroße Partitionen unterteilt (Abbildung 4.2). Die Unterteilungsrichtung wechselt dabei in jedem Partitionierungsschritt und beginnt mit der in y-Richtung. Jede Partition wird einzeln für sich gelöst (Abbildung 4.3). Im folgenden werden die Teillösungen entsprechend der vorherigen Unterteilung zu einer Gesamtlösung kombiniert (Abbildung 4.4).

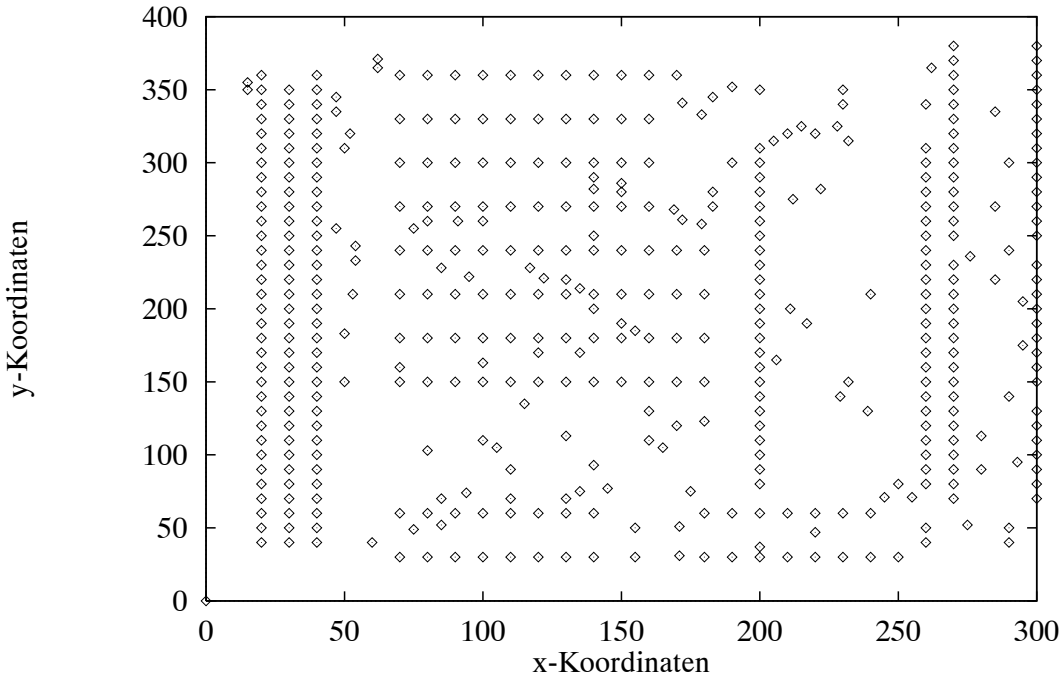


Abbildung 4.1: Verteilung der Bohrpunkte des Grötschel-Problems

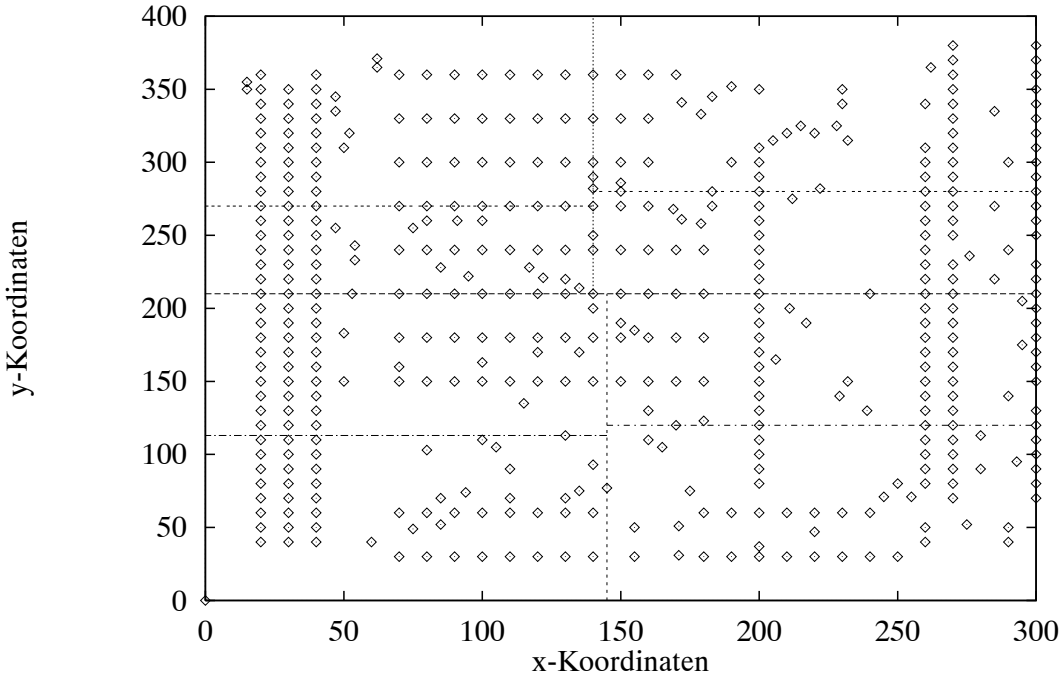


Abbildung 4.2: Unterteilung des Grötschel-Problems in acht Partitionen

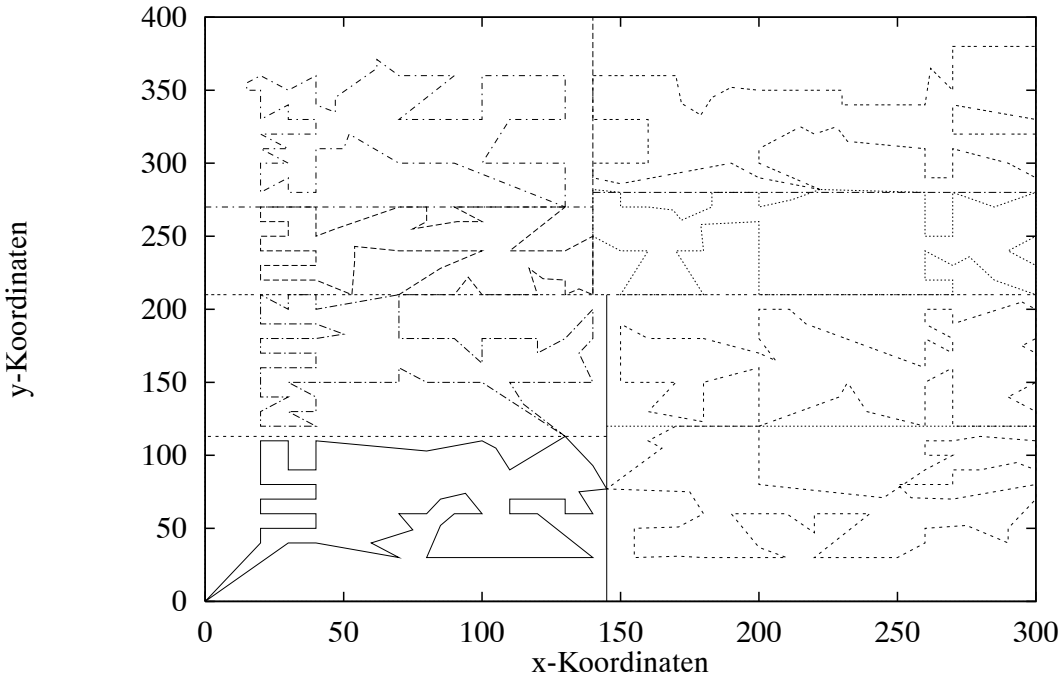


Abbildung 4.3: Teillösungen der acht Partitionen des Grötschel-Problems

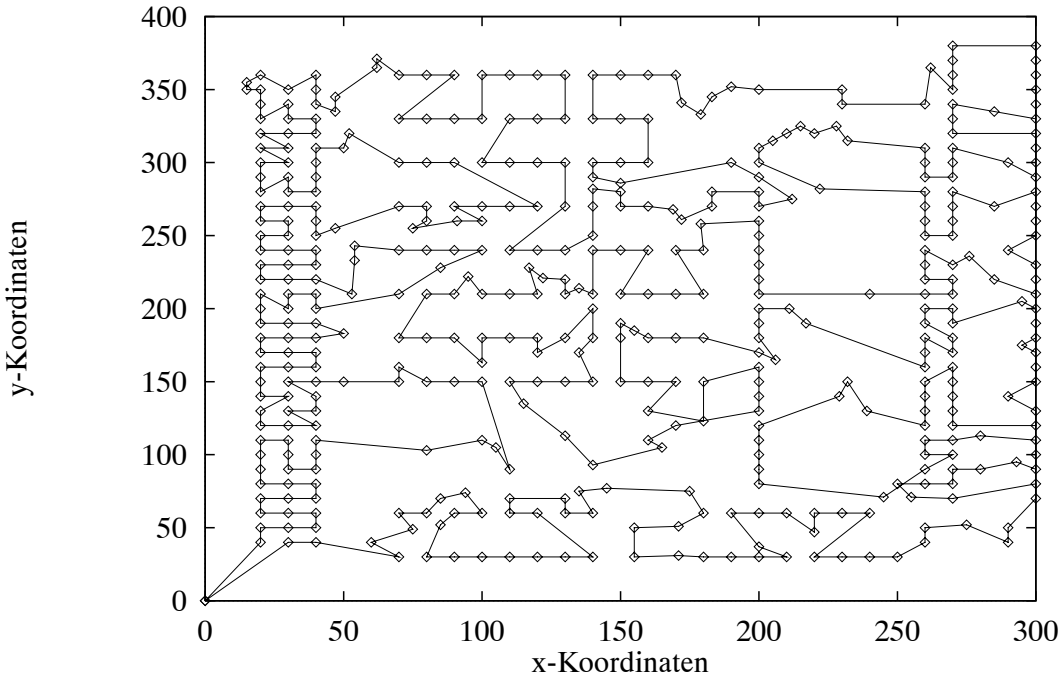


Abbildung 4.4: Lösung des Grötschel-Problems bei acht Partitionen mit einer Länge von 61.09 inch

4.2 Heuristik des weitesten Einfügens

Die Heuristik des weitesten Einfügens (*Farthest Insertion*) ist, wie oben erwähnt, ein Tour-Konstruktionsalgorithmus. Er geht von der Idee aus, die gesuchte Tour durch sukzessives Hinzufügen des am weitesten entfernten Knotens von der bisher konstruierten Teiltour zu bestimmen. Dieser Idee liegt die Annahme zugrunde, daß das grobe Aussehen der Tour durch weit auseinanderliegende Knoten konstruiert werden kann und deren feinere Details aus der Hinzunahme nahe liegender Knoten resultieren. Denn durch die Hinzunahme von in der Nähe liegender Knoten wird die Tour im Vergleich zur Hinzunahme weit entfernter nur unwesentlich verlängert.

Die Heuristik kann wie folgt beschrieben werden: Der Ausgangsknoten oder die **Quelle** (Source) wird willkürlich gewählt. Nun wird der Knoten bestimmt, dessen Entfernung von der Quelle maximal ist und eine Tour von der Quelle zu diesem Knoten und zurück gebildet. Im weiteren Verlauf wird zu der bereits gebildeten Teiltour immer derjenige Knoten hinzugefügt, dessen Entfernung maximal von allen in der Teiltour enthaltenen ist. Er wird so eingefügt, daß die neu entstandenen Tourkosten minimal werden. Bei n Knoten wird dieser Schritt $(n-2)$-mal wiederholt.
Für symmetrische euklidische Graphen, die die Dreiecksungleichung erfüllen, findet die Heuristik des weitesten Einfügens suboptimale Touren, deren Längen bis zu 2% schlechter sind als die minimal mögliche Tourlänge [Qui83]. Die Zeitkomplexität dieser Heuristik beträgt $O(n^2)$ und wird von anderen Näherungsalgorithmen kaum unterboten [Qui83]. Nur kürzlich entwickelte Heuristiken wie der Sintflut-Algorithmus oder das Threshold Accepting erreichen eine Tour, deren Länge um 0,5% über der optimal möglichen liegen [DSW93].
Aus diesen Gründen läßt sich die Heuristik des weitesten Einfügens sehr gut für die Bestimmung einer Näherungslösung für das euklidische Traveling-Salesman-Problem nutzen, falls auf eine sequentielle Implementation, wie in der vorliegenden Arbeit, zurückgegriffen werden muß.

Abbildung 4.5 zeigt ein Beispiel für die Vorgehensweise der Heuristik des weitesten Einfügens. Willkürlich wird Knoten 3 als Quelle gewählt a). Knoten 6 besitzt die weiteste Entfernung zu Knoten 3. Mit ihm wird die erste Teiltour gebildet b). Zu einer bereits gebildeten Teiltour wird nun sukzessive immer der zu dieser Teiltour am weitesten entfernte Knoten hinzugenommen, bis alle Knoten in der Tour enthalten sind c) – f).

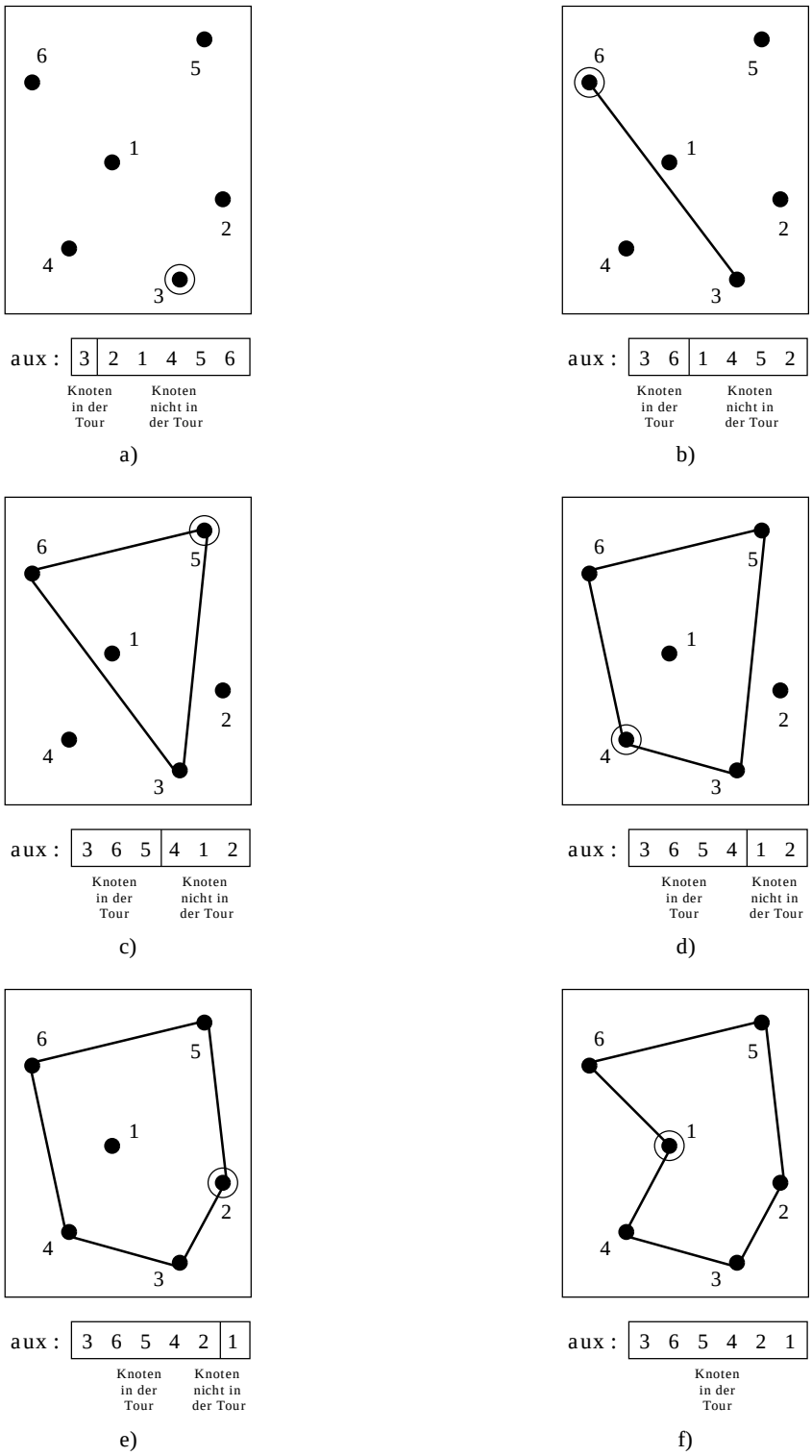


Abbildung 4.5: Vorgehensweise bei der Heuristik des weitesten Einfügens

4.3 Implementation auf iPSC/860 und Paragon

Die in dieser Arbeit gewählte Form der Implementation weicht in einem Punkt von der oben (Abschnitt 4.1) beschriebenen Variante des Algorithmus ab. Die Anzahl der beherrschbaren Partitionen ist dort abhängig von der frei wählbaren maximalen Anzahl von Knoten in einer Partition des untersten Niveaus. Das heißt, es wird solange partitioniert, bis diese Knotenanzahl erreicht oder unterschritten wird. In der hier vorliegenden Implementation ist die Partitionsgröße nun abhängig von der Anzahl der Partitionen. Die Anzahl der zu bildenden Partitionen wird gleich der Anzahl der zur Verfügung stehenden Prozessoren gewählt. Das erstellte dynamische Programm liest zu Anfang die Zahl der Prozessoren ein und unterteilt das Originalproblem rekursiv absteigend in ebenso viele Partitionen.
Als Eingabe erhält der Algorithmus die Koordinaten der Knoten, die Ausgabe besteht aus der Länge der Tour und die Reihenfolge der Tourknoten.

4.3.1 Paralleler Unterteilungsalgorithmus von Karp

Das hier vorgestellte parallele Programm, welches den Unterteilungsalgorithmus von Karp realisiert, wurde direkt, das heißt ohne vorherige Implementation eines sequentiellen Programms für die beiden in Kapitel 2 vorgestellten massiv-parallelen Rechner iPSC/860 und Paragon entwickelt. Dabei sind zwei Arten der Kommunikation implementiert und untersucht worden. Das im nächsten Unterabschnitt beschriebene sequentielle Programm wurde für Vergleichszwecke auf der Basis des parallelen Programms entworfen. Im weiteren Verlauf der Beschreibung werden die Begriffe Partitionen und Prozessoren für die parallele Implementation gleichgesetzt.

Im Unterteilungsalgorithmus von Karp ist ein impliziter *Datenparallelismus* enthalten. Denn die Partitionierung eines Problems in zwei Teilprobleme beinhaltet als virtuelle Verzweigungstopologie einen Binärbaum, der sich sehr gut in die Hypercube-Topologie des iPSC/860 einbetten läßt. Da die Gitter-Struktur des Paragon-Verbindungsnetzwerkes keine Restriktionen bezüglich der logischen räumlichen und zahlenmäßigen Festlegung der Rechenknoten einer Partition vorsieht, konnte auch dort der virtuelle Binärbaum auf die Rechenknoten abgebildet werden. Die durch mit dem Fortschreiten der Verzweigungstiefe in diesem Binärbaum fehlende dynamische Allokation der Rechenknoten bedingt eine nicht vollständige Balancierung der Rechenlast auf die einzelnen Rechenknoten. Eine verbesserte Lastbalancierung benötigt jedoch ein verstärktes Maß an Kommunikation.
Aufgrund der Ergebnisse in Abschnitt 2.4 mußte jedoch eine Form der Implementation gefunden werden, bei der die Kommunikation so wenig wie möglich zur Ausführungszeit beiträgt. Der rekursive Aufbau des Unterteilungsalgorithmus von Karp mußte wegen der Programmiersprache (FORTRAN) und der Realisation des Parallelismus iterativ nachgebildet werden. Dafür wurden zwei Schleifen implementiert, deren gleiche Anzahl k von Durchläufen sich aus dem Zweierlogarithmus der vorgegebenen Zahl p der Prozessoren bestimmt. Der Ablauf des parallelen

Programms läßt sich an einer Art Binärbaum (Schema in Abbildung 4.6, für 4 Prozessoren bzw. Partitionen) verfolgen. Der Binärbaum wird im weiteren als *Unterteilungsschema* bezeichnet, das in drei Phasen unterteilt ist. Die erste, die **Unterteilungsphase**, wird durch die erste der oben erwähnten Schleifen realisiert. Die **Lösungsphase** als zweite, wird mit Hilfe eines Unterprogramms parallel durchlaufen, und die **Kombinationsphase** als letzte, wird durch die oben erwähnte zweite Schleife realisiert.

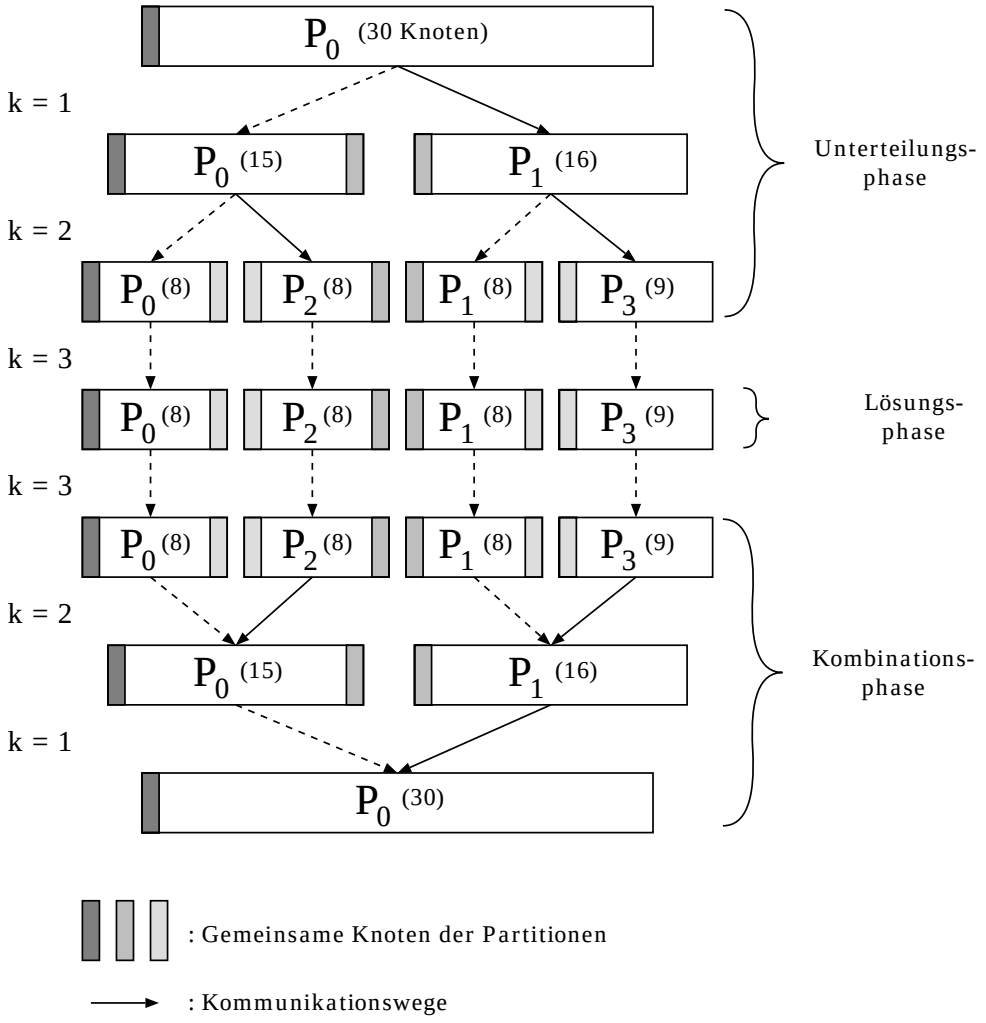


Abbildung 4.6: Unterteilungsschema der parallelen Implementation

Die zu einer Tour zu kombinierenden Knoten des Originalproblems werden dargestellt durch x- und y-Koordinaten in der euklidischen Ebene. Diese Koordinaten stehen in den Feldern *xloc* bzw. *yloc* zur Verfügung.
Die Unterteilungsrichtung einer Partition ist nun abhängig von der räumlichen Verteilung der x- und y-Koordinaten. Es wird immer in die Richtung unterteilt, in der die Verteilung am größten ist. Die Richtung der Unterteilung ist stark abhängig von der Eingabe. Bei Eingabe einer Menge von in einem Quadrat gleichverteilten

Knoten wechselt die Unterteilungsrichtung nach jedem Partitionierungsschritt. Liegen die Knoten jedoch in einem Band, so wird bei jedem Partitionierungsschritt die Unterteilungsrichtung beibehalten.
Vor der Unterteilung müssen die Koordinaten sortiert und deren Position in die Felder *xsort* und *ysort* abgespeichert werden. Zusätzlich benötigt der Algorithmus zur korrekten Unterteilung einer Partition zwei Zeigerfelder, *xassoc* und *yassoc*, die die Positionen der Knoten im jeweiligen anderen sortierten Feld enthalten.
Die exakten Koordinaten des Knotens l seien in $xloc(l)$ und $yloc(l)$ gespeichert. Nach Sortierung aller Koordinaten nimmt Knoten l in *xsort* die Position i, in *ysort* die Position j ein, also $xsort(i) = l = ysort(j)$. Die Felder *xassoc* und *yassoc* dienen nun als Zeiger zwischen *xsort* und *ysort*. Die Position j des Knotens l im Feld *ysort*, der in $xsort(i)$ gespeichert ist, wird durch $xassoc(i)$ angegeben, also $xassoc(i) = j$. Analog gilt im umgekehrten Fall $xsort(yassoc(j)) = ysort(j)$.
Jede Partition ist gekennzeichnet durch ihre eigenen *xsort*-, *ysort*-, *xassoc*- und *yassoc*-Felder.

Der parallele Algorithmus arbeitet wie folgt:

Jeder der bei der Berechnung der Problemlösung beteiligten Prozessoren liest in einem ersten Schritt die exakten x- und y-Koordinaten ein und speichert sie in die dafür vorgesehenen Felder ab. Alle Prozessoren kennen somit zu diesem Zeitpunkt nur die Koordinaten der Knoten, durch die eine Tour gebildet werden soll.
Nun bildet der logische erste Prozessor die erste Partition, indem er, ausgehend von den Koordinaten, die sortierten Felder und die Zeigerfelder aufbaut. Diese Aufgabe wird durch das Unterprogramm *BUILD* durchgeführt, welches nur vom logischen ersten Prozessor aufgerufen wird.

Jetzt beginnt mit dem Eintritt in die erste Schleife die Phase der Unterteilung. Im weiteren Verlauf werden die Prozessoren (in iterativ aufsteigender Folge der Schleifendurchläufe) in **Sender** und **Empfänger** unterschieden. Sender ist ein Prozessor dann, falls er eine Partition besitzt, von der eine Hälfte an einen anderen Prozessor abgegeben wird.
Alle in einem Schleifendurchlauf als Sender gekennzeichneten Prozessoren besitzen die eine Partition darstellenden Daten in den Feldern *xsort*, *ysort*, *xassoc* und *yassoc*. Diese Partition wird vom Sender in zwei Subpartitionen unterteilt. Dies leistet das Unterprogramm *DIVIDE*. Es entscheidet, ob die Partition in x- oder in y-Richtung aufgeteilt wird. Ist die Unterteilungsrichtung festgelegt, werden die neuen Subpartitionen aufgebaut. Der mediane Knoten der Vaterpartition wird bestimmt und dessen Knotenmenge gleichmäßig in zwei Teilmengen aufgeteilt. Dazu ruft *DIVIDE* das Unterprogramm *SETUP* auf. *SETUP* speichert das sortierte Feld, in dessen Richtung unterteilt wird, so um, daß der mediane Knoten doppelt vorkommt. Er steht in der Mitte des neuen Feldes zweimal hintereinander. Die alte Reihenfolge der Knoten bleibt erhalten. Der mediane Knoten wird als **Quelle** bezeichnet. Dieser gemeinsame Knoten wird für die Kombination der Teiltouren benötigt. In Abbildung 4.6 sind die gemeinsamen Knoten der Subpartitionen markiert. Durch das von *SETUP* aufgerufene Unterprogramm *COPY* werden nun das andere sortierte Feld und die Zeigerfelder so aufgebaut, daß dort ebenfalls ein Element doppelt vorkommt.

Nachdem alle Felder neu aufgebaut wurden, besitzen die Sender zwei Partitionen, deren Daten in den jeweiligen Feldern hintereinander stehen. Die erste Hälfte eines Feldes wird durch die erste, die zweite Hälfte durch die zweite Subpartition belegt. Gemeinsam ist beiden Subpartitionen die Quelle, die zusätzlich noch einmal gesondert abgespeichert wird. Daneben wird die Zahl der Knoten einer neu entstandenen Subpartition ebenfalls zwischengespeichert. Die Quelle und die Knotenanzahl werden für das spätere Lösen der Partitionen und das Kombinieren der daraus entstehenden Teillösungen benötigt.
Nachdem der Sender aus einer Partition zwei Subpartitionen berechnet hat, bestimmt er die Addresse seines ihm, in diesem Schleifendurchlauf k, zugewiesenen Empfängers mittels der Formel

$$empfänger = sender + 2^{k-1}.$$

Die erste Subpartition behält der Sender zwecks Weiterbearbeitung im nächsten Schleifendurchlauf. Die zweite, bestehend aus den zweiten Hälften der Felder *xsort*, *ysort*, *xassoc* und *yassoc* und der Quelle schickt der Sender seinem zugewiesenen Empfänger. Dazu ruft er das Unterprogramm *DOWNSEND* auf. Der Empfänger hingegen empfängt mit dem Unterprogramm *DOWNRECV* die zweite Subpartition. Damit ist ein Schleifendurchlauf der Unterteilungsphase beendet.
Im nächsten Schleifendurchlauf bleiben die Sender Sender und die Empfänger des letzten Durchlaufs werden nun ebenfalls zu Sendern. Ein neuer Unterteilungsschritt beginnt. Die Partitionierung sowie die Kommunikation wird in einem Schleifendurchlauf von den Sendern sowie von den Empfängern parallel durchgeführt. Die Schleife der Unterteilungsphase wird solange durchlaufen, bis alle Prozessoren eine Partition besitzen. Damit ist die Partitionierung beendet und die Lösungsphase kann beginnen.

Jeder Prozessor, der in die Lösungsphase eintritt, beginnt mit dem Lösen seiner Partition, unabhängig davon, ob alle anderen Prozessoren ihre zu lösenden Partitionen bereits besitzen.
Der Algorithmus, mit dem eine Näherungslösung einer Partition berechnet wird, ist die **Heuristik des weitesten Einfügens**. Das von allen Prozessoren parallel aufgerufene Unterprogramm *SOLVE* realisiert diese Heuristik, deren Implementation in Unterabschnitt 4.3.3 näher erläutert wird. Die Lösungsphase wird von allen Prozessoren parallel durchlaufen.
Jede Lösung einer Partition besteht aus der Länge der Teiltour und der Reihenfolge der Knoten, die diese Teiltour bilden. Die Länge ist in *mylen* und die Reihenfolge in *cycle* gespeichert.

In der Kombinationsphase wird die implizit vorhandene Rekursion wieder durch eine iterative Schleife, deren Schleifenindex hier rückwärts läuft, simuliert. Eine Unterscheidung der Prozessoren in Sender und Empfänger wird ebenfalls vorgenommen. Sie geschieht hier umgekehrt zur Unterscheidung in der Unterteilungsphase. Das heißt, der Prozessor, der im letzten Schleifendurchlauf der Unterteilungsphase Sender war, ist nun im ersten Schleifendurchlauf der Kombinationsphase Empfänger und umgekehrt.

Der Sender schickt im Schleifendurchlauf mit Index k die von ihm in der Lösungsphase berechnete Teillösung demjenigen Prozessor, von dem er die zugehörige Subpartition im Schleifendurchlauf mit Index k der Unterteilungsphase erhalten hat. Der Empfängerprozessor berechnet sich durch die Formel (vgl. oben)

$$empfänger = sender - 2^{k-1}.$$

Das Verschicken einer Teillösung wird durch den Sender mittels des Unterprogramms *UPSEND* realisiert. Mit dem Ende dieser Kommunikation tritt er aus der Ermittlung der Gesamttour, das heißt aus der aktiven Phase des Programms, aus.
Der Empfänger empfängt mit dem Unterprogramm *UPRECV* die Daten. Die Länge einer empfangenen Teiltour C speichert er in einer temporären Variable *othlen* zwischen. Die in *cycle* stehende Reihenfolge der Knoten von C wird als zweite Hälfte hinter den Einträgen der eigenen Teiltour B im eigenen *cycle*-Feld abgelegt. Ähnlich wie die Felder der Partitionen in der Unterteilungsphase besitzt *cycle* ebenfalls zwei Hälften, in denen die zu einer neuen Lösung A zu kombinierenden Teillösungen B und C abgespeichert sind.
Die Kombination der Teiltouren wird mit dem Unterprogramm *COMBINE* realisiert. Dafür ist nun die Kenntnis der gemeinsamen Knoten, der Quellen, der entsprechenden Partitionen, wichtig. Für die Kombination zweier Teiltouren gibt es zwei Möglichkeiten. Dazu sind die Knoten vor und nach der Quelle der beiden Teiltouren B und C zu ermitteln. Dann wird der resultierende Kostengewinn berechnet, falls der Knoten vor der Quelle von Partition B mit dem Knoten nach der Quelle von Partition C verbunden wird und die beiden Kanten von den Knoten zur Quelle entfernt werden. Entsprechend werden die Kosten im anderen Fall ermittelt. Die Kombination, bei der die Kostenersparnis am größten ist, wird ausgewählt. Dann wird die Länge der neu entstandenen Tour berechnet und die Reihenfolge der Knoten in *cycle* neu ermittelt. Eine der doppelt vorkommenden Quellen der Partitionen B und C wird entfernt. Die Knoten werden so umgespeichert, daß *cycle* mit der Quelle der folgenden Partition A beginnt und die richtige Reihenfolge der Tourknoten erhalten bleibt. Der Empfänger besitzt nach Ausführen von *COMBINE* in *mylen* die Länge und in *cycle* die Reihenfolge der Tourknoten der neuen Teillösung. Damit ist ein Schleifendurchlauf der Kombinationsphase beendet und die Hälfte der Empfänger wird für einen erneuten Durchlauf zu Sendern. Auch in dieser Phase wird die Kommunikation und die Kombination zweier Teiltouren parallel ausgeführt.
Nach ebenso vielen Schleifendurchläufen wie in der Unterteilungsphase ist die Kombinationsphase abgeschlossen.
Nach Abarbeiten der drei Phasen ist damit eine Näherungslösung für das gegebene Problem gefunden. Die Länge der Lösungstour ist in *mylen* und die Reihenfolge der Tourknoten in *cycle* des logischen ersten Prozessors gespeichert, der auch die Berechnung der Gesamtlösung initiiert hat.

Der folgende Code beschreibt die Implementation des parallelen Unterteilungsalgorithmus von Karp.

```
      PROGRAM KARP
      . . .
      me = MYNODE()
      p  = NUMNODES()
      k  = log(p)
      IF (me .EQ. 0) THEN
         CALL BUILD(n, xkord, ykord, xsort, ysort, xassoc, yassoc)
         sources(0, me)  = xsort(1)
         sindices(0, me) = 1
         elements(0, me) = n
         high            = n
         begintime       = DCLOCK()
      ENDIF

      DO 100 i = 1, k, 1                        ! Unterteilungsphase
         IF (me .LE. ((2 ** (i - 1)) - 1)) THEN             !Sender
            median = INT((high + 1) / 2)
            length = 4 * (INT(high - median) + 1)
            CALL DIVIDE(xkord, ykord, xsort, ysort, xassoc, yassoc,
     >                  source1, source2, sindex1, sindex2,
     >                  median, high)
            address = me + 2 ** (i - 1)
            CALL DOWNSEND(i, xsort, ysort, xassoc, yassoc, source2,
     >                    sindex2, median, length, address)
            high            = median
            elements(i, me) = high
            sources(i, me)  = source1
            sindices(i, me) = sindex1
         ELSEIF ((me .GT. ((2 ** (i - 1)) - 1)) .AND.    !Empfaenger
     >           (me .LE. ((2 ** i) - 1)))          THEN
            CALL DOWNRECV(i, xsort, ysort, xassoc, yassoc,
     >                    source2, sindex2, length)
            high            = length / 4
            elements(i, me) = high
            sources(i, me)  = source2
            sindices(i, me) = sindex2
         ENDIF
  100 CONTINUE

      CALL SOLVE(k, xkord, ykord, xsort, sindices,    !Loesungsphase
     >           elements, cycle, mylen)
```

```
      DO 200 i = k, 1, -1                          !Kombinationsphase
      IF ((me .GT. ((2 ** (i - 1)) - 1)) .AND.              !Sender
     >    (me .LE. ((2 ** i) - 1)))           THEN
         high    = elements(i, me)
         length  = 4 * high
         address = me - 2 ** (i - 1)
         CALL UPSEND(i, cycle, mylen, length, address)
      ELSEIF (me .LE. ((2 ** (i - 1)) - 1)) THEN        !Empfaenger
         high   = elements(i, me)
         median = high
         CALL UPRECV(i, cycle, othlen, median, length)
         length = length / 4
         high   = median + length - 1
         CALL COMBINE(i, xkord, ykord, high, median, length,
     >                sources, cycle, mylen, othlen)
      ENDIF
  200 CONTINUE

      IF (me .EQ. 0) endtime = DCLOCK()

      END
```

Zu Anfang dieses Abschnittes wurde erwähnt, daß zwei parallele Varianten des Unterteilungsalgorithmus von Karp implementiert wurden. Sie sollen im weiteren mit *send* und *packed_send* bezeichnet werden. Die beiden Varianten unterscheiden sich in der Art der Kommunikation.
In der Implementation *send* wurde für jedes zu versendende Feld eine eigene Nachricht erzeugt. Das heißt, es mußten für eine zu verschickende Partition 7 Nachrichten erzeugt, versendet und empfangen werden. Auch wenn dieser Kommunikationsschritt in einem Schleifendurchlauf für jeden Sender bzw. Empfänger parallel geschieht, so muß doch für jede Nachrichtenübertragung eine 14-fache Startup-Zeit angerechnet werden. Dadurch erhöht sich die Rechenzeit des Algorithmus.
Im Hinblick auf eine Verringerung der Kommunikationszeit, und damit der Ausführungszeit, wurde in der zweiten Implementation *packed_send* auf einen mehrmaligen Nachrichtenaustausch in einem Schleifendurchlauf verzichtet. Statt der 7 Nachrichten wurden hier nur zwei versendet. Dies wurde dadurch gelöst, daß im Unterprogramm *DOWNSEND* vor der Nachrichtenübertragung ein großes Feld $hilf$, bestehend aus allen erforderlichen Angaben einer Partition, erstellt wurde. Nur dieses eine Feld wurde verschickt. Es muß in dieser Implementation aber die Zeit für die Umspeicherung der Daten in $hilf$ mit berücksichtigt werden.

Nachfolgend sind die beiden Kommunikationsvarianten der Implementationen *send* und *packed_send* aufgeführt.

Die Kommunikationsroutine *DOWNSEND* der Implementation *send*:

```
      SUBROUTINE DOWNSEND(i, xsort, ysort, xassoc, yassoc, source2,
     >                    sindex2, median, length, address)
      . . .
      CALL CSEND(11 * i, length, 4, address, 0)
      CALL CSEND(12 * i, xsort(median+1), length, address, 0)
      CALL CSEND(13 * i, ysort(median+1), length, address, 0)
      CALL CSEND(14 * i, xassoc(median+1), length, address, 0)
      CALL CSEND(15 * i, yassoc(median+1), length, address, 0)
      CALL CSEND(16 * i, source2, 4, address, 0)
      CALL CSEND(17 * i, sindex2, 4, address, 0)

      END
```

Die Kommunikationsroutine *DOWNSEND* der Implementation *packed_send*:

```
      SUBROUTINE DOWNSEND(i, xsort, ysort, xassoc, yassoc, source2,
     >                    sindex2, median, length, address)
      . . .
      lang = length / 4

      DO 100 j = 1, lang, 1
         hilf(j) = xsort(median + j)
100   CONTINUE
      DO 200 j = 1, lang, 1
         hilf(lang + j) = ysort(median + j)
200   CONTINUE
      DO 300 j = 1, lang, 1
         hilf(2 * lang + j) = xassoc(median + j)
300   CONTINUE
      DO 400 j = 1, lang, 1
         hilf(3 * lang + j) = yassoc(median + j)
400   CONTINUE
      hilf(4 * lang + 1) = source2
      hilf(4 * lang + 2) = sindex2
      CALL CSEND(11 * i, length, 4, address, 0)
      CALL CSEND(12 * i, hilf(1), 4 * length + 8, address, 0)

      END
```

4.3.2 Sequentieller Unterteilungsalgorithmus von Karp

Zur Ermittlung der Performance mußte auf der Basis der parallelen Implementation ein sequentielles Programm entwickelt werden. Dieses sollte die Kommunikation und die Kontrollstruktur der Verteilung der Partitionen simulieren und das Problem wie in der parallelen Implementation lösen.

Die Dreiteilung in Unterteilungs-, Lösungs- und Kombinationsphase wurde beibehalten. Ebenso wurde die Partitionierung, die Berechnung und die Kombination der Näherungslösungen durch die Unterprogramme realisiert, die von der parallelen Implementation übernommen und modifiziert wurden.
Die Simulation der Kommunikation und damit der korrekten Verteilung der Partitionen gestaltete sich schwierig. Da nur ein Prozessor die Berechnung durchführt, mußte eine Form der Abspeicherung der Partitionen gefunden werden, die es erlaubt, ebenso große Probleme zu berechnen, wie es mit dem parallelen Programm möglich war. Aufgrund der vorgegebenen Speicherplatzbeschränkung von etwa 16 bzw. 8 MByte pro Prozessor konnten alle gebildeten Partitionen und Subpartitionen nicht in einem Feld hintereinander geschrieben werden. Stattdessen wurden für jeden Schleifendurchlauf die Daten der Subpartitionen B und C einer Partition A an die gleiche Stelle der Felder gespeichert, an denen vorher die Daten von Partition A abgelegt waren. Die Summe der Längen der Subpartitionen B und C war wegen der Verdopplung der Quelle um ein Element größer als die Länge von Partition A. Um keine Daten von nachfolgenden Partitionen zu überschreiben, mußten alle in einem Schleifendurchlauf gebildeten Tochterpartitionen in temporäre *xsorttemp*-, *ysorttemp*-, *xassoctemp*- und *yassoctemp*-Felder zwischengespeichert werden. Der Ablauf des sequentiellen Programms läßt sich anhand von Abbildung 4.7 (hier für 4 Partitionen) verfolgen.

Der sequentielle Algorithmus kann folgendermaßen beschrieben werden:

Zu Anfang werden wieder die exakten Koordinaten der Knoten eingelesen und in *xloc* und *yloc* gespeichert. Das Unterprogramm *BUILD* baut die initialen *xsort*-, *ysort*-, *xassoc*- und *yassoc*-Felder einer Partition auf. Da alle Partitionen in einem einzigen Feld abgespeichert werden, müssen die Indizes der Anfangselemente jeder Partition in Abhängigkeit des Unterteilungsniveaus in *bases* abgespeichert werden.

Mit dem Eintritt in die Unterteilungsphase beginnt die Partitionierung des Originalproblems. In die von der parallelen Implementation her bekannten Schleifen, die den Weg von oben nach unten, der in den Abbildungen 4.6 und 4.7 dargestellten Unterteilungsschemata realisieren, wird nun eine zweite Schleife implementiert. Diese simuliert die parallele Weiterbearbeitung der Partitionen durch die einzelnen Prozessoren in der parallelen Implementation. Erreicht wird dies durch den Lauf von links nach rechts in dem entsprechenden Unterteilungsniveau (siehe Abbildung 4.7). Die Abspeicherung der Partitionen ist identisch mit der Verteilung, das heißt der Versendung der Partitionen in der parallelen Implementation. Mit Hilfe der Unterprogramme *DIVIDE*, *SETUP* und *COPY* werden wie in der parallelen Implementation aus einer Partition zwei Subpartitionen erstellt, deren Daten in den oben

erwähnten temporären Feldern zwischengespeichert werden. Die innere Schleife wird solange ausgeführt, bis alle Partitionen eines Baumniveaus abgearbeitet sind. Dann werden die Daten aus den temporären Feldern in die ursprünglichen *xsort*-, *ysort*-, *xassoc*- und *yassoc*-Felder zurückgespeichert.
Ein neuer äußerer Schleifendurchlauf beginnt, das heißt im Unterteilungsschema wird auf das nächste Niveau gesprungen. Die Unterteilungsphase endet, wenn so viele Partitionen gebildet wurden, wie für den Vergleich mit der parallelen Implementation benötigt werden.

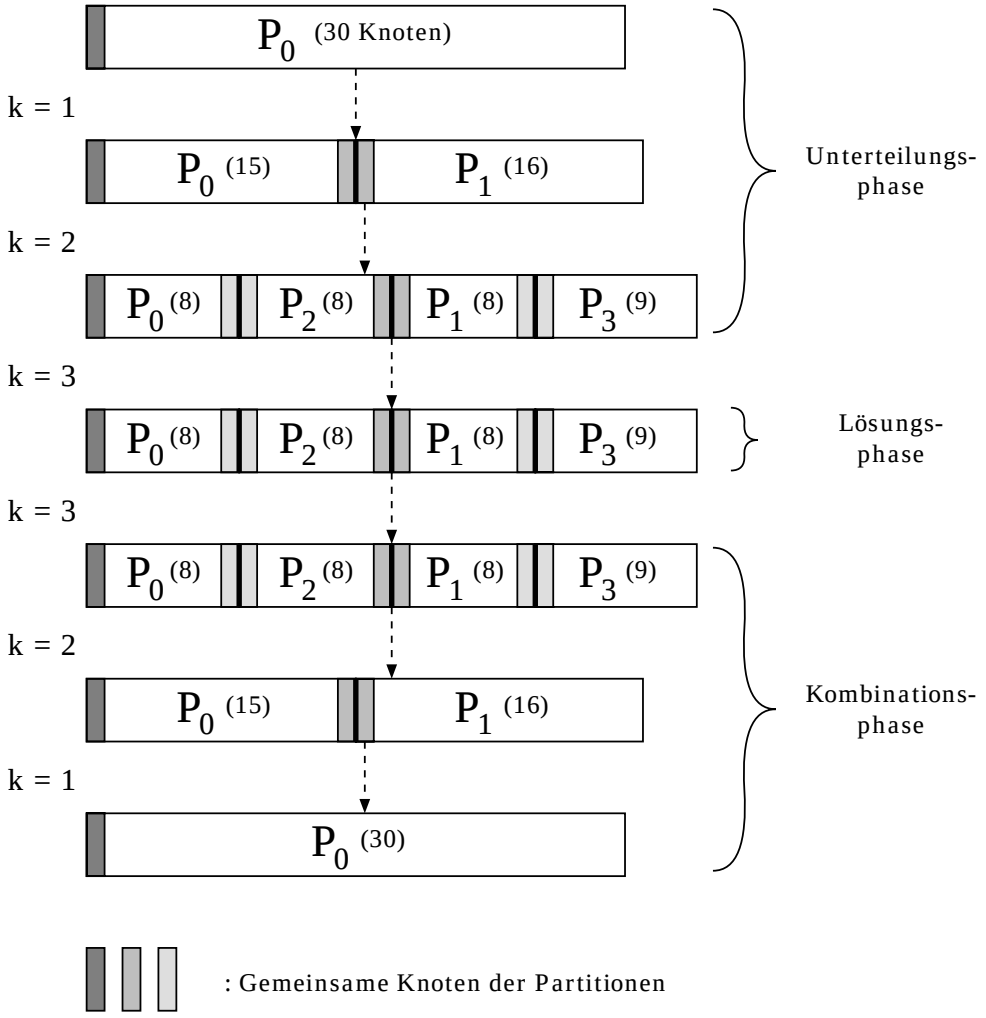


Abbildung 4.7: Unterteilungsschema bzw. Speicherbelegung der sequentiellen Implementation

In der Lösungsphase sind die gebildeten Partitionen nun hintereinander in einem Feld abgespeichert. Mit dem in einer Schleife ausgeführten Unterprogramm *SOLVE* werden sie gemäß ihrer Partitionsnummer gelöst. Eine Teillösung besteht aus der Länge einer Teiltour (*seqlen*) und der Reihenfolge der Tourknoten (*cycle*).

Die Kombinationsphase wird ebenfalls durch zwei ineinandergeschachtelte Schleifen realisiert. Wie in der Unterteilungsphase entspricht die äußere der Unterteilungstiefe und die innere simuliert die parallele Bearbeitung der Teillösungen.
COMBINE wurde im Vergleich zur parallelen Implementation nicht verändert. Aus den gleichen Gründen wie in der Unterteilungsphase muß aber auch hier die Reihenfolge der Tourknoten zwischengespeichert werden. Zwei Teiltouren werden durch das Verfahren kombiniert, welches bereits bei der parallelen Implementation beschrieben wurde.
Am Ende der Kombinationsphase enthält *seqlen* die Länge der Gesamttour und *cycle* die Reihenfolge der Tourknoten. Eine Näherungslösung für das ETSP ist gefunden.

Der folgende Code verdeutlicht die Vorgehensweise des Algorithmus bei der sequentiellen Implementation.

```
      PROGRAM KARP
      . . .
      CALL BUILD(n, xkord, ykord, xsort, ysort, xassoc, yassoc)
      sources(0, 1)  = xsort(1)
      sindices(0, 1) = 1
      elements(0, 1) = n
      bases(0, 1)    = 1
      high           = n
      k              = log(# Partitionen)
      begintime      = DCLOCK()

      DO 100 i = 1, k, 1                          !Unterteilungsphase
         DO 110 j = 1, (2 ** (i - 1)), 1
            high                            = elements(i - 1, j)
            median                          = INT((high + 1) / 2)
            length                          = high - median + 1
            bases(i, j)                     = els + 1
            bases(i, j + (2 ** (i - 1)))    = els + median + 1
            elements(i, j)                  = median
            elements(i, j + (2 ** (i - 1))) = length
            els                             = els + median + length
            CALL DIVIDE(i, j, xkord, ykord, xsort, ysort, xassoc,
     >                  yassoc, xsorttemp, ysorttemp, xassoctemp,
     >                  yassoctemp, source1, source2, sindex1,
     >                  sindex2, bases, median, high)
            sources(i, j)                   = source1
            sources(i, j + (2 ** (i - 1)))  = source2
            sindices(i, j)                  = sindex1
            sindices(i, j + (2 ** (i - 1))) = sindex2
  110    CONTINUE
```

```
        DO 120 j = 1, els, 1                              !Umspeichern
           xsort(j)  = xsorttemp(j)
           ysort(j)  = ysorttemp(j)
           xassoc(j) = xassoctemp(j)
           yassoc(j) = yassoctemp(j)
120     CONTINUE
100  CONTINUE

     DO 200 j = 1, 2 ** k, 1                            !Loesungsphase
        CALL SOLVE(j, k, xkord, ykord, xsort, bases,
    >              sindices, elements, cycle, seqlen)
200  CONTINUE

     DO 300 i = k, 1, -1                             !Kombinationsphase
        DO 310 j = 1, (2 ** (i - 1)), 1
           length = elements(i, j + 2 ** (i - 1))
           median = elements(i, j)
           high   = elements(i - 1, j)
           els    = els + median + length - 1
           CALL COMBINE(i, j, xkord, ykord, high, median, length,
    >                   sources, cycle, cycletemp, bases, seqlen)
310     CONTINUE
        DO 320 j = 1, els, 1                              !Umspeichern
           cycle(j) = cycletemp(j)
320     CONTINUE
300  CONTINUE

     endtime = DCLOCK()

     END
```

4.3.3 Heuristik des weitesten Einfügens

Mit der Heuristik des weitesten Einfügens wird im Unterprogramm *SOLVE* eine Näherungslösung für eine Subpartition berechnet.
Sie geht so vor, daß zu einer bereits gebildeten Teiltour immer der am weitesten entfernte Knoten zu dieser Teiltour hinzugefügt wird. Die bei diesem Vorgang verwendete Datenstruktur ist ein Feld (aux) der Länge n. n ist die Anzahl der Knoten einer Partition. aux gibt an, welche Knoten bereits in der Tour enthalten sind und welche nicht. In Abbildung 4.5 sind sie für die einzelnen Teiltouren unter deren Darstellungen aufgeführt. Der Aufbau der Tour wird mittels einer Schleife realisiert. In der k-ten Iteration enthalten die ersten k Elemente von aux die Namen derjenigen Knoten, die bereits in der Tour enthalten sind. Soll nun im $(k+1)$-ten Durchlauf der Knoten in die Tour aufgenommen werden, dessen Name in $aux(j)$, $j \geq (k+1)$,

gespeichert ist, so werden die Inhalte von $aux(j)$ und $aux(k+1)$ miteinander vertauscht (siehe Abbildung 4.5). Zu Beginn enthält $aux(i)$ den Namen von Knoten i, $1 \leq i \leq n$, für alle Knoten der Partition. Das erste Element, welches die Tour bildet, ist die Quelle. Die Entfernungen aller Knoten zu der bereits gebildeten Tour sind in *dist* abgespeichert. Der Knoten mit dem maximalem Wert in *dist* wird in die Tour aufgenommen. Er wird zwischen den beiden Tourknoten eingefügt, bei denen die neuentstehende Tourlänge minimal wird. Mit Hilfe der Gewichtsmatrix W werden die möglichen Kombinationen berechnet. Die Matrix ist vollbesetzt und symmetrisch, die Elemente der Hauptdiagonalen sind gleich Null. Die Kantenbewertungen wurden in eine Gewichtsmatrix abgespeichert, da bei der Heuristik des weitesten Einfügens alle Kanten untersucht werden. Der direkte Zugriff auf die Elemente der Matrix erlaubt so ein schnelleres Auffinden der längsten Kante und somit des am weitesten entfernten Knotens.
Für die Kombination der Teiltouren durch das Unterprogramm *COMBINE* ist es wichtig, daß alle Touren den gleichen Richtungssinn aufweisen. Deshalb wird, wenn die Tour aus drei Knoten besteht, geprüft, ob der Richtungssinn der Tour gegen den Uhrzeigersinn läuft. Ist das nicht der Fall, so wird die Reihenfolge der Knoten in der Tour entsprechend umgekehrt.
Nach Ende der Berechnung enthält *len* die Gesamtlänge der Teiltour. Die Reihenfolge der Tourknoten, beginnend mit der Quelle, ist in *cycle* abgespeichert.

Die Heuristik des weitesten Einfügens wird wie folgt in *SOLVE* realisiert:

```
      SUBROUTINE SOLVE(k, pn, xkord, ykord, xsort,
     >                 sindices, elements, cycle, len)
      . . .
      high   = elements(k, pn)          !pn ist die Partitionsnummer
      sindex = sindices(k, pn)
      DO 100 j = 2, high, 1                          !Bestimmen der
         DO 100 l = 1, j - 1, 1                !Entfernungsmatrix W
            w(j, l) = DSQRT((xkord(xsort(j)) -
     >                       xkord(xsort(l))) ** 2 +
     >                      (ykord(xsort(j)) -
     >                       ykord(xsort(l))) ** 2)
            w(l, j) = w(j, l)
100   CONTINUE
      aux(1)         = sindex
      aux(sindex)    = 1
      chosen         = sindex
      tcycle(sindex) = sindex
```

```
      DO 200 itnum = 1, high - 1, 1             !Aufbau der Teiltour
         DO 210 j = itnum + 1, high, 1
            vtxa       = aux(j)                        !Bestimmen des
            dist(vtxa) = MIN(dist(vtxa), w(vtxa, chosen))  !Knotens,
            IF (dist(vtxa) .GT. max) THEN                    !der am
               max   = dist(vtxa)                        !weitesten von
               index = j                  !der Teiltour entfernt ist
            ENDIF
210      CONTINUE
         chosen = aux(index)
         DO 220 j = 1, itnum, 1             !Bestimmen der Stelle, an
            vtxa = aux(j)                  !der der Knoten eingefuegt
            vtxb = tcycle(vtxa)                          !werden soll
            cost = w(vtxa, chosen) + w(chosen, vtxb)
     >                             - w(vtxa, vtxb)
            IF (cost .LT. min) THEN
               min  = cost
               end1 = vtxa
               end2 = vtxb
            ENDIF
220      CONTINUE
         IF (Teiltour besteht aus drei Knoten) THEN     !Pruefen der
            IF (Teiltour in Uhrzeigersinn) THEN        !Laufrichtung
               Aendere den Richtungssinn der Teiltour!     !der Tour
            ENDIF
         ENDIF
         tcycle(chosen) = end2          !Einfuegen des neuen Knotens
         tcycle(end1)   = chosen
         aux(index)     = aux(itnum + 1)
         aux(itnum + 1) = chosen
         len            = len + min
200   CONTINUE
      index = sindex
      DO 300 j = 1, high, 1                !Aufbau von cycle mit der
         cycle(j) = xsort(index)          !Quelle als Anfangselement
         index    = tcycle(index)
300   CONTINUE

      END
```

4.4 Komplexitätsbetrachtungen

Vorab sei die Komplexität des Unterprogramms *BUILD* betrachtet. Dessen Zeitkomplexität ist bei der sequentiellen wie bei der parallelen Implementation gleich. Sie beträgt bei einer Problemgröße von n Knoten $O(n^2)$. Der bestimmende Faktor dieser Zeitkomplexität ist der verwendete Sortieralgorithmus. In dieser Arbeit wurde ein Sortieralgorithmus verwendet, der auf Vergleich mit dem jeweiligen Minimum mit Vertauschung beruht. Wird ein optimaler Sortieralgorithmus, wie etwa Quicksort, verwendet, reduziert sich die Zeitkomplexität von *BUILD* auf $O(n \cdot \log n)$. Zu beachten ist jedoch, daß *BUILD* in den Programmen außerhalb der Zeitmessung liegt.

Die Zeitkomplexität des Unterteilungsalgorithmus von Karp berechnet sich aus drei Summanden. Diese sind die Zeitkomplexitäten der Unterteilungs-, der Lösungs- und der Kombinationsphase.
Im weiteren wird mit n die Größe des mit dem Unterteilungsalgorithmus von Karp zu lösenden Problems bezeichnet. p bezeichnet die Anzahl der Partitionen, in die das Originalproblem unterteilt wird. Bei der parallelen Implementation ist die Anzahl p der Partitionen identisch mit der Anzahl der zur Verfügung stehenden Prozessoren. Mit log ist im weiteren der Zweierlogarithmus gemeint.

Die Unterteilungsphase wird innerhalb einer Schleife realisiert, die $\log(p)$-mal ausgeführt wird. *DIVIDE* besitzt bei einer Problemgröße von n eine Zeitkomplexität von $O(n)$. In jeder Schleife halbiert sich die Größe einer Partition. Damit beträgt die Zeitkomplexität der Unterteilungsphase der parallelen Implementation

$$O(\sum_{i=1}^{\log(p)} n/2^i).$$

Bei der sequentiellen Implementation wird die Unterteilungsphase von einem Prozessor ausgeführt. Er muß alle Partitionen eines Niveaus des Unterteilungsschemas in Subpartitionen aufteilen. Die Summe der Elemente der Subpartitionen erhöht sich jeweils um die doppelte Quelle. Somit beträgt in diesem Fall die Zeitkomplexität der Unterteilungsphase

$$O(\sum_{i=1}^{\log(p)} 2^i \cdot (n/2^i + (2^i - 1))) = O(\sum_{i=1}^{\log(p)} n).$$

Die Zeitkomplexität der Heuristik des weitesten Einfügens beträgt $O(n^2)$ bei einer Problemgröße von n. Soll das Originalproblem in p Partitionen unterteilt werden, so muß mit der Heuristik des weitesten Einfügens eine suboptimale Tour für n/p Knoten gefunden werden. Somit beträgt die Zeitkomplexität der Lösungsphase $O((n/p)^2)$. Bei der sequentiellen Implementation wird die Lösungsphase durch einen Prozessor ausgeführt, das heißt, er muß p Partitionen lösen. Damit erhöht sich die Zeitkomplexität der Lösungsphase der sequentiellen Implementation auf $O(n^2/p)$ im Vergleich zu $O((n/p)^2)$ der parallelen Implementation.

Die Zeitkomplexität der Kombinationsphase berechnet sich ähnlich wie die der Unterteilungsphase. *COMBINE* besitzt bei einer Problemgröße von n, wie *DIVIDE*, eine Zeitkomplexität von $O(n)$. Die Schleifendurchläufe sind so wie in der Unterteilungsphase. Damit ergibt sich bei der parallelen Implementation eine Zeitkomplexität von

$$O(\sum_{i=1}^{\log(p)} n/2^i)$$

und bei der sequentiellen Implementation von

$$O(\sum_{i=1}^{\log(p)} n).$$

Der parallele Unterteilungsalgorithmus von Karp besitzt damit eine Zeitkomplexität von

$$O(\sum_{i=1}^{\log(p)} n/2^i) + O((n/p)^2) + O(\sum_{i=1}^{\log(p)} n/2^i) = O(n + (n/p)^2)$$

und der sequentielle von

$$O(\sum_{i=1}^{\log(p)} n) + O(n^2/p) + O(\sum_{i=1}^{\log(p)} n) = O(n \cdot \log(p) + n^2/p).$$

4.5 Ergebnisse

Die in Abschnitt 4.3 beschriebenen Programme zur näherungsweisen Lösung eines ETSP mit Hilfe des Unterteilungsalgorithmus von Karp und der Heuristik des weitesten Einfügens sind auf den in Kapitel 2 beschriebenen massiv-parallelen Rechnern iPSC/860 und Paragon XP/S 5 der Firma Intel implementiert worden.

Die Programme aller drei Implementationen wurden mit der Compiler-Optimierungsstufe *1* übersetzt. Die ausführbaren Programme wurden auf alle Prozessoren gleichzeitig geladen und einzeln und unabhängig voneinander ausgeführt. Die reellwertigen Eingabedaten für *xloc* und *yloc* im Bereich zwischen 0 und 1 wurden mittels eines Zufallszahlengenerators ermittelt.
An dieser Stelle muß noch einmal deutlich gemacht werden, daß im Zusammenhang mit den parallelen Implementationen die Anzahl der Prozessoren grundsätzlich gleichzusetzen ist mit der Anzahl der Partitionen und umgekehrt.
Die Kommunikation der Prozessoren untereinander findet im synchronen Modus statt. Dafür kann aber auch der asynchrone Modus gewählt werden. Doch weil sich die Kommunikation nicht mit der Berechnung überlappen läßt, wird dadurch keine Verringerung der Ausführungszeit erzielt.
Die Messungen auf dem iPSC/860 wurden für die Prozessor- und damit Partitionsanzahlen 2 und 4 im interaktiven, für 8, 16 und 32 im Batch-Betrieb durchgeführt. Im Gegensatz dazu sind alle Messungen auf dem Paragon im Batch-Betrieb durchgeführt worden. Auf dem Paragon konnte im Gegensatz zum iPSC/860 auch eine Untersuchung mit 64 Prozessoren bzw. Partitionen durchgeführt werden. Aufgrund zeitlicher Beschränkungen im Batch-Betrieb mußten auf beiden Rechnern alle Messungen für die unterschiedlichen Problemgrößen in Blöcken stattfinden. Die Schrittweiten der Knotenanzahlen der Probleme in den Blöcken ist unterschiedlich. Sie stimmen mit den Meßpunkten in den Abbildungen der Speedup-Werte überein.
Die Problemgrößen bewegten sich zwischen einem minimalen und einem maximalen Wert. Die beiden Werte waren abhängig von der Implementationsvariante, der Anzahl der zu bildenden Partitionen und dem Speicherplatz eines Rechenknotens. Die Größe (`wmax, wmax`) des zweidimensionalen Feldes `w`, in dem die Gewichtsmatrix W einer Partition abgespeichert war, zeigte sich als beschränkender Faktor in der Skalierung der Problemgröße.
Das Minimum berechnet sich durch

$$min = (2 * \#Partitionen) + 1$$

und entsprechend das Maximum durch

$$max = (wmax * \#Partitionen) - (\#Partitionen - 1) \ .$$

Für die Implementation *packed_send* kam zusätzlich der Speicherplatzbedarf für das benötigte Feld *hilf* hinzu. Die Größe einer Partition der untersten Unterteilungsstufe muß mindestens drei Knoten betragen.

In den Tabellen 4.1 und 4.2 werden die auf dem iPSC/860 erzielten Ausführungszeiten den Problemgrößen, das heißt der Anzahl der Knoten des Graphen, gegenübergestellt.

# Part.	Größe *min*	Ausführungszeit [ms] *min*	Größe *max*	Ausführungszeit [ms] *max*
1	3	$7.43 \cdot 10^{-2}$	1380	9793.71
2	5	0.31	2773	20203.56
4	9	0.71	5513	40559.75
8	17	1.56	10873	80051.47
16	33	3.01	21169	158170.11
32	65	6.50	40097	289259.28

Tabelle 4.1: Vergleich der Ausführungszeiten in Abhängigkeit von der Problemgröße bei der sequentiellen Implementation auf dem iPSC/860

Tabelle 4.1 zeigt für die sequentielle Implementation die von der Anzahl der Partitionen abhängigen kleinstmöglichen Problemgrößen *min* und deren Ausführungszeit in Millisekunden auf einem Prozessor. Die minimalen Problemgrößen gelten auch für die parallelen Implementationen. Weiter sind die von den verschiedenen Partitionsanzahlen abhängigen maximalen Problemgrößen *max* sowie deren Ausführungszeiten in Millisekunden aufgeführt.

# Part. = # Proz.	*send*		*packed_send*	
	Größe *max*	Ausführungszeit [ms]	Größe *max*	Ausführungszeit [ms]
1	1380	9802.23	1380	9805.37
2	2773	10099.93	2769	10053.03
4	5501	10088.08	5493	10115.31
8	10833	10118.74	10793	10138.26
16	21057	10226.59	20849	9914.23
32	39681	9569.21	38977	9308.57

Tabelle 4.2: Vergleich der Ausführungszeiten in Abhängigkeit von der Problemgröße bei den parallelen Implementationen auf dem iPSC/860

Im Gegensatz dazu beinhaltet Tabelle 4.2 für die beiden parallelen Implementationen *send* und *packed_send* die maximal möglichen Problemgrößen *max* und deren Ausführungszeiten in Millisekunden. Die maximalen Problemgrößen sind abhängig

von der Anzahl der Partitionen und sind für die Implementationen *packed_send* und *send* unterschiedlich. Die Ausführungszeiten wurden mit so vielen Prozessoren erreicht, wie Partitionen gebildet wurden. Die Anzahl der verwendeten Prozessoren ist hier also gleich der Anzahl der Partitionen. Man erkennt, daß sich mit der Erhöhung der Anzahl von Partitionen und damit des Parallelitätsgrads wesentlich größere Probleme lösen lassen.
Die Ausführungszeiten der Implementationen *send* und *packed_send* unterscheiden sich kaum voneinander. Ein Vergleich zeigte, daß die Zeiten von *packed_send* zum großen Teil sogar schlechter sind als die von *send*. Die Unterschiede betragen maximal 8%. Nur für kleine Probleme ist *packed_send* schneller. Bei großen Problemen überwiegt der Zeitaufwand für die Umspeicherung der zu versendenden Felder in *hilf* die durch das einmalige Versenden gewonnene Zeit. Im weiteren wird aufgrund dessen nur die parallele Implementation *send* betrachtet.

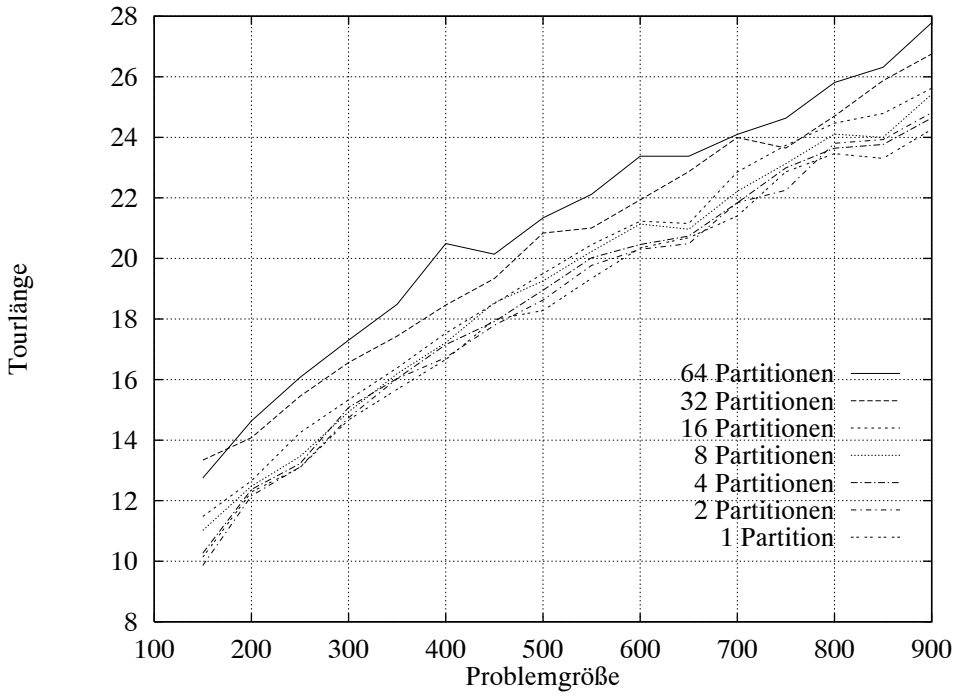


Abbildung 4.8: Unterschied der Tourlängen bei verschiedenen Partitionen und unterschiedlicher Problemgröße

Die in Abschnitt 4.1 aufgestellte Vermutung, daß sich die Länge einer Tour bei feinerer Unterteilung in kleinere Partitionen in der Regel vergrößert, bestätigte sich. In Abbildung 4.8 sind die Tourlängen von unterschiedlichen Problemgrößen bei verschiedenen Partitionsanzahlen dargestellt. Untersucht wurden Problemgrößen zwischen 150 und 900 Knoten im Abstand von jeweils 50 Knoten. Die resultierende Tour wächst abhängig vom Problem mit der Anzahl der Partitionen der untersten Verteilungsstufe.

Im folgenden werden Speedup und Effizienz für den iPSC/860 und den Paragon miteinander verglichen. Der erreichte Speedup ist in Abhängigkeit von den betrachteten Problemgrößen aufgetragen. Die Anzahl der bei der Messung gebildeten Partitionen ist gleich der Anzahl der verwendeten Prozessoren.

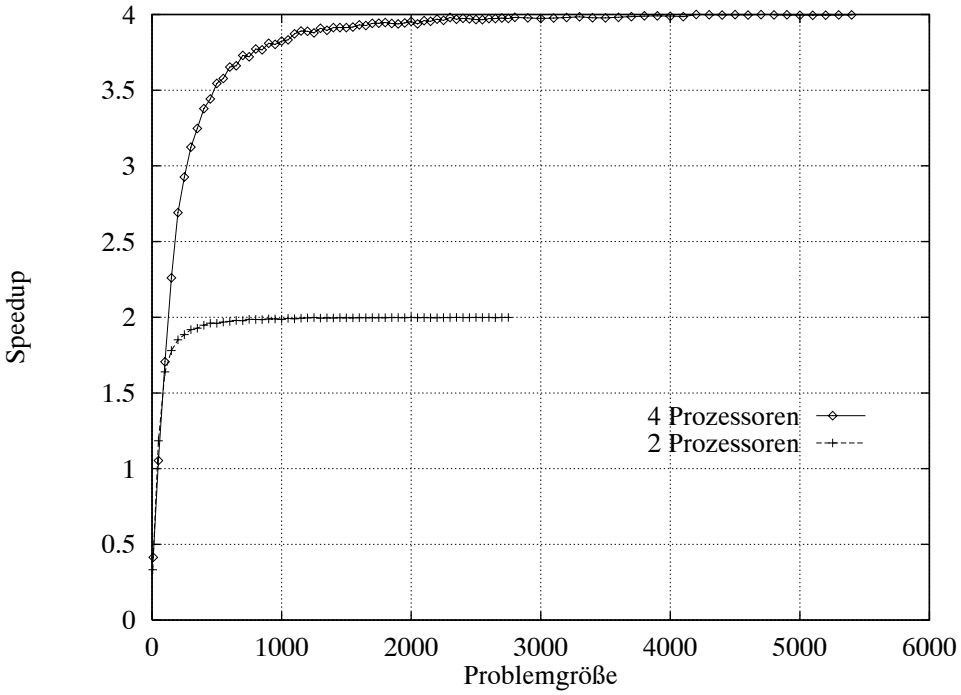


Abbildung 4.9: Erreichter Speedup in Abhängigkeit von der Problemgröße bei 2 und 4 Prozessoren auf dem iPSC/860 (# Partitionen = # Prozessoren)

Abbildung 4.9 zeigt den Speedup, der für zwei und vier Prozessoren und damit entsprechend vielen Partitionen auf dem iPSC/860 erzielt wurde. Abbildung 4.10 gibt den auf dem iPSC/860 erreichten Speedup für 8, 16 und 32 Prozessoren an.

Problemgröße	Speedup	Effizienz
100	1.23	0.03
1000	4.92	0.15
2500	11.13	0.35
5000	17.67	0.55
10000	23.68	0.74
20000	27.68	0.86
35000	29.49	0.92

Tabelle 4.3: Speedup und Effizienz für 32 Prozessoren des iPSC/860 und damit 32 Partitionen bei verschiedenen Problemgrößen

Schon bei geringen Problemgrößen werden für Prozessoranzahlen bis zu 8, Speedup-Werte erreicht, die nur geringfügig unter den optimal möglichen liegen oder diese sogar erreichen. Erhöht sich jedoch die Anzahl der Prozessoren von 8 auf 16 oder 32, so verringert sich der Speedup. Er liegt bei den hier betrachteten größten Problemgrößen, die durch die jeweilige Anzahl von Partitionen und damit Prozessoren gegeben ist, um 3.5% bzw. 8.5% unter dem optimalen Wert.

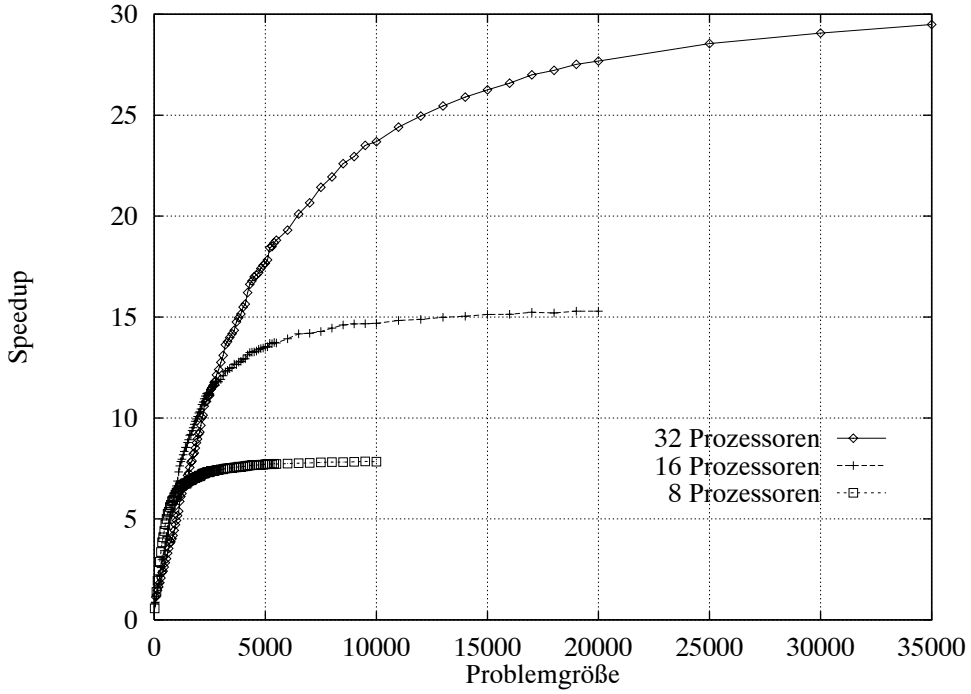


Abbildung 4.10: Erreichter Speedup in Abhängigkeit von der Problemgröße bei 8, 16 und 32 Prozessoren auf dem iPSC/860 (# Partitionen = # Prozessoren)

Um diese Ergebnisse deutlicher herauszustellen, werden in Tabelle 4.3 Speedup und Effizienz verschiedener Problemgrößen für den iPSC/860 nach Berechnung mit 32 Prozessoren einander gegenüber gestellt.

Die in den Abbildungen 4.11 und 4.12 dargestellten Speedup-Werte des Paragon zeigen den gleichen Verlauf wie die des iPSC/860. Sie erreichen jedoch nicht die beim iPSC/860 erzielten hohen Werte, sondern bleiben ein wenig darunter.

Wegen dem schnelleren Applikationsprozessor eines Paragon-Rechenknotens verläuft die sequentielle Berechnung einer Näherungslösung eines ETSP dort schneller als auf dem iPSC/860. Die Kommunikationsleistung des Paragon ist zwar auch größer, doch ist deren Einfluß auf die Performance nicht so entscheidend. Dadurch wirkt sich der Kommunikations-Overhead bei der parallelen Berechnung auf dem Paragon stärker auf den erreichten Speedup aus als auf dem iPSC/860.

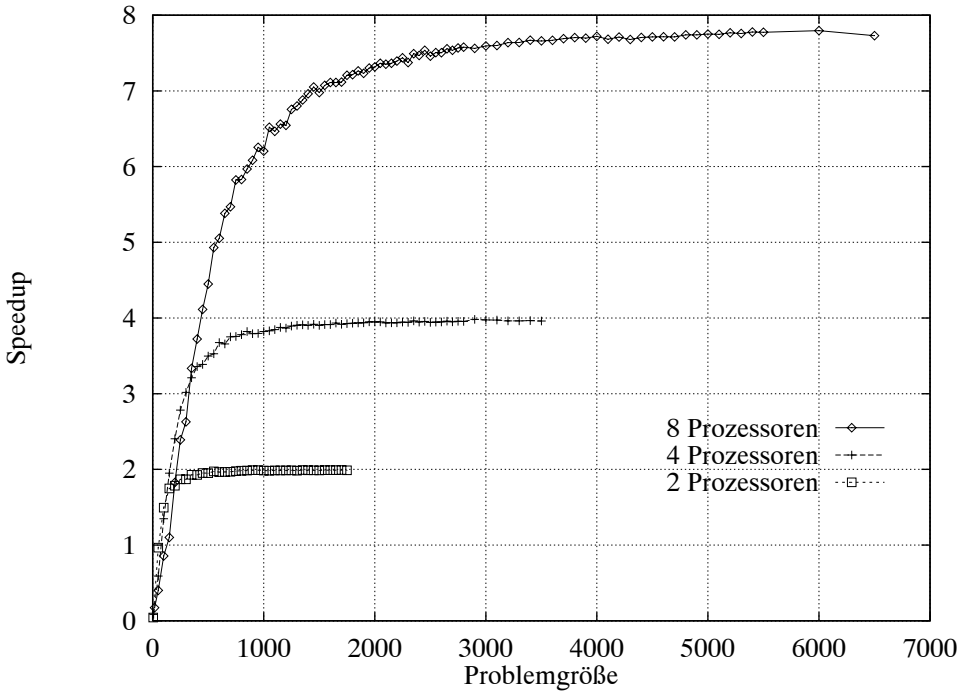


Abbildung 4.11: Erreichter Speedup in Abhängigkeit von der Problemgröße bei 2, 4 und 8 Prozessoren auf dem Paragon (# Partitionen = # Prozessoren)

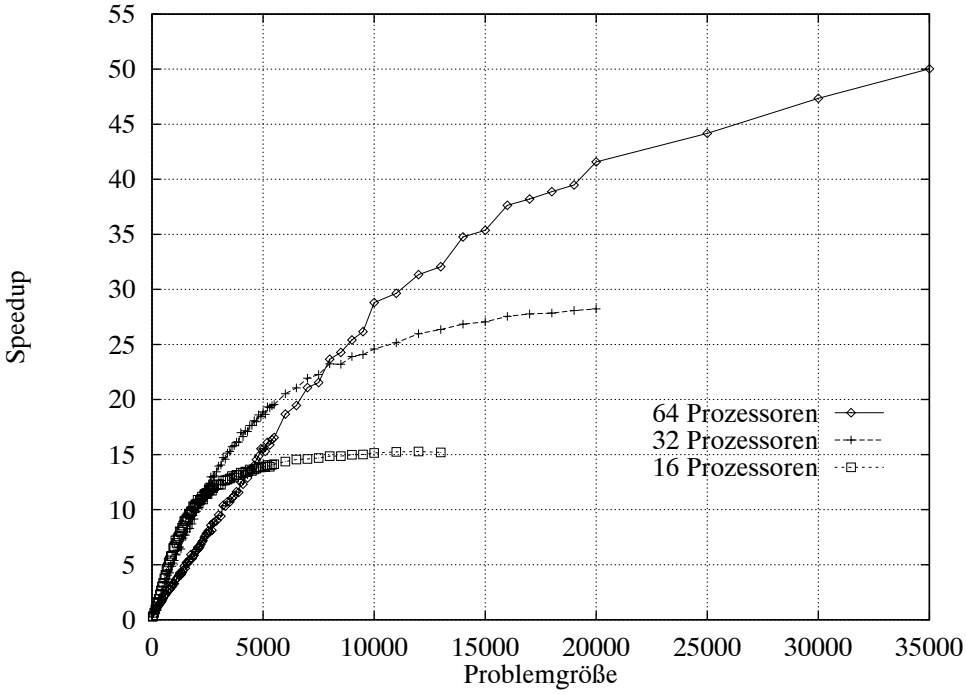


Abbildung 4.12: Erreichter Speedup in Abhängigkeit von der Problemgröße bei 16, 32 und 64 Prozessoren auf dem Paragon (# Partitionen = # Prozessoren)

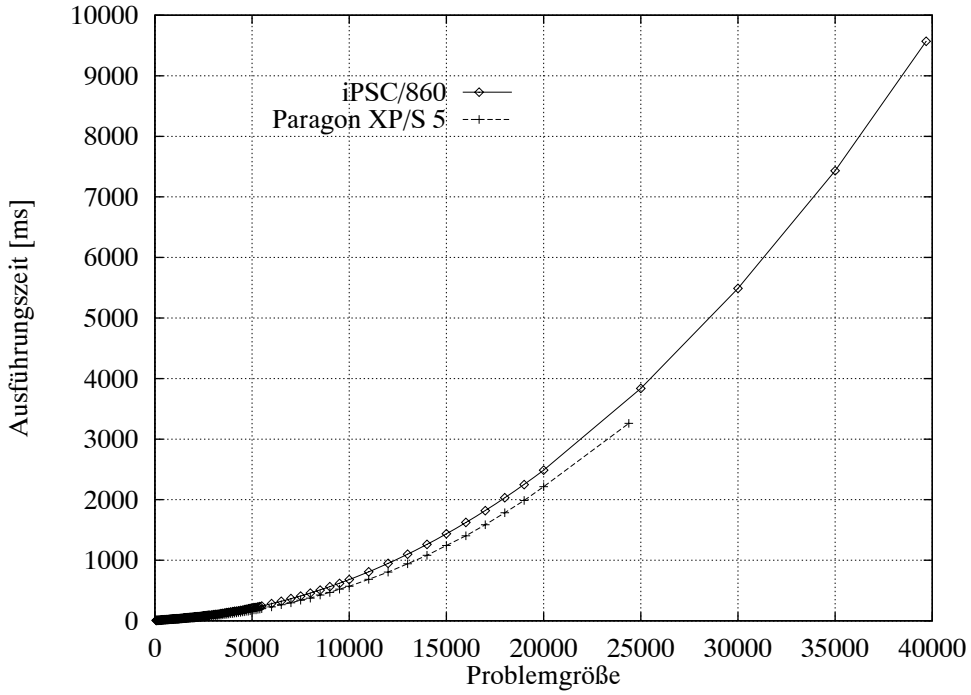


Abbildung 4.13: Vergleich der Ausführungszeiten in Abhängigkeit von der Problemgröße auf 32 Prozessoren des iPSC/860 und des Paragon

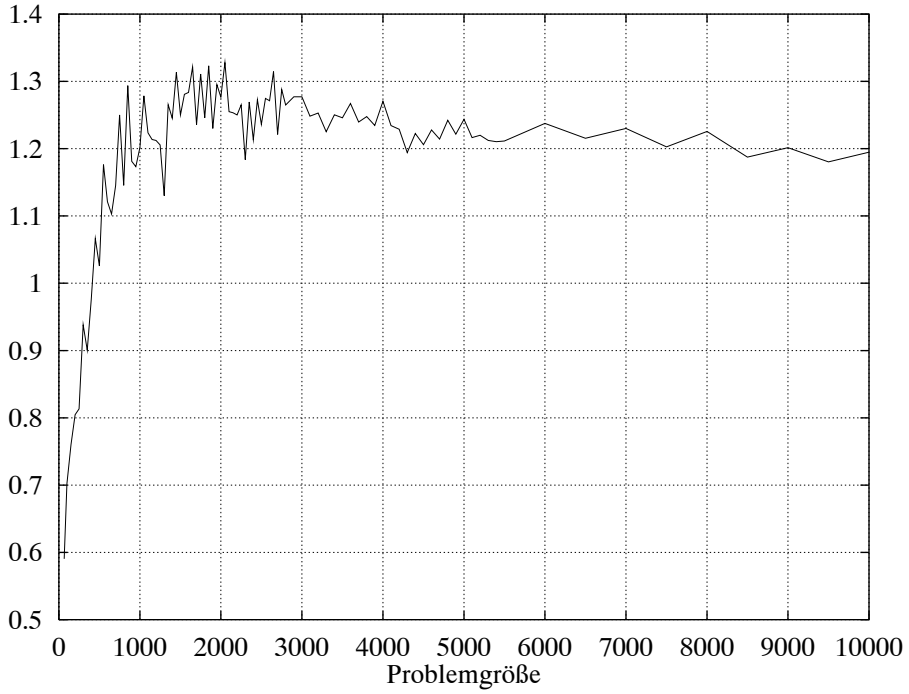


Abbildung 4.14: Quotient aus den Ausführungszeiten von iPSC/860 und Paragon bei 32 Prozessoren

Die Ausführungszeit einer Applikation auf dem Paragon ist jedoch niedriger als die auf dem iPSC/860. Abbildung 4.13 zeigt die unterschiedliche Rechenzeit von iPSC/860 und Paragon mit 32 Prozessoren und damit 32 Partitionen. Deutlich erkennt man die durch die höhere Taktfrequenz des i860 XP-Prozessors gegenüber dem i860 XR-Prozessor bedingte schnellere Bearbeitungszeit. Für Problemgrößen von 65 bis 10000 ist der Quotient der Ausführungszeiten in Abbildung 4.14 aufgeführt. Man erkennt, daß nur bei sehr kleinen Problemen der iPSC/860 aufgrund der geringeren Startup-Zeit schneller ist als der Paragon.

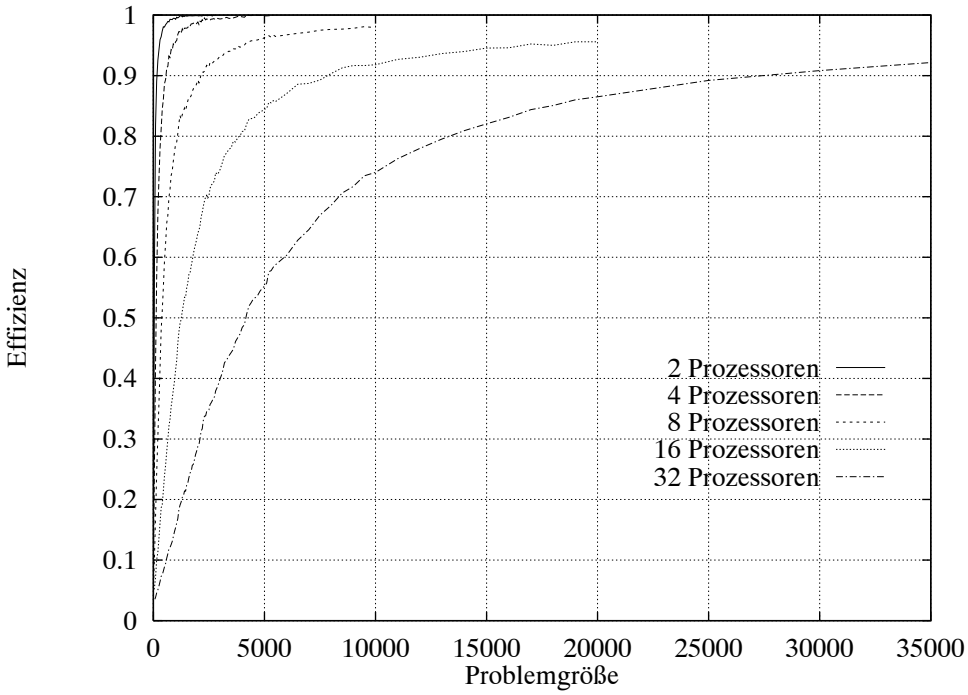


Abbildung 4.15: Effizienz in Abhängigkeit von der Problemgröße auf dem iPSC/860 bei 2 bis 32 Prozessoren (# Partitionen = # Prozessoren)

Abbildung 4.15 zeigt die Effizienz des Unterteilungsalgorithmus von Karp bei verschiedenen Prozessoranzahlen und entsprechender Anzahl von Partitionen nach Ausführung auf dem iPSC/860. Sehr schnell, das heißt, schon bei relativ kleinen Problemen wird eine Effizienz von über 50% erreicht. Man beachte dazu auch Tabelle 4.3. Da auf dem Paragon ähnliche Speedup-Werte erzielt wurden wie auf dem iPSC/860, sind auch die Effizienzen, wie Abbildung 4.16 zeigt, in etwa gleich.

Die Frage, warum die Implementationen des Unterteilungsalgorithmus von Karp auf dem iPSC/860 und dem Paragon XP/S 5 die in diesem Abschnitt vorgestellten Performance-Werte erreicht, stellt sich nun zurecht.

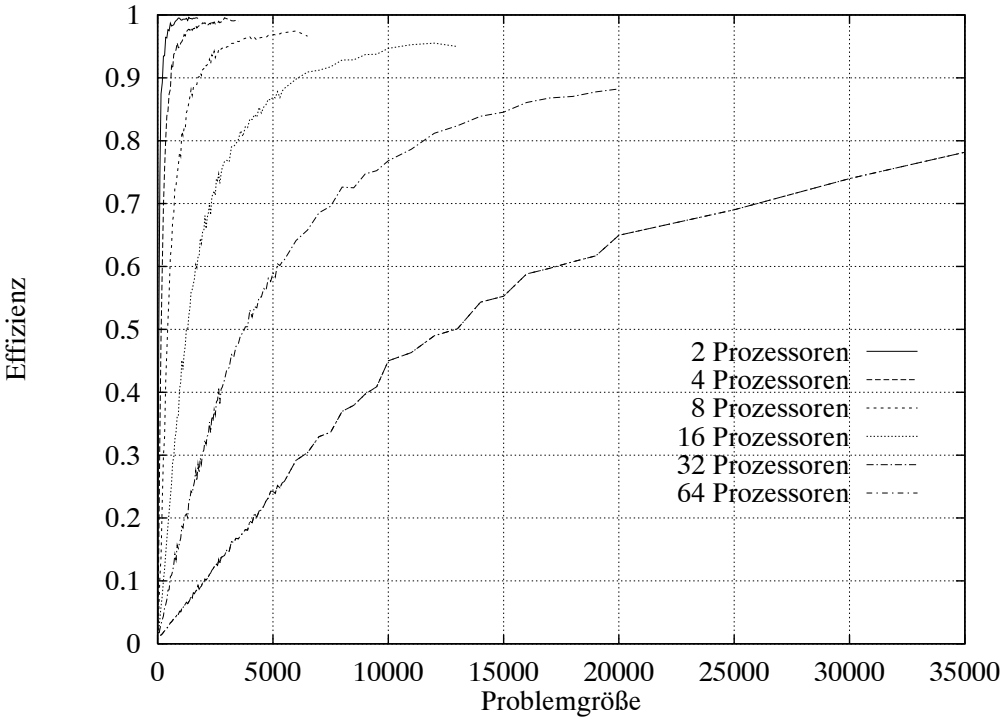


Abbildung 4.16: Effizienz in Abhängigkeit von der Problemgröße auf dem Paragon bei 2 bis 64 Prozessoren (# Partitionen = # Prozessoren)

Die Struktur des Unterteilungsalgorithmus von Karp ließ es zu, eine Implementationsform zu entwickeln, bei der die Kommunikation weitgehend parallel verläuft. Der größte Teil der Ausführungszeit wurde durch die Berechnung der Teillösungen mit der Heuristik des weitesten Einfügens verbraucht. Dessen Zeitkomplexität beträgt $O(n^2)$ bei einer Problemgröße von n Knoten. Bei einem Problem der halben Größe benötigt die Heuristik des weitesten Einfügens also etwa nur ein viertel der Zeit.

Da im Gegensatz zur parallelen bei der sequentiellen Berechnung die Teilprobleme hintereinander gelöst werden müssen, dauert die Ausführung einer Applikation entsprechend länger. Da der Kommunikationsaufwand gegenüber dem Rechenaufwand sehr gering ist, werden diese hohen Performance-Werte erreicht.

Die Tabellen 4.4 und 4.5 stellen für den iPSC/860 und den Paragon die möglichen Partitions- und Prozessoranzahlen den erzielten Ausführungszeiten und Tourlängen bei einer Problemgröße gegenüber. Die Ausführungszeiten werden in Millisekunden angegeben, die Tourlängen sind dimensionslos.

Tabelle 4.4 zeigt, daß auf dem iPSC/860 mit 32 Prozessoren und damit 32 Partitionen ein ETSP mit 1350 Knoten 250 mal schneller gelöst werden kann als mit nur einem Prozessor und einer Partition. Dies bedeutet eine Verringerung der Ausführungszeit um 99.57%. Die Länge der Lösungstour vergrößert sich dabei nur um 5.5%.

Seq.	# Proz.	1	1	1	1	1	1
	# Part.	1	2	4	8	16	32
	Zeit [ms]	9384.46	4585.31	2201.08	1047.42	499.82	264.04
Par.	# Proz.	1	2	4	8	16	32
	# Part.	1	2	4	8	16	32
	Zeit [ms]	9389.56	2297.07	565.07	155.79	59.92	40.13
	Tourlänge	29.59	30.08	30.63	30.98	31.49	31.63

Tabelle 4.4: Erzielte Ausführungszeiten in Millisekunden auf dem iPSC/860 für Partitionierungen von 1 bis 32 und Prozessoranzahlen von 1 bis 32 bei einer Problemgröße von 1350 Knoten mit den errechneten Tourlängen

In Tabelle 4.5 sind die entsprechenden Werte für den Paragon XP/S 5 aufgetragen.

Seq.	# Proz.	1	1	1	1	1	1	1
	# Part.	1	2	4	8	16	32	64
	Zeit [ms]	3514.61	1674.44	791.96	370.30	177.23	118.27	72.89
Par.	# Proz.	1	2	4	8	16	32	64
	# Part.	1	2	4	8	16	32	64
	Zeit [ms]	3527.71	841.34	208.85	60.89	30.65	24.32	24.39
	Tourlänge	24.26	24.81	24.63	25.42	25.62	26.75	27.78

Tabelle 4.5: Erzielte Ausführungszeiten in Millisekunden auf dem Paragon für Partitionierungen von 1 bis 64 und Prozessoranzahlen von 1 bis 64 bei einer Problemgröße von 900 Knoten mit den errechneten Tourlängen

Abschließend sei das Grötschel-Problem, dessen Graph 442 Knoten besitzt, nochmals erwähnt. Tabelle 4.6 zeigt die Ergebnisse, die mit dem Paragon XP/S 5 erzielt wurden. Ein Beispiel für die rekursiv absteigende Unterteilung in 8 Partitionen zeigen die Abbildungen 4.1 bis 4.4 in Abschnitt 4.1. Das von Olaf Holland berechnete globale Optimum des Grötschel-Problems beträgt 50.69 inch [DSW93].

Die mit dem Unterteilungsalgorithmus von Karp mit einer Partition, das heißt der Heuristik des weitesten Einfügens, erzielte beste Lösung beträgt 56.88 inch, ist also um 12% schlechter. Die schnellste parallele Lösung wurde auf dem Paragon XP/S 5 mit 32 Prozessoren bei 32 Partitionen erzielt. Deren Länge ist mit 60.88 inch um 7% schlechter als die hier erzielte beste Lösung von 56.88 inch, braucht jedoch mit 14.1 ms nur 1.8% der dafür benötigten sequentiellen Ausführungszeit.

# Part.	Tourlänge [inch]	seq. Zeit [ms]	par. Zeit [ms]
1	56.88	777.62	783.06
2	58.19	366.88	186.61
4	58.10	169.09	51.34
8	61.09	79.27	19.47
16	59.89	44.77	14.47
32	60.88	30.83	14.10
64	65.48	27.86	15.74
128	70.07	30.76	—
Opt.	50.69	—	—

Tabelle 4.6: Tourlängen in inch und Rechenzeit in Millisekunden des Grötschel-Problems von 442 Knoten auf dem Paragon. Die Anzahl der Partitionen entspricht bei der parallelen Implementation der Anzahl der verwendeten Prozessoren.

In Unterabschnitt 2.1.4 wurde erwähnt, daß zur Laufzeitanalyse einer Applikation Performance Analysis Tools zur Verfügung stehen.
Für das auf vier Prozessoren des iPSC/860 mit der parallelen Implementation *send* berechnete Grötschel-Problem ist in Abbildung 4.17 die Graphik einer solchen Laufzeitanalyse dargestellt, die mit dem **ctool** erstellt wurde. Dort wird die Anzahl der Aufrufe der einzelnen Kommunikations- und Ein-/Ausgaberoutinen aufgeführt.
Anhand von CSEND und CRECV läßt sich deutlich das auf einem Binärbaum beruhende Unterteilungsschema und damit die Kommunikations- bzw. Verteilungsstruktur erkennen. Um eine Partition zu versenden, werden sieben Aufrufe von CSEND bzw. CRECV benötigt (a) und eine Teillösung einer Partition wird durch jeweils drei dieser Aufrufe verschickt (b).
Mit Hilfe des in Abbildung 4.6 dargestellten Unterteilungsschemas für vier Prozessoren können nun die Anzahlen der in Tabelle 4.7 aufgeführten Aufrufe der Kommunikationsroutinen ermittelt werden.

Die Übereinstimmung der ermittelten (Tabelle 4.7) zu den mit dem **ctool** gemessenen Werte hebt Abbildung 4.17 nochmals hervor. Weiter läßt die Graphik erkennen, daß alle Prozessoren einmal synchronisiert werden (GSYNC), um einen gemeinsamen Startstatus für die Berechnung zu erhalten.

Node	CSEND	CRECV
0	$14 = 2 \times a$	$6 = 2 \times b$
1	$10 = 1 \times a + 1 \times b$	$10 = 1 \times a + 1 \times b$
2	$3 = 1 \times b$	$1 \times a$
3	$3 = 1 \times b$	$1 \times a$

Tabelle 4.7: Anzahl der Aufrufe von CSEND bzw. CRECV bei der Implementation *send* bei Berechnung des Grötschel-Problems auf 4 Prozessoren des iPSC/860

Die Routine READ dient zur Eingabe der Punktkoordinaten und mit WRITE wird das Ergebnis ausgegeben. Die weiteren aufgeführten Routinen werden intern von Systemfunktionen und -routinen aufgerufen.

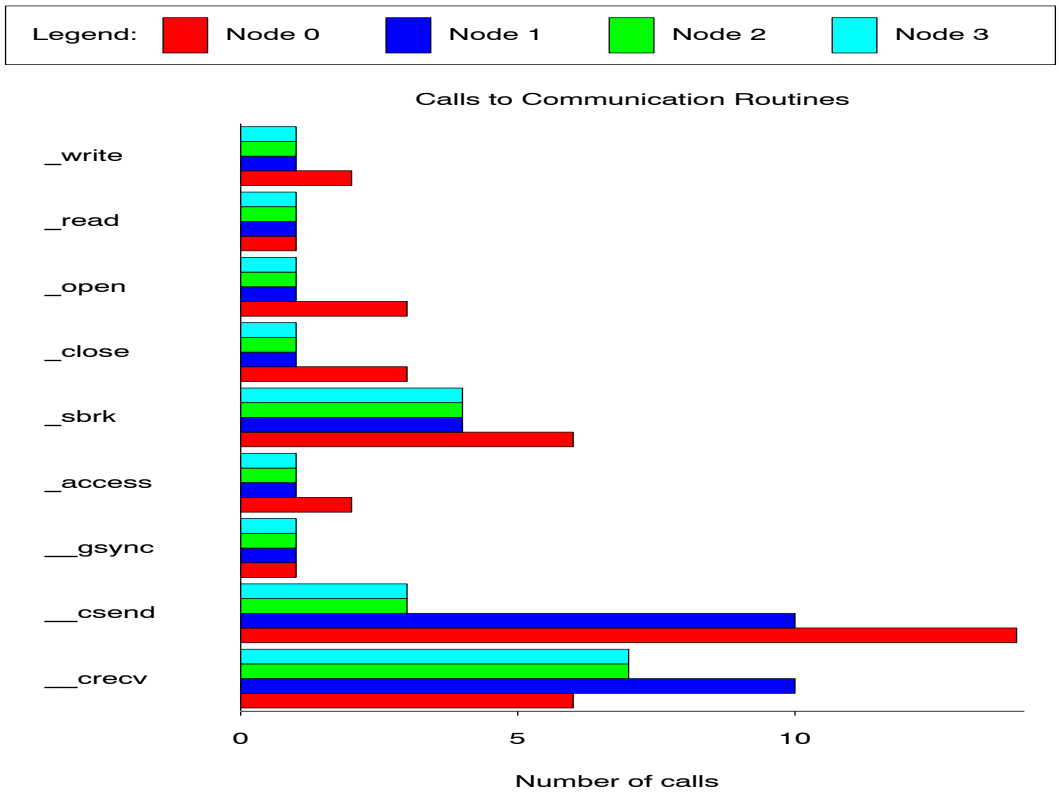


Abbildung 4.17: **ctool**: Anzahl der Aufrufe der Kommunikations- und Ein-/Ausgaberoutinen für das Grötschel-Problem auf dem iPSC/860 mit 4 Prozessoren

4.6 Vergleich der Ergebnisse mit denen anderer Rechnersysteme

Der Unterteilungsalgorithmus von Karp zur näherungsweisen Lösung des euklidischen Traveling-Salesman-Problems wurde im Rahmen von Diplomarbeiten im Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich bereits auf Rechnersystemen implementiert, denen im Gegensatz zum verteilten Speicher bei iPSC/860 und Paragon das Konzept des gemeinsamen Hauptspeichers zugrunde liegt.

In [Gur86] wird eine Variante beschrieben, die auf einem Rechner des Typs CRAY X-MP/22 implementiert wurde. Die CRAY X-MP/22 ist ein Vektor-Supercomputer mit zwei Prozessoren und gemeinsamem Hauptspeicher. Die Datenkommunikation der Prozessoren untereinander wird mittels dieses gemeinsamen Speichers realisiert. Das bei der damaligen Implementation verwendete Programmierkonzept war das *Macrotasking*.
Der erzielte Speedup auf der CRAY X-MP/22 liegt, unabhängig von der Problemgröße, in dem untersuchten Bereich um etwa 10% bis 15% niedriger als der Speedup auf den beiden hier benutzten massiv-parallelen Rechnern.
Eine neuere Untersuchung der Implementierung des Traveling-Salesman-Problems mittels Macrotasking auf einer CRAY X-MP/48 mit 4 Prozessoren brachte die gleichen Ergebnisse wie die Messung auf der CRAY X-MP/22.
Daneben wurde das TSP auf einer CRAY X-MP/416 mit Hilfe des Programmierkonzeptes *Microtasking* neu implementiert. Die bei dieser Untersuchung erzielten Speedup-Werte sind für Problemgrößen über 1000 Knoten vergleichbar mit denen der in dieser Arbeit beschriebenen Implementationen. Sie erreichen ebenfalls hohe Werte und liegen maximal um etwa 5% darunter. Für Problemgrößen unter 1000 Knoten sind die auf der CRAY X-MP/416 mit Microtasking erzielten Speedup-Werte jedoch deutlich besser als die auf den massiv-parallelen Rechnern erzielten [Kne92]. Bei diesen kleinen Problemen wirkt sich die Kommunikation bei der Berechnung des TSP auf den massiv-parallelen Rechnern entscheidend nachteilig auf die erreichte Performance aus.

In [Mül93] wurde der Unterteilungsalgorithmus von Karp auf einem Cluster von IBM RS/6000-Workstations implementiert. Das Cluster bestand aus vier Workstations und war mit einem Ethernet Local Area Network verbunden. Die Kommunikation der Workstations untereinander wurde mittels der Programmierumgebungen Network-Linda und Express realisiert. Bei dieser Implementation wurde die Kontroll- und Kommunikationsstruktur durch eine Kombination aus einem Binärbaum und einem Warteschlangensystem realisiert.
Der auf dem Workstation-Cluster erzielte Speedup lag zwischen 1.5 und 3.9 und war von der Problemgröße und von der Partitionsgröße der untersten Unterteilungsstufe abhängig. Bei großen Problemen überwog auch hier der Rechenaufwand gegenüber dem Kommunikationsaufwand, so daß auch dort ein nur geringfügig unter der Anzahl der Workstations liegender Speedup erreicht wurde.

Bei kleinen Problemgrößen wurde, wie bei der Messung auf massiv-parallelen Rechnern, der Speedup aufgrund des Kommunikations-Overhead geringer.
Die Untersuchungen mit den damals verfügbaren Software-Systemversionen zeigten, daß die Implementation mit Express bessere Ergebnisse liefert als die Implementation mit Network-Linda. Der Unterschied zwischen den erzielten Speedup-Werten beträgt maximal 10%.

Die auf den CRAY-Rechnern und dem IBM-Workstation-Cluster ermittelten Ergebnisse lassen sich jedoch nicht direkt mit denen auf den massiv-parallelen Rechnern iPSC/860 und Paragon XP/S 5 erzielten vergleichen. Den beiden in diesem Abschnitt behandelten früheren Implementationen ist gemeinsam, daß sie die in Unterabschnitt 4.1 beschriebene Variante realisieren. Anders als bei den in dieser Arbeit vorgestellten Implementationen definieren die Implementationen für CRAY-Rechner und IBM-Workstation-Cluster die Beherrschbarkeit einer Partition durch die Angabe der maximalen Knotenanzahl einer Partition. Diesen damaligen Implementationen liegt damit eine andere Partitionierungsstrategie, als die in dieser Arbeit vorgestellte, zugrunde. Ein weiterer wesentlicher Unterschied ist der, daß bei den Implementationen für CRAY-Rechner und IBM-Workstation-Cluster nur kleine Problemgrößen untersucht wurden.

Kapitel 5

Minimale spannende Bäume

Algorithmen für die Bestimmung von minimalen spannenden Bäumen (MST = minimum spanning tree) sind in der Graphentheorie sowie in praktischen Anwendungen von großer Bedeutung. In der Graphentheorie dienen sie als Basis, wenn zur Erstellung von anderen Algorithmen minimale spannende Bäume vorausgesetzt sind oder benötigt werden. Praktische Anwendung findet die Problematik des MST oft dann, wenn Punkte derart miteinander verbunden werden müssen, daß die Gesamtkosten der Verbindung minimal sein soll und deren Topologie frei wählbar ist.
Beispiele für solche praktischen Anwendungen finden sich in der Verkabelung von Versorgungs- und Verteilungsnetzen sowie in der Vernetzung von Rechensystemen durch Busse.

Das Problem des minimalen spannenden Baumes kann folgendermaßen formuliert werden:
Gegeben sei ein ungerichteter Graph $G = (V, E)$, dessen Kanten durch Gewichte bewertet sind (Abbildung 5.1).

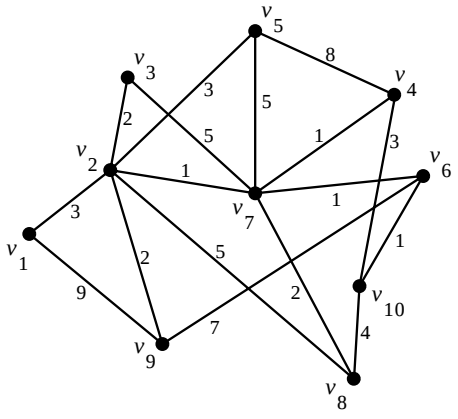


Abbildung 5.1: Ungerichteter Graph G

Aufgabe ist es, alle Knoten des Graphen durch einen minimalen spannenden Baum zu verbinden. In Abbildung 5.2 ist ein minimaler spannender Baum des Graphen G aus Abbildung 5.1 dargestellt. In der Regel gibt es keinen eindeutig bestimmten minimalen spannenden Baum, sondern mehrere.

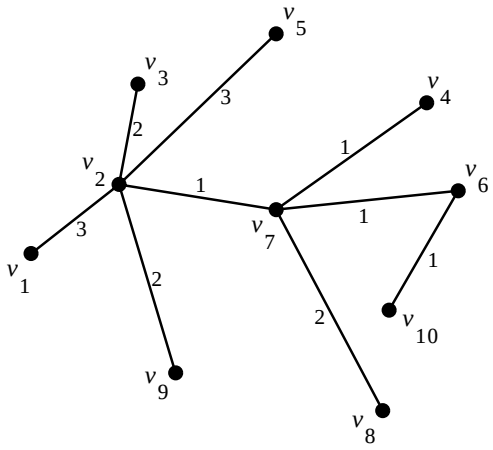


Abbildung 5.2: Minimaler spannender Baum des Graphen G

Zur Ermittlung eines minimalen spannenden Baumes eines Graphen stehen drei klassische Algorithmen zur Verfügung, die alle sogenannte **Greedy**-Algorithmen sind. Greedy-Algorithmen sehen in jedem Schritt nur die momentan beste Möglichkeit, das heißt hier die Kante mit minimalem Gewicht. Diese wird zu dem bereits bestehenden minimalen Teilbaum angefügt. Mit *minimalem Teilbaum* wird im folgenden der Teilbaum bezeichnet, dessen Gewicht minimal ist und der mit den noch anzufügenden Kanten den minimalen spannenden Baum bilden wird. Die drei Algorithmen sind:

- Algorithmus von Kruskal [Kru56]
- Algorithmus von Sollin [Sol77]
- Algorithmus von Prim-Dijkstra [Pri57], [Dij59]

Der Algorithmus von Kruskal baut in einem ersten Schritt eine Liste von Kanten aus E nach aufsteigenden Gewichten auf. Dann wird die Kante mit dem geringsten Gewicht gewählt. Diese bildet einen ersten minimalen Teilbaum. In den nächsten Schritten wird jeweils die Kante des Graphen hinzugefügt, die das kleinste Gewicht besitzt und mit den bereits gewählten Kanten keinen Kreis bildet. Der letzte Schritt wird solange ausgeführt, bis keine Kante mehr ohne Verletzung der Kreisfreiheit hinzugenommen werden kann. Der Algorithmus von Kruskal bestimmt einen minimalen spannenden Baum mit der Zeitkomplexität $O(|E| \cdot \log(|E|))$. Diese Zeitkomplexität wird im wesentlichen durch den ersten Schritt des Algorithmus bestimmt, da das Sortieren der $|E|$ Kanten des Graphen $O(|E| \cdot \log(|E|))$ erfordert.

Falls die Kante mit dem kleinsten Gewicht leicht zu finden ist, etwa durch Abspeicherung aller Kanten in einem *Heap*, kann auf das Sortieren der Kanten am Anfang des Algorithmus verzichtet werden. Der Heap ist ein spezieller Binärbaum, bei dem die Gewichte aller Sohnknoten größer oder gleich dem Gewicht des Vaterknotens sind [AHU83]. Das heißt, die Wurzel besitzt das kleinste Gewicht. Die gesuchte Kante ist dann in $O(\log(|E|))$ Schritten zu ermitteln. Der Aufbau des Heap erfordert eine Zeitkomplexität von $O(|E|)$, so daß auch diese Variante ohne Sortieren eine Zeitkomplexität von $O(|E| \cdot \log(|E|))$ besitzt [Gur86].
Einen Ansatz zur Parallelisierung des Algorithmus von Kruskal bietet im wesentlichen das Sortieren der $|E|$ Kanten des Graphen nach aufsteigenden Gewichten bzw. das Suchen der minimalen Kante in dem Heap.

Anders geht der Algorithmus von Sollin vor. In einem ersten Schritt wird jeder Knoten des Graphen mit seinem eindeutig bestimmten nächsten Nachbarn verbunden, das heißt mit dem Knoten, zu dem er die geringste Entfernung besitzt. Dann wird für jede so entstandene Komponente die kleinste Kante gesucht, mit der sie mit einer anderen Komponente verbunden werden kann. Diese Kanten werden so hinzugenommen, daß zwei Komponenten immer nur durch eine Kante verbunden und auftretende Kreise aufgelöst werden. Dieser Schritt wird solange wiederholt, bis nur noch eine Komponente existiert, die dann auch einen minimalen spannenden Baum bildet. Die Zeitkomplexität des Algorithmus von Sollin beträgt $O(|V|^2 \cdot \log(|V|))$. Die Suche nach der Kante mit dem geringsten Gewicht, die mit jeder Komponente inzident ist, benötigt $O(|V|^2)$ Vergleiche. Die Anzahl der Komponenten reduziert sich in jeder Iteration um einen Faktor von zwei, so daß folglich $O(\log(|V|))$ Iterationen nötig sind, um den minimalen spannenden Baum zu finden [Gur86].
Möglichkeiten zur Parallelisierung bietet der Algorithmus von Sollin bei der Suche nach der kleinsten Kante.

Der Algorithmus von Prim-Dijkstra ist implementiert worden und wird im folgenden Abschnitt näher erläutert.

5.1 Algorithmus von Prim-Dijkstra

Dieser Algorithmus wurde unabhängig voneinander von Prim (1957) und Dijkstra (1959) entwickelt. Er geht von der Idee aus, daß zu einem bereits gebildeten minimalen Teilbaum immer die Kante mit dem kleinsten Gewicht hinzugefügt wird, die mit einem Knoten innerhalb und einem außerhalb des Baumes inzident ist. Die Knoten außerhalb des Baumes erfahren eine temporäre Markierung, die innerhalb eine permanente. Die Abspeicherung des Graphen geschieht durch die Gewichtsmatrix.

Der Algorithmus kann wie folgt beschrieben werden:

Ein beliebiger Startknoten erhält eine permanente Marke. Dieser wird in dem Feld *near* mit einer 0 gekennzeichnet. Alle anderen Knoten, die noch nicht in dem minimalen Teilbaum enthalten sind, erhalten als temporäre Marke in *near* den Knoten des Baumes, zu dem sie die geringste Entfernung besitzen. Diese Entfernung wird in dem Feld *dist* abgespeichert. Nun wird unter allen Knoten mit temporärer Marke, der mit dem minimalen Wert in *dist* gesucht. In den Implementationen wird dies mittels einer Schleife realisiert. Der ausgewählte Knoten wird in den Baum aufgenommen und erhält eine permanente Marke. In einer weiteren Schleife werden nun für jeden Knoten außerhalb des Teilbaumes die in *dist* und *near* stehenden Daten, also seine geringste Entfernung zu dem Baum und damit der mit ihm adjazente Baumknoten, aktualisiert. Der Algorithmus endet, wenn n Knoten ausgewählt wurden, der minimale spannende Baum also aus $(n-1)$ Kanten besteht.
In der hier vorliegenden Implementation wurden nur ungerichtete Graphen betrachtet. Infolgedessen wird der Graph durch eine symmetrische Gewichtsmatrix dargestellt. Auch hier bietet, wie bei dem Traveling-Salesman-Problem (vgl. Unterabschnitt 4.3.3), die Matrix den Vorteil des direkten Zugriffs auf deren Elemente. Aus Speicherplatzgründen und zur Untersuchung größerer Probleme wurden nur die Elemente der Gewichtsmatrix oberhalb der Hauptdiagonalen betrachtet und in einem eindimensionalen Feld spaltenweise abgespeichert. Die Gewichte der Kanten wurden mittels eines Zufallszahlengenerators erzeugt und stellen INTEGER-Werte dar.

Die Ausgabe des Algorithmus besteht aus dem Gesamtgewicht des minimalen spannenden Baumes und aus einer Liste der Kanten, die den minimalen spannenden Baum bilden. Ist der zu untersuchende Graph nichtzusammenhängend, so erfolgt eine Fehlermeldung. Der Algorithmus stoppt, wenn keine auszuführende Kante mehr gefunden wird.

Die äußere Schleife, in der der minimale spannende Baum aufgebaut wird, wird $(n-1)$-mal durchlaufen. Die darin enthaltene Suche nach dem jeweiligen Minimum und die Aktualisierung benötigt höchstens n Schritte. Somit besitzt der sequentielle Algorithmus eine Zeitkomplexität von $O(n^2)$ für eine Problemgröße von n Knoten.

Der sequentielle Algorithmus wird auf den folgenden Seiten dargestellt.

Sequentieller Algorithmus von Prim-Dijkstra:

```
     PROGRAM PRIM_DIJKSTRA
     . . .
     begintime = DCLOCK()
     near(1)   = 0                                    !Permanente Marke
                                                     !des Anfangsknotens
     DO 1000 j = 2, n, 1
        near(j) = 1                                   !Temporaere Marken
        dist(j) = feldw(bases(j))                    !der anderen Knoten
1000 CONTINUE

     DO 2000 k = 2, n, 1      !Aufbau des minimalen spannenden Baumes

        DO 3000 i = 2, n, 1
           IF (near(i) .NE. 0) THEN           !Suche nach dem minimal
              IF (dist(i) .LT. min) THEN           !entfernten Knoten
                 next = i                    !und damit der naechsten
                 min  = dist(i)                 !einzufuegenden Kante
              ENDIF
           ENDIF
3000    CONTINUE

IF (Graph nicht zusammenhaengend) THEN
   Fehlermeldung und Abbruch!
ENDIF

        tree(k - 1, 1) = near(next)      !Einfuegen des neuen Knotens
        tree(k - 1, 2) = next
        weight         = weight + dist(next)
        near(next)     = 0                            !Permanente Marke
                                                     !des neuen Knotens
```

```
        DO 4000 j = 2, n, 1                          !Aktualisieren der
           IF (near(j) .NE. 0) THEN                 !Angaben der Knoten
              IF (j .LT. next) THEN                    !ausserhalb des
                 IF (j .GT. near(j)) THEN                  !Teilbaumes
                    IF (feldw(bases(next) + j - 1) .LT.
    >                      feldw(bases(j) + near(j) - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(next) + j - 1)
                    ENDIF
                 ELSEIF (j .LT. near(j)) THEN
                    IF (feldw(bases(next) + j - 1) .LT.
    >                     feldw(bases(near(j)) + j - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(next) + j - 1)
                    ENDIF
                 ENDIF
              ELSEIF (j .GT. next) THEN
                 IF (j .GT. near(j)) THEN
                    IF (feldw(bases(j) + next - 1) .LT.
    >                      feldw(bases(j) + near(j) - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(j) + next - 1)
                    ENDIF
                 ELSEIF (j .LT. near(j)) THEN
                    IF (feldw(bases(j) + next - 1) .LT.
    >                      feldw(bases(near(j)) + j - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(j) + next - 1)
                    ENDIF
                 ENDIF
              ENDIF
           ENDIF
4000    CONTINUE

2000 CONTINUE

     endtime = DCLOCK()

     END
```

5.2 Parallelisierung des Algorithmus von Prim-Dijkstra

Anders als bei der Implementation des Unterteilungsalgorithmus von Karp wurde ausgehend von dem sequentiellen der parallele Algorithmus entwickelt. Die Parallelisierung des Algorithmus von Prim-Dijkstra basiert auf der Aufteilung der Spalten der Gewichtsmatrix und damit der zu untersuchenden Knoten auf die einzelnen Prozessoren. Dies ist in der oben aufgeführten sequentiellen Implementation dadurch erreichbar, daß die Laufindizes der Schleifen, in denen einerseits der einzufügende Knoten ermittelt und andererseits die Aktualisierung der Angaben für die Knoten außerhalb des Baumes durchgeführt wird, durch andere Werte als die in der sequentiellen Implementation charakterisiert sind. Die Schleife beginnt für jeden Prozessor bei $(me + 2)$ und endet bei n. Die Schrittweite beträgt n/p. (me = logische Prozessornummer, n = Knotenanzahl, p = Prozessoranzahl.) Jeder Prozessor untersucht somit n/p Spalten. Gleichzusetzen damit ist aber, daß soviele kleinste Entfernungen und potentielle neue Baumknoten ermittelt werden, wie an der Berechnung beteiligte Prozessoren vorhanden sind. Es darf jedoch nur ein Knoten mit der dazugehörenden minimalen Entfernung gefunden werden. Das heißt, dieser Knoten und seine Entfernung zu dem bereits gebildeten Teilbaum muß in einem weiteren Schritt global ermittelt und dann allen Prozessoren bekannt gemacht werden. Ist der gesuchte Knoten allen Prozessoren bekannt, kann jeder Prozessor für sich den neuen minimalen Teilbaum aufbauen und dessen Gewicht neu berechnen. Den Neuaufbau der *near*- und *dist*-Felder und die Berechnung des nächsten lokalen Minimums führt jeder Prozessor dann wieder auf seinem eigenen Datensatz aus.

Die Aufteilung der Spalten und damit der zu untersuchenden Knoten kann sowohl zyklisch als auch blockweise vorgenommen werden. Bei vier Prozessoren und zehn Spalten werden bei der zyklischen Verteilung dem ersten Prozessor die Spalten 1, 5 und 9 zugeordnet, dem zweiten die Spalten 2, 6 und 10. Entsprechend werden die restlichen Spalten auf die verbleibenden zwei Prozessoren verteilt. Bei der blockweisen Verteilung erhält jeder Prozessor eine bestimmte Anzahl von hintereinander liegenden Spalten der Gewichtsmatrix.
In der hier vorliegenden Implementation wurde die zyklische Verteilung der Spalten auf die Prozessoren gewählt. Grundsätzlich ist auch die blockweise Aufteilung möglich, doch bietet die zyklische den Vorteil der besseren Lastverteilung. Da in der vorliegenden Implementation eine obere Dreiecksmatrix aufgeteilt werden muß, wird mit der zyklischen Verteilung sichergestellt, daß alle Prozessoren etwa gleich viele Matrixelemente bei der Berechnung zu untersuchen haben. Würde blockweise verteilt werden, so müßte der erste Prozessor sehr wenige und der logische letzte Prozessor sehr viele Elemente untersuchen.

Für die Ermittlung des globalen Minimums der Entfernungen, und damit auch des in den minimalen Teilbaum einzufügenden Knotens, sind drei Varianten entwickelt und implementiert worden. Die Kommunikation geschieht in allen drei Varianten im synchronen Modus.

Die erste Variante besteht darin, daß für die Bestimmung des globalen Minimums eine globale Operation, **GILOW** (*min, 1, work*), benutzt wurde, die sowohl auf dem iPSC/860 als auch auf dem Paragon verfügbar ist. Jeder Prozessor multipliziert sein lokales Minimum mit der gleichen sehr großen Zahl, zum Beispiel 1000000, und addiert seine logische Prozessornummer dazu. Dann ruft er **GILOW** mit diesem Wert in *min* auf. Nach Ausführung von **GILOW** besitzt nun jeder Prozessor in *min* das globale Minimum dieses Wertes. Mit dem Ausdruck ($me = min$ `mod` 1000000) stellt er fest, ob er die Kante besitzt, die in den minimalen Teilbaum aufgenommen werden soll. Nur der Prozessor, der diese Kante besitzt, fügt sie zu seinem Teilbaum hinzu und ermittelt das Teilgewicht seines Baumes. Bei dieser Variante werden von den Prozessoren kantendisjunkte minimale Teilbäume erstellt, die am Ende zu dem resultierenden minimalen spannenden Baum vereinigt werden. Zur Ermittlung des Gesamtgewichtes des minimalen spannenden Baumes aus den Teilgewichten (*weight*) der kantendisjunkten Teilbäume, die auf allen Prozessoren verteilt liegen, muß am Ende der Berechnung die globale Summenoperation **GISUM**(*weight, 1, work*) aufgerufen werden. Mit dieser Variante konnten Untersuchungen mit allen möglichen Prozessoranzahlen vorgenommen werden.
Bei der zweiten Variante wurde die in Unterabschnitt 4.3.1 beschriebene Baumstruktur verwendet. Der Sender schickt sein lokales Minimum und den dazugehörenden Knoten dem Empfänger, der dann den kleinsten Wert berechnet. Die Kommunikationsstruktur folgt dabei dem Kombinationsschema des parallelen Unterteilungsalgorithmus von Karp. Am Ende eines solchen Schrittes besitzt der logische erste Prozessor den einzubindenden Knoten und dessen Entfernung zum bereits gebildeten minimalen Teilbaum. Mit einem Broadcast werden diese Informationen allen anderen Prozessoren bekannt gemacht. Die Kommunikationstiefe beträgt dabei $O(\log(p))$. Zu beachten ist hierbei jedoch, daß nur Prozessoranzahlen, die Potenzen von 2 sind, verwendet werden können.
Um beliebige Prozessoranzahlen verwenden zu können, wurde in der dritten Variante ein Ring als Kommunikationsstruktur implementiert. Dessen Tiefe beträgt $O(p/2)$. Jeder Prozessor schickt seinem ihm zugeordneten Empfänger sein momentanes lokales Minimum mit dazugehörendem Knoten. Gleichzeitig empfängt er von einem Sender dessen Werte. Er berechnet aus diesen und seinen eigenen wiederum das Minimum mit dem dazugehörenden Knoten. Nach $(p/2 + 1)$ Schritten besitzt jeder Prozessor den einzufügenden Knoten und dessen Entfernung. Bei dieser Variante kann auf ein Broadcast verzichtet werden.

Die Schleife, in der der minimale spannende Baum aufgebaut wird, wird in der parallelen Implementation ebenfalls $(n-1)$-mal ausgeführt. Im Gegensatz zur sequentiellen Implementation werden die Schleifen zur Bestimmung des lokalen Minimums und der Aktualisierung jedoch nur (n/p)-mal ausgeführt. Zu dieser Zeitkomplexität wird dann noch die Zeitkomplexität der Kommunikation addiert, die bei der Baumstruktur $O(\log(p))$ und bei der Ringstruktur $O(p/2)$ beträgt. Die Zeitkomplexität der globalen Operationen konnte nicht ermittelt werden. So summiert sich die Zeitkomplexität des parallelen Algorithmus mit Baumstruktur zu $O(n^2/p+n\log(p))$ und die des Algorithmus mit Ringstruktur zu $O(n^2/p + (n \cdot p)/2)$ bei einer Problemgröße von n Knoten und bei p Prozessoren.

Stellvertretend wird auf den folgenden zwei Seiten die Implementation mit der Ringstruktur vorgestellt.

```
     PROGRAM PRIM_DIJKSTRA
     . . .
     me = MYNODE()
     p  = NUMNODES()
     IF (me .EQ. 0) begintime = DCLOCK()

     near(1) = 0                !Permanente Marke des Anfangsknotens
     DO 1000 j = me + 2, n, p                      !Temporaere Marken
        near(j) = 1                               !der anderen Knoten
        dist(j) = feldw(bases(j))
1000 CONTINUE
     zaehl = 1

     DO 2000 k = 2, n, 1                        !Aufbau des minimalen
                                                   !spannenden Baumes
        DO 3000 i = me + 2, n, p               !Suche nach dem minimal
           IF (near(i) .NE. 0) THEN               !entfernten Knoten
              IF (dist(i) .LT. help(1, 2)) THEN       !und damit der
                 help(1, 1) = i             !naechsten einzufuegenden
                 locali     = i                              !Kante
                 help(1, 2) = dist(i)
              ENDIF
           ENDIF
3000    CONTINUE

IF (Graph nicht zusammenhaengend) THEN
   Fehlermeldung und Abbruch!
ENDIF
                         !Globales Ermitteln der minimalen Entfernung
        DO 4000 j = 1, INT(p/2) + 1, 1                        !Sender
           CALL CSEND(11 * j * k, help, 8, MOD((me + j), p), 0)
           CALL CRECV(11 * j * k, hilf, 8)                !Empfaenger
           IF (help(1, 2) .GT. hilf(1, 2)) THEN
              help(1, 2) = hilf(1, 2)
              help(1, 1) = hilf(1, 1)
           ELSEIF ((help(1, 2) .EQ. hilf(1, 2)) .AND.
    >              (help(1, 1) .GT. hilf(1, 1)))     THEN
              help(1, 1) = hilf(1, 1)
           ENDIF
4000    CONTINUE

        next = help(1, 1)
```

```
        IF (next .EQ. locali) THEN       !Einfuegen des neuen Knotens
           tree(zaehl, 1) = near(next)
           tree(zaehl, 2) = next
           zaehl          = zaehl + 1
        ENDIF
        weight     = weight + help(1, 2)
        near(next) = 0            !Permanente Marke des neuen Knotens
        DO 5000 j = me + 2, n, p                   !Aktualisieren der
           IF (near(j) .NE. 0) THEN               !Angaben der Knoten
              IF (j .LT. next) THEN            !ausserhalb des Baumes
                 IF (j .GT. near(j)) THEN
                    IF (feldw(bases(next) + j - 1) .LT.
     >                   feldw(bases(j) + near(j) - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(next) + j - 1)
                    ENDIF
                 ELSEIF (j .LT. near(j)) THEN
                    IF (feldw(bases(next) + j - 1) .LT.
     >                   feldw(bases(near(j)) + j - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(next) + j - 1)
                    ENDIF
                 ENDIF
              ELSEIF (j .GT. next) THEN
                 IF (j .GT. near(j)) THEN
                    IF (feldw(bases(j) + next - 1) .LT.
     >                   feldw(bases(j) + near(j) - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(j) + next - 1)
                    ENDIF
                 ELSEIF (j .LT. near(j)) THEN
                    IF (feldw(bases(j) + next - 1) .LT.
     >                   feldw(bases(near(j)) + j - 1)) THEN
                       near(j) = next
                       dist(j) = feldw(bases(j) + next - 1)
                    ENDIF
                 ENDIF
              ENDIF
           ENDIF
5000    CONTINUE
2000 CONTINUE

     IF (me .EQ. 0) endtime = DCLOCK()

     END
```

Im folgenden sind die Programmbereiche, die die Verteilung bei den anderen beiden Implementationsvarianten realisieren, dargestellt.

Variante mit Baumstruktur:

```
        . . .
        DO 4000 j = log(p), 1, -1
           IF ((me .GT. ((2 ** (j - 1)) - 1)) .AND.           !Sender
     >        (me .LE. ((2 ** j) - 1)))          THEN
              address = me - (2 ** (j - 1))
              CALL CSEND(11 * j * k, help, 8, address, 0)
           ELSEIF (me .LE. ((2 ** (j - 1)) - 1)) THEN     !Empfaenger
              CALL CRECV(11 * j * k, hilf, 8)
              IF (help(1, 2) .GT. hilf(1, 2)) THEN
                 help(1, 2) = hilf(1, 2)
                 help(1, 1) = hilf(1, 1)
              ELSEIF ((help(1, 2) .EQ. hilf(1, 2)) .AND.
     >                (help(1, 1) .GT. hilf(1, 1)))     THEN
                 help(1, 1) = hilf(1, 1)
              ENDIF
           ENDIF
4000    CONTINUE
        IF (me .EQ. 0) THEN                                !Broadcast
           CALL CSEND(12 * k, help, 8, -1, 0)
        ELSE
           CALL CRECV(12 * k, help, 8)
        ENDIF
```

Variante mit globalen Operationen:

```
        . . .
        min = min * 1000000 + me
        CALL GILOW(min, 1, work)                    !Globales Minimum
        node = MOD(min, 1000000)
        IF (me .EQ. node) THEN           !Einfuegen des neuen Knotens
           CALL CSEND(11, next, 4, -1, 0)                    !Sender
           tree(zaehl, 1) = near(next)       !(next fuer die Aktual.)
           tree(zaehl, 2) = next
           zaehl          = zaehl + 1
           weight         = weight + dist(next)
        ELSE
           CALL CRECV(11, next, 4)                       !Empfaenger
        ENDIF
        near(next) = 0            !Permanente Marke des neuen Knotens
        . . .
        CALL GISUM(weight, 1, work)                   !Globale Summe
```

5.3 Ergebnisse

Der oben beschriebene Algorithmus wurde sequentiell und parallel in den drei verschiedenen Varianten auf dem iPSC/860 und dem Paragon XP/S 5 implementiert.

Die ausführbaren Programme wurden mit der Compiler-Optimierungsstufe *4* übersetzt. Diese Stufe erreichte sequentiell sowie parallel die schnellsten Ausführungszeiten. Die Ausführungszeiten und der daraus berechnete Speedup der einzelnen Optimierungsstufen unterscheiden sich bis zu 35% voneinander. Die Messungen bei den verschiedenen Prozessoranzahlen wurden auf dem iPSC/860 sowie auf dem Paragon bis auf die größtmöglichen, das heißt 32 Prozessoren beim iPSC/860 und 72 beim Paragon, im interaktiven Betrieb und die anderen im Batch-Betrieb durchgeführt.
Bei der Auswertung der Ergebnisse zeigte es sich, daß die parallele Implementation mit globaler Minimumssuche und Summenoperation bei kleineren Prozessoranzahlen die längsten Ausführungszeiten erreichte. In Tabelle 5.1 werden die auf dem iPSC/860 mit 8 Prozessoren erzielten Ausführungszeiten in Millisekunden der drei Kommunikationsvarianten miteinander verglichen.

# Knoten	Baum	Ring	Globale Op.
700	465.66	629.20	652.37
1700	1705.42	2107.53	2405.45
2700	3568.90	4213.55	5025.77

Tabelle 5.1: Ausführungszeiten der drei verschiedenen Kommunikationsvarianten in Millisekunden auf 8 Prozessoren des iPSC/860 bei verschiedenen Problemgrößen

Obwohl die globalen Operationen GILOW und GISUM optimal implementiert sind, bewirkt deren Ausführung die, im Vergleich, ungünstigeren Werte. Nur bei vielen Prozessoren, das heißt mehr als 10, war eine geringfügige Verbesserung der Zeiten gegenüber der Ringstruktur zu erkennen. Die kleineren Ausführungszeiten der Variante mit globalen Operationen gegenüber denen mit Ringstruktur machte sich bei der Ermittlung der Speedup- und Effizienzwerte jedoch nicht bemerkbar. Es wurde daher auf weitere Messungen mit der Variante, die globale Operationen verwendet, verzichtet. Die Baumstruktur lieferte bei allen Untersuchungen aufgrund der geringen Kommunikationstiefe von $O(\log(p))$ die schnellsten Ausführungszeiten.

Auf dem iPSC/860 lassen sich nur Cubes allokieren, deren Prozessoranzahlen Potenzen von 2 sind. Deshalb wurde auf diesem Rechner ausschließlich mit der schnellsten parallelen Implementation, der mit Baumstruktur, gearbeitet.

Auf dem Paragon hingegen können Partitionen mit beliebiger Prozessoranzahl angefordert werden. Hier wurde sowohl mit der Baum- als auch mit der Ringstruktur gemessen. Bei Potenzen von 2 als Prozessoranzahl wurde auch auf dem Paragon nur die Variante mit Baumstruktur eingesetzt. Standen andere Prozessoranzahlen zur Verfügung, konnte die Variante mit Baumstruktur nicht benutzt werden. Dann wurde auf die Variante mit Ringstruktur zurückgegriffen.

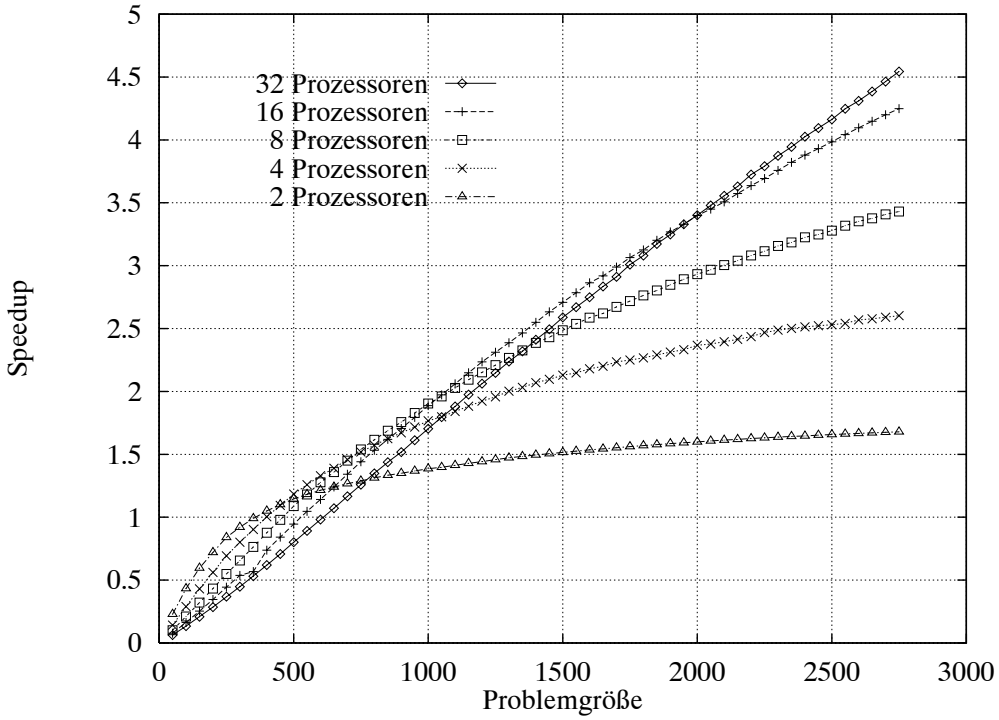


Abbildung 5.3: Erreichter Speedup in Abhängigkeit von der Problemgröße bei der Implementation mit Baumstruktur auf dem iPSC/860 bei Prozessoranzahlen von 2 bis 32

In den Abbildungen 5.3 und 5.4 sind die auf dem iPSC/860 und dem Paragon erzielten Speedup-Werte aufgetragen. Die Abbildungen 5.5 und 5.6 zeigen die daraus ermittelten Effizienzen.
Deutlich erkennt man, daß nur bei kleinen Prozessoranzahlen ein befriedigendes Ergebnis erreicht wird. Der Grund dafür liegt in der feinen Granularität des Algorithmus von Prim-Dijkstra. Das Verhältnis von Rechenaufwand zu Kommunikationsaufwand ist sehr ungünstig. Im Gegensatz zum Unterteilungsalgorithmus von Karp müssen hier zu viele und zu kleine Nachrichten versendet werden. Der Kommunikations-Overhead wirkt sich so besonders bei vielen Prozessoren sehr nachteilig aus. Erst bei sehr großen Problemen, die hier aufgrund der geringen Kapazität des Speichers der Rechenknoten nicht berechnet werden konnten, kann mit einer Effizienz größer als 50% gerechnet werden.

Wie schon bei den Ergebnissen der Implementation des Unterteilungsalgorithmus von Karp (siehe Abschnitt 4.5) deutlich wurde, sind auch hier die Speedup- und damit auch die Effizienzwerte, die auf dem Paragon erzielt wurden, niedriger als die auf dem iPSC/860. Der Grund ist der gleiche wie dort. Der im Vergleich zum iPSC/860 schnellere Prozessor des Paragon XP/S 5 bewirkt, daß sich, trotz der größeren Übertragungsleistung des Verbindungsnetzwerkes, der erhöhte Kommunikations-Overhead beim Paragon negativer in den erzielten Beschleunigungen bemerkbar macht als beim iPSC/860.

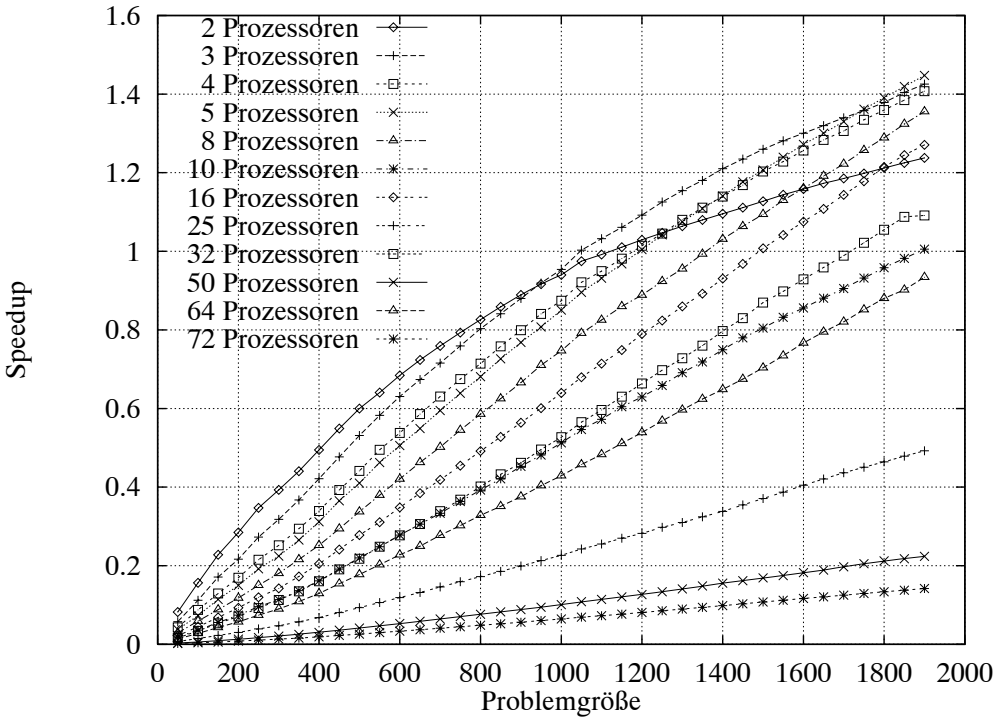


Abbildung 5.4: Erreichter Speedup in Abhängigkeit von der Problemgröße bei der Implementation mit Baumstruktur für Potenzen von 2 und der Implementation mit Ringstruktur für andere Prozessoranzahlen auf dem Paragon

Die Speedup- und Effizienzwerte zeigen recht deutlich, daß die Bearbeitung feingranularer Algorithmen, wie der von Prim-Dijkstra, mit einer großen Prozessoranzahl schlechtere Ergebnisse liefert als die mit einer geringeren. Im Falle sehr großer Prozessoranzahlen wird sogar eine Verlangsamung erreicht. Die Ergebnisse, die auf dem iPSC/860 erzielt wurden, zeigen, daß maximal 8 Prozessoren verwendet werden sollten. Durch den Einsatz von mehr Prozessoren wird keine wesentliche Verbesserung erreicht. Die Problemgrößen, die auf dem Paragon berechnet werden konnten, waren zu gering und ließen keine eindeutige Aussage zu. Aber auch hier ist zu erkennen, daß durch den Einsatz vieler Prozessoren keine verbesserten Werte gegenüber dem Einsatz weniger Prozessoren erzielt werden.

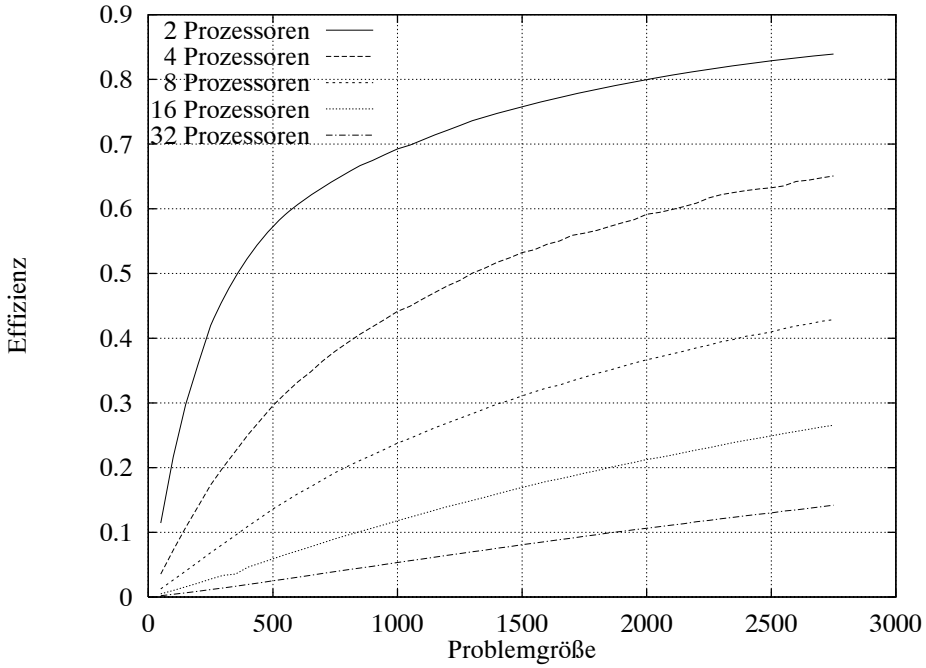


Abbildung 5.5: Effizienzen in Abhängigkeit von der Problemgröße bei der Implementation mit Baumstruktur auf dem iPSC/860 bei Prozessoranzahlen von 2 bis 32

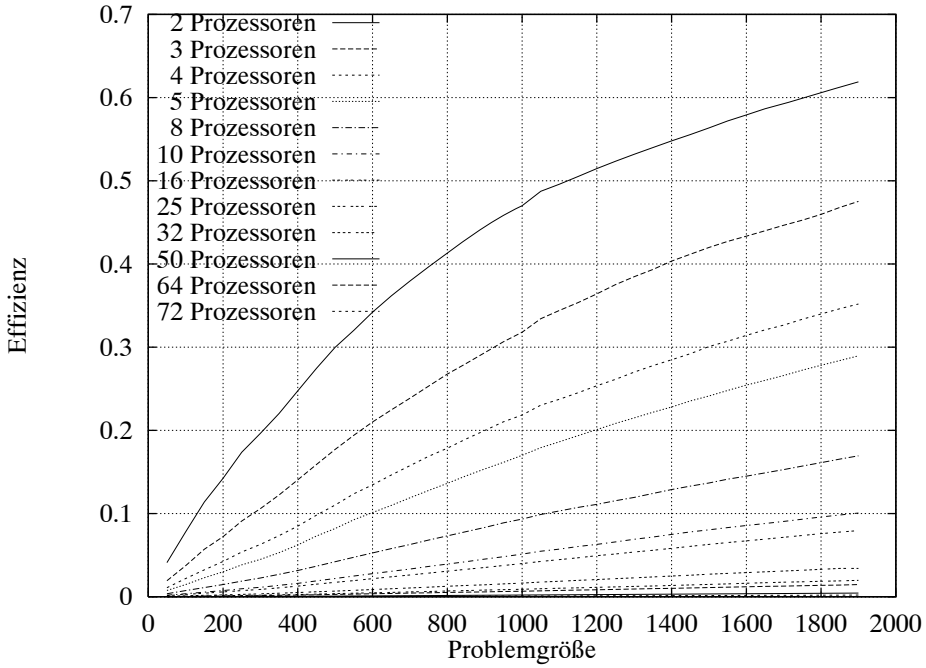


Abbildung 5.6: Effizienzen in Abhängigkeit von der Problemgröße bei der Implementation mit Baumstruktur für Potenzen von 2 und der Implementation mit Ringstruktur für andere Prozessoranzahlen auf dem Paragon

Mit den oben vorgestellten drei parallelen Implementationen konnten aufgrund der Speicherplatzbeschränkung nur kleine Problemgrößen untersucht werden. Um auch Ausführungszeiten für größere Probleme ermitteln zu können, wurde die Eingabe der Spalten der Gewichtsmatrix parallelisiert. Jeder Prozessor liest nur genau die Spalten ein, die er zu untersuchen hat, also anstatt n nur n/p. Die Problemgröße n kann so in Abhängigkeit von der Prozessoranzahl p deutlich erhöht werden.

Neben dem parallelen Erzeugen der Spalten für die Gewichtsmatrix durch die einzelnen Prozessoren wurde auch ein Verfahren implementiert, mit dessen Hilfe die Prozessoren die Spalten parallel aus mehreren Dateien einlesen können. Das parallele Lesen aus einer Datei läßt sich mit dem Concurrent File System (CFS) realisieren. Das CFS wurde hier jedoch nicht benutzt. Das Verfahren des parallelen Einlesens aus mehreren Dateien wurde bei 2 Prozessoren und für sehr kleine Problemgrößen eingesetzt.

Der Nachteil bei dieser Vorgehensweise ist, daß kein Vergleich mit der sequentiellen Implementation möglich ist, denn mit dieser können nur Problemgrößen bis 2750 Knoten untersucht werden. Vorteilhaft ist aber, daß durch diese Parallelisierung größere Probleme gelöst werden können, auch wenn die Beschleunigung gering ist. Als weitere Schwierigkeit stellte sich die Aktualisierung der *near*- und *dist*-Felder der noch nicht in den minimalen Teilbaum eingefügten Knoten heraus. Dazu werden Informationen benötigt, die aufgrund der parallelen Eingabe nur auf einem Prozessor zur Verfügung stehen. Nur bei der Variante mit globalen Operationen läßt sich ohne zusätzlichen hohen Verwaltungsaufwand feststellen, welcher Prozessor diese Informationen, das heißt den als nächsten einzufügenden Knoten und die entsprechende Spalte der Gewichtsmatrix, besitzt. Mit einem zusätzlichen Broadcast wird diese den anderen Prozessoren zugesandt.

Das Ergebnis der Untersuchung zeigt Tabelle 5.2. Bei verschiedenen Prozessoranzahlen werden die ermittelten Ausführungszeiten den von der Anzahl der Prozessoren abhängigen Problemgrößen gegenübergestellt. Die Kantengewichte wurden mit einem Zufallszahlengenerator berechnet.

# Prozessoren	1	2	4	8	16
Problemgröße	2750	5500	10500	13000	15500
Ausführungszeit [ms]	8034.82	49919.40	224419.26	439653.63	755039.89

Tabelle 5.2: Ausführungszeiten in Millisekunden der von den Prozessoranzahlen abhängigen größtmöglichen Problemgrößen auf dem iPSC/860 bei der Implementationsvariante mit globalen Operationen

Die hohen Ausführungszeiten sind auf die globalen Operationen und deren oftmaliges Wiederholen wegen den großen Problemgrößen zurückzuführen.

Die Kommunikation einer MST-Berechnung von 2700 Knoten auf vier Prozessoren des iPSC/860 wurde mit dem **ctool** untersucht. Von den drei oben beschriebenen Kommunikationsvarianten wurde die Baumstruktur benutzt. In Abbildung 5.7 ist das Ergebnis dargestellt. Die Graphik zeigt die Zeit in Millisekunden an, die die Prozessoren für die Ausführung der relevanten Kommunikations- und der Ein-/Ausgaberoutinen verbrauchen.
Deutlich ist die lange Wartezeit auf den Empfang der Nachrichten bei CRECV erkennbar. Im Gegensatz dazu verbraucht das Versenden der Nachrichten nur etwa 20% der zum Empfang benötigten Zeit.
Die Prozessoren 1, 2 und 3 benötigen mehr Zeit zur Synchronisation als Prozessor 0, da dieser mehr Knoten zu untersuchen hat und so die anderen drei auf Prozessor 0 warten müssen.
Die Werte von WRITE zeigen die parallele Ausgabe der zum minimalen spannenden Baum gehörenden Kanten. Deutlich wird deren Verteilung auf die Prozessoren sichtbar. Prozessor 0 steuert bei diesem Beispiel im Gegensatz zu den anderen nur wenige Kanten zum MST bei, obwohl er hier die meisten Knoten, also auch Spalten der Gewichtsmatrix, zu untersuchen hat.

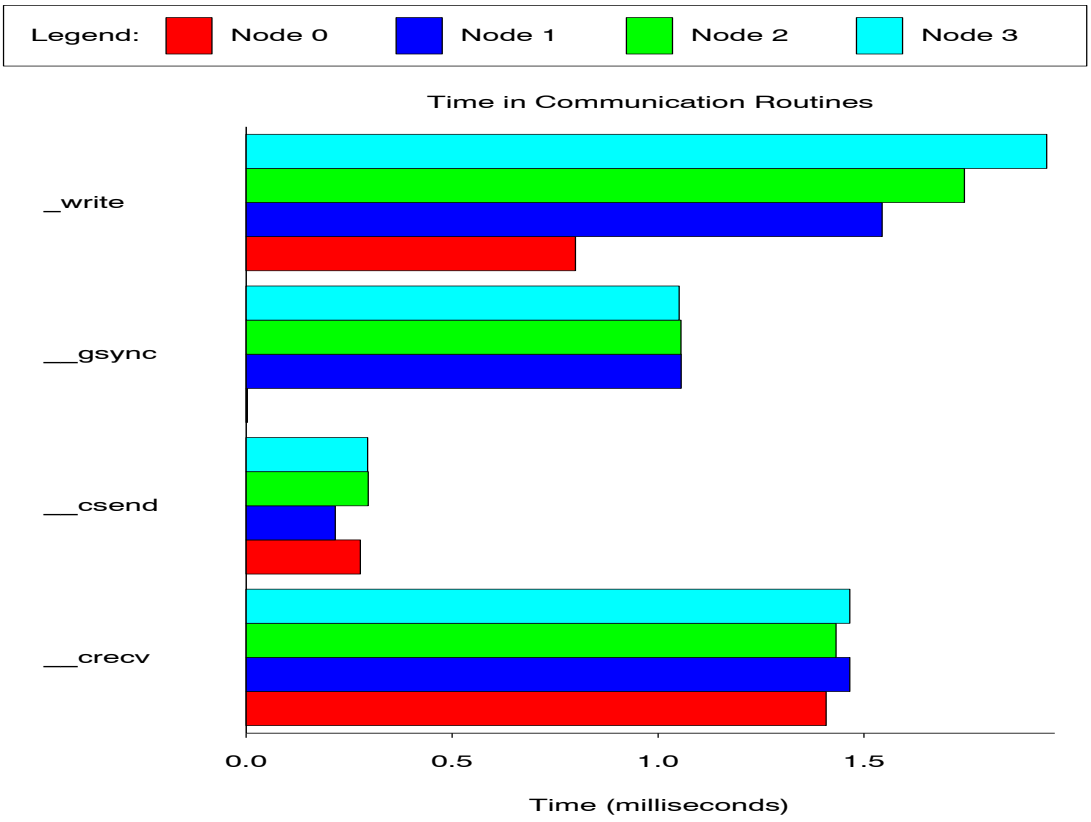


Abbildung 5.7: **ctool**: Verbrauchte Zeit der relevanten Kommunikations- und Ein-/Ausgaberoutinen für die Berechnung eines MST auf dem iPSC/860 mit 4 Prozessoren mittels der Variante mit Baumstruktur

5.4 Vergleich der Ergebnisse mit denen anderer Rechnersysteme

Der Algorithmus von Prim-Dijkstra wurde bereits im Rahmen einer Diplomarbeit im Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich auf einem Rechner des Typs CRAY X-MP/22 implementiert [Gur86]. Die CRAY X-MP/22 ist ein Shared-Memory-Rechner und besitzt zwei Prozessoren. Das Programmierkonzept, mit dem der Algorithmus von Prim-Dijkstra damals implementiert wurde, ist das *Macrotasking*.
Da der Kommunikationsaufwand bei den massiv-parallelen Rechnern gegenüber dem Rechenaufwand sehr groß ist und auf der CRAY X-MP/22 nicht kommuniziert werden muß, konnten dort sehr viel bessere Ergebnisse als die in dieser Arbeit vorgestellten erzielt werden. Schon bei relativ kleinen Problemgrößen wird auf der CRAY ein akzeptabler Speedup erreicht. Es wurde dort jedoch nur bis 600 Knoten gemessen. Auf dem iPSC/860 wurde bei 600 Knoten ein um 20% und auf dem Paragon ein um 50% geringerer Speedup erzielt. In der damaligen Arbeit wurden dicht- und dünnbesetzte Adjazenzmatrizen unterschieden. Eine solche Unterscheidung wurde hier nicht vorgenommen, sondern es wurden nur dichtbesetzte Matrizen untersucht.

Kapitel 6

Zusammenhangskomponenten

Eine weitere interessante Fragestellung in der Graphentheorie ist das Problem des Zusammenhangs von Graphen. Für diese Problemstellung können die gleichen theoretischen und praktischen Anwendungsgebiete wie bei den minimalen spannenden Bäumen genannt werden.

Gegeben sei ein ungerichteter Graph $G = (V, E)$, der aus mehreren Zusammenhangskomponenten besteht. Die Kanten des Graphen sind durch Gewichte bewertet. In Abbildung 6.1 ist ein nichtzusammenhängender Graph dargestellt. Der Graph G besitzt drei Zusammenhangskomponenten, wobei Knoten v_5 ein isolierter Knoten ist.

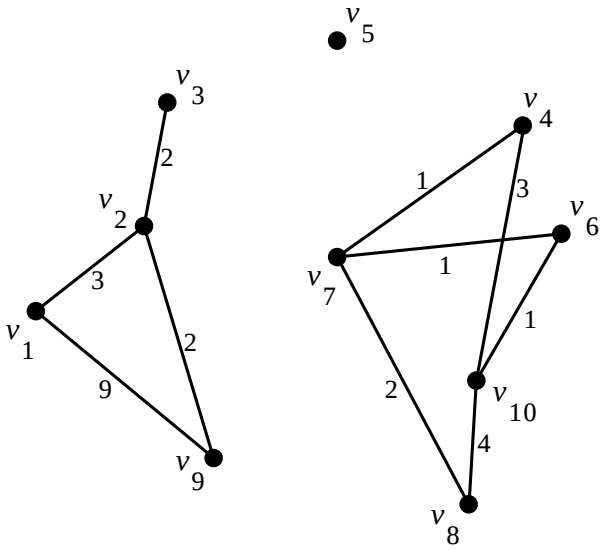


Abbildung 6.1: Nichtzusammenhängender ungerichteter Graph G

Gesucht sind nun neben der Anzahl der Zusammenhangskomponenten auch die Knoten, die diese Komponenten bilden.

Jede Komponente besitzt einen spannenden Baum, dessen Gewicht für diese Komponente minimal ist. Er wird im weiteren als *minimaler spannender Komponentenbaum* bezeichnet. Die Gesamtheit der minimalen spannenden Komponentenbäume aller Komponenten bildet einen minimalen spannenden Wald.
In Abbildung 6.2 sind die minimalen spannenden Komponentenbäume, die den minimalen spannenden Wald des Graphen G bilden, dargestellt. Wie minimale spannende Bäume sind auch minimale spannende Wälder nicht eindeutig bestimmt.

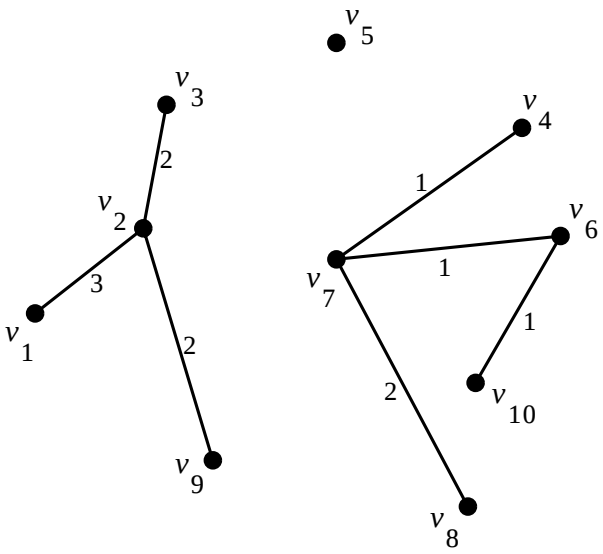


Abbildung 6.2: Minimaler spannender Wald des Graphen G

Auf der Basis des Algorithmus von Prim-Dijkstra zur Bestimmung eines minimalen spannenden Baumes wurde ein Algorithmus entwickelt, mit dessen Hilfe Zusammenhangskomponenten und minimale spannende Wälder bestimmt werden können. Dieser erweiterte Algorithmus wird in den folgenden Abschnitten näher erläutert.

6.1 Algorithmus zur Bestimmung von Zusammenhangskomponenten

Die Grundlage des hier beschriebenen Algorithmus zur Bestimmung der Zusammenhangskomponenten und eines minimalen spannenden Waldes eines Graphen bildet der Algorithmus von Prim-Dijkstra.
Dieser in Abschnitt 5.1 beschriebene Algorithmus dient zur Bestimmung eines minimalen spannenden Baumes. Bei Eingabe eines nichtzusammenhängenden Graphen stoppt er genau an der Stelle, an der keine weitere einzufügende Kante mehr gefunden wird, also alle Knoten einer Zusammenhangskomponente vollständig durchsucht wurden.

Der, aus dem von Prim-Dijkstra, neu entwickelte Algorithmus setzt an dieser Stelle wieder an. Jeder Knoten erhält darin mit *marked* eine zweite permanente Markierung. Die vom Algorithmus von Prim-Dijkstra her bekannte permanente Marke des *near*-Feldes ist hier lokal innerhalb der Berechnung einer Zusammenhangskomponente zu verstehen. Sie wird benötigt, um einen minimalen spannenden Komponentenbaum aufzubauen.
Die zweite permanente Marke ist global. Sie markiert alle schon in den minimalen spannenden Wald eingefügten Knoten. Alle anderen noch nicht durch *marked* gekennzeichneten Knoten werden als *frei* betrachtet.

Nachfolgend werden die Neuerungen erläutert:

Nachdem die *near*- und *dist*-Felder initialisiert worden sind, wird in einer äußeren Schleife ein erster minimaler spannender Komponentenbaum bestimmt. Die Vorgehensweise ist die gleiche, die in Abschnitt 5.1 beschrieben wurde. Kann kein weiterer einzufügender Knoten ermittelt werden, ist eine Zusammenhangskomponente gefunden.
In einem nächsten Schritt wird nun ein neuer Anfangsknoten für die neue zu bildende Zusammenhangskomponente gesucht. Dies wird in einer Schleife realisiert, die den freien Knoten sucht, der den kleinsten Index besitzt. Er wird mit *marked* permanent markiert.
In einer folgenden Schleife werden für alle nun noch freien Knoten die Werte in den *dist*- und *near*-Feldern, also die Entfernung zu dem neuen Komponentenbaum (zu diesem Zeitpunkt besteht er nur aus dem Anfangsknoten) aktualisiert. Dann wird wieder, wie im Algorithmus von Prim-Dijkstra, fortgefahren und ein neuer Komponentenbaum erstellt. Dies wird solange wiederholt, bis alle Knoten des Graphen durch *marked* markiert sind und so ein minimaler spannender Wald aufgebaut wurde.

Als Eingabe für den Algorithmus dient die obere Dreiecksmatrix der symmetrischen Gewichtsmatrix, die wie beim Algorithmus von Prim-Dijkstra spaltenweise in einem eindimensionalen Feld abgespeichert wird.
Als Ausgabe liefert der Algorithmus die Anzahl der Zusammenhangskomponenten, das Gewicht des minimalen spannenden Waldes und die Liste der Kanten, die diesen

minimalen spannenden Wald bilden. Gleichzeitig läßt sich daraus ablesen, welche Knoten eine Zusammenhangskomponente bilden und wie groß das Gewicht des dazugehörenden minimalen spannenden Komponentenbaumes ist.

Die Zeitkomplexit des sequentiellen Algorithmus zur Berechnung von Zusammenhangskomponenten beträgt bei einer Problemgröße von n Knoten, wie die vom Algorithmus von Prim-Dijkstra, $O(n^2)$.

6.2 Parallelisierung

Die Parallelisierung dieses Algorithmus erfolgt in gleicher Weise wie die des Algorithmus von Prim-Dijkstra. Die zu untersuchenden Spalten der Gewichtsmatrix werden zyklisch auf die Prozessoren verteilt.
Von den beiden neu hinzugekommenen Schleifen wird nur diejenige parallelisiert, die die Aktualisierung der *near*- und *dist*-Felder realisiert. Als neuer Anfangsknoten wird immer der mit dem kleinsten Index gewählt. So ist eine Parallelisierung dieser Schleife nicht erforderlich. Würde sie ebenfalls parallelisiert werden, so müßte der globale kleinste Anfangsknoten wieder berechnet und allen Prozessoren bekannt gemacht werden (siehe Berechnung des globalen Minimums in Abschnitt 5.2). Dies würde zu einer Verlängerung der Ausführungszeiten und damit einer Verschlechterung der Performance führen.

Aufgrund der Erfahrungen mit den drei Varianten, die beim Algorithmus von Prim-Dijkstra implementiert wurden, wurden bei der Parallelisierung hier nur die Kommunikationsstrukturen des Baumes und des Ringes verwendet. Grundsätzlich ließe sich auch die Variante mit globalen Operationen verwenden, doch da sie die für die Performance-Bestimmung ungünstigste ist, wurde darauf verzichtet. Die Kommunikation verlief auch hier im synchronen Modus.

Die Zeitkomplexitäten der beiden parallelen implementierten Varianten müssen wegen der nicht parallelisierten Schleife, die den neuen Anfangsknoten sucht, für den durchschnittlichen und den schlechtesten Fall unterschieden werden.
Bei der Variante mit Baumstruktur beträgt die Zeitkomplexität bei einer Problemgröße von n Knoten und bei p Prozessoren im Mittel $O(n^2/2 + n\log(p))$. Im schlechtesten Fall beträgt sie $O(n^2)$. Die Zeitkomplexität der Variante mit Ringstruktur beträgt bei eine Problemgröße von n Knoten und bei p Prozessoren im Mittel $O(n^2/2 + n \cdot n/2) = O(n^2)$, sowie im schlechtesten Fall $O(n^2)$.
Als Beispiel wird nachfolgend die parallele Implementation mit Baumstruktur aufgelistet.

Parallele Implementation mit Baumstruktur:

```
      PROGRAM CONN_COMP
      . . .
      me  = MYNODE()
      p   = NUMNODES()

      IF (me .EQ. 0) begintime = DCLOCK()

      near(1) = 0                !Permanente Marke des Anfangsknotens

      DO 1000 j = me + 2, n, p
         near(j) = 1                              !Temporaere Marken
         dist(j) = feldw(bases(j))                !der anderen Knoten
1000  CONTINUE
      zaehl = 1

      DO 2000 k = 2, n, 1        !Aufbau der Zusammenhangskomponenten
                                 !und des minimalen spannenden Waldes
         IF (.NOT. connect) THEN           !Beginn der Erweiterungen
            connect = .TRUE.                 !des neuen Algorithmus

            DO 10 i = 2, n, 1                 !Suche nach dem neuen
               IF (marked(i) .EQ. 0) THEN    !Anfangsknoten der neuen
                  naechster         = i      !Zusammenhangskomponente
                  marked(naechster) = 1
                  GOTO 10
               ENDIF
10          CONTINUE

            DO 20 i = me + 2, n, p          !Aktualisieren der Werte
               IF (marked(i) .EQ. 0) THEN               !aller noch
                  near(i) = naechster                  !freien Knoten
                  IF (naechster .LT. i) THEN
                     dist(i) = feldw(bases(i) + naechster - 1)
                  ELSEIF (naechster .GT. i) THEN
                     dist(i) = feldw(bases(naechster) + i - 1)
                  ENDIF
               ENDIF
20          CONTINUE

         ENDIF          !Ende der Erweiterungen des neuen Algorithmus
```

```
      DO 3000 i = me + 2, n, p            !Suche nach dem minimal
         IF (marked(i) .EQ. 0) THEN            !entfernten Knoten
            IF (near(i) .NE. 0) THEN      !und damit der naechsten
               IF (dist(i) .LT. help(1, 2)) THEN   !einzufuegenden
                  help(1, 1) = i                          !Kante
                  locali     = i
                  help(1, 2) = dist(i)
               ENDIF
            ENDIF
         ENDIF
3000     CONTINUE
      DO 4000 j = log(p), 1, -1
         IF ((me .GT. ((2 ** (j - 1)) - 1)) .AND.
     >        (me .LE. ((2 ** j) - 1)))          THEN
            address = me - (2 ** (j - 1))
            CALL CSEND(11 * j * k, help, 8, address, 0)    !Sender
         ELSEIF (me .LE. ((2 ** (j - 1)) - 1)) THEN
            CALL CRECV(11 * j * k, hilf, 8)           !Empfaenger
            IF (help(1, 2) .GT. hilf(1, 2)) THEN
               help(1, 2) = hilf(1, 2)
               help(1, 1) = hilf(1, 1)
            ELSEIF ((help(1, 2) .EQ. hilf(1, 2)) .AND.
     >              (help(1, 1) .GT. hilf(1, 1)))     THEN
               help(1, 1) = hilf(1, 1)
            ENDIF
         ENDIF
4000     CONTINUE
      IF (me .EQ. 0) THEN                          !Broadcast
         CALL CSEND(12 * k, help, 8, -1, 0)           !Sender
      ELSE
         CALL CRECV(12 * k, help, 8)                 !Empfaenger
      ENDIF
      next = help(1, 1)
      IF (next .EQ. 0) THEN
         connect = .FALSE.
         cc      = cc + 1
      ELSE
         marked(next) = 1
         IF (next .EQ. locali) THEN
            tree(zaehl, 1) = near(next)           !Einfuegen des
            tree(zaehl, 2) = next                 !neuen Knotens
            zaehl          = zaehl + 1
         ENDIF
         weight     = weight + help(1, 2)
         near(next) = 0        !Permanente Marke des neuen Knotens
```

```
            DO 5000 j = me + 2, n, p              !Aktualisieren der
               IF (marked(j) .EQ. 0) THEN         !Angaben der Knoten
                  IF (near(j) .NE. 0) THEN     !ausserhalb des Waldes
                     IF (j .LT. next) THEN
                        IF (j .GT. near(j)) THEN
                           IF (feldw(bases(next) + j - 1) .LT.
     >                         feldw(bases(j) + near(j) - 1)) THEN
                              near(j) = next
                              dist(j) = feldw(bases(next) + j - 1)
                           ENDIF
                        ELSEIF (j .LT. near(j)) THEN
                           IF (feldw(bases(next) + j - 1) .LT.
     >                         feldw(bases(near(j)) + j - 1)) THEN
                              near(j) = next
                              dist(j) = feldw(bases(next) + j - 1)
                           ENDIF
                        ENDIF
                     ELSEIF (j .GT. next) THEN
                        IF (j .GT. near(j)) THEN
                           IF (feldw(bases(j) + next - 1) .LT.
     >                         feldw(bases(j) + near(j) - 1)) THEN
                              near(j) = next
                              dist(j) = feldw(bases(j) + next - 1)
                           ENDIF
                        ELSEIF (j .LT. near(j)) THEN
                           IF (feldw(bases(j) + next - 1) .LT.
     >                         feldw(bases(near(j)) + j - 1)) THEN
                              near(j) = next
                              dist(j) = feldw(bases(j) + next - 1)
                           ENDIF
                        ENDIF
                     ENDIF
                  ENDIF
               ENDIF
5000        CONTINUE

         ENDIF

2000  CONTINUE

      IF (me .EQ. 0) endtime = DCLOCK()

      END
```

6.3 Ergebnisse

Die Ergebnisse, die mit der sequentiellen und den beiden parallelen Implementationen auf dem iPSC/860 und dem Paragon XP/S 5 erzielt wurden, ähneln stark denen des Algorithmus von Prim-Dijkstra.

Die verwendete Compiler-Optimierungsstufe war Stufe *2*. Mit dieser Optimierungsstufe wurden bei den parallelen Implementationen die schnellsten Ausführungszeiten erzielt. Stufe *2* war um 30% schneller als Stufe *0*, um 15% schneller als Stufe *1* und um 2% schneller als die Stufen *3* und *4*.
Die Messungen mit 2, 4 und 8 Prozessoren auf dem iPSC/860 wurden interaktiv vorgenommen, die mit 16 und 32 Prozessoren im Batch-Betrieb. Auf dem Paragon wurde nur die Messung mit 72 Prozessoren im Batch-Betrieb durchgeführt, die anderen erfolgten im interaktiven Betrieb.

Bei der Untersuchung der parallelen Implementationen wurde auf dem iPSC/860 aufgrund der Anzahl der allokierbaren Prozessoren (Zweierpotenzen) ausschließlich die Variante mit Baumstruktur verwendet. Auf dem Paragon wurde bei Prozessoranzahlen, die Potenzen von 2 sind, ebenfalls nur mit der Variante mit Baumstruktur gemessen. Bei den Messungen mit anderen Prozessoranzahlen wurde, wie beim Algorithmus von Prim-Dijkstra, auf die Variante mit Ringstruktur zurückgegriffen.
Die Gewichte der Kanten eines Graphen sind INTEGER-Werte und wurden mittels eines Zufallszahlengenerators erzeugt. Die Anzahl der Zusammenhangskomponenten des Graphen konnte frei gewählt werden. Aufgrund der hier benutzten automatischen Generierung der Eingabe besaßen alle Zusammenhangskomponenten eines Graphen gleich viele Knoten.

# Komp.	2 Proz.	4 Proz.	8 Proz.	16 Proz.	32 Proz.
2	1.53	2.29	2.85	3.44	3.56
5	1.52	2.28	2.81	3.40	3.50
10	1.52	2.27	2.80	3.36	3.46
50	1.53	2.30	2.80	3.35	3.43
100	1.54	2.30	2.80	3.35	3.43

Tabelle 6.1: Vergleich der Speedup-Werte für Prozessoranzahlen von 2 bis 32 in Abhängigkeit von der Anzahl der Zusammenhangskomponenten bei einer Problemgröße von 2700 Knoten auf dem iPSC/860 (ermittelt mit der Variante mit Baumstruktur)

Bei einer Messung, in der verschiedene Anzahlen von Zusammenhangskomponenten bei einer festen Problemgröße von 2700 und Prozessoranzahlen von 2 bis 32 untersucht wurden, zeigte es sich, daß die Performance-Werte bei dem hier untersuchten Problem abhängig waren von der Anzahl der Zusammenhangskomponenten. Bei kleinen Prozessoranzahlen stiegen die Werte mit der Anzahl der Komponenten leicht an, bei größeren Prozessoranzahlen wurden die Werte mit zunehmender Komponentenanzahl etwas schlechter.
Die verwendete parallele Implementationsvariante war die mit Baumstruktur. In Tabelle 6.1 sind die bei diesem Problem erzielten Speedup-Werte dargestellt.

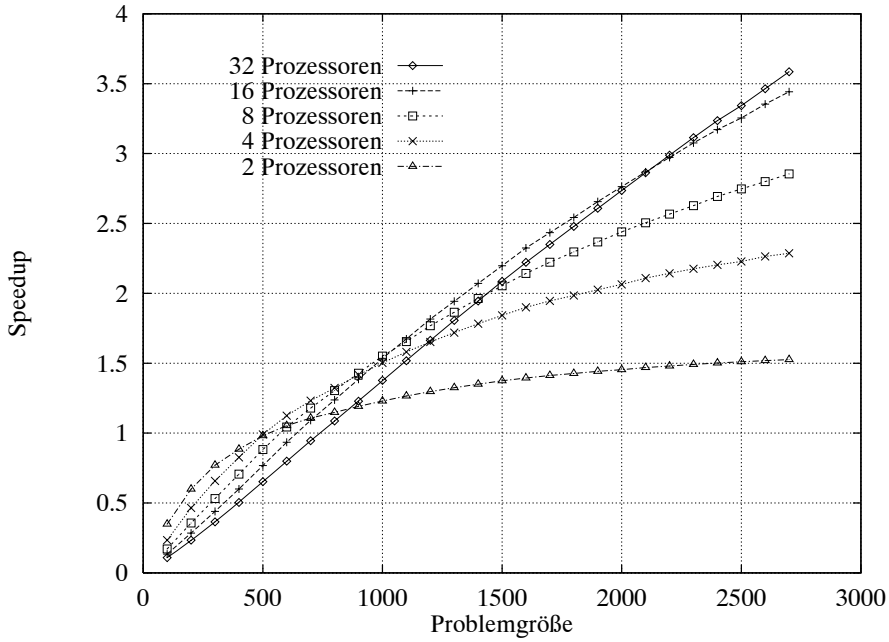


Abbildung 6.3: Speedup in Abhängigkeit von der Problemgröße bei einem Graphen mit 2 Zusammenhangskomponenten bei der Implementation mit Baumstruktur auf dem iPSC/860

Im weiteren werden nur Graphen mit zwei Zusammenhangskomponenten betrachtet. Die Abbildungen 6.3 und 6.4 zeigen den in Abhängigkeit von der Problemgröße auf dem iPSC/860 und dem Paragon XP/S 5 erzielten Speedup.

Wie beim Algorithmus von Prim-Dijkstra werden nur bei kleinen Prozessoranzahlen akzeptable Werte erreicht. Der Grund dafür liegt wieder in der zu feinen Granularität des Algorithmus.
Diese Tatsache unterstreichen auch die Abbildungen 6.5 und 6.6, in denen die aus dem Speedup ermittelten Effizienzen in Abhängigkeit von der Problemgröße dargestellt sind.

Wie auch bei den in den Kapiteln 4 und 5 untersuchten Algorithmen liegen die auf dem Paragon ermittelten Performance-Werte deutlich unter denen, die auf dem iPSC/860 erzielt wurden. Auch hier wirkt sich, trotz des schnelleren Verbindungsnetzwerkes, der Kommunikations-Overhead aufgrund des schnelleren Paragon-Prozessors im Vergleich zum iPSC/860 nachteilig aus.

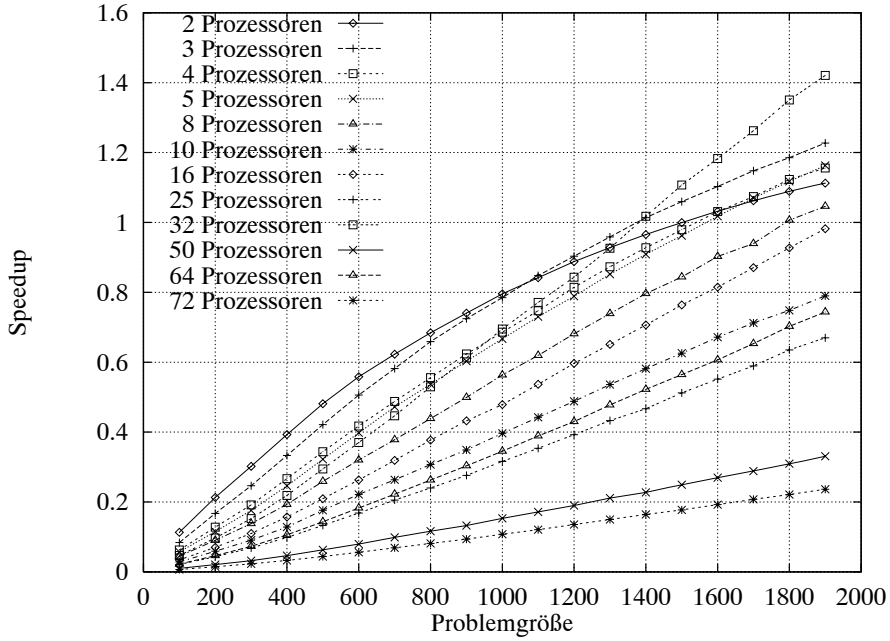


Abbildung 6.4: Speedup in Abhängigkeit von der Problemgröße bei einem Graphen mit 2 Zusammenhangskomponeten bei der Implementation mit Baumstruktur für Potenzen von 2 und der Implementation mit Ringstruktur für andere Prozessoranzahlen auf dem Paragon

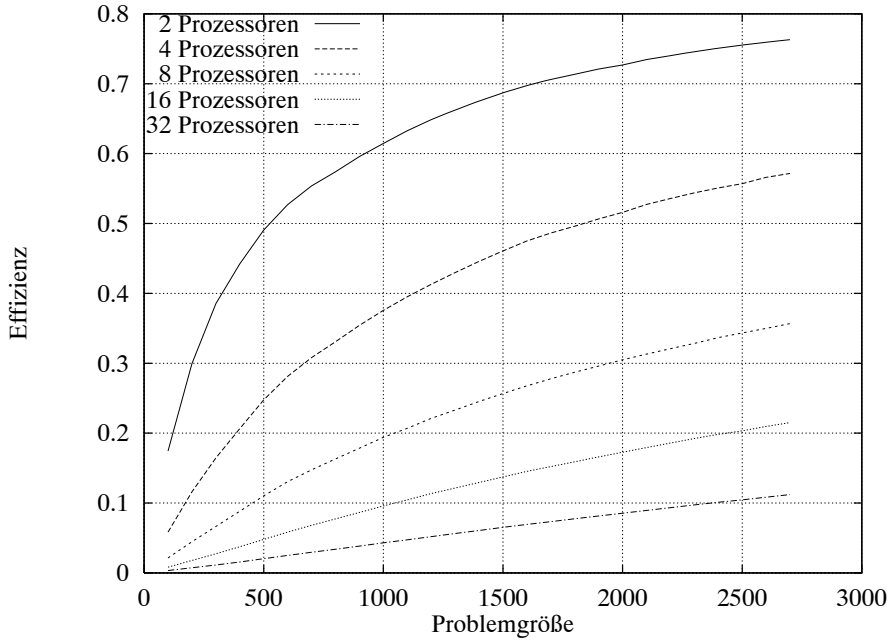


Abbildung 6.5: Effizienz in Abhängigkeit von der Problemgröße bei einem Graphen mit 2 Zusammenhangskomponenten bei der Implementation mit Baumstruktur auf dem iPSC/860

Abbildung 6.7 zeigt eine mit dem **xtool** erstellte Performance-Analyse. Dargestellt ist der prozentuale Anteil der aufgeführten Unterprogramme an der Ausführungszeit der Berechnung von 50 Zusammenhangskomponenten eines Graphen mit 2700 Knoten auf 4 Prozessoren des iPSC/860. Die verwendete Variante war die Baumstruktur, und es wurde die Optimierungsstufe *2* benutzt.
In CONN_COMP wird der minimale spannende Wald berechnet. Deswegen besitzt CONN_COMP bei allen Prozessoren, bis auf Prozessor 0, den größten Anteil an der Dauer der Applikation. Der zweitgrößte Anteil fällt an FLICK. FLICK wird im blockierenden Kommunikationsmodus aufgerufen, wenn die Prozessoren auf zu empfangende Nachrichten warten. RANGEN bezeichnet den Zufallszahlengenerator, mit dem die Kantengewichte erzeugt werden. Bei der Ermittlung der Ausführungszeit wird RANGEN allerdings nicht mit gemessen. Die für die Nachrichtensendung, das heißt die Übergabe der Nachricht an das Verbindungsnetzwerk, benötigte Zeit (CSEND) ist im Vergleich dazu gering. Das gleiche gilt auch für die Zeit, die für den Nachrichtenempfang benötigt wird (CRECV).

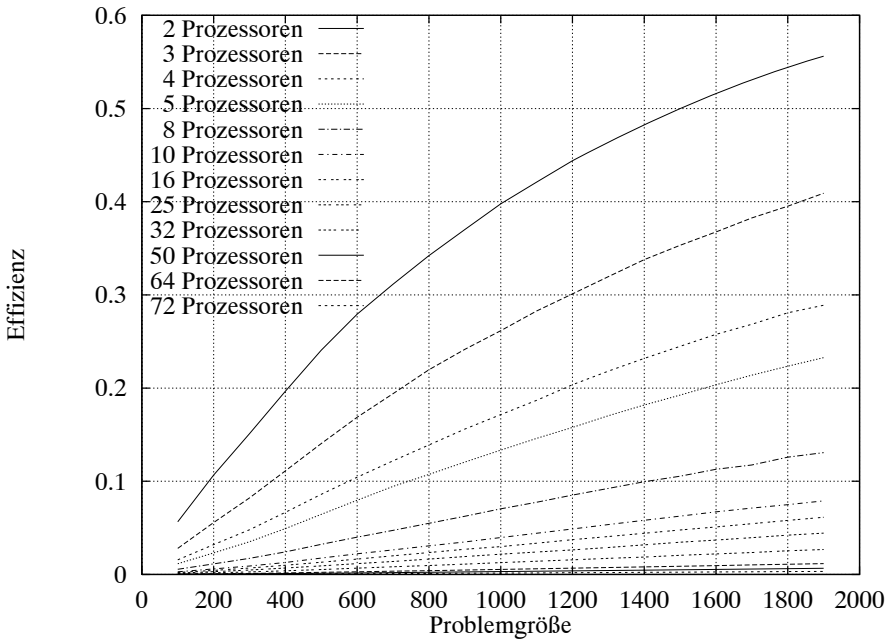


Abbildung 6.6: Effizienz in Abhängigkeit von der Problemgröße bei einem Graphen mit 2 Zusammenhangskomponenten bei der Implementation mit Baumstruktur für Potenzen von 2 und der Implementation mit Ringstruktur für andere Prozessoranzahlen auf dem Paragon

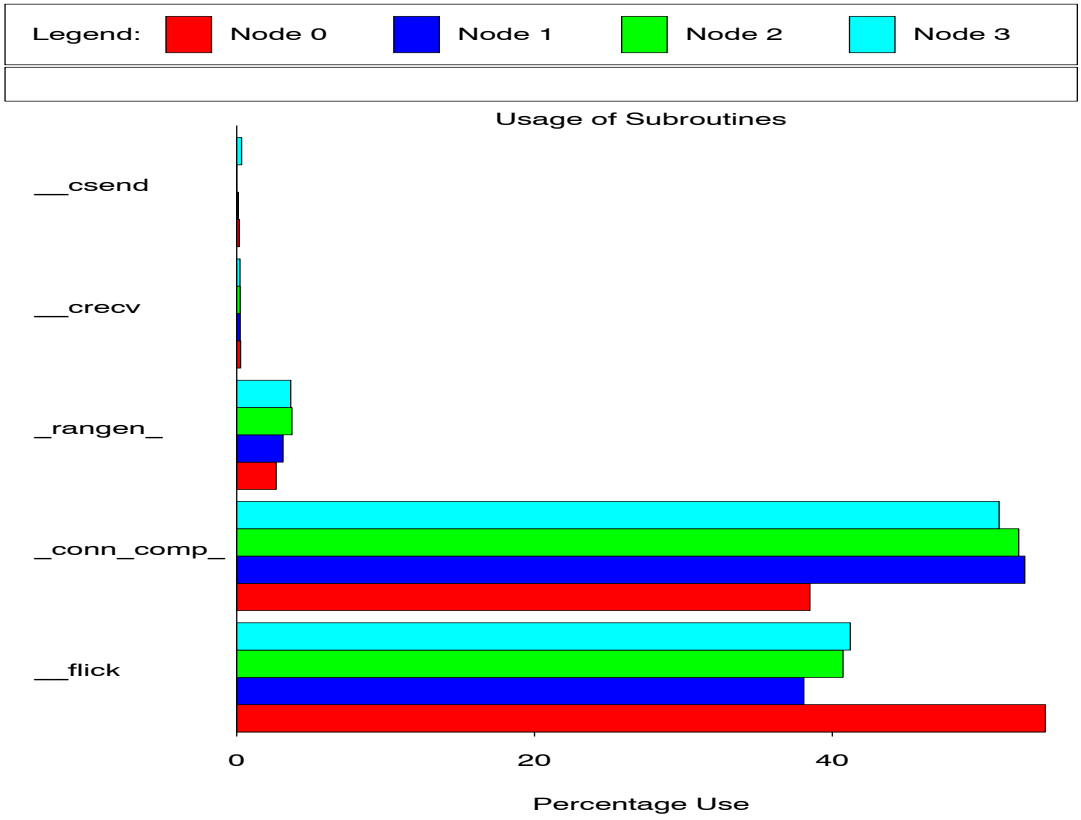


Abbildung 6.7: **xtool**: Prozentualer Anteil der Unterprogramme bei der Berechnung von 50 Zusammenhangskomponenten eines Graphen mit 2700 Knoten auf 4 Prozessoren des iPSC/860 mittels der Variante mit Baumstruktur und Optimierungsstufe *2*

Kapitel 7

Zusammenfassende Bewertung und Ausblick

Ziel dieser Arbeit war die Parallelisierung ausgewählter Graphenalgorithmen auf massiv-parallelen Rechnern. Zur Verfügung standen der Intel iPSC/860 und der Intel Paragon XP/S 5.
Beide Rechner besitzen einen auf die Rechenknoten verteilten Hauptspeicher und realisieren die Interprozessorkommunikation mittels Message Passing.
Das Nachfolgermodell Paragon XP/S 5 wurde gegenüber dem iPSC/860 verbessert. Neben einem schnelleren i860-RISC-Prozessor besitzt er ein Verbindungsnetzwerk mit Gitter-Struktur, das eine sehr viel höhere Übertragungsleistung aufweist als das Hypercube-Verbindungsnetzwerk des iPSC/860.

Der Abstand der Prozessoren hat beim Paragon XP/S 5 im Gegensatz zum iPSC/860 faktisch keinen Einfluß auf die Übertragungsdauer einer Nachricht. Der Programmierer braucht so die Verbindungstopologie und damit die möglichen Abbildungsmuster der Prozesse auf die einzelnen Prozessoren nicht zu berücksichtigen.

Der erste untersuchte Graphenalgorithmus war der Unterteilungsalgorithmus von Karp, der mit Hilfe der Heuristik des weitesten Einfügens eine Näherungslöung für das Traveling-Salesman-Problem bestimmt. Dieser Algorithmus ist von grobgranularer Struktur, die Parallelisierung findet auf Unterprogrammebene statt. Für die Kommunikation wurden zwei Varianten mit unterschiedlicher Nachrichtenzahl und Nachrichtenlänge implementiert, die sich in der Performance als annähernd gleich erwiesen. Mit diesen Implementationen konnten große Probleme untersucht werden. Weiterhin wurde zur Bestimmung eines minimalen spannenden Baumes der Algorithmus von Prim-Dijkstra implementiert.
Aufbauend auf dem Algorithmus von Prim-Dijkstra ist ein dritter Algorithmus implementiert worden, der die Zusammenhangskomponenten von Graphen berechnet. Die beiden zuletzt genannten Algorithmen haben im Gegensatz zum Unterteilungsalgorithmus von Karp eine feingranulare Struktur. Die Parallelisierung findet hier auf der Ebene von Schleifen statt.

Während der Untersuchung dieser beiden Algorithmen wurden drei Kommunikationsvarianten implementiert. Die erste bildet einen binären Baum nach, bei der zweiten wurden die Prozessoren in einem Ring angeordnet. Die dritte Variante benutzt globale Operationen, die auf dem iPSC/860 und dem Paragon zur Verfügung stehen. Die Variante mit Baumstruktur lieferte im Vergleich zu den anderen die schnellsten Ausführungszeiten, hat jedoch den Nachteil, daß sie nur bei Prozessoranzahlen anwendbar ist, die Potenzen von zwei bilden. Die beiden anderen Varianten können bei beliebiger Anzahl von Prozessoren eingesetzt werden. Die Variante mit Ringstruktur führte bei geringer Prozessoranzahl zu besseren Ergebnissen als die Variante mit globalen Operationen. Diese Aussage kehrt sich bei hoher Prozessoranzahl um. Diese Ergebnisse wurden für kleine Problemgrößen ermittelt. Die geringe Speicherkapazität der Rechenknoten ließ eine Untersuchung von großen Problemen nicht zu.

Die Ausführungszeiten aller drei Algorithmen waren auf dem Paragon schneller als auf dem iPSC/860. Dies war auch aufgrund des schnelleren i860 XP-Prozessors zu erwarten gewesen. Doch auch die wesentlich höhere Kommunikationsleistung des Paragon vermag den Kommunikations-Overhead zur Zeit noch nicht aufzuwiegen. Die Performance-Werte des Paragon sind so etwas schlechter als die des iPSC/860.

Deutliche Unterschiede waren bei den drei untersuchten Algorithmen festzustellen. Beim grobgranularen Unterteilungsalgorithmus von Karp wurde eine gute Performance erzielt. Durch den im Vergleich zum Rechenaufwand sehr geringen Kommunikationsaufwand werden durch die Parallelisierung nahezu optimale Beschleunigungen erreicht.
Im Gegensatz dazu ist bei den anderen beiden feingranularen Algorithmen der Kommunikationsaufwand im Vergleich zum Rechenaufwand sehr hoch. Daraus resultiert eine schlechtere Performance, die nur bei geringen Prozessoranzahlen akzeptable Werte liefert. Werden zur Berechnung von minimalen spannenden Bäumen und Wäldern viele Prozessoren eingesetzt, so tritt bei den hier untersuchten "kleinen" Problemgrößen auf dem Paragon XP/S 5 anstatt der erwarteten Beschleunigung sogar eine Verlangsamung ein.

Bei der Untersuchung des Unterteilungsalgorithmus von Karp läßt sich deutlich erkennen, daß sich insbesondere dieser grobgranulare, also auf der Ebene von Unterprogrammen parallelisierbare Algorithmus für die Parallelisierung auf MIMD-Rechnern eignet.
Die anderen beiden feingranularen Algorithmen, der Algorithmus von Prim-Dijkstra und dessen Erweiterung, lassen sich weniger gut auf MIMD-Rechnern implementieren. Nur der Einsatz von einigen wenigen Prozessoren verspricht in diesem Fall einen Gewinn an Effizienz.
Der Vergleich der Ergebnisse mit denen, die auf Rechnersystemen mit gemeinsamem Hauptspeicher erzielt wurden, zeigt, daß der hier untersuchte grobgranulare Algorithmus auf massiv-parallelen Rechnern mit verteiltem Speicher bessere Beschleunigungen erreicht. Für die untersuchten feingranularen Algorithmen muß jedoch festgestellt werden, daß sie auf Rechnern mit gemeinsamem Hauptspeicher größere Beschleunigungen erreichen als auf Rechnern mit verteiltem Hauptspeicher.

Die Zukunft läßt aufgrund der rasch fortschreitenden Hardware-Entwicklung hoffen, daß auch feingranulare Algorithmen auf MIMD-Rechnern mit verteiltem Speicher erfolgreich und effizient parallelisiert werden können. Neben Verbesserungen der Prozessoren liegt hier insbesondere die Aufgabe darin, die Kapazität des lokalen Speichers und die Übertragungsrate der Verbindungsnetzwerke drastisch zu erhöhen.

Literaturverzeichnis

[AlGo89] G. S. Almasi, A. Gottlieb: *Highly Parallel Computing.* Benjamin/ Cummings, Redwood City, California, 1989.

[AHU83] A. V. Aho, J. E. Hopcroft, J. D. Ullman: *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[BeHe90] R. Berrendorf, J. Helin: *Evaluating the Basic Performance of the Intel iPSC/860 Parallel Computer.* KFA Jülich, KFA-ZAM-IB-9102, Dezember 1990.

[Bod90] A. Bode (Hrsg.): *RISC-Architekturen.* BI-Wissenschaftsverlag, Mannheim, 1990.

[Bol79] B. Bollobás: *Graph Theory — An Introductory Course.* Springer, Heidelberg, Dezember 1979.

[BDG90] R. Berrendorf, J. Doctor, F. Gutbrod: *Benchmarks auf dem iPSC/860.* KFA Jülich, KFA-ZAM-IB-9030, Oktober 1990.

[BEK93] T. Bönniger, R. Esser, D. Krekel: *CM-5, KSR1, Paragon XP/S: a comparative description of massively parallel computers on the basis of a catalog of classifying characteristics.* KFA Jülich, KFA-ZAM-IB-9320, Oktober 1993.

[Car79] B. Carré: *Graphs and Networks.* Oxford University Press, Oxford, 1979.

[Chr75] N. Christofides: *Graph Theory — An Algorithmic Approach.* Academic Press, London, 1975.

[CMTS79] N. Christofides, A. Mingozzi, P. Toth, C. Sandi: *Combinatorial Optimization.* John Wiley & Sons, Chichester, 1979.

[DaSe86] W.J. Dally, C.L. Seitz: *The Torus Routing Chip.* J. Distributed Computing, Vol. 1, No. 3, pp. 187–196, 1986.

[Dij59] E. Dijkstra: *A note on two problems in connection with graphs.* Numerische Math. 1, pp. 269–271, 1959.

[DöMü73] W. Dörfler, J. Mühlbacher: *Graphentheorie für Informatiker.* Sammlung Göschen, Band 6016, de Gruyter, Berlin, 1973.

[DDP90] S. K. Das, N. Deo, S. Prasad: *Parallel graph algorithms for hypercube computers.* Parallel Computing 13, pp. 143–158, 1990.

[DSW93] G. Dueck, T. Scheuer, H.-M. Wallmeier: *Toleranzschwelle und Sintflut: neue Ideen zur Optimierung.* Spektrum der Wissenschaft, pp. 42–51, März 1993.

[EsKn93] R. Esser, R. Knecht: *Intel Paragon XP/S — Architecture and Software Environment.* In: H.-W. Meuer: Supercomputer '93, pp. 121–141, Springer, Heidelberg, Juni 1993.

[Eve79] S. Even: *Graph Algorithms.* Computer Science Press, Rockville, Maryland, 1979.

[Fly66] M. J. Flynn: *Very high-speed computing systems.* Proceedings of the IEEE 54, pp. 1901–1909, 1966.

[Gil93] W. K. Giloi: *Rechnerarchitektur.* Springer, Heidelberg, 1993.

[GoMr89] M. Gonauser, M. Mrva: *Multiprozessorsysteme — Architektur und Leistungsbewertung.* Springer, Heidelberg, 1989.

[GrTe93] Ch. Großmann, J. Terno: *Numerik der Optimierung.* Teubner, Stuttgart, 1993.

[Gur86] R. Gurke: *Graphenalgorithmen für MIMD-Rechner.* KFA Jülich, Jül-Spez-355, 1986.

[Har69] F. Harary: *Graph Theory.* Addison-Wesley, Reading, Massachusetts, 1969.

[HeBe90] J. Helin, R. Berrendorf: *Analyzing the Performance of Message Passing Hypercubes: a Study with the Intel iPSC/860.* KFA Jülich, KFA-ZAM-9101, November 1990.

[HoJe88] R. W. Hockney, C. R. Jesshope: *Parallel Computers 2.* Adam Hilger, Bristol, 1988.

[Hoß92] F. Hoßfeld: *Algorithmen für Parallelrechner.* Folienkopien zur Vorlesung an der RWTH Aachen, Aachen, 1992.

[HwBr84] K. Hwang, F. A. Briggs: *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[HPFF92] High Performance FORTRAN Forum: *High Performance FORTRAN Language Specification (DRAFT), Version 4.0.* Rice University, Houston Texas, 1992.

[Int90] *i860^{TM} 64-Bit Microprocessor — Hardware Reference Manual.* Intel Corporation, 1990.

[Int91] *Paragon XP/S — Product Overview.* Intel Supercomputer Systems Division, Beaverton, Oregon, 1991.

[Int92] *i860^{TM} Microprocessor Family — Programmer's Reference Manual.* Intel Corporation, 1992.

[Int92a] *iPSC/860 FORTRAN System Calls — Reference Manual.* Intel Supercomputer Systems Division, Beaverton, Oregon, März 1992.

[Int92b] *iPSC/860 System Users Guide.* Intel Supercomputer Systems Division, Beaverton, Oregon, März 1992.

[Int93a] *iPSC/860 FORTRAN Compiler Users Guide.* Intel Supercomputer Systems Division, Beaverton, Oregon, Januar 1993.

[Int93b] *Paragon OSF/1 FORTRAN System Calls — Reference Manual.* Intel Supercomputer Systems Division, Beaverton, Oregon, April 1993.

[Int93c] *Paragon OSF/1 Users Guide.* Intel Supercomputer Systems Division, Beaverton, Oregon, April 1993.

[Jun87] D. Jungnickel: *Graphen, Netzwerke und Algorithmen.* B.I.-Wissenschaftsverlag, Mannheim, 1987.

[Kar77] R. Karp: *Probabilistic Analysis of Partitioning Algorithms for the Traveling Salesman Problem in the Plane.* Mathematics of Operations Research 2, pp. 200–224, 1977.

[Kar90] St. Karsch: *Bestimmung maximaler Flüsse in gerichteten Graphen auf MIMD-Rechnern.* KFA Jülich, Jül-Spez-556, März 1990.

[Kne92] R. Knecht: *Implementation of Divide-and-Conquer Algorithms on Multiprocessors.* In: J. D. Becker, I. Eisele, F. W. Mundemann (eds.): Parallelism, Learning, Evolution. Lecture Notes in Artificial Intelligence 565, pp. 121–136, Springer, Heidelberg, 1992.

[Kru56] J. B. Kruskal: *On the subtree of a graph and the traveling salesman problem.* Proc. Amer. Math. Soc. 7 (Feb.), pp. 48–50, 1959.

[KrSm88] C. P. Kruskal, C. H. Smith: *On the Notion of Granularity.* The Journal of Supercomputing 1, pp. 395–408, 1988.

[Lei92] F. Th. Leighton: *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

[Li86] K. Li: *Shared Virtual Memory on Loosely-coupled Multiprocessors.* Ph.D. dissertation, Yale University, Technical Report YALEU-RR-492, Oktober 1986.

[LLRS86] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys: *The Traveling Salesman Problem.* John Wiley & Sons, Chichester, 1986.

[LMSK63] J. D. C. Little, K. G. Murty, D. W. Sweeney, C. Karel: *An Algorithm for the Traveling Salesman Problem.* Operations Research 11 (6), pp. 972–989, 1963.

[Mly93] A. Mlynski-Wiese: *Leistungsuntersuchung des iPSC/860 RISC-Knoten-Prozessors: Architekturanalyse und Programmoptimierung.* KFA Jülich, Jül-2766, Mai 1993.

[Moh82] J. Mohan: *A study in parallel computation — the traveling salesman problem.* Technical Report CMU-CS-82-136, Computer Science Department, Carnegie-Mellon University, 1982.

[Moh83] J. Mohan: *Experience with two parallel programs solving the traveling salesman problem.* Proceedings of the 1983 International Conference on Parallel Processing, IEEE, New York, pp. 191–193, 1983.

[Mül93] J. Müller: *Parallelverarbeitung auf Workstation-Clustern mittels Express und Network-Linda.* KFA Jülich, Jül-2819, September 1993.

[MGPO89] M. Malek, M. Guruswamy, M. Pandya, H. Owens: *Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem.* Ann. Oper. Res., vol. 21, (1–4), pp. 59–84, November 1989.

[NiMc93] L. M. Ni, P. K. McMinley: *A Survey of Wormhole Routing Techniques in Direct Networks.* IEEE Computer, pp. 62–76, February 1993.

[ObSc92] W. Oberschelp, R. Schneiders: *Algorithmen zur diskreten Optimierung.* RWTH Aachen, Lehrstuhl für Angewandte Mathematik, insbesondere Informatik; Schriften zur Informatik und Angewandten Mathematik, September 1992.

[OSF92] *OSF/1 Operating System, Users Guide.* Open Software Foundation, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[PaSt82] Ch. H. Papadimitriou, K. Steiglitz: *Combinatorial Optimization — Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[Pri57] R. C. Prim: *Shortest connection networks and some generalizations.* Bell Syst. Tech. J. 36, pp. 1389–1401, 1957.

[Qui83] M. J. Quinn: *The design and analysis of algorithms and data structures for the efficient solution of graph theoretic problems on MIMD computers.* Ph.D. dissertation, Computer Science Dept., Washington State Univ., Pullman, Wash., 1983.

[Qui87] M. J. Quinn: *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York, 1987.

[Röd91] St. Rödder: *Aufbau und Eigenschaften von Hypercube-Rechnern und Untersuchung von Datentransferalgorithmen.* KFA Jülich, Jül-2428, Januar 1991.

[RND77] E. Reingold, J. Nievergelt, N. Deo: *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[Sed89] R. Sedgewick: *Algorithms.* Addison-Wesley, Reading, Massachusetts, 1989.

[Sol77] M. Sollin: *An Algorithm Attributed to Sollin.* In: S. E. Goodman, S. T. Hedetniemi (Hrsg.): Introduction to the Design and Analysis of Algorithms, McGraw-Hill, New York, Kap. 5.5, 1977.

[Tho90] R. Thomas: *Parallelisierung des Max-Flow-Algorithmus von Shiloach/Vishkin auf CRAY-Rechnern.* KFA Jülich, Jül-Spez-557, März 1990.

[Zim87] H. J. Zimmermann: *Methoden und Modelle des Operations Research.* Rechnerorientierte Ingenieurmathematik, Vieweg & Sohn, Braunschweig, 1987.

Die vorliegende Diplomarbeit wurde am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen erstellt.

Dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich (KFA) anfertigen zu können.
Mein besonderer Dank gilt Frau R. Knecht für die Betreuung und für viele konstruktive Diskussionen. Weiter möchte ich Herrn Dr. P. Weidner und allen anderen Mitarbeitern des Zentralinstituts für Angewandte Mathematik sowie Herrn H. Bast von der Firma Intel für die wertvollen Hinweise zur Erstellung dieser Arbeit danken.