Digitale Bildverarbeitung mit kombinierter Pipeline- und Parallel-Architektur

Vom Promotionsausschuß der
Technischen Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Doktor-Ingenieur
genehmigte Dissertation

von
Bernhard Lang
aus
Groß-Rechtenbach

1991

1. Gutachter: Prof. Dr.-Ing. H. Burkhardt
2. Gutachter: Prof. Dr.-Ing. O. Lange
Tag der mündlichen Prüfung: 28. Oktober 1991

Erschienen im VDI-Verlag GmbH, Düsseldorf 1992, ISBN 3-18-140810-7

Danksagung

Die vorliegende Arbeit entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Arbeitsbereich Technische Informatik I der Technischen Universität Hamburg-Harburg.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. H. Burkhardt, der durch wertvolle Diskussionen und Gespräche die Arbeit angeregt und betreut hat. Seine Förderung durch die Mittel des Arbeitsbereichs ermöglichte weiterhin die Realisierung der praktischen Teile dieser Arbeit.

Herrn Prof. Dr.-Ing. O. Lange danke ich für die freundliche Übernahme des 2. Gutachtens und die hilfreichen Hinweise zur Arbeit.

Herzlich bedanken möchte ich mich bei den Mitarbeitern Dipl.-Ing. M. Perezowsky und Herrn O. Zeuge im Hardware-Labor des Arbeitsbereichs für die gute Zusammenarbeit und die große Hilfe beim Aufbau des Prototypen des Harburger Systems.

Mein Dank gilt schließlich den Mitarbeitern und Studenten des Arbeitsbereichs sowie Kollegen und Freunden, welche durch ihre Gespräche und Zusammenarbeit zum Gelingen dieser Arbeit beigetragen haben.

Hamburg, im Januar 1992 Bernhard Lang

Inhaltsverzeichnis

1 Einleitung **4**

2 Anforderungen an den Entwurf eines Rechnersystems **6**

2.1 Algorithmische Anforderungen 7

2.2 Physikalische Anforderungen 9

2.3 Globale Anforderungen 11

3 Parallelisierung von Algorithmen **12**

3.1 Zerlegung von Algorithmen 12

3.1.1 Parallele Beschreibung von Algorithmen durch Graphen 12

3.1.2 Definitionen und Kenngrößen 16

3.2 Zerlegungsprinzipien 19

3.2.1 Lokale Operationen 19

3.2.2 Globale Transformationen 21

4 Parallele Programmformulierung **31**

4.1 Ebenen der Parallelität 31

4.2 Parallele Programmierung 33

4.2.1 Datenflußsprachen 33

4.2.2 Parallelität in sequentiellen Sprachen 34

4.2.3 Parallele Sprachen 35

5 Parallele Programmausführung **38**

5.1 Operationsprinzipien von Einzelprozessoren 38

5.2 Organisation von Multiprozessorsystemen 47

5.2.1 Prozessorauswahl 47

5.2.2 Speicherorganisation 48

5.3 Multiprozessor Verbindungsnetzwerke 50

5.3.1 Bewertung von Netzwerken 51

5.3.2 Einstufige Netzwerke 55

5.3.3 Mehrstufige, schaltbare Netzwerke 63
5.3.4 Mehrstufige, selbstroutende Netzwerke 67

6 Einordnung und Bewertung paralleler Rechner 70
6.1 Klassifikation von Rechnern 70
6.2 Bewertung der Systemleistung 73
6.2.1 Leistungsmessung bei Einzelprozessoren 73
6.2.2 Leistung von Multiprozessorsystemen 74

7 Ein paralleler Bildverarbeitungsrechner 77
7.1 Systemübersicht 78
7.1.1 Funktionseinheiten des Systems 80
7.1.2 Rechenleistung 81
7.1.3 Kommunikationsleistung 82
7.2 Module des Systems 83
7.2.1 Transputer-Modul 84
7.2.2 Bildaufnahmemodul 85
7.2.3 Verbindungsnetzwerkmodul 87
7.2.4 Host-Interface-Modul 88
7.2.5 Prototypsystem 88

8 Ein schneller Pipelinebus 91
8.1 Token-unterstützte Adressierung 91
8.2 Betriebsarten des Pipelinebus-Interface 95
8.3 Realisierung von Multiprozessorlademodi 96
8.3.1 ‘Scattering’, ‘Broadcast’ und ‘Gathering’ 97
8.3.2 Modularität und Ladezeit 101
8.4 Hardware-Struktur und Steuerung des Pipelinebus-Interface 101
8.4.1 Steuerung der Eingangsschicht 103
8.4.2 Steuerung der Ausgangsschicht 105
8.5 Weitere Anmerkungen zum Pipelinebus-Interface 108

9 Eine parallele Betriebssoftware 110
9.1 Anforderungen 110
9.2 Globale Organisation der verteilten Betriebssoftware 111
9.3 Lokale Betriebssoftware eines Transputer-Moduls 112

10 Parallele Implementierung von Algorithmen 115
10.1 Übertragung von Daten auf dem schnellen Pipelinebus 116
10.2 Bildrotation 118
10.3 Schnelle Fouriertransformation 120
10.4 Schnelle Bewegungsschätzung 126

11 Zusammenfassung 130

Literaturverzeichnis 132

A Verwendete Symbole und Operatoren 139
A.1 Symbole 139
A.2 Operatoren 141

Kapitel 1

Einleitung

Die digitale Bildverarbeitung ist durch die großen Datenmengen, die bei der digitalen Auswertung von Bildern anfallen, ein wichtiges Anwendungsgebiet für Rechner mit hohen bis höchsten Rechenleistungen. Dabei besteht der Wunsch, solche Rechner effizient einzusetzen, so daß ein möglichst großer Anteil der vorhandenen Rechenleistung für Bildverarbeitungsanwendungen verfügbar wird.

Beim Einsatz vorhandener Rechner muß zu deren effizienter Ausnutzung bei der Implementierung von Algorithmen die Rechnerarchitektur und -konfiguration mit berücksichtigt werden. Programme werden dann effizient ausgeführt, wenn sie an die leistungsfähigen Verarbeitungseinheiten (z.B. Vektoreinheiten, Array-Prozessoren) des jeweiligen Rechnertyps angepaßt werden.

Vielfältige Arbeiten beschäftigen sich mit der Abbildung von Algorithmen auf vorhandene Superrechner. Durch die hohen Kosten, die durch die Verwendung von Superrechnern entstehen, beschränken sich solche Anwendungen jedoch meist auf den Forschungs- und Entwicklungsbereich. Weiterhin erfordert eine Programmübertragung auf einen anderen Rechnertyp, z.B. zur Erzielung einer höheren Rechenleistung, meist wieder viel Anpassungs- und Optimierungsarbeit.

Bildverarbeitung auf kostengünstigen Rechnern stößt bei komplexen Anwendungen schnell an die Leistungsgrenze von Einzelprozessoren. Der Ausweg, zur Erhöhung der Rechenleistung Spezialhardware einzusetzen, führt zwar zu leistungsfähigen Einzellösungen, das Problem der Übertragbarkeit stellt sich jedoch auch hier.

Der Aufbau leistungsfähiger und kostengünstiger Multirechnersysteme auf Mikroprozessorbasis ist durch die steigende Integrationsdichte digitaler Bauelemente und die damit erfolgte Verkleinerung der Rechnerhardware heute möglich. Insbesondere die Familie der Transputer-Prozessoren hat diese Entwicklung stark beeinflußt und ermöglicht einen weitgehend problemlosen Aufbau komplexer Parallelrechnersysteme mit in weiten Grenzen frei einstellbaren Kommunikationstopologien. Damit ist die Anpassung der Rechnertopologie an Algorithmen möglich, wodurch einerseits deren effiziente Ausführung und andererseits die Übertragung auf zukünftige, schnellere Systeme durch den Erhalt algorithmischer Topologie unterstützt wird. Der Einsatz solcher Parallelsysteme in der Bildverarbeitung führt zu neuen Möglichkeiten, insbesondere bei der Entwicklung modular skalierbarer Implementierungen komplexer Algorithmen höherer Verarbeitungsstufen.

Die vorliegende Arbeit behandelt den Entwurf und den Einsatz paralleler Multirechner-

systeme für die Bildverarbeitung. Dazu wird exemplarisch die Breite der Anforderungen betrachtet, welche durch Algorithmen, durch die Struktur der anfallenden Massendaten und auch durch die Notwendigkeit der Systemverwaltung bestimmt sind. Damit fließen sowohl Gesichtspunkte zum Entwurf allgemein verwendbarer Systeme als auch spezielle Überlegungen zur Verarbeitung von Bilddaten in den Entwurf ein.

Das Thema gliedert sich in vier Teileinheiten. Im ersten Teil (Kapitel 2) werden Anforderungen der Bildverarbeitung an den Systementwurf herausgestellt. Der zweite Teil (Kapitel 3 bis 6) geht auf die notwendigen Schritte zur Implementierung von Algorithmen, vom Problem bis zur parallelen Programmausführung auf Multiprozessorsystemen, ein. Der dritte Teil (Kapitel 7 bis 9) stellt das Architekturkonzept des im Rahmen dieser Arbeit an der TU Hamburg-Harburg entwickelten Parallelrechners für Bildverarbeitungsaufgaben vor. Der vierte Teil (Kapitel 10) behandelt schließlich die beispielhafte Implementierung von Bildverarbeitungsalgorithmen auf einem Prototypen dieses Rechners.

Kapitel 2

Anforderungen an den Entwurf eines Rechnersystems zur digitalen Bildverarbeitung

Bildverarbeitung beinhaltet den gesamten Prozeß vom Erfassen einer Szene über die oft komplexe, mehrschichtige Verarbeitung bis hin zu einer Aussage über das Bild, welche z.B. beim Einsatz in einer industriellen Umgebung übergeordnete Aktionen steuert. Der Entwurf eines Rechnersystems muß diesen gesamten Vorgang berücksichtigen, so daß in keiner Schicht Engpässe auftreten.

Die zu implementierenden Algorithmen nehmen Einfluß auf entstehende Anforderungen zum Entwurf eines System. Die Analyse der Algorithmen und deren Zerlegung in einzelne Verarbeitungsschichten führt zu Anforderungen an die Rechenleistung und Topologie eines Systems.

Weitere Gesichtspunkte betreffen die Kosten und die Flexibilität von Systemen. Dabei stehen die Kosten eines Systems in Konkurrenz zur Systemleistung und, im Vergleich mit isoliert betrachtet billigeren Einzellösungen, auch zur Flexibilität. Abhilfe kann ein modularer, an benötigte Leistungsanforderungen anpaßbarer Systementwurf schaffen. Er ist Voraussetzung für eine von der endgültigen Leistung unabhängige, flexible Problemlösung und ermöglicht eine spätere Abstimmung zwischen Kosten und benötigter Leistung. Die parallele Zerlegung von Algorithmen hat entscheidenden Einfluß auf den Erfolg dieses Vorgehens, denn sie ist Grundlage für die Modularisierung von Problemen. Ist eine Modularisierung möglich, erlaubt ein flexibler, modularer Systementwurf auch die Verwendung gleichartiger Systeme für unterschiedliche Problemklassen und damit letztendlich eine Kostenersparnis im Vergleich zu vielen Einzellösungen.

Weiterhin wird Bildverarbeitung beeinflußt durch physikalische Randbedingungen. Diese betreffen die eingesetzten Bildgeber und die daraus resultierenden Datenraten und -formate. Aufgrund dieser Randbedingungen für Daten entstehen Anforderungen an den Entwurf der Systemeinheiten. Der Entwurf wird weiterhin durch die Struktur und Komplexität verfügbarer Bauteile zum Aufbau dieser Einheiten beeinflußt. Hier können technologische Fortschritte Lösungen ermöglichen, welche in der Vergangenheit kaum realisierbar erschienen.

Ist die Bildverarbeitung Teil einer komplexen Anordnung, interessieren nicht ihre De-

Hohe Datenraten	Geringe Datenraten
Einfache Datenstrukturen	Komplexe Datenstrukturen
Hohe Rechenleistung	Mittlere Rechenleistung
Einfache Algorithmen	Komplexe Algorithmen

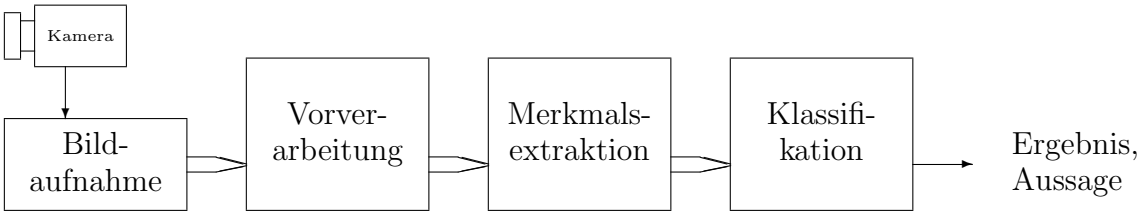

Abbildung 2.1: Schichten der Bildverarbeitung.

tails der Verarbeitung, sondern sie wird als Problemlösung in eine (z.B. industrielle) Umgebung eingebunden. Dann ergeben sich globale Randbedingungen, welche von dieser Problemlösung zu erfüllen sind. In den meisten Fällen ist die Verarbeitungszeit von der Erfassung eines Bildes bis zu einem darauf basierten Ergebnis die entscheidende globale, physikalische Randbedingung, welche es zu erfüllen gilt.

2.1 Algorithmische Anforderungen

Zur Betrachtung algorithmischer Anforderungen gilt es, die gesamte Verarbeitungskette der Bildverarbeitung von der Aufnahme eines Objekts bis hin zu einer Aussage bzw. einer Bedeutung zu berücksichtigen. Im allgemeinen ist wegen der Komplexität der Aufgabenstellung kein geschlossener Lösungsweg möglich. Man wendet daher ein strukturiertes Vorgehen an, welches zu einer mehrstufigen Verarbeitungskette führt. Zur Untergliederung dieser Kette gibt es verschiedene Ansätze.

Eine mögliche Untergliederung unterscheidet zwischen ikonischer und symbolischer Verarbeitung und orientiert sich an der Datenrepräsentation aufeinanderfolgender Verarbeitungsschichten [LE89]. Die ikonische Schicht umfaßt alle Vorverarbeitungsschritte bis hin zur Segmentierung eines Bildes. Beim Übergang in die symbolische Schicht werden die Bilddaten aufgrund der Segmentierung in eine symbolische Bildrepräsentation überführt. Die Verarbeitung und Interpretation dieser symbolischen Daten führt schließlich zu Aussagen über ein Bild.

Die in Abbildung 2.1 aufgezeigte Schichtenaufteilung orientiert sich am methodischen Vorgehen bei der Verarbeitung von Bildern. Sie wird insbesondere im Bereich der Mustererkennung verwendet [Nie83, Bur89, Fu82, HsF83]. Aus einer Einheit zur digitalen Bildgenerierung (Kamera, Bildspeicher) erhält man ein Bild als Funktion im digitalen Musterraum. Es dient als Eingangssignal der Vorverarbeitungsstufe, welche Algorithmen zur Bildverbesserung und Restauration umfaßt. Ergebnis ist wiederum ein durch Pixel repräsentiertes, visuell darstellbares Bild. Aus diesem Bild werden in der zweiten Stufe, der Merkmalsextraktion, interessierende Bildmerkmale gewon-

nen. Beispiele für die Gewinnung von Bildmerkmalen sind die Berechnung von Fourierdeskriptoren [Bur79, ASBH90], von Bildmomenten oder von schnellen lageinvarianten Transformationen [Bur79]. Der Merkmalsraum, in dem die aus der Merkmalsextraktion gewonnenen Merkmale dargestellt werden, hängt von den Algorithmen zur Merkmalsgewinnung ab. So besitzt er im Fall der schnellen Transformationen noch die gleiche Dimension wie der Musterraum, seine Dimension kann aber auch, wie bei den Fourierdeskriptoren, wesentlich kleiner als die des Musterraums sein. Die dritte Schicht, die Klassifikation, ordnet einen Merkmalsvektor einer bestimmten Bedeutungsklasse zu.

Aus den nach Abbildung 2.1 aufeinanderfolgenden Verarbeitungsschichten ergeben sich algorithmische Anforderungen an ein Rechnersystem zur Bildverarbeitung. In der Vorverarbeitungsschicht müssen sehr große Datenmengen verarbeitet werden. Die Algorithmen (z.B. lokale lineare Bildfilterung, nichtlineare lokale Filter) weisen üblicherweise eine einfache, reguläre Struktur auf, was sich in einfachen, vielfach wiederholten Grundoperationen widerspiegelt. Diese vielfache Wiederholung der einfachen Grundoperationen benötigt eine sehr hohe Rechenleistung mit einfacher Komplexität. Weiterhin ergeben sich durch die großen Datenmengen Anforderungen an die Kommunikationsleistung eines Systems. Es müssen große Datenmengen den Verarbeitungseinheiten zugeführt werden, das Format der Bilddaten ist jedoch einfach in seiner Struktur.

Beim Übergang in die zweite Schicht der Merkmalsextraktion ist der Bedarf an Rechenleistung ebenfalls hoch. Die Rechenleistung wird jedoch zur Realisierung komplexerer Operationen als im Vorverarbeitungsbereich benötigt und beschränkt sich nicht mehr auf eine lokale Bildumgebung. Der Bedarf an Kommunikationsleistung zur Datenzuführung hängt von den Eingangsdaten ab und entspricht damit den Anforderungen der Vorverarbeitung. Die Struktur der Ergebnisdaten im Merkmalsraum und die sich daraus ergebenden Kommunikationsanforderungen werden durch den jeweiligen Algorithmus bestimmt. Eine hohe Flexibilität im Zugriff auf die Ergebnisse ist hier wünschenswert. In vielen Fällen erhält man am Ausgang der Merkmalsschicht schon eine deutliche Datenreduktion bei gleichzeitigem Anstieg der Komplexität zugehöriger Datenstrukturen.

Im Bereich der Klassifikation ergibt sich sowohl bezüglich der Algorithmen als auch der Datenstrukturen eine weitere Erhöhung der Komplexität bei gleichzeitiger Abnahme der Rechenleistung und Datenraten. Im Datenbereich kommt man in dieser Schicht zu einer Bedeutung, die sich auf sehr wenige Datenworte abbilden läßt. Gegenüber dem Bild, aus dem diese Bedeutung gewonnen wurde, ergibt sich somit eine Datenreduktion um mehrere Größenordnungen.

In der Verarbeitungskette von der Bildaufnahme bis zur Extraktion einer Aussage im Bedeutungsraum nehmen somit die benötigten Rechenleistungen und Datenraten ab. Gleichzeitig steigen die Anforderungen an die Komplexität der Berechnungen und der Datenstrukturen.

Betrachtet man heutige Bildverarbeitungssysteme, so wird insbesondere der erste Teil der Verarbeitungskette nach Abbildung 2.1, der Vorverarbeitungsbereich, mit hoher Leistungsfähigkeit abgedeckt. Verarbeitungseinheiten basieren auf Spezialhardware oder auf Pipelineprozessoren, welche zur Ausführung lokaler Operatoren optimiert sind.

Für die Algorithmen der Merkmals- und Klassifikationsschichten wird eine höhere Flexibilität der Verarbeitungseinheiten benötigt. Hier bietet sich der Einsatz moderner, paral-

leler Systeme an, wobei durch geeignete Parallelisierung der Algorithmen dieser Schichten und durch ein geeignetes Verbindungsnetzwerk zwischen parallelen Prozessoren die Konfigurierung der benötigten Rechenleistung durch die Prozessoranzahl P möglich wird. Eine effiziente Implementierung basiert dann auf einer effizienten Zerlegung der Algorithmen und einer guten Anpassung an eine parallele Hardware.

Die Übertragung von Algorithmen auf eine parallele Hardware läßt sich mit einem dreistufigen Vorgehen beschreiben:

1. Parallelisierung von Algorithmen,
2. Übertragung parallelisierter Algorithmen in eine parallele Programmformulierung,
3. Parallele Ausführung der parallelen Programmformulierung.

Nur wenn jede dieser drei Stufen effizient durchgeführt wird, ist die effiziente Implementierung eines Algorithmus auf einer parallelen Hardware gewährleistet. In den Kapiteln 3 bis 5 werden diese drei Stufen eingehender betrachtet.

2.2 Physikalische Anforderungen

Physikalische Anforderungen zum Aufbau eines Bildverarbeitungssystems werden durch verwendete Bildgeber und durch technologische Randbedingungen beim Aufbau von Speicher-, Peripherie- und Verarbeitungseinheiten bestimmt. Auch kann ein Bildverarbeitungssystem nicht isoliert, sondern muß in Interaktion mit seiner Umgebung gesehen werden. Dadurch ergeben sich unterschiedliche Anforderungen z.B. bezüglich des Datenformats der zu verarbeitenden Datenraten, der Organisation der Hardware sowie zeitlicher Größen.

Bildgeber

Kameras als Bildgeber liefern üblicherweise einen sequentiellen, zeilenorientierten Datenstrom [Sch72]. Nur Spezialkameras erlauben einen andersartigen Zugriff auf den Aufnahmesensor (z.B. 'Image-Disector'-Kamera mit beliebigem örtlichen Zugriff). Die Datenrate einer standardmäßigen Videokamera bestimmt sich aus der Bildwechselfrequenz und der 2-dimensionalen Bildauflösung. Herkömmliche Kameras erreichen bei einer Bildwechselfrequenz W von 25 Hz eine Vertikalauflösung V von 625 Zeilen (CCIR-Norm). Nimmt man ein geometrisches Seitenverhältnis S des Kamerabilds von $4:3$ an, so ergibt sich:

$$\begin{aligned} H &= V \cdot S \\ &= 625 \cdot 4/3 \approx 833 \end{aligned} \tag{2.1}$$

mit H als Horizontalauflösung einer Zeile in Pixel bei quadratischer Abtastung[1]. Daraus resultiert eine maximale Frequenz f_{Max}, mit der Daten generiert werden:

$$f_{Max} = V \cdot H \cdot W \tag{2.2}$$

[1]Die Ermittlung der Horizontalauflösung und der daraus resultierenden maximalen Abtastfrequenz f_{Max} ist an dieser Stelle vereinfacht dargestellt. Bei genauer Betrachtung müssen eine horizontale und vertikale Austastlücke mit in die Überlegungen einbezogen werden. Die hier ermittelten Ergebnisse weichen jedoch weniger als 10% von den korrekten Werten ab.

$$= \quad 625^2 \cdot \frac{4}{3} \cdot 25\ Hz \approx 13\ MHz.$$

Mit dieser Frequenz f_{Max} muß ein Wandler getaktet werden. Da der interessierende Bildausschnitt jedoch kleiner ist als diese maximale Bildgröße, liegt die resultierende mittlere Frequenz $\bar{f}$ unter dem maximalen Wert f_{Max}. Will man bei obigem Bildformat einen Fensterausschnitt der Größe $V_F \times H_F$ bearbeiten, so erhält man:

$$\bar{f} = \frac{V_F \cdot H_F}{V \cdot H} f_{Max} = V_F \cdot H_F \cdot W. \tag{2.3}$$

Bei einem üblichen Bildausschnitt von 512^2 Pixeln ergibt sich ein Wert $\bar{f}_{512^2} \approx 6,55 MHz$. Bei der Aufnahme von Schwarz-Weiß Bildern mit 8 Bit Auflösung pro Pixel führt dies zur maximalen und mittleren Datenrate von $R_{Max} \approx 13\ MBytes/s$ und $\bar{R} \approx 6,55\ MBytes/s$. Bei der gleichzeitigen Aufnahme dreier Farbkanäle mit jeweils gleicher Auflösung müssen diese Werte mit dem Faktor 3 multipliziert werden.

Damit ergibt sich eine physikalische Forderung bei der Verwendung zeilenorientierter Kameras: Werden diese Kameras als Bildgeber in einem System eingesetzt, müssen an den Bildaufnehmer angeschlossene Einheiten während der Aufnahme Pixeldaten in dem physikalisch vorgegebenen, zeilensequentiellen Format kurzzeitig mit der Rate R_{Max} aufnehmen. Im Mittel über einen Bildzyklus braucht die Aufnahme jedoch nur mit der Rate $\bar{R}$ erfolgen.

Technologische Randbedingungen

Der Hardware-Entwurf digitaler Systeme orientiert sich in starkem Maß an den verfügbaren Bauteilen. Diese beeinflussen den Entwurf von Peripherie-, Speicher- und Verarbeitungseinheiten. Nur der konsequente Einsatz technologisch moderner Bauteile führt zu modernen und kompakten Systemen. Daher wird die Auswahl geeigneter Bauteile zur wichtigen Anforderung beim Hardware-Entwurf.

Bei den zum Aufbau von Peripherieeinheiten verfügbaren Umsetzerbauteilen ist in den letzten Jahren eine starke funktionale Integration zu beobachten, so daß die Anpassung standardisierter Kameras und Monitore nur noch wenige Bauteile benötigt und eine programmierbare Parameteranpassung (z.B. Auflösung V, H oder Bildwechselfrequenz W) möglich wird [Boo90].

Die Masse der Daten und deren physikalisch bedingte Struktur muß Einfluß auf den Entwurf von Speichereinheiten nehmen und in der Zugrifforganisation Beachtung finden. Bei der Verwendung dynamischer Speicher, welche heutzutage die größte Halbleiterspeicherdichte zur Verfügung stellen, erzielt man im wahlfreien Zugriff zwar schon kurze Zeiten (ca. 150ns/Speicherwort), ein sequentiell organisierter Zugriff (Nibble Mode, Static Column Mode, etc.) verkürzt diese Zeiten jedoch nochmals deutlich. Zum sehr schnellen Ein- und Auslesen zeilensequentieller Datenströme eignen sich besonders die für Grafikanwendungen entwickelten dynamischen Video-RAMs [Tos89]. Sie erlauben mit moderatem Hardware-Aufwand einen sehr schnellen sequentiellen Zugriff auf komplette Speicherzeilen (ca. 30ns/Speicherwort) und ermöglichen dazu weitgehend parallel den wahlfreien Zugriff eines übergeordneten Prozessors.

Zum Entwurf von Verarbeitungseinheiten gewinnen neben verfügbaren Standardprozessoren spezielle Bildverarbeitungsbausteine [LSI90, INM88a] und frei programmierbare

Logikzellenbausteine mit hoher Gatterdichte (EPLD, LCA) [Int87, Xil88] immer mehr Bedeutung. Die erste Klasse umfaßt Videoschieberegister und funktionsspezifische Signalverarbeitungsbausteine. Programmierbare Logikzellen erlauben die konfigurierbare Abbildung einer Verarbeitungsstruktur auf die Zellstruktur des Bausteins und ermöglichen damit dessen Einsatz als frei programmierbare Verarbeitungseinheit.

2.3 Globale Anforderungen

Die Umgebung, aus der heraus ein Bildverarbeitungssystem betrieben wird, bedingt globale Anforderungen. Neben der Schnittstellenfrage und mechanischen Anforderungen stellt sie hauptsächlich globale zeitliche Anforderungen an ein System.

Beispiel für eine Umgebung ist ein interaktiver Betrieb, bei dem ein Benutzer in einer akzeptablen Zeit ein Ergebnis der Verarbeitung sehen will. Ein weiteres Beispiel ist durch die Einbindung in ein übergeordnetes System gegeben, bei dem Anforderungen sowohl bezüglich der Verarbeitungszeit vom Bild bis zum Ergebnis als auch — bei Verarbeitung von Bildsequenzen — bezüglich der Zykluszeit konsekutiv aufgenommener Bilder bestehen. Insbesondere in einer industriellen Umgebung ist die Zykluszeit eine entscheidende Größe, denn sie begrenzt den möglichen Durchsatz des übergeordneten Systems.

Die durch die Verarbeitungszeit und Zykluszeit beschriebenen Anforderungen können als *Echtzeitbedingungen* interpretiert werden. Dabei bestimmt die Bildwechselfrequenz W des Aufnehmers die untere Schranke der Zykluszeit.

Kapitel 3

Parallelisierung von Algorithmen

Der erste Schritt zur parallelen Implementierung eines Algorithmus ist die mit *Parallelisierung* (Seite 9) bezeichnete Erfassung der inhärenten Parallelität und dessen Zerlegung in Teileinheiten. Zur Veranschaulichung dieses Vorgehens unter Erhalt möglicher inhärenter Parallelität dienen Darstellungen durch Graphen. Anhand einer Graphenbeschreibung, die im allgemeinen nicht eindeutig ist, können Zerlegungsprinzipien erläutert und zugehörige Kenngrößen definiert werden.

Die optimale Parallelisierung eines Algorithmus ist im allgemeinen kein trivialer Schritt. Daher können Zerlegungsprinzipien für Algorithmenklassen bei erkannter Zugehörigkeit zu einer Klasse diesen ersten Schritt deutlich vereinfachen und auch Hinweise zum Vorgehen bei den beiden nachfolgenden Schritten einer Implementierung geben.

Nach der Einführung einer Notation zur Algorithmenbeschreibung durch azyklische Graphen wird in diesem Kapitel die Parallelisierung zweier Algorithmenklassen betrachtet. Die erste Klasse umfaßt Algorithmen mit lokalen Operationen, welche in der Bildvorverarbeitung eingesetzt werden. Die zweite Klasse beschreibt globale Transformationen, welche primär in den hinteren Bildverarbeitungsschichten (siehe Abbildung 2.1) Anwendung finden.

3.1 Zerlegung von Algorithmen

3.1.1 Parallele Beschreibung von Algorithmen durch gerichtete azyklische Graphen

Mit Graphen lassen sich Objekte und deren gegenseitige Abhängigkeiten ausdrücken. Daher eignen sie sich zur Beschreibung von Algorithmen bei deren Auflösung in Grundoperationen. Der Vorteil gegenüber sequentiellen Beschreibungsformen (z.B. Flußdiagramm, Struktogramm) liegt im Erhalt einer bei der Auflösung sichtbar werdenden inhärenten Parallelität.

Zwischen den Grundoperationen eines Algorithmus bestehen Abhängigkeiten, z.B. dienen Ergebnisse einer Operation als Operanden einer Folgeoperation. Dabei sind Datenabhängigkeiten eindeutig und müssen, aufgrund der notwendigen Kausalität, frei von

Zyklen sein.[1].

Unter Beachtung der gerichteten Abhängigkeiten und der Zyklenfreiheit eignen sich besonders *gerichtete azyklische Graphen* (DAGs)[2] zur Algorithmenbeschreibung [BT89]. Unter einem DAG versteht man einen Graphen $\mathcal{G}$, bestehend aus *Ecken* e_i und gerichteten *Kanten* k_{ij}, wobei jede Kante jeweils durch ein geordnetes Eckenpaar (e_i, e_j) repräsentiert wird:

$$\begin{array}{rcll} \mathcal{G} & = & (\mathcal{E}, \mathcal{K}) & \\ \text{mit:} & & & \\ \mathcal{E} & = & \{e_i\} & \text{(Menge der Ecken)} \\ \mathcal{K} & = & \{k_{ij}\} = \{(e_i, e_j)\} & \text{(Menge der Kanten).} \end{array} \tag{3.1}$$

Es gilt: $(e_i, e_j) \neq (e_j, e_i)$. Weiterhin darf aufgrund der Zyklenfreiheit jede Aneinanderreihung gerichteter Kanten $(e_{i_0}, e_{i_1}), (e_{i_1}, e_{i_2}), \ldots, (e_{i_{k-1}}, e_{i_k})$ keinen geschlossenen Zyklus bilden, d.h. $e_{i_1} \neq e_{i_k}$.

Bei der Darstellung von Algorithmen durch DAGs ordnet man jeder Ecke jeweils eine Operation zu und repräsentiert die Daten durch die Kanten zwischen den Ecken. Zuführende Kanten einer Ecke beinhalten die Operanden, wegführende Kanten die Ergebnisse der Operation. Der algorithmisch interpretierte Graph wird oft als *Datenflußgraph* bezeichnet.

Die Abbildung eines Algorithmus A auf einen Graphen $\mathcal{G}$ ist nicht eindeutig. Wie auch bei der seriellen Beschreibung hängt ein zugehöriger Datenflußgraph von der Auflösung des Algorithmus ab.

Ausführungszeit $t(\mathcal{G})$:
Nimmt man für alle durch Ecken repräsentierte Operationen einschließlich der Datenzugriffe eine gleiche Ausführungszeit $t_\mathcal{E}$ an, dann hängt die Gesamtzeit $t(\mathcal{G})$ zur Ausführung des Algorithmus auf einem Einzelprozessor linear von der Mächtigkeit $|\mathcal{E}|$ der Eckenmenge ab:

$$t(\mathcal{G}) = t_\mathcal{E} \cdot |\mathcal{E}|. \tag{3.2}$$

Die Beschränkung auf konstante Zeiten für alle Ecken vereinfacht zugehörige Überlegungen. Ecken mit größerer Komplexität können dann z.B. durch eine Sequenz mehrerer gleichgewichtiger Ecken angenähert werden. Will man unterschiedliche Operationszeiten zulassen, ist dies durch Zuordnung eines Komplexitätsattributs an die Ecken möglich, wobei dann dieses Attribut in allen Kenngrößen zur Ermittlung von Ausführungszeiten berücksichtigt werden muß. Dies soll im weiteren jedoch nicht ausgeführt werden.

Einen Einfluß der Datenzugriffe auf die Ausführungszeit läßt sich durch attribuieren der Kanten berücksichtigen. Dann kann bei der im folgenden noch behandelten Zerlegung eines Graphen in Subgraphen unterschieden werden, ob eine Kante lokal zu einem Subgraphen gehört oder zwei Subgraphen miteinander verbindet. Erster Fall führt bei der Ausführung des zerlegten Algorithmus auf einem parallelen System zu einem lokalen Datenzugriff, zweiter Fall zu einer Datenkommunikation.

[1] Mit heutigen parallelen Programmiersprachen ist, um den Sprachen Flexibilität bei akzeptablem Übersetzungsaufwand zu geben, die Formulierung zyklischer Datenabhängigkeiten und somit das Auftreten von Blockierungen ('Dead Locks') nicht verhindert. Somit ist es noch Aufgabe des Programmierers, für einen kausalen, von Datenzyklen freien Programmablauf zur Laufzeit zu sorgen.

[2] 'DAG' ist die Abkürzung für die englische Bezeichnung 'Directed Acyclic Graph'

Basierend auf der Graphenbeschreibung in [BT89] werden weitere Kenngrößen und Definitionen eingeführt:

Pfad $\mathbf{P}(\mathcal{G})$, Positiver Pfad $\mathbf{P}^+(\mathcal{G})$, Länge $|\mathbf{P}(\mathcal{G})|$:
Eine Sequenz von Ecken $(e_{p_0}, \ldots, e_{p_{k-1}})$, welche durch Kanten $(e_{p_j}, e_{p_{j+1}})$ (oder auch $(e_{p_{j+1}}, e_{p_j})$ bei gerichteten Graphen) miteinander verbunden sind und eine Ecke nur einmal beinhalten, wird als *Pfad* $\mathbf{P}$ bezeichnet:

$$\begin{aligned} \mathbf{P}(\mathcal{G}) \;=\; (e_{p_0}, e_{p_1}, \ldots, e_{p_{k-1}}) \quad \Rightarrow \quad & e_{p_i} \in \mathcal{E}; \\ & e_{p_i} \neq e_{p_j} \quad \forall\, i \neq j; \\ & (e_{p_i}, e_{p_{i+1}}) \in \mathcal{K} \;\vee\; (e_{p_{i+1}}, e_{p_i}) \in \mathcal{K}. \end{aligned} \tag{3.3}$$

Gehören alle aufeinanderfolgende Eckenpaare $(e_{p_i}, e_{p_{i+1}})$ eines Pfads zur Kantenmenge $\mathcal{K}$ des gerichteten Graphen, bezeichent man den Pfad als *positiven Pfad* $\mathbf{P}^+$:

$$\begin{aligned} \mathbf{P}^+(\mathcal{G}) \;=\; (e_{p_0}, e_{p_1}, \ldots, e_{p_{k-1}}) \quad \Rightarrow \quad & e_{p_i} \in \mathcal{E}; \\ & e_{p_i} \neq e_{p_j} \quad \forall\, i \neq j; \\ & (e_{p_i}, e_{p_{i+1}}) \in \mathcal{K}. \end{aligned} \tag{3.4}$$

Eine Kenngröße des Pfads ist seine Länge $|\mathbf{P}|$. Sie ermittelt sich aus der Anzahl der zugehörigen Ecken:

$$|\mathbf{P}(\mathcal{G})| \;=\; |(e_{p_0}, e_{p_1}, \ldots, e_{p_{k-1}})| \;=\; k. \tag{3.5}$$

Tiefe $T(\mathcal{G})$:
Über die Längen aller möglichen positiven Pfade läßt sich die *Tiefe* $T(\mathcal{G})$ eines Graphen bestimmen:

$$T(\mathcal{G}) \;=\; \max_i(|\mathbf{P}_i^+(\mathcal{G})|). \tag{3.6}$$

Die Tiefe beschreibt die maximale Anzahl von Grundoperationen des zum Graphen gehörigen Algorithmus, welche aufgrund der Datenabhängigkeiten in Sequenz auszuführen sind (inhärente Serialität), und ermöglicht, bei angenommener konstanter Ausführungszeit für Grundoperationen, die Ermittlung einer unteren Schranke für die Ausführungszeit t.

Vorgänger V:
Mit der Definititon des positiven Pfads (Gleichung 3.4) läßt sich die Abhängigkeit einer Ecke von vorgeordneten Ecken ausdrücken. Eine Ecke e_j soll als *Vorgänger* V von e_i bezeichnet werden, wenn ein positiver Pfad $\mathbf{P}^+$ von e_j nach e_i existiert:

$$e_j \;=\; V(e_i) \quad \Rightarrow \quad \exists\, \mathbf{P}^+ = (e_j, \ldots, e_i). \tag{3.7}$$

Dann benötigt, bei algorithmischer Interpretation des Graphen, die Ausführung der Operation zu e_i direkt oder indirekt Ergebnisse der Operation zu e_j als Operanden.

Eingangs- und Ausgangsecken:
Besitzt eine Ecke e_i keine Vorgänger, soll sie als *Eingangsecke* bezeichnet werden. Mit den Eingangsecken wird in diesem Modell die Bereitstellung der Eingangsdaten berücksichtigt.

Weiter erhält eine Ecke, welche zu keiner anderen Ecke Vorgänger ist, die Bezeichnung *Ausgangsecke*. Diese Ecken enthalten nach der Ausführung eines Datenflußgraphen die Ergebnisse der Berechnung.

Zerlegung $\mathcal{Z}(\mathcal{G})$:
Zur parallelen Ausführung muß ein Algorithmus in Segmente aufgeteilt werden. Diese Aufteilung läßt sich durch Zerlegung des zugehörigen Graphen beschreiben. Eine *Zerlegung* $\mathcal{Z}(\mathcal{G})$ eines Graphen $\mathcal{G}$ wird charakterisiert durch eine Menge von *Subgraphen* $\{\mathcal{G}_0, \ldots, \mathcal{G}_{P-1}\}$ und eine Menge *globaler Kanten* $\mathcal{K}^G$, welche zu $\mathcal{G}$ aber zu keinem der Subgraphen $\mathcal{G}_k$ gehören:

$$\begin{aligned} \mathcal{Z}(\mathcal{G}) &= (\{\mathcal{G}_k,\ k \in I\!N_P\}, \mathcal{K}^G) && (3.8)\\ \mathcal{G}_k &= (\mathcal{E}_k, \mathcal{K}_k) \\ \mathcal{K}^G &= \mathcal{K} \setminus \bigcup_{k=0}^{P-1} \mathcal{K}_k. && (3.9) \end{aligned}$$

Die vollständige Definition der Zerlegung benötigt noch eine Beschreibung der Subgraphen: Die Eckenmenge $\mathcal{E}_k$ eines Subgraph $\mathcal{G}_k$ ist eine Teilmenge von $\mathcal{E}$. Für eine Zerlegung muß jede Ecke e_i zu mindestens einer Eckenmenge eines Subgraphen gehören. Die Kantenmenge $\mathcal{K}_k$ beinhaltet alle Kanten von $\mathcal{K}$ zwischen Ecken des Subgraphen:

$$\begin{aligned} \mathcal{E} &= \bigcup_{k=0}^{P-1} \mathcal{E}_k \\ \mathcal{E}_k &\subset \mathcal{E} \\ \mathcal{K}_k &= \{k_{ij} = (e_i, e_j) | e_i, e_j \in \mathcal{E}_k,\ k_{ij} \in \mathcal{K}\}. \end{aligned} \qquad (3.10)$$

Diese Beschreibung der Subgraphen bewirkt, daß die Menge der globalen Kanten $\mathcal{K}^G$ alle diejenigen Kanten umfaßt, welche von einem Subgraphen zu einem anderen weisen. Interpretiert man die Zerlegung des Graphen als die Aufteilung eines Algorithmus auf P Prozessoren, so beschreiben die globalen Kanten die notwendige Kommunikation zwischen Prozessoren während der Ausführung des Algorithmus. Ein Graph $\mathcal{G}_k$ beinhaltet dann alle Operationen und alle lokalen Datenzugriffe für Prozessor k.

Eine disjunkte Aufteilung der Eckenmenge $\mathcal{E}$ des Gesamtgraphen führt zu einer disjunkten Zerlegung $\mathcal{Z}^D(\mathcal{G})$. Dann besitzten sowohl die Ecken- als auch die Kantenmengen der Subgraphen keine gleichen Elemente:

$$\mathcal{Z}(\mathcal{G}) = \mathcal{Z}^D(\mathcal{G}) \quad \Rightarrow \quad \begin{array}{ll} \mathcal{E}_k \bigcap \mathcal{E}_l = \emptyset & \text{und} \\ \mathcal{K}_k \bigcap \mathcal{K}_l = \emptyset, & \text{falls } k \neq l. \end{array} \qquad (3.11)$$

Schicht $s; s \in S \subset I\!N$, Verarbeitungsschicht $\mathcal{E}^{(s)}$, Kommunikationsschicht $\mathcal{K}^{G,(s)}$:
Zur parallelen Ausführung eines Algorithmus gemäß einer getroffenen Zerlegung $\mathcal{Z}$ erfolgt die Zuordnung der P Subgraphen $\mathcal{G}_k$ zu P Prozessoren. Aufgrund der globalen, durch $\mathcal{K}^G$ beschriebenen Datenabhängigkeiten, kann jedoch die Berechnung der Subgraphen nicht unabhängig erfolgen, sondern wird durch notwendige globale Kommunikationen unterbrochen. Dieser Vorgang läßt sich durch Einführung von Verarbeitungs- und und Kommunikationsschichten modellieren.

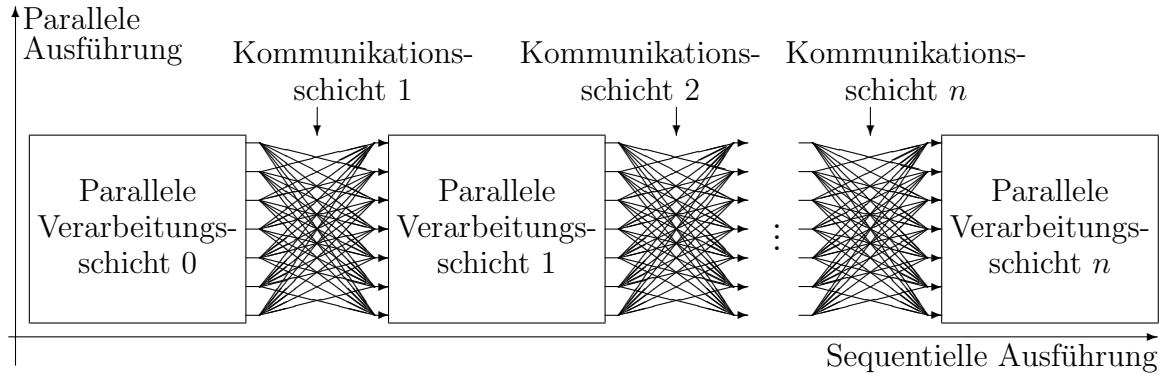

Abbildung 3.1: Parallel/sequentielle Ausführung eines Algorithmus.

Eine *Verarbeitungsschicht* $\mathcal{E}^{(s)}$ wird durch alle Untermengen $\mathcal{E}_k^{(s)}$ gebildet. Eine Menge $\mathcal{E}_k^{(s)}$ besteht aus Ecken, deren zugehörige Operationen auf Prozessor k nach einer Kommunikation mit den übrigen Prozessoren lokal berechnet werden können:

$$\mathcal{E}^{(s)} = \bigcup_{k=0}^{P-1} \mathcal{E}_k^{(s)} \tag{3.12}$$

$$\mathcal{E}_k^{(s)} = \Big\{ e_i \Big| \forall V(e_i) : (V(e_i) \in \bigcup_{l=0}^{s-1} \mathcal{E}^{(l)}) \vee (V(e_i) \in \mathcal{E}_k^{(s)}), \quad e_i \in (\mathcal{E}_k \setminus \bigcup_{l=0}^{s-1} \mathcal{E}_k^{(l)}) \Big\}.$$

Die Auswahl $e_i \in (\mathcal{E}_k \setminus \bigcup_l \mathcal{E}_k^{(l)})$ zur Ermittlung der Mengen $\mathcal{E}_k^{(s)}$ schließt alle Ecken vorgeordneter Schichten für folgende Schichten aus. Die Bedingung zur Bildung dieser Mengen beinhaltet eine kausale Ausführung, so daß zu jeder zugehörigen Ecke nur Kanten von Ecken vorgeordneter Schichten $(V(e_i) \in \bigcup_l \mathcal{E}^{(l)})$ oder von lokalen Ecken der gleichen Schicht $(V(e_i) \in \mathcal{E}_k^{(s)})$ zeigen dürfen.

Entsprechend der Verarbeitungsschichten lassen sich Untermengen der globalen Kanten $\mathcal{K}^G$ einer Zerlegung $\mathcal{Z}$ als *Kommunikationsschichten* definieren, welche den notwendigen Datenaustausch zwischen Prozessoren vor Ausführung einer Verarbeitungsschicht beschreiben. Eine Kommunikationsschicht $\mathcal{K}^{G,(s)}$ umfaßt alle globalen Kanten, welche auf Ecken einer Verarbeitungsschicht $\mathcal{E}^{(s)}$ weisen:

$$\mathcal{K}^{G,(s)} = \{k_{ij} = (e_i, e_j) \mid e_i \in \bigcup_{l=0}^{s-1} \mathcal{E}^{(l)} \ \wedge \ e_j \in \mathcal{E}^{(s)} \ \wedge \ k_{ij} \in \mathcal{K}^G\}. \tag{3.13}$$

Dieses Schichtenmodell führt zur parallel/sequentiellen Ausführung eines Graphen. Dies ist in Abbildung 3.1 graphisch angedeutet: Verarbeitungs- und Kommunikationsschichten werden, gegebenenfalls mit einer mit der Kausalität verträglichen zeitlichen Überlappung, sequentiell nacheinander ausgeführt. Die Ausführung einer jeden Schicht auf einem parallelen System geschieht hingegen parallel mit der gewählten Zerlegung.

3.1.2 Definitionen und Kenngrößen

Bei der Beschreibung paralleler Algorithmen sowie deren Implementierungen findet man in der Literatur Bezeichnungen und Kenngrößen, welche nicht immer einheitlich definiert

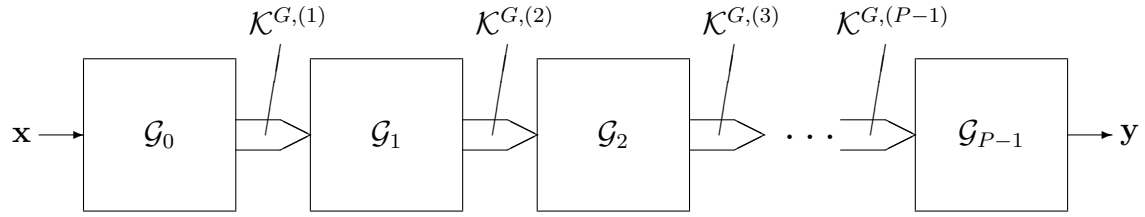

Abbildung 3.2: Blockdiagramm einer Funktionszerlegung.

sind. Im folgenden werden daher die Begriffe *Funktionszerlegung*, *Datenzerlegung*, '*Load-Balancing'* und *Granularität* aus der Sicht der im vorangegangenen Abschnitt eingeführten Graphenbeschreibung erläutert.

Bei der Betrachtung der einzelnen Begriffe wird eine Zerlegung $\mathcal{Z}(\mathcal{G})$ eines Graphen $\mathcal{G}$ vorausgesetzt.

Funktionszerlegung:
Eine Funktionszerlegung liegt dann vor, wenn in einer Verarbeitungsschicht s einer Zerlegung $\mathcal{Z}(\mathcal{G})$ alle Ecken $\mathcal{E}_k$ eines und nur eines Subgraphen ausgeführt werden. Mit der Zerlegung in P Subgraphen ergeben sich somit P Schichten des Gesamtgraphen (Abbildung 3.2). Bei einer Funktionszerlegung gilt dann:

$$\begin{array}{rcl} \mathcal{E}_k^{(k)} & = & \mathcal{E}_k \\ \mathcal{E}_k^{(s)} & = & \emptyset \quad \forall s \neq k. \end{array} \tag{3.14}$$

Algorithmisch läßt sich mit der Funktionszerlegung die Auflösung einer Funktion f in Teilfunktionen $f^{(i)}$ beschreiben, wobei deren Komposition wieder die Gesamtfunktion ergibt.

$$\mathbf{y} = f(\mathbf{x}) = f^{(P-1)} \circ \ldots \circ f^{(1)} \circ f^{(0)}(\mathbf{x}). \tag{3.15}$$

Als Datenflußgraph realisiert dabei $\mathcal{G}_k$ genau die Teilfunktion $f^{(k)}$.

Eine Funktionszerlegung ist dann sinnvoll, wenn mehrere Eingangsdatensätze verarbeitet werden müssen (Datensequenzverarbeitung). Nach Ausführung der ersten Schicht kann, während die Zwischenergebnisse in der zweiten Schicht weiterverarbeitet werden, in der ersten Schicht schon wieder die Bearbeitung des nächsten Datensatzes erfolgen. Diese Art der Verarbeitung wird auch als *'Pipelining'* bezeichnet. Damit diese parallele Ausführung sukzessiver Eingangsdaten nach einer Initialisierung effizient erfolgt, muß der zeitliche Aufwand zur Berechnung jeder Schicht gleich sein. Bei gleichem Aufwand für jede Ecke bedeutet dies eine gleiche Mächtigkeit der Eckenmengen $\mathcal{E}_k$ jeder Schicht.

Die Funktionszerlegung führt zu einer einfachen Kommunikationstopologie, denn globale Kanten $\mathcal{K}^{G,(s)}$ einer Kommunikationsschicht verbinden nur sukzessive Subgraphen $\mathcal{G}_{s-1}$ und $\mathcal{G}_s$ und benötigen damit lediglich eine lineare Verbindung benachbarter Prozessoren.

Datenzerlegung:
Eine *Datenzerlegung* soll, im Gegensatz zu obiger Funktionszerlegung, eine hohe Parallelarbeit innerhalb jeder Verarbeitungsschicht ermöglichen. Dazu muß jede Verarbeitungsschicht s aus möglichst gleichmächtigen Eckenmengen $\mathcal{E}_k^{(s)}$ aller Subgraphen bestehen. Mit

der Zuordnung jedes Subgraphen zu einem Prozessor ist dann der zeitliche Aufwand zur Berechnung zugehöriger Operationen jeder Menge $\mathcal{E}_k^{(s)}$ gleich und eine gleichmäßige Auslastung paralleler Prozessoren gegeben.

Die Datenzerlegung führt zu komplexeren Anforderungen an die Kommunikationsschichten $\mathcal{K}^{G,(s)}$. Ohne Einführung algorithmischer Nebenbedingungen müssen Kommunikationen zwischen beliebigen Subgraphen und somit zwischen beliebigen Prozessoren in jeder Kommunikationsschicht angenommen werden. Erst die Einführung algorithmischer Nebenbedingungen kann diese Kommunikationsanforderung einschränken.

Wünschenswert ist bei einer Datenzerlegung eine einheitliche Topologie in allen Kommunikationsschichten, da dann eine nur einmalige Abstimmung dieser Topologie mit einer Verbindungstopologie zwischen Prozessoren eines parallelen Systems notwendig ist und diese sukzessive in allen Kommunikationsschichten Anwendung findet.

'Load-Balancing'
Der Begriff des *'Load Balancing'* dient zur Beschreibung der Auslastung paralleler Prozessoren. Einen optimalen Wert nimmt es an, wenn die Berechnungen einer Schicht auf allen P Prozessoren gleichzeitig beginnen und gleichlange dauern.

In Verbindung mit der Funktionszerlegung, wo nach einer Initialisierung alle Prozessoren parallel Subgraphen sukzessiver Eingangsdatensätzen verarbeiten, bedeutet dies die gleiche Mächtigkeit aller P Eckenmengen $\mathcal{E}_k$. Bei der Datenzerlegung bedeutet dies die gleiche Mächtigkeit aller Eckenmengen $\mathcal{E}_k^{(s)}$ in jeder Schicht s.

Eine Unausgewogenheit der jeweiligen Eckenmengen schlägt sich in den Leistungsdaten (siehe Abschnitt 6.2.2), welche sich bei der Implementierung eines Algorithmus auf einem Multiprozessorsystem ergeben, nieder.

Granularität
Zur Beschreibung der Granularität gibt es unterschiedliche Modelle. Stone [Sto90] bezeichnet die Zeit, die ein Teilalgorithmus unabhängig auf einem Prozessor ablaufen kann, als Laufzeitquantum. Es entspricht bei obiger Graphenbeschreibung der Mächtigkeit der Eckenmenge $|\mathcal{E}_k^{(s)}|$ eines ausführbaren Subgraphen. Nach dessen Ausführung ist eine Kommunikation und Synchronisation mit den übrigen, parallel ablaufenden Teilalgorithmen notwendig. Dies führt zu einem Overhead $C_k^{(s)}$, der im obigem Graphenmodell aus der Kommunikationsschicht $\mathcal{K}^{G,(s+1)}$ und dem Verbindungsnetzwerk (siehe Abschnitt 5.3) zwischen den Prozessoren für jeden Prozessor k bestimmt werden muß. Die Granulariät $G_k^{(s)}$ eines Algorithmus in der Schicht s bestimmt sich in dem Modell von Stone durch den Quotienten aus Laufzeitquantum und Overhead:

$$G_k^{(s)} \sim \frac{|\mathcal{E}_k^{(s)}|}{C_k^{(s)}}. \tag{3.16}$$

Ein hohes Verhältnis beschreibt eine *grobe* (coarse) Granularität, eine kleiner Wert für G bezeichnet eine *feine* Granularität.

Andere Definitionen [MG89] bezeichnen die Granularität eines Programmes als die Menge von Programmschritten, die sequentiell auf einem einzelnen Prozessor ohne Unterbrechung ausgeführt werden können. Dies entspricht direkt der Mächtigkeit der Eckenmenge $|\mathcal{E}_k^{(s)}|$ und berücksichtigt nicht den aus der Zerlegung resultierenden Kommunikationsaufwand.

Auch wenn unterschiedliche Modelle zur Definition der Granularität G verwendet werden, steht diese jeweils in Konkurrenz zu der bei einer parallelen Programmausführung notwendigen Unterbrechung des linearen Programmablaufes.

3.2 Zerlegungsprinzipien

Für bestimmte Algorithmenklassen lassen sich einheitliche Zerlegungsprinzipien finden und deren effiziente Datenflußgraphen und Zerlegungen ermitteln. In den folgenden beiden Unterabschnitten werden Prinzipien für zwei Algorithmenklassen betrachtet, welche in der Bildverarbeitung Bedeutung besitzen. Zunächst wird die Zerlegung lokaler Operationen der Vorverarbeitung untersucht, danach die Zerlegung globaler Transformationen, welche in den Bildverarbeitungsschichten mittlerer bis hoher Komplexität Anwendung finden.

3.2.1 Lokale Operationen

Lokale Operationen auf einer geordneten Menge $\mathbf{I}_1(\mathcal{I})$ von Eingangsdaten sind dadurch gekennzeichnet, daß ihre Ergebnisse von einer lokalen Umgebung der Eingangsdaten abhängen. Es entsteht wiederum eine geordnete Menge $\mathbf{I}_2(\mathcal{I})$. Im weiteren werden diese geordneten Mengen als *Bilder* angesprochen.

Die zugrundeliegende Ordnung der Eingangsdaten wird durch eine *Indexmenge* $\mathcal{I}$ ausgedrückt. Bei zweidimensionalen $N \times M$ Bildern gilt beispielsweise: $\mathcal{I} = I\!N_N \times I\!N_M$.

Eine *Umgebung* $\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F})$ entnimmt einem Bild $\mathbf{I}_1(\mathcal{I})$ Elemente in der Umgebung eines Index $\mathbf{i} \in \mathcal{I}$, deren Positionen $\mathbf{p}$ relativ zu $\mathbf{i}$ durch eine Fenstermenge $\mathcal{F}$ bestimmt werden:

$$\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F}) = \{\mathbf{I}_1(\mathbf{p}) | \, \mathbf{p} = \mathbf{i} - \mathbf{f}, \;\; \mathbf{f} \in \mathcal{F}\}. \tag{3.17}$$

Der Ausdruck $\mathbf{I}_1(\mathbf{p})$ beschreibt ein Element (Pixel) des Bildes $\mathbf{I}_1(\mathcal{I})$ an der Position $\mathbf{p}$ [3]. Die Subtraktion $\mathbf{i} - \mathbf{f}$ wird als eine elementweise, vektorielle Subtraktion durchgeführt.

In der Bildverarbeitung finden häufig quadratische Fenster auf zweidimensionaler Indexmenge Anwendung. Ein symmetrisches Fenster der Größe $k \times k$ ergibt sich dann beispielsweise zu:

$$\mathcal{F}^{k\times k} = \{\, \mathbf{f} = (i,j) | \, -\frac{k-1}{2} \leq i \leq \frac{k-1}{2}, \;\; -\frac{k-1}{2} \leq j \leq \frac{k-1}{2}\}. \tag{3.18}$$

Mit der Umgebungsdefinition lassen sich lokale Operationen f auf einem Bild $\mathbf{I}_1(\mathcal{I})$ wie folgt darstellen:

$$\mathbf{I}_2(\mathcal{I}) = f(\mathbf{I}_1(\mathcal{I}), \mathcal{F}) = \{\mathbf{I}_2(\mathbf{i}) \,|\, \mathbf{I}_2(\mathbf{i}) = f(\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F})), \;\; \mathbf{i} \in \mathcal{I}\}. \tag{3.19}$$

[3] Die relative Definition der Umgebung $\mathcal{U}_{\mathbf{i}}$ führt an den Rändern eines Bildes $\mathbf{I}_1(\mathcal{I})$ zu Problemen, dort werden bei endlichen Bildern Positionen ermittelt, welche nicht mehr zum Bild gehören.

Mögliche Lösungen ignorieren nichtdefinierte Umgebungswerte und verkleinern damit die Mächtigkeit der Umgebungsmenge am Rand, oder sie setzen nichtdefinierte Umgebungelemente auf einem festen, konstanten Wert. Weiter kann man einen Torus als Indexmenge $\mathcal{I}$ dem Bild zugrunde legen, wodurch das Bild keinen Rand mehr besitzt und somit Umgebungswerte einem gegenüberliegenden Bildteil entnommen werden.

Eine weitere Lösung ist die Verkleinerung der Indexmenge eines Ergebnisbildes $\mathbf{I}_2$, so daß durch eine lokale Bildoperation nur Ergebniswerte bestimmt werden, deren zugehörige Umgebungen vollständig in $\mathbf{I}_1(\mathcal{I})$ enthalten sind.

Bei der zweidimensionalen Faltung [Wah84] werden beispielsweise durch die Funktion f die Elemente der Umgebungen $\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F})$, $\mathbf{i} \in \mathcal{I}$ komponentenweise mit Elementen einer Gewichtungsmenge multipliziert und die entstehenden Produkte aufsummiert zu Pixeln $\mathbf{I}_2(\mathbf{i})$, $\mathbf{i} \in \mathcal{I}$ des Ergebnisbildes. Die Menge $\mathcal{W}$ entspricht in dieser Darstellung dem Faltungskern:

$$\begin{aligned} \mathbf{I}_2(\mathcal{I}) &= \mathbf{I}_1(\mathcal{I}) * \mathcal{W} \\ &= \{\mathbf{I}_2(\mathbf{i}) \,|\, \mathbf{I}_2(\mathbf{i}) = \textstyle\sum (\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F}) \cdot \mathcal{W}),\ \ \mathbf{i} \in \mathcal{I}\}. \end{aligned} \tag{3.20}$$

Bei morphologischen Bildfilterungen [Ser82] beschreibt f nichtlineare Funktionen, im Falle der Erosion z.B. das Minimum über die Umgebungselemente:

$$\begin{aligned} \mathbf{I}_2(\mathcal{I}) &= \mathbf{I}_1 \ominus \mathcal{F} \\ &= \{\mathbf{I}_2(\mathbf{i}) \,|\, \mathbf{I}_2(\mathbf{i}) = \min (\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F})),\ \ \mathbf{i} \in \mathcal{I}\}. \end{aligned} \tag{3.21}$$

Zerlegung lokaler Operationen

Zur Zerlegung lokaler Operationen auf Bildern werden drei Möglichkeiten vorgeschlagen: Die Segmentierung der Indexmenge, die Auflösung der Funktion f sowie die Trennung der Umgebungsbildung von der Berechnung der Ergebniselemente $\mathbf{I}_2(\mathbf{i})$.

Segmentierung der Indexmenge

Bei der Segmentierung der Indexmenge $\mathcal{I}$ teilt man diese in zusammenhängende Segmente mit möglichst wenig Randpunkten auf, so daß Umgebungen zur Berechnung eines Ergebniselements $\mathbf{I}_2(\mathbf{i})$ meist in einem Segment des Eingangsbildes $\mathbf{I}_1(\mathcal{I})$ enthalten sind. Nur an Segmentgrenzen werden dann Umgebungselemente von Nachbarsegmenten — und somit bei der parallelen Implementierung von benachbarten Prozessoren durch Kommunikation — benötigt.

Diese Form der Zerlegung entspricht der in Abschnitt 3.1.2 beschriebenen Datenzerlegung. Übliche Segmente bei quadratischen Bildern sind horizontale oder vertikale Streifen und auch die Aufteilung in Quadrate oder Rechtecke.

Auflösung der Funktion f

Eine Funktionszerlegung nach Abschnitt 3.1.2 ergibt sich durch Auflösung der Funktion f in Analogie zu Gleichung 3.15. Man erhält direkt das dort beschriebene 'Pipelining' mit dem Bild $\mathbf{I}_1(\mathcal{I})$ als Eingangsdaten, $P-1$ temporären Bildern zwischen den Pipeline-Schichten und dem Bild $\mathbf{I}_2(\mathcal{I})$ als Ergebnisdaten.

Wenn beispielsweise bei der Faltung die Auflösung des Faltungskerns $\mathcal{W} = \mathcal{W}_0 * \mathcal{W}_1$ in kleinere Kerne möglich ist, kann direkt dieses Pipeline-Prinzip angewendet werden. Dann gilt: $\mathbf{I}_2(\mathcal{I}) = \mathbf{I}_1(\mathcal{I}) * \mathcal{W} = (\mathbf{I}_1(\mathcal{I}) * \mathcal{W}_0) * \mathcal{W}_1$.

Die alleinige Auflösung der Funktion f bringt nur einen Gewinn bei der Verarbeitung von Bildsequenzen, da bei der Funktionszerlegung erst dann eine Parallelarbeit möglich ist.

Trennung der Umgebungsbildung von der Berechnung

Ein weiterer Ansatz zur schnellen Berechnung lokaler Bildoperationen ist die Trennung der Umgebungsbildung von der Berechnung. Die Berechnung aller Ergebniswerte $\mathbf{I}_2(\mathbf{i});\ \mathbf{i} \in \mathcal{I}$

aus den zugehörigen Umgebungen kann als Sequenzverarbeitung $|\mathcal{I}|$ unabhängiger Berechnungen interpretiert werden. Damit ist in Kombination mit einer möglichst weitgehenden Funktionszerlegung von $f(\mathcal{U}_\mathbf{i}(\mathbf{I}_1(\mathcal{I}), \mathcal{F}))$ eine effiziente Parallelarbeit auch bei Verarbeitung eines einzelnen Bildes möglich.

Im Zusammenhang mit den Operationsprinzipien von Prozessoren wird in Abschnitt 5.1 näher auf die Realisierung solcher Zerlegungen eingegangen.

3.2.2 Globale Transformationen

Bei globalen Transformationen werden, im Gegensatz zu lokalen Algorithmen, Punkte miteinander verknüpft, die nicht in einer gemeinsamen lokalen Umgebung liegen. In der Bildverarbeitung treten besonders in den Verarbeitungsschichten mittlerer bis hoher Komplexität (siehe Bild 2.1) solche Algorithmen mit globaler Verknüpfungstopologie auf.

Beschreibung einer rekursiv definierten Klasse globaler Transformationen

Die im folgenden betrachtete Klasse linearer und nichtlinearer, globaler Algorithmen [Bur79] läßt sich effizient durch eine rekursive Definition beschreiben, welche eine Zerlegung der Verarbeitung bewirkt. Zur Formulierung der Rekursion wird eine Beschränkung der Eingangsdaten auf eindimensionale Vektoren vorgenommen. Für Eingangsdaten $\mathbf{x}$ gilt somit:

$$\mathbf{x} = \{x_i \mid i \in \mathcal{I} = I\!N_N\}. \tag{3.22}$$

Die Einschränkung auf eindimensionale Eingangsdaten ist nicht zwingend, sie dient der anschaulichen Darstellung. Es lassen sich jedoch auch höherdimensionale Eingangsdaten (z.B. Bildmatrizen) auf die Vektordarstellung zurückführen, so daß damit gewonnenen Aussagen auch über die Vektordarstellung hinaus Relevanz besitzen.

Die Mächtigkeit N der Indexmenge sei Potenz einer Basis B, d.h. $|I\!N_N| = N = B^n$. Im folgenden wird $B = 2$ angenommen. Burkhardt benutzt in [Bur79] eine Notation zur Beschreibung einer allgemeinen Zerlegung eines Vektors in gleichgroße Subvektoren:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_{1|2} \\ \mathbf{x}_{2|2} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{1|4} \\ \mathbf{x}_{2|4} \\ \mathbf{x}_{3|4} \\ \mathbf{x}_{4|4} \end{bmatrix} = \ldots = \begin{bmatrix} \mathbf{x}_{1|N} \\ \vdots \\ \mathbf{x}_{N|N} \end{bmatrix}. \tag{3.23}$$

Mit der eingeführten Notation wird eine rekursiv auflösbare Transformation $\tilde{\mathbf{x}} = T(\mathbf{x})$ der Dimension N beschrieben und durch den Rekursionsschritt in zwei gleichartige Transformationen der halben Dimension überführt:

$$\tilde{\mathbf{x}} = T(\mathbf{x}) = \begin{bmatrix} T(\mathbf{f_1}^{(1)}(\mathbf{x}_{1|2}, \mathbf{x}_{2|2})) \\ T(\mathbf{f_2}^{(1)}(\mathbf{x}_{1|2}, \mathbf{x}_{2|2})) \end{bmatrix} = \begin{bmatrix} T(\mathbf{x}^{(1)}_{1|2}) \\ T(\mathbf{x}^{(1)}_{2|2}) \end{bmatrix}. \tag{3.24}$$

Die dyadischen Vektorfunktionen $\mathbf{f_1}^{(1)}$ und $\mathbf{f_2}^{(1)}$ berechnen die Eingangsvektoren $\mathbf{x}^{(1)}_{1|2}$ und $\mathbf{x}^{(1)}_{2|2}$ des folgenden Rekursionsschritts. Die Vektorfunktionen bewirken eine komponentenweise Verknüpfung ihrer Operandenvektoren $\mathbf{x}_{1|2}$ und $\mathbf{x}_{2|2}$. Die rekursive Auflösung führt zu einem Datenflußgraphen $\mathcal{G}_G$, dessen Struktur für $N = 8$ beispielhaft in Abbildung 3.3

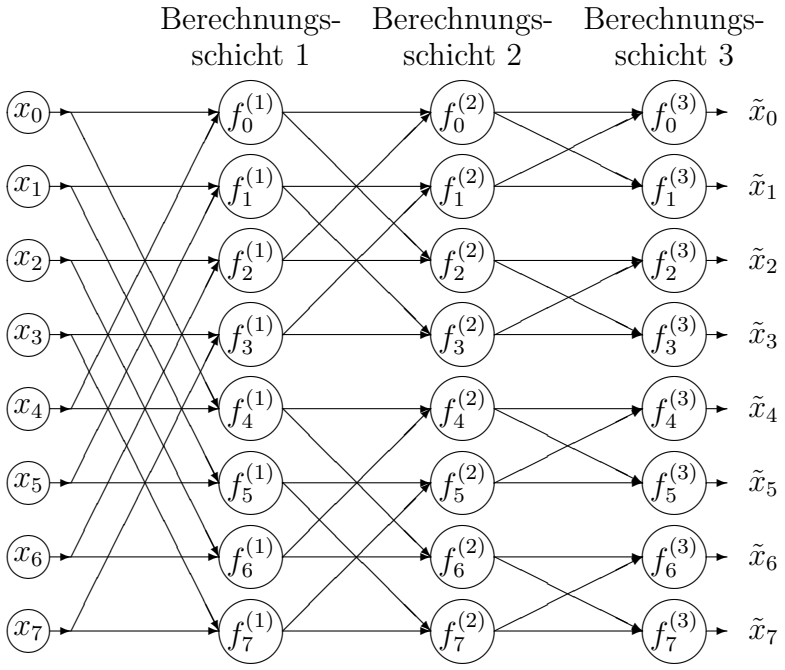

Abbildung 3.3: 1. kanonische Form $\mathcal{G}_G$ eines Graphen zur Berechnung globaler Transformationen, Beispiel: $N = 8$.

dargestellt ist. Man erkennt eine schichtweise Unterteilung. Burkhardt bezeichnet diesen Graphen als 1. *kanonische Form*.

Die Vektorfunktionen $\mathbf{f_i}^{(m)}$ eines Rekursionsschritts m bilden zusammengefaßt eine Berechnungsschicht $\mathbf{f}^{(m)}$:

$$\mathbf{f}^{(m)} = \{f_i^{(m)} |\ i \in I\!N_n,\ n = \log_2 N\}. \tag{3.25}$$

Die Elemente des Eingangsvektors sollen als Eingangsecken die Berechnungsschicht 0 bilden:

$$\mathbf{f}^{(0)} = \{f_i^{(0)} | f_i^{(0)} = x_i,\ i \in I\!N_N\}. \tag{3.26}$$

Dann läßt sich ein Datenflußgraph $\mathcal{G}_G$ zu dieser Klasse globaler Transformationen über seine Ecken- und Kantenmengen beschreiben:

$$\mathcal{G}_G = (\mathcal{E}_G, \mathcal{K}_G) \tag{3.27}$$

$$\mathcal{E}_G = \{f_i^{(m)}|, \quad i \in I\!N_N,\ m \in I\!N_{n+1}\} \tag{3.28}$$

$$\mathcal{K}_G = \{(f_i^{(m-1)}, f_j^{(m)}) | j = i \vee j = i \oplus 2^{n-m}, \quad i \in I\!N_N,\ (m-1) \in I\!N_n\}. \tag{3.29}$$

Korrespondierende Funktionen:
Die Kanten des Graphen weisen immer nur von Ecken einer Berechnungsschicht $m-1$ zur Folgeschicht m. Jede Ecke $f_i^{(m)}$ besitzt nach Gleichung 3.29 die zwei Vorgänger $f_i^{(m-1)}$ und $f_j^{(m-1)},\ j = i \oplus 2^{n-m}$. Es existiert nun in Berechnungsschicht m eine zweite Ecke $f_j^{(m)},\ j = i \oplus 2^{n-m}$ mit den gleichen beiden Vorgängern. Die beiden Ecken $f_i^{(m)}$ und $f_j^{(m)}$ werden im folgenden als *korrespondierende Funktionen* bezeichnet.

Beispiele zur Klasse globaler Transformationen

Die beschriebene Klasse schneller Transformationen kann durch geeignete Wahl der Funktionen $f_i^{(m)}$ an vielfältige lineare und nichtlineare Transformationen angepaßt werden. Als Beispiele sind in [BB85, BLN90] die schnelle Fouriertransformation (FFT), die schnelle Walsh-Transformation (FWT), eine Klasse lageinvarianter Transformationen (CT), schnelles paralleles Sortieren, die *maximum a posteriori* Schätzung von Markov-Prozessen mit dem Viterbi-Algorithmus, die Polynomberechnung, die Matrixtransposition sowie die Berechnung des Innenproduktes angeführt.

In vielen der Beispiele finden sich in den $N \cdot \log_2 N$ Funktionsecken $f_i^{(m)}$ nur zwei unterschiedliche Operatoren f_1 und f_2, die jeweils von korrespondierenden Funktionen realisiert werden: Die Walsh Transformation beinhaltet das Operatorpaar $(f_1, f_2) = (+, -)$, das schnelle Sortieren nach [Sto71] baut auf einem *Bitonen Sortierer* mit dem Operatorpaar (min, max) auf, und die Klasse CT benötigt ein Paar kommutativer Operatoren wie z.B. die Addition und betragsmäßige Differenz.

In die Topologie des beschriebenen Datenflußgraphen können weiterhin Baumtopologien der Tiefe n eingebettet werden. Die Berechnung eines Ergebniswertes in der Baumspitze aus N Eingangswerten verwendet dann nur einen Teil der gesamten Funktionsecken: Von Berechnungsschicht 1 wird die Hälfte der Ecken benötigt, in den folgenden Schichten halbiert sich sukzessive die benötigte Eckenzahl bis schließlich die Spitze des Baumes in Schicht n nur noch eine Ecke belegt. Die Auswahl der Funktionsecken in den einzelnen Berechnungsschichten bestimmt sich aus der Wahl der Vektorkomponente $\tilde{x_i}$, auf der das Ergebnis ausgegeben werden soll.

Bei der Datenverarbeitung mit Baumtopologie kann weiterhin durch gleiche Wahl von $\mathbf{f_1}^{(m)}$ und $\mathbf{f_2}^{(m)}$ und somit durch redundante Berechnung von Zwischenergebnissen auf vorher unbenutzten Funktionsecken erreicht werden, daß die Baumspitze gleichzeitig in allen Ergebnisecken endet. Damit steht das Ergebnis in allen Komponenten $\tilde{x_i}$ des Ergebnisvektors parallel zur Verfügung und braucht nicht mehr kommuniziert werden.

Ein Beispiel zur Berechnung mit Baumtopologie ist die Ermittlung der Summe über N Eingangswerte durch Halbierungsstrategie. Mit der Addition als Operation in benötigten Funktionsecken $f_i^{(m)}$ erhält man diese Summe in einem frei wählbaren Element $\tilde{x_j}$ des Ergebnisvektors. Mit Wahl von $\mathbf{f_1}^{(m)}$ als komponentenweiser Addition legt man beispielsweise die Baumspitze in $\tilde{x_0}$:

$$\tilde{\mathbf{x}} = T(\mathbf{x}) = \begin{bmatrix} T(\mathbf{x_{1|2}} + \mathbf{x_{2|2}}) \\ T(\mathbf{f_2}(\mathbf{x_{1|2}}, \mathbf{x_{2|2}})) \end{bmatrix}. \tag{3.30}$$

Die Funktion $\mathbf{f_2}$ braucht nicht spezifiziert werden, da das Ergebnis $\tilde{x_0}$ davon unabhängig ist. Wählt man $\mathbf{f_2}$ ebenfalls als komponentenweise Addition, so erhält durch die redundante Berechnung jede Komponente des Ergebnisvektors die Summe über die Eingangsdaten.

Parallelisierung der Klasse globaler Transformationen

Funktionszerlegung globaler Transformationen:
Eine Funktionszerlegung der beschriebenen Klasse globaler Algorithmen nach Ab-

schnitt 3.1.2 erhält man direkt durch Zerlegung des Gesamtgraphen in seine n Berechnungsschichten. Es gilt dann $P = n$. Die Ecken $f_i^{(m)}$ einer Schicht m bilden die Eckenmenge eines Subgraphen $\mathcal{G}_G^m$.

Das 'Pipelining', welches sich nach Abschnitt 3.1.2 bei der sukzessiven Ausführung dieser Subgraphen ergibt, schlagen z.B. Rabiner und Gold [RG75] zur Ausführung der schnellen Fouriertransformation vor. Das dort beschriebene Vorgehen läßt sich weitgehend auf diese ganze Transformationsklasse übertragen.

Datenzerlegung globaler Transformationen:
Eine mögliche disjunkte Datenzerlegung $\mathcal{Z}^D(\mathcal{G}_G)$ basiert auf einer horizontalen Partitionierung des globalen Graphen in $P = 2^p$ Subgraphen, wobei N/P Ecken mit aufeinanderfolgendem Index über alle $n+1$ Schichten dem gleichen Subgraphen $\mathcal{G}_{G,k}$ zugeordnet werden. Dies ergibt bei der 1. kanonischen Form die folgende Eckenmenge $\mathcal{E}_{G,k}$ eines Subgraphen:

$$\mathcal{E}_{G,k} = \{f_i^{(m)} | \lfloor i/P \rfloor = k,\ i \in I\!N_N,\ m \in I\!N_{n+1}\}. \tag{3.31}$$

Diese Zerlegung führt nach Gleichung 3.12 zu $\log_2(P)+1$ Verarbeitungsschichten $\mathcal{E}_G^{(s)}$:

$$\mathcal{E}_G^{(s)} = \begin{cases} \{f_i^{(s)} | i \in I\!N_N\} & \text{für } s \in I\!N_{\log_2(P)} \\ \{f_i^{(m)} | i \in I\!N_N,\ m \in \{\log_2(P), \ldots, n\}\} & \text{für } s = \log_2(P) \end{cases}. \tag{3.32}$$

Davon beinhaltet die Menge $\mathcal{E}_G^{(0)}$ alle Eingangsecken. Weiterhin beinhaltet die Menge $\mathcal{E}_G^{(\log_2(P))}$ alle Berechnungsschichten mit $\{m \in \log_2(P), \ldots, n\}$, welche somit lokal ausgeführt werden können.

Betrachtet man die zugehörigen Kommunikationsschichten $\mathcal{K}_G^{G,(s)}$ nach Gleichung 3.13, so ergibt sich aufgrund der gewählten Zerlegung und der Kantenmenge $\mathcal{K}_G$ des Graphen (Gleichung 3.29) eine Abhängigkeit der durch die Kanten beschriebenen Kommunikationstopologie von s:

$$\mathcal{K}_G^{G,(s)} = \{(f_i^{(s-1)}, f_j^{(s)}) | j = i \oplus 2^{n-s},\ i \in I\!N_N,\ s-1 \in I\!N_{\log_2(P)}\}. \tag{3.33}$$

Die durch $j = i \oplus 2^{n-s}$ ausgedrückte Abhängigkeit benötigt bei der schichtweisen Ausführung (Abbildung 3.1) des Graphen auf P Prozessoren für jede Kommunikationsschicht eine jeweils andere Verbindungstopologie der parallelen Prozessoren.

Eine einheitliche Topologie der Kanten zwischen allen Berechnungsschichten bietet der Graph $\mathcal{G}_{G2}$ zur Klasse globaler Transformationen in der *2. kanonischen Form* [BB85] (siehe Abbidung 3.4). Die verwendete σ-Funktion wird unten erläutert:

$$\mathcal{G}_{G2} = (\mathcal{E}_{G2}, \mathcal{K}_{G2}) \tag{3.34}$$

$$\mathcal{E}_{G2} = \{f_{i'}^{(m)} | i = \sigma^{-m}(i') \wedge f_{i'}^{(m)} = f_i^{(m)},\ f_i^{(m)} \in \mathcal{E}_G,\ i' \in I\!N_N,\ m \in I\!N_{n+1}\} \tag{3.35}$$

$$\mathcal{K}_{G2} = \{(f_{i'}^{(m-1)}, f_{j'}^{(m)}) | j' = \sigma(i') \vee j' = \sigma(i') \oplus 1,\ i' \in I\!N_N,\ m+1 \in I\!N_n\}. \tag{3.36}$$

Er geht aus der ersten Form $\mathcal{G}_G$ durch Permutation der inneren Ecken $f_i^{(s)},\ s+1 \in I\!N_{n-1}$ hervor. Die Ecken $f_i^{(0)}$ und $f_i^{(n)}$ der ersten und letzten Schicht bleiben unverändert. Damit besitzen $\mathcal{G}_G$ und $\mathcal{G}_{G2}$ gleiches Ein-/Ausgangsverhalten. Dies wird durch die Gleichungen 3.42 bis 3.45 für die gewählten Graphendarstellungen noch bewiesen.

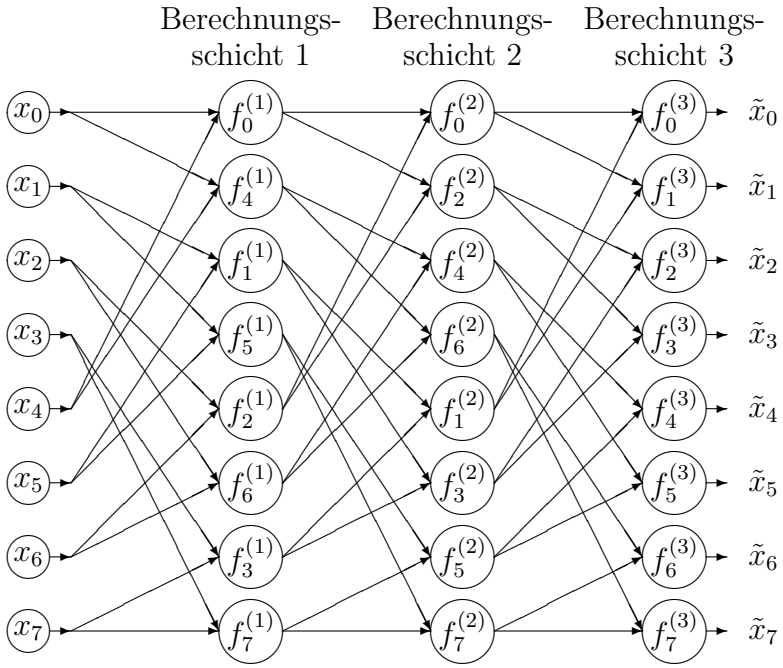

Abbildung 3.4: 2. kanonische Form $\mathcal{G}_{G2}$ eines Graphen zur Berechnung globaler Transformationen, Beispiel: $N = 8$.

Zur Beschreibung der 2. kanonischen Form des Graphen wird eine Funktion $\sigma(i)$ verwendet. Sie beschreibt eine Indexpermutation durch Rotation der Indexziffern und wird als 'Shuffle'-Permutation bezeichnet [BLN90]:

$$\begin{aligned} \sigma_{q,r}(i) &= \sigma_{q,r}((i_{n-1},\ldots,i_{q+r},i_{q+r-1},\ldots,i_{r+1},i_r,i_{r-1},\ldots,i_0)_B) \\ &= (i_{n-1},\ldots,i_{q+r},\underbrace{i_{q+r-2},\ldots,i_{r+1},i_r,i_{q+r-1}}_{q \text{ rotierte Indexziffern ab Stelle } r},i_{r-1},\ldots,i_0)_B \qquad (3.37) \\ \sigma(i) &= \sigma_{n,0}(i). \qquad (3.38) \end{aligned}$$

Mehrfache Anwendung der 'Shuffle'-Permutation wird durch eine entsprechende Potenz ausgedrückt, so beschreibt $\sigma_{q,r}^k(i)$ ein k-faches 'Shuffle' des Index i:

$$\sigma_{q,r}^k(i) = \underbrace{\sigma_{q,r}(\sigma_{q,r}(\ldots\sigma_{q,r}(}_{k\text{-mal}} i)\ldots)). \qquad (3.39)$$

Die zu $\sigma_{q,r}^k(i)$ inverse Funktion soll mit einem negativen Exponenten gekennzeichnet werden. Dann gilt:

$$\sigma_{q,r}^{-k}(\sigma_{q,r}^k(i)) = i. \qquad (3.40)$$

Die mehrfache Anwendung der 'Shuffle'-Permutation zeigt periodisches Verhalten. So gilt, wie man sich anhand von Gleichung 3.37 leicht veranschaulichen kann:

$$\sigma_{q,r}^{l\cdot q}(i) = \sigma_{q,r}^{-l\cdot q}(i) = i, \quad l \in I\!N. \qquad (3.41)$$

Die Beschreibung der σ-Funktion bestätigt die behauptete Ein-/Ausgangsäquivalenz zwischen der 1. und 2. kanonischen Form:

In Gleichung 3.35 in den Berechnungsschichten $m = 0$ und $m = n$ werden die Korrespondenzen $i = \sigma^{-m}(i')$ zur identischen Abbildung $i = i'$, und somit sind Ecken $f_i^{(m)}$ und $f_{i'}^{(m)}$ dieser Schichten in beiden Graphendarstellungen identisch.
Die topologische Korrespondenz der Kanten beider Graphendarstellungen läßt sich dadurch zeigen, daß man die Beziehung zwischen i' und j' (Gleichung 3.36) zur Beschreibung der Kanten in der 2. kanonischen Form wieder auf die Indexwerte i und j (Gleichung 3.29) der 1. Form zurückführt:

Fall 1:

$$\begin{aligned} j' &= \sigma(i') && (3.42)\\ \underbrace{\sigma^{-m}(j')}_{\text{Indexwerte von Schicht } (m)} &= \underbrace{\sigma^{-m}(\sigma(i'))}_{\text{Indexwerte von Schicht } (m-1)} \\ j &= \sigma^{-(m-1)}(i') \\ j &= i && (3.43) \end{aligned}$$

Fall 2:

$$\begin{aligned} j' &= \sigma(i') \oplus 1 && (3.44)\\ \underbrace{\sigma^{-m}(j')}_{\text{Indexwerte von Schicht } (m)} &= \underbrace{\sigma^{-m}(\sigma(i') \oplus 1)}_{\text{Indexwerte von Schicht } (m-1)} \\ j &= \sigma^{-(m-1)}(i') \oplus \sigma^{-m}(1) \\ j &= i \oplus 2^{n-m}. && (3.45) \end{aligned}$$

Gleichungen 3.43 und 3.45 stellen genau die Kantenbeziehungen der 1. kanonischen Form her (Gleichung 3.29), womit die topologische Korrespondenz der Kantenbeschreibungen beider Graphenformen bewiesen ist.
Aus der Identität der Ein- und Ausgangsecken und der topologischen Korrespondenz der Kanten ergibt sich das behauptete gleiche Ein-/Ausgangsverhalten beider Graphenformen. □

Eine horizontale Zerlegung entsprechend Gleichung 3.31 für den Graphen $\mathcal{G}_{G2}$ führt zu $n+1$ Verarbeitungsschichten $\mathcal{E}_{G2}^{(s)}$:

$$\mathcal{E}_{G2}^{(s)} = \bigcup_{k=0}^{P-1} \mathcal{E}_{G2,k}^{(s)} = \{f_{i'}^{(s)} | i' \in I\!N_N\}. \tag{3.46}$$

In der 2. Form des Graphen stimmen bei horizontaler Zerlegung somit Berechnungs- und Verarbeitungsschichten überein. Jede globale Kommunikationsschicht $\mathcal{K}_{G2}^{G,(s)}$ umfaßt dann alle Kanten[4] zwischen zwei Berechnungsschichten:

$$\mathcal{K}_{G2}^{G,(s)} = \{(f_{i'}^{(s-1)}, f_{j'}^{(s)}) | j' = \sigma(i') \vee j' = \sigma(i') \oplus 1,\ i' \in I\!N_N\}. \tag{3.47}$$

[4]Bei den Subgraphen $\mathcal{G}_{G2,0}$ und $\mathcal{G}_{G2,P-1}$ entstehen nach Definition (Gleichung 3.10) mit dieser Zerlegung jeweils $n \cdot N/P$ lokale Kanten. Zur einheitlichen Beschreibung werden diese jedoch den n globalen Kommunikationsschichten nach Gleichung 3.47 zugeordnet und somit alle Kanten des Graphen als globale Kanten betrachtet.

Diese Kommunikationsschichten weisen eine von der aktuellen Schicht s unabhängige Topologie auf, was sich in der von s unabhängigen Berechnung des Index j' in Gleichung 3.47 widerspiegelt.

Die Mächtigkeit der Menge $\mathcal{K}_{G2}^{G,(s)}$ ist doppelt so groß wie bei Kommunikationsschichten $\mathcal{K}_{G}^{G,(s)}$ der 1. kanonischen Form (Gleichung 3.33) und bedingt somit die doppelte Kommunikationsleistung. Mit $P = 2^p < N$ weisen jedoch beide von einer Ecke $f_{i'}^{(s-1)}$ wegführenden Kanten zum gleichen Subgraphen $\mathcal{G}_{G2,j'/P}$. Damit braucht bei einer Implementierung auf P Prozessoren die Kommunikation nur einmal stattfinden, und der durch die zwei Kanten beschriebene doppelte Datenzugriff wird lokal auf dem Zielprozessor durchgeführt. Die Einschränkung $P < N$ bewirkt weiterhin die Zuordnung korrespondiernder Funktionen zu einem Subgraphen bzw. zu einem Prozessor und damit in vielen Fällen ein gutes 'Load Balancing'.

Die horizontalen Zerlegungen beider Formen des Graphen der Klasse globaler Transformationen in P Subgraphen zeigen unterschiedliche Eigenschaften:

- Die horizontale Zerlegung der 1. kanonische Form benötigt bei P Subgraphen nur $\log_2(P)$ Kommunikationsschichten und erlaubt damit bei einer Implementierung die Minimierung der globalen Kommunikation bezüglich der Prozessoranzahl. Nachteil ist die unterschiedliche Kommunikationstopologie in jeder Schicht und bei unausgewogenem Aufwand korrespondierender Funktionen ein schlechtes 'Load Balancing'.
- Die horizontale Zerlegung der 2. kanonische Form besitzt eine einheitliche Topologie in allen Kommunikationsschichten und weist für $P < N$ korrenspondierende Funktionen einem Subgraphen zu. Jedoch benötigt sie immer n Kommunikationsschichten, unabhängig von der Anzahl P der Subgraphen bzw. der Prozessoren.

Eine dritte, zweistufige Form $\mathcal{G}_{G3}$ soll die Vorteile obiger beider Formen, d.h. wenige Kommunikationsschichten, einheitliche Topologie in diesen Schichten und Zuweisung korrespondierender Funktionen zu einem Subgraphen, kombinieren. Sie ermöglicht eine modulare Anpassung der Subgraphen an vorgebbare Prozessoranzahlen durch einen frei einstellbaren Parameter q:

$$\mathcal{G}_{G3}(q) = (\mathcal{E}_{G3}(q), \mathcal{K}_{G3}(q)) \tag{3.48}$$

$$\mathcal{E}_{G3}(q) = \mathcal{E}_{G3a}(q) \cup \mathcal{E}_{G3b}(q) \tag{3.49}$$

$$\mathcal{K}_{G3}(q) = \mathcal{K}_{G3a}(q) \cup \mathcal{K}_{G3b}(q) \tag{3.50}$$

$$\mathcal{E}_{G3a}(q) = \{f_{i''}^{(m)} | (i = \sigma_{q,n-q}^{-m}(i'')) \wedge (f_{i''}^{(m)} = f_{i}^{(m)}), \; f_{i}^{(m)} \in \mathcal{E}_G, \; i'' \in I\!N_N, \; m \in I\!N_{q+1}\} \tag{3.51}$$

$$\mathcal{E}_{G3b}(q) = \{f_{i''}^{(m)} | (i = i'') \wedge (f_{i''}^{(m)} = f_{i}^{(m)}), \; f_{i}^{(m)} \in \mathcal{E}_G, \; i'' \in I\!N_N, \; m - q - 1 \in I\!N_{n-q}\} \tag{3.52}$$

$$\mathcal{K}_{G3a}(q) = \{(f_{i''}^{(m-1)}, f_{j''}^{(m)}) | j'' = \sigma_{q,n-q}(i'') \vee j'' = \sigma_{q,n-q}(i'') \oplus 2^{n-q}, \; i'' \in I\!N_N, \; m - 1 \in I\!N_q\} \tag{3.53}$$

$$\mathcal{K}_{G3b}(q) = \{(f_{i''}^{(m-1)}, f_{j''}^{(m)}) | j'' = i'' \vee j'' = i'' \oplus 2^{n-m} \; i'' \in I\!N_N, \; m - q - 1 \in I\!N_{n-q}\}. \tag{3.54}$$

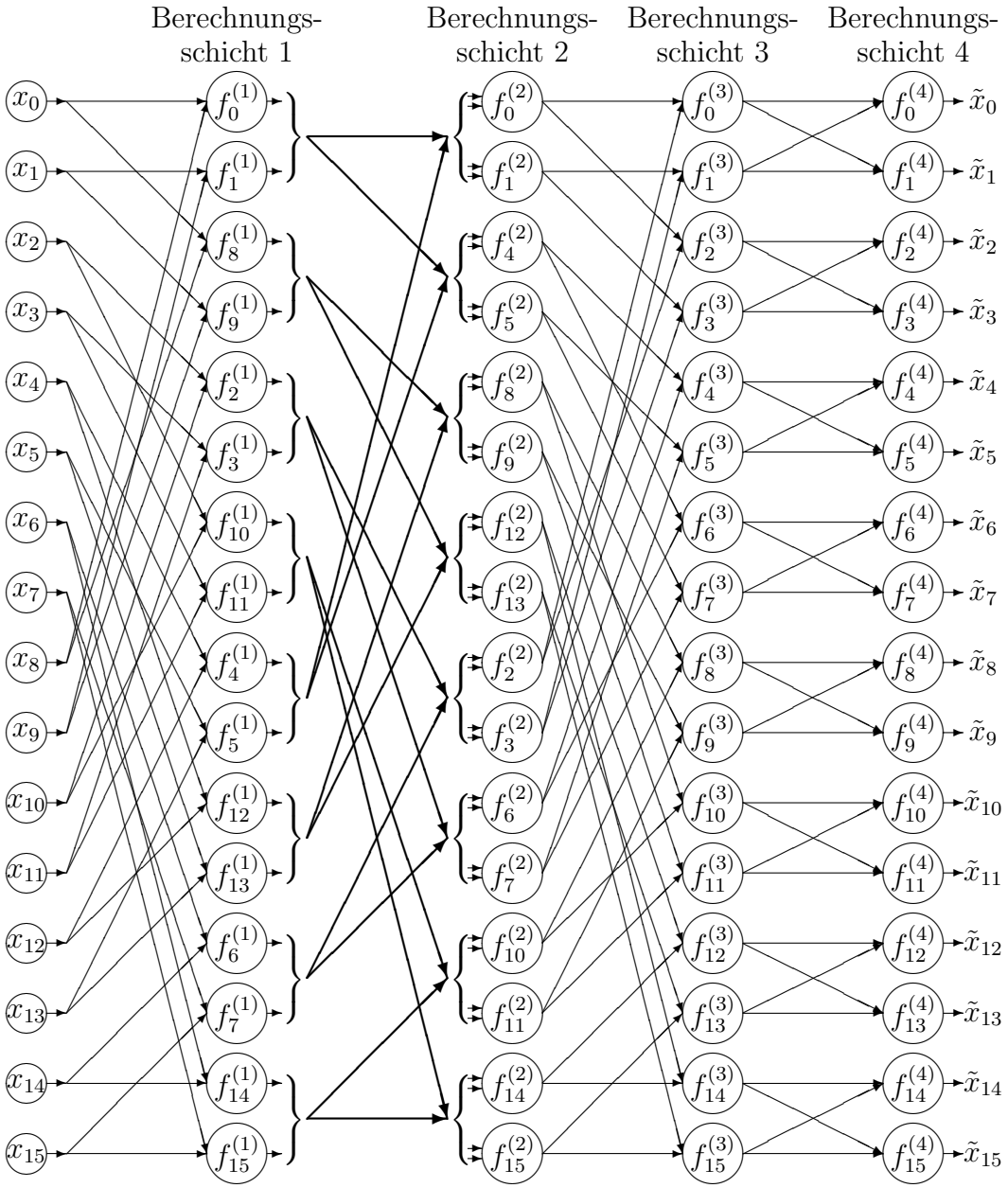

Abbildung 3.5: Zweistufige Form $\mathcal{G}_{G3}$ eines Graphen zur Berechnung globaler Transformationen, Beispiel: $N = 16$, $q = 3$.

Die erste Stufe des Graphen wird aus der Ecken- und Kantenmenge $\mathcal{E}_{G3a}$ und $\mathcal{K}_{G3a}$ gebildet. Die Ecken der ersten und letzten Berechnungsschicht $f_{i''}^{(0)}$ und $f_{i''}^{(q)}$ dieser Stufe stimmen mit den Schichten der 1. kanonischen Form überein. Die Funktionsecken $f_{i''}^{(m)}$ der Schichten $m \in \{1, \ldots, q-1\}$ sind so permutiert, daß die Kanten $(f_{i''}^{(m-1)}, f_{j''}^{(m)})$ aus $\mathcal{K}_{G3a}$ schichtweise mit m eine einheitliche, nur durch i'' und j'' bestimmte Topologie unabhängig von m aufweisen.

Die zweite Stufe des Graphen $\mathcal{G}_{G3}$, gebildet aus $\mathcal{E}_{G3b}$ und $\mathcal{K}_{G3b}$, ist identisch mit dem Graphen $\mathcal{G}_G$:

Die Kantenkorrespondenz der Graphen $\mathcal{G}_{G3}$ und $\mathcal{G}_G$ innerhalb der ersten Stufe läßt sich wieder dadurch zeigen, daß man die Beziehung zwischen i'' und j'' (Gleichung 3.53) auf die Indizes i und j der 1. kanonischen Form (Gleichung 3.29) zurückführt:

Für $m-1 \in \mathbb{N}_q$ gilt:

Fall 1:

$$j'' = \sigma_{q,n-q}(i'') \tag{3.55}$$

$$\underbrace{\sigma_{q,n-q}^{-m}(j'')}_{\text{Indexwerte von Schicht } (m)} = \underbrace{\sigma_{q,n-q}^{-m}(\sigma_{q,n-q}(i''))}_{\text{Indexwerte von Schicht } (m-1)}$$

$$j = \sigma_{q,n-q}^{-(m-1)}(i'')$$

$$j = i \tag{3.56}$$

Fall 2:

$$j'' = \sigma_{q,n-q}(i'') \oplus 2^{n-q} \tag{3.57}$$

$$\underbrace{\sigma_{q,n-q}^{-m}(j'')}_{\text{Indexwerte von Schicht } (m)} = \underbrace{\sigma_{q,n-q}^{-m}(\sigma_{q,n-q}(i'') \oplus 2^{n-q})}_{\text{Indexwerte von Schicht } (m-1)}$$

$$j = \sigma_{q,n-q}^{-(m-1)}(i'') \oplus \sigma_{q,n-q}^{-m}(2^{n-q})$$

$$j = i \oplus 2^{n-m}. \tag{3.58}$$

Die Gleichungen 3.56 und 3.58 zeigen nach der Umformung die Beziehung der 1. Form. Somit ist zusammen mit der Übereinstimmung der ersten und letzten Berechnungsschicht die Äquivalenz beider Graphenformen in der ersten Stufe hergestellt. In der zweiten Stufe ist die Äquivalenz direkt gegeben. Damit ist die Gleichheit des Ein-/Ausgangsverhaltens beider Darstellungen bewiesen. □

Abbildung 3.5 zeigt einen Graphen $\mathcal{G}_{G3}$ in dieser dritten, zweistufigen Form. Mit den gewählten Parametern $N = 16$ und $q = 3$ ergeben sich 3 identische Kantenschichten in der Menge $\mathcal{K}_{G3a}$. Sie können, wie in der zweiten Kantenschicht grafisch angedeutet ist, als Vektor-'Shuffle'-Schichten mit Vektorlänge $N/2^q$ interpretiert werden. Die vierte Kantenschicht des Graphen entspricht einer korrespondierenden Schicht der 1. kanonischen Form.

Die Anpassung einer horizontalen Zerlegung entsprechend Gleichung 3.31 für $\mathcal{G}_{G3}$ an eine vorgegebene Anzahl von P Subgraphen bzw. Prozessoren kann mit dem noch freien Parameter q vorgenommen werden. Damit ergeben sich folgende Eigenschaften mit der Wahl von q:

$q \leq 1$: Der zweistufige Graph $\mathcal{G}_{G3}$ wird mit $q \leq 1$ identisch mit der 1. kanonischen Form $\mathcal{G}_G$.

$q = \log_2(P)$: Die Wahl von $q = \log_2(P)$ ergibt wie in der 1. kanonischen Form $\log_2(P)$ Kommunikationsschichten, nun jedoch mit einheitlicher Topologie. Bei Abbildung 3.5 entspricht dies der horizontalen Aufteilung des Graphen auf 8 Prozessoren. Bei dieser Aufteilung liegen jedoch korrespondierende Funktionen in den Berechnungsschichten 1 bis q auf unterschiedlichen Prozessoren. Auch ist die benötigte Kommunikationsleistung einer Schicht doppelt so hoch wie in der 1. Form.

$q = \log_2(P) + 1$: Mit einer zusätzlichen Kommunikationsschicht werden korrespondierende Funktionen einem Prozessor zugeordnet. Daher zeigen beide wegführenden Ergebniskanten einer Ecke $f_{i''}^{(m-1)}$ zum gleichen Subgraphen und können bei einer Implementierung durch eine einzelne Kommunikation mit anschließendem doppelten Datenzugriff des Zielprozessors realisiert werden.

$q = n$: Mit $q = n$ wird der zweistufige Graph $\mathcal{G}_{G3}$ identisch mit der 2. kanonischen Form $\mathcal{G}_{G2}$.

Somit umfaßt die dritte vorgestellte Form eines Graphen $\mathcal{G}_{G3}$ zur Beschreibung globaler Transformationen die beiden kanonischen Darstellungen und kann durch einen wählbaren Parameter q bei einer Implementierung an Randbedingungen angepaßt werden.

Kapitel 4

Parallele Programmformulierung

Die parallele Programmformulierung umfaßt den Schritt zwischen der Extraktion der Parallelität eines Algorithmus und dessen Ausführung auf einer parallelen Hardware. Damit beim Übergang vom parallelen Algorithmus zur parallelen Ausführung kein Effizienzverlust auftritt, muß die Programmformulierung die extrahierte Parallelität in geeigneter Weise berücksichtigen. Dies kann auf unterschiedlichen Ebenen geschehen. Das Spektrum reicht von der Betriebssystemebene bis hinunter zur Wortbreite der Verarbeitungseinheiten.

Die traditionelle sequentielle Programmformulierung beschäftigte sich aus der Sicht der Datenflußgraphen damit, eine serielle und kausale Reihenfolge der Ecken zu finden. Daß dabei mögliche inhärente Parallelitäten nicht berücksichtigt werden können, liegt auf der Hand.

Mit der parallelen Programmformulierung sollen die extrahierten Parallelitäten erhalten bleiben, somit reichen die Konstrukte herkömmlicher serieller Sprachen nicht mehr aus. Vielmehr müssen parallele Sprachen Möglichkeiten bieten, Nebenläufigkeit, Kommunikation und Nichtdeterminismus auszudrücken.

4.1 Ebenen der Parallelität

Die extrahierte Parallelität eines Algorithmus läßt sich auf verschiedene Ebenen eines Systems abbilden. In [HJ88] werden vier solcher Parallelisierungsebenen unterschieden. Dies sind in hierarchischer Reihenfolge:

Job-Ebene: Parallele Ausführung eigenständiger Job-Phasen.

Programmebene: Parallele Ausführung von Teilprogrammen eines zerlegten Gesamtprogramms.

Instruktionsebene: Parallele Ausführung verschiedener Phasen der Instruktionsverarbeitung.

Arithmetikebene: Bit-Parallelität in den Verarbeitungseinheiten eines Prozessors.

Die hierarchisch höchste Ebene der *Job*-Ausführung bedingt Anforderungen an Betriebssysteme für Parallelrechner. Beim Start eines Jobs kann beispielsweise jeweils ein geeigneter, wenig belasteter Prozessor vom Betriebssystem ermittelt und dort die Ausführung der anstehenden Aufgabe gestartet werden. Eine gute Aufgabenverteilung auf dieser Ebene durch das Betriebssystem bewirkt eine gleichmäßige Rechnerauslastung und damit eine effiziente Ausnutzung eines Parallelsystems, besonders beim Betrieb mit mehreren Benutzern. Einige solcher Betriebssysteme zur Verwaltung von Parallelrechnern sind in ersten Versionen verfügbar. Ein verbreitetes Beispiel für Transputersysteme ist Helios [Ltd89], welches dem Anwender eine UNIX-ähnliche Oberfläche zur Verfügung stellt.

Der Programmzugriff auf diese Parallelitätsebene geschieht meist über Laufzeitbibliotheken, welche dem Programmierer das Aufsetzen nebenläufiger Jobs einschließlich ihrer Kommunikation erlauben.

Bei paralleler Ausführung mehrerer Job-Phasen mit hohem Kommunikationsbedarf zwischen den Phasen nimmt die Systemleistung aufgrund der Betriebssystemverwaltung sehr schnell ab. Dies beschränkt eine sinnvolle parallele Programmierung in dieser Ebene auf Probleme mit grober Granularität und wenig Kommunikationsbedarf.

Die darunterliegende *Programmebene* umfaßt die Beschreibung von Parallelität in der Programmformulierung. Auf dieser Ebene ist das Aufsetzen nebenläufiger Programmteile sowie deren Kommunikation und Synchronisation in die Programmiersprache eingebettet. Die Formulierung erfolgt wesentlich direkter als in der Job-Ebene. Zur Unterstützung findet man spezielle Bibliotheken als Ergänzung herkömmlicher Programmiersprachen, die Erweiterung standardisierter Sprachen um neue Anweisungen sowie die Entwicklung neuer, paralleler Sprachen.

Die Programmebene soll im folgenden Abschnitt genauer betrachtet werden.

Die *Instruktionsebene* der Parallelität beinhaltet die parallele Ausführung einzelner Instruktionen. Moderene Prozessoren besitzen mehrere Verarbeitungseinheiten (Adreßrechenwerk, Ganzzahl-ALU, Fließkomma-ALU, usw.) welche alle gleichzeitig arbeiten können. Die Nebenläufigkeit dieser Einheiten in Verbindung mit 'Pipelining' bei der Befehlsdekodierung ermöglicht hohe Rechenleistungen. Will man diese jedoch effizient ausnutzen, muß meist aufwendig in Maschinensprache programmiert werden. Die Entwicklung von 'Compilern' zur Ausnutzung paralleler Verarbeitungseinheiten ist ein aktuelles Forschungsgebiet. Häufig ist der Unterschied zwischen maximaler und mittlerer Leistung bei der Ausführung typischer Testprogramme und Benutzung herkömmlicher 'Compiler' noch sehr groß.

Die Parallelität auf der niedrigsten *Arithmetikebene* war beim CPU-Entwurf bis Mitte der 70er Jahre noch Bestandteil der Entwicklung. 'Bit-Slice'-Arithmetikeinheiten erlaubten die Anpassung der Verarbeitungswortbreite an Erfordernisse der Rechenleistung. Durch die Integration von Prozessoren mit bis zu 64 *Bit* Wortbreite hat diese scheibenweise Anpassung heute kaum mehr Bedeutung.

Beachtung findet diese Ebene noch beim Entwurf von Systemen mit sehr hohen Prozessoranzahlen, wo viele Prozessoren in einem Baustein untergebracht werden sollen und damit

die Verarbeitungsbreite in Konkurrenz zur Prozessoranzahl steht.

Bedeutung gewinnt diese Ebene wieder beim Entwurf kundenspezifischer integrierter Schaltungen. Parallele Verarbeitungseinheiten bewirken einerseits hohe Verarbeitungsleistungen, andererseits aber auch hohe Kosten. Durch Einstellen der Wortbreite ist eine Abstimmung dieser Parameter mit aktuellen Erfordernissen möglich.

4.2 Parallele Programmierung

Die parallele Programmierung umfaßt im wesentlichen die ersten beiden Ebenen der Parallelität. Im folgenden sollen daher Programmiersprachen innerhalb dieser Ebenen bezüglich ihrer Ausdrucksmöglichkeiten für Parallelität betrachtet werden.

Die Möglichkeit, Parallelität implizit auszudrücken, wie es funktionale Sprachen in einer sehr mächtigen Weise erlauben [Veg84], soll an dieser Stelle nur im Hinblick auf Datenflußsprachen ausgeführt werden.

Die explizite Formulierung von Parallelität wird für herkömmliche sequentielle Sprachen und für parallele Sprachen diskutiert.

4.2.1 Datenflußsprachen

Datenflußsprachen erlauben die direkte Übertragung von Datenflußgraphen in eine Programmformulierung. Diese Sprachen verlassen den traditionellen sequentiellen Kontrollfluß vollständig.

Dennis gibt in [Den79] eine Übersicht über die Grundlagen der Datenflußprogrammierung und den Einzel- und Multiprozessorentwurf nach dem Datenflußprinzip. Bei Datenflußsprachen wird jede Ecke eines Datenflußgraphen auf ein *'activity template'* abgebildet. Dies ist eine Datenstruktur, welche aus einem Operatorfeld, Operandenfeldern sowie aus Zeigerfeldern besteht, wobei die Zeigerfelder auf Operandenfelder nachfolgender 'Templates' zeigen, welche das Ergebnis der Operation übernehmen.

Eine Operation im 'Template' wird durch das Eintreffen notwendiger Eingangsdaten aktiviert und sie aktiviert weiterhin über das eigene Ergebnis Operationen in Folge-'Templates'.

Abbildung 4.1 zeigt einen von Dennis angeführten einfachen Beispielgraphen zur Berechnung von $z = (x+y)\cdot(x-y)$ und die zugehörige Datenflußformulierung durch verzeigerte 'activity templates'.

Aufbauend auf dieser mit der Assembler-Ebene vergleichbaren Programmiermöglichkeit über 'Templates' existieren Datenflußsprachen höherer Ebene. Beispiele sind die Sprachen SISAL [GKB87], 'Id' [HH89] und die speziell für Bildverarbeitungsanforderungen entworfene Sprache 'IPL' [Gun90].

Der Vorteil von Datenflußsprachen ist der Erhalt inhärenter Parallelität von Datenflußgraphen. Weiterhin wird die Ausführung eines Graphen über die Verfügbarkeit der Daten getriggert, wodurch sich automatisch eine Synchronisierung paralleler Teilprogramme ergibt.

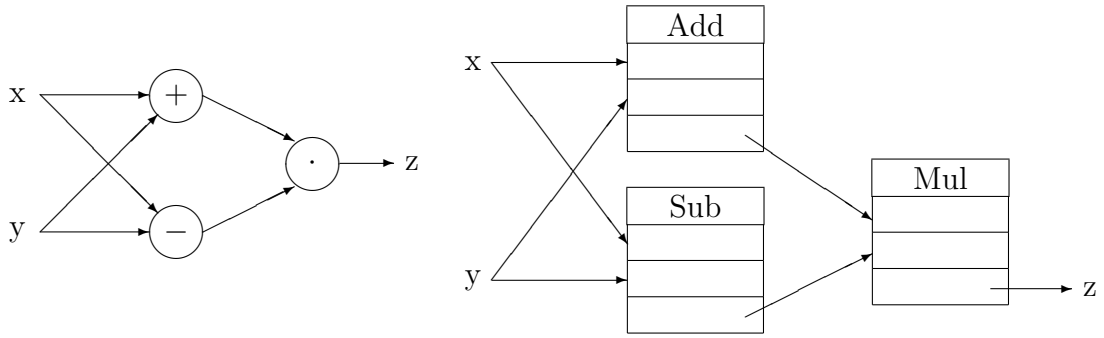

Abbildung 4.1: Beispiel eines einfachen Datenflußgraphen und dessen Formulierung durch 'Templates' nach [Den79].

Die Realisierung bedingter Ausführungen (IF-THEN-ELSE) und sequentieller Schleifen bereitet hingegen Probleme. Die bedingte Ausführung führt zu 'Templates' in der Graphenbeschreibung, welche niemals aktiviert werden, und sequentielle Schleifen bewirken Zyklen in der Datenflußformulierung.

4.2.2 Parallelität in sequentiellen Sprachen

Die Erweiterung herkömmlicher sequentieller Sprachen zur parallelen Programmausführung erfolgt durch geeignete Unterprogrammaufrufe. Diese Unterprogramme rufen Teile eines Betriebssystemkerns oder einer Laufzeitumgebung auf, welche den parallelen Programmablauf überwachen. Die Parallelität in sequentiellen Sprachen ist somit kein Sprachbestandteil, sondern basiert auf einer Betriebsumgebung, in der das Programm ausgeführt wird, und somit auf der obersten Job-Ebene.

Ein Beispiel zur Formulierung von Nebenläufigkeit ist der *fork*-Aufruf, der z.B. in UNIX-Bibliotheken zur Verfügung gestellt wird [BR84]. Dieser Aufruf bewirkt die nebenläufige Ausführung zweier gleichartiger Programminstanzen als Eltern- und Kindprozeß, wobei jeder Prozeß feststellen kann, ob er die Eltern- oder Kindinstanz repräsentiert.

Die Kommunikation nebenläufiger Prozesse erfolgt entweder über einen gemeinsamen Speicherbereich oder über 'Message-Passing', d.h. das Versenden von Nachrichten zwischen nebenläufigen Prozessen.

Beim gemeinsamen Speicher muß der Zugriff mehrerer Prozesse auf gleiche Datenstrukturen durch geeignete Kontrollmechanismen synchronisiert werden. Bekannteste Mechanismen dazu sind die von Dijkstra eingeführte Semaphore oder der darauf aufbauende Monitor [Hoa74], welcher Datenstrukturen gemeinsam mit Routinen kapselt, wobei der Zugriff auf die Datenstrukturen nur über diese Monitorroutinen möglich ist.

Zur zweiten Möglichkeit, dem 'Message-Passing', gibt es unterschiedliche Kommunikationsmodelle [Din89]. Sie unterscheiden sich beispielsweise darin, ob ein Sender auf den Empfang einer Nachricht durch den Zielprozeß wartet, ob ein simultanes Senden an viele Zielprozesse oder ob ein gegenseitiger Austausch von Daten möglich ist. Letztere Möglichkeit wird als 'Rendezvous' bezeichnet. Neuere Kommunikationsmechanismen auf Task-Ebene versuchen beide Modelle zu erweitern und zu verbinden. So kann

z.B. das Konzept 'Linda' [Lel90] einen gemeinsamen Speicher und ein 'Message-Passing' unterstützen.

Ein Beispiel für Parallelität in sequentiellen Sprachen findet man in der Beschreibung von Modula-2 [Wir85]. Dort schlägt Wirth die Einführung von Nebenläufigkeit auf Taskebene genau über Prozeduraufrufe vor und beschreibt die Realisierung eines Monitormoduls zur sicheren Kommunikation nebenläufiger Tasks.

4.2.3 Parallele Sprachen

Eine tiefergehende Unterstützung zum Ausdrücken von Nebenläufigkeit bieten Sprachen, welche neben den traditionellen sequentiellen Anweisungen weitere Anweisungen zur parallelen Programmformulierung umfassen. Unterschiedliche Sprachen, welche solche Erweiterungen beinhalten, sind verfügbar. Das Spektrum reicht von der Erweiterung sequentieller Sprachen bis hin zu Sprachen, welche direkt für die Belange der parallelen Programmierung entwickelt wurden.

Im folgenden sollen die Möglichkeiten der parallelen Formulierung in den Sprachen ADA, CSP, Occam und Par.C kurz diskutiert werden.

ADA:
In der Sprache ADA [SS89b] erfolgt die Formulierung von Parallelität durch die Deklaration von Tasks innerhalb abgeschlossener Einheiten (z.B. Prozeduren oder Funktionen). Der Start der Tasks erfolgt implizit vor dem Start der Einheit, in deren Geltungsbereich die Tasks deklariert wurden. Die Anweisungsteile von Tasks und anderer Programmeinheiten können nur rein sequentiell formuliert werden. Eine feine Parallelität auf der Instruktionsebene ist in ADA nicht vorgesehen.

Die Kommunikation nebenläufiger Tasks ist in ADA über '`entry`'-Punkte organisiert. An diesen Punkten hält die Ausführung einer Task an und eine zweite Task kann mit der ersten kommunizieren. Die Kommunikation erfolgt nach dem 'Rendezvous'-Prinzip, d.h. erreicht eine Task einen Kommunikationspunkt, so wartet diese bis auch die zweite Task zur Kommunikation bereit ist. Dann findet ein Datenaustausch statt. Anschließend setzen beide Tasks unabhängig voneinander die Ausführung fort.

Zur nichtdeterministischen Programmausführung besitzt ADA die '`select`'-Anweisung, mit der ein Prozeß auf alternative Kommunikationen mit mehreren Tasks warten kann. Ist eine Task bereit, wird eine Kommunikation mit dieser Task durchgeführt und die Anweisung terminiert.

Die Möglichkeiten der parallelen Programmformulierung in ADA sind noch stark an die Job-Ebene angelehnt. Damit ist eine parallele Formulierung grob granularer Probleme gut unterstützt. Die Formulierung einer bis ins Detail reichenden inhärenten Parallelität eines Algorithmus ist aber nur bedingt möglich.

Communicating Sequential Processes (CSP):
Weitergehende Möglichkeiten zur parallelen Programmformulierung beschreibt Hoare [Hoa85] durch die Sprache CSP. Diese sehr formale Sprachdefinition erlaubt die Formulierung elementarer, sequentieller Prozesse, welche über Kanäle mit nebenläufigen

Prozessen Meldungen austauschen können. Weiterhin können durch Komposition elementarer Prozesse komplexe Prozesse hierarchisch generiert werden. Hoare definiert unterschiedliche Operatoren zur Komposition von Prozessen und schafft damit die Möglichkeit, nebenläufige, nichtdeterministische und sequentielle Anordnungen kommunizierender Prozesse zu beschreiben. Da die Prozessdefinition von Hoare auf ganz elementarer Ebene basiert — eine einzele Programmanweisung stellt z.B. schon einen Prozeß dar — kann mit CSP eine sehr feine Granularität beschrieben und damit inhärente algorithmische Parallelität bis ins Detail ausgedrückt werden.

OCCAM:
Basierend auf CSP wurde die Sprache Occam [PM87] entwickelt. Im Gegensatz zu CSP ist das Erscheinungsbild von Occam enger an die Syntax herkömmlicher Sprachen angelehnt.

Im Vergleich zu sequentiellen Sprachen, wie z.B. Pascal, C, Modual-2 oder auch der Sprache ADA, ist Occam eine einfache, teilweise noch in der Evolution befindliche Sprache. Besonders bezüglich Datenstrukturen und dynamischer Datenobjekte werden dem Programmierer kaum Unterstützungen gegeben.

In seinen Anweisungen folgt Occam weitgehend dem in CSP eingeführten Prozeßkonzept. Zur parallelen Anordnung zweier oder mehrerer Prozesse existiert eine `PAR`-Anweisung, eine sequentielle Prozessanordnung wird durch eine `SEQ`-Anweisung eingeleitet.

Eine Kommunikation findet in Occam über Kanäle statt. Jede Kommunikation muß eindeutig sein: Ein Kanal verbindet immer einen Sender mit nur einem Empfänger. Eine Kommunikation wird durch einen Kanalnamen, den Sende- '`!`' oder Empfangsoperator '`?`' und durch Sende- oder Empfangsvariablen beschrieben. Das Protokoll muß für jeden Kanal eindeutig definiert werden, hier bietet Occam Strukturierungsmöglichkeiten, die denen von strukturierten Datentypen in sequentiellen Sprachen entsprechen. Sehr einheitlich erfolgt in Occam die Inter- und Intraprozessorkommunikation. Der Unterschied besteht lediglich in der Deklaration der Kanäle, die zugehörigen Anweisungen sind identisch.

Das in Abbildung 4.1 angeführte Beispiel läßt sich unter Erhalt feiner Parallelität durch die folgenden beiden Occam Fragmente formulieren, der Gültigkeitsbereich einer Anweisung ist durch Einrücken gekennzeichnet:

```
SEQ                          PAR
  PAR                          tmp1.CHAN ! (x+y)
    tmp1:=x+y                  tmp2.CHAN ! (x-y)
    tmp2:=x-y                  SEQ
  z:=tmp1*tmp2                   PAR
                                   tmp1.CHAN ? tmp1
                                   tmp2.CHAN ? tmp2
                                 z:=tmp1*tmp2
```

Der links aufgeführte Programmtext beschreibt die Berechnug von `z` in einem einzelnen Prozeß, wobei die Ausführung von `x+y` und `x-y` durch eine `PAR`-Anweisung als nebenläufig durchführbar gekennzeichnet ist. Der rechte Programmtext beschreibt — in direkter

Korrespondenz zum Datenflußgraph von Abbildung 4.1 — die Berechnung durch drei kommunizierende Prozesse. Diese sind durch die obere `PAR`-Anweisung alle nebenläufig aufgesetzt. Trotz des parallelen Aufsetzens wird der dritte, durch das `SEQ`-Statement eingeleitete Prozeß zur Berechnung von `z` erst in Sequenz nach den ersten beiden Prozessen ausgeführt. Diese sequentielle Ausführung resultiert aus den Datenabhängigkeiten dieses Beispiels und zeigt, daß eine Programmausführung in parallelen Sprachen nicht allein durch den Kontrollfluß sondern auch — in Analogie zu den Datenflußsprachen — durch die Verfügbarkeit von Operanden gesteuert wird.

Die Behandlung alternativer Kommunikationen und damit von Nichtdeterminismen im Programmablauf wird in OCCAM durch eine `ALT`-Anweisung ermöglicht. Diese Anweisung erlaubt, ähnlich wie die `select`-Anweisung in ADA, das alternative Warten auf die Kommunikationsbereitschaft mehrerer Eingangskanäle. Sobald ein Kanal bereit wird, erfolgt eine Kommunikation über diesen Kanal und anschließend die Ausführung eines der Alternative zugeordneten Prozesses.

OCCAM ist nach der vorliegenden Definition eine Sprache mit statischem Speichermodell. Dadurch ist weder die Vereinbarung dynamischer Datenobjekte noch ein rekursiver Funktions- oder Prozeduraufruf möglich. Weiterhin ist auch das Aufsetzen paralleler Prozesse durch `PAR`-Anweisungen nur statisch zur Übersetzungszeit möglich. Diese Einschränkungen erschweren die Formulierung parametrisierbarer Programme, welche sich zur Laufzeit dynamisch an Problemgrößen anpassen lassen.

Par.C:
Mit `Par.C` [Par90] wird der Sprachumfang von `C` einschließlich einer Erweiterung zur parallelen Programmformulierung zur Verfügung gestellt. Diese Erweiterung beinhaltet Anweisungen zur nebenläufigen Ausführung von Prozessen, zur Kommunikation paralleler Prozesse und zur Behandlung von Nichtdeterminismen. Diese Anweisungen sind den beschriebenen OCCAM-Anweisungen sehr ähnlich. Somit gelten die dortigen Möglichkeiten auch für diese Sprache. Ein Vorteil von `Par.C` gegenüber OCCAM besteht in dem wesentlich größeren Sprachumfang von `C` und durch die Verfügbarkeit strukturierter Datentypen. Weiterhin erlaubt das dynamische Speichermodell von `C` die Formulierung dynamischer Parallelität und unterstützt das Vereinbaren dynamischer Datenobjekte.

Kapitel 5

Parallele Programmausführung

Nach der Parallelisierung eines Algorithmus und seiner Formulierung als paralleles Programm schließt sich als dritter Schritt der parallelen Implementierung die Ausführung des Programms auf einer Hardware an. Diese sollte die Mechanismen der parallelen Programmformulierung in geeigneter Weise unterstützen.

Eine Zielhardware wird im einfachsten Fall ein Einzelprozessorsystem sein, höhere Leistungsanforderungen werden Multiprozessorsysteme bedingen. Kennzeichen eines Zielsystems sind die Operationsprinzipien (d.h. die Verwaltung der Operationen und Operanden bei der Algorithmenausführung) seiner Einzelprozessoren sowie die Topologie und Bandbreiten der Verbindungsnetzwerke, welche die Einzelprozessoren zu einem parallelen System verbinden. Die Auswahl der Einzelprozessoren und geeigneter Verbindungsnetzwerke wird beeinflußt durch Nebenbedingungen wie z.B. Klassen zu realisierender Algorithmen, angestrebte Leistungsdaten, Bedingungen an die Modularität eines Systems und unterstützte Programmiersprachen.

5.1 Operationsprinzipien von Einzelprozessoren

Die Leistungsfähigkeit eines Einzelprozessors wird beeinflußt durch seine technologische Realisierung und die Breite seiner Verarbeitungseinheiten. Entscheidend bestimmt jedoch das zugrundeliegende Operationsprinzip, wie effizient eine Programmformulierung auf einen Prozessor abgebildet werden kann und wie leistungsfähig somit deren Ausführung sein wird. Eine hohe Korrespondenz zwischen der Programmformulierung und dem Operationsprinzip, welches die innere Architektur eines Prozessors bestimmt, ist Grundlage für eine hohe Verarbeitungsleistung.

Treleaven [Tre84] verweist auf zwei grundlegende Mechanismen zur Organisation einer Programmausführung: den Kontroll- und den Datenmechanismus. Ein Prozessor stellt durch sein Operationsprinzip eine Kombination spezieller Realisierungen beider Mechanismen zur Verfügung. Zur Ausführung muß eine Programmformulierung auf die verfügbaren Mechanismen eines Prozessors abgebildet werden. Die Abbildung übernehmen meist für einen Prozessortyp entwickelte Übersetzerprogramme ('Compiler'). Eine hohe Korrespondenz zwischen der Programmformulierung und den verfügbaren Mechanismen ist für die

Ausführung nicht zwingend, bewirkt jedoch eine einfache Übersetzung und die Generierung effizienter Instruktionen für den Zielprozessor.

Kontroll- und Datenmechanismen
Zur Erläuterung unterschiedlicher Operationsprinzipien werden verschiedene Möglichkeiten der Kontroll- und Datenmechanismen in Anlehnung an [Tre84] angeführt:

Der *Kontrollmechanismus* beschreibt, wie eine Instruktion die Ausführung einer Folgeinstruktion anstößt. Treleaven führt dazu drei Möglichkeiten an:

Sequentieller Mechanismus: Instruktionen werden in einer zur Übersetzungszeit determinierten Reihenfolge ausgeführt.

Paralleler Mechanismus: Eine Instruktion wird durch Verfügbarkeit der benötigten Operanden aktiv und kann zur Ausführung kommen. Sind mehrere Instruktionen aktiv, werden diese auf einem Einzelprozessor sequentiell in beliebiger Reihenfolge ausgeführt.

Rekursiver Mechanismus: Der Bedarf an Operanden zur Ausführung einer Instruktion bewirkt die Aktivierung weiterer Instruktionen, welche genau die benötigten Operanden als Ergebnis liefern. Nach Ermittlung aller Operanden erfolgt die Ausführung der zugehörigen Instruktion.

Der *Datenmechanismus* beschreibt den Zugriff der Instruktionen auf die Daten bei einer Programmausführung. Dazu führt Treleaven drei Möglichkeiten an, welche sich am Zugriff auf Einzeldaten orientieren. Die vierte Möglichkeit zur Betrachtung höherdimensionaler, strukturierter Datenobjekte wurde hinzugefügt:

Direkter Zugriff ('by literal'): Die Kopie eines Werts, welcher zur Übersetzungszeit bekannt sein muß, wird in jeder Instruktion gespeichert, die auf den Wert zugreift.

Wert-Zugriff: Für einen zur Laufzeit generierten Datenwert werden für alle Instruktionen, welche auf diesen Wert zugreifen müssen, Kopien erzeugt und zusammen mit diesen Instruktionen gespeichert.

Referenzierter Zugriff: Ein zur Laufzeit generierter Datenwert wird in einer Speicherzelle abgelegt. Instruktionen, welche auf diesen Wert zugreifen müssen, enthalten eine Referenz auf diese Speicherzelle.

Sequenzgesteuerter Zugriff: Einer Instruktion wird die Referenz auf eine Datensequenz zugeordnet, welche Werte in einer von der Verarbeitung abhängigen Reihenfolge enthält. Der Zugriff der Instruktion ist nur auf die vollständige Sequenz möglich.

Die Kombination einzelner Daten- und Kontrollmechanismen ergibt eine grobe Untergliederung in drei Operationsprizipien: Kontrollfluß, Datenfluß und Reduktion. Das Prinzip der Reduktion, welches den rekursiven Kontrollmechanismus mit dem Wert- oder Referenzzugriff verbindet, führt zu Maschinen, welche vielfach bei der Ausführung funktionaler Sprachen Anwendung finden. Eine Übersicht findet man in [Veg84]. Dieses Prinzip soll jedoch in dieser Arbeit nicht weiter vertieft werden.

Im folgenden sollen insbesondere Architekturen zur Realisierung von Prinzipien betrachtet werden, welche auf dem Kontroll- und Datenfluß basieren.

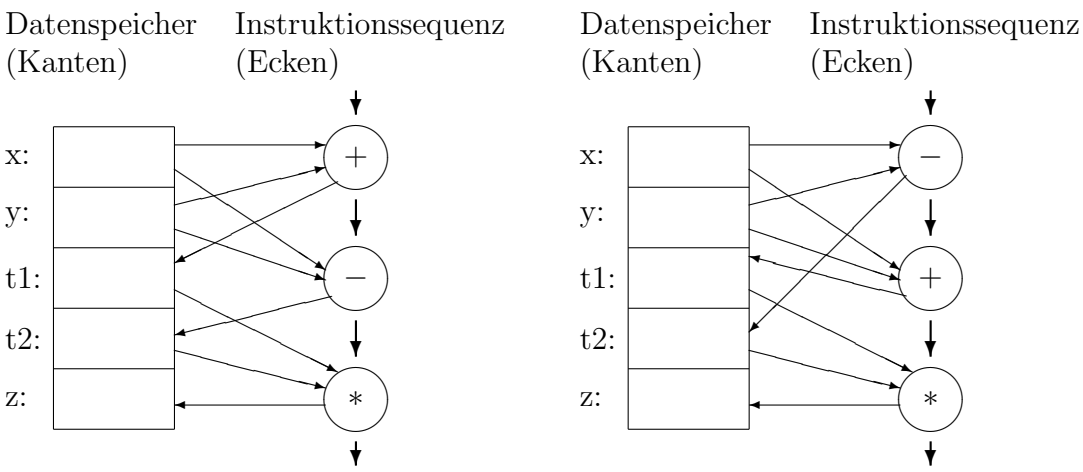

Abbildung 5.1: Zwei Möglichkeiten zur sequentiellen Ausführung des Graphen aus Abbildung 4.1 nach dem Kontrollflußprinzip.

Kontrollfluß

Die traditionelle Programmausführung nach dem Prinzip des Kontrollflusses verbindet die sequentielle Ausführung der Instruktionen mit einem referenzierten Datenzugriff. Die Ausführung wird durch eine teilweise willkürlich gewählte, sequentielle Instruktionsreihenfolge bestimmt, welche als Linearisierung des Datenflußgraphen unter Beachtung der Kausalitätsbedingung interpretiert werden kann. Abbildung 5.1 zeigt beispielhaft die Ausführung des sehr einfachen Graphen aus Abbildung 4.1 nach dem Kontrollflußprinzip, für den sich schon zwei mögliche Reihenfolgen ergeben.

Realisiert wird dieses Verarbeitungsprinzip durch 'von Neumann'-Typ Rechner und davon abgeleiteten Architekturen wie beispielsweise der bei Signalprozessoren gebräuchlichen Harvard-Architektur. Kennzeichen dieser Rechner ist ein zentrales Instruktionswerk, welches die sequentielle Instruktionsausführung steuert und notwendige Transfers von Daten zwischen Speicher und Prozessor vornimmt. Limitierender Engpaß dieser Rechner ist der bekannte 'Bottleneck' des Speicherzugriffs über den Bus des Rechners. Vielfältige Möglichkeiten, diesen Engpaß aufzuweiten wurden untersucht und realisiert. Beispiele sind unter vielen anderen: 'Cache'-Speicher oder getrennte Instruktions- und Programmspeicher.

Zunächst führte die Entwicklung der Kontrollflußrechner zu Prozessoren mit immer komplexerem Instruktionssatz (CISC), wobei einzelne Instruktionen durch Mikroprogramme realisiert werden. Die Entwicklung moderner, schneller Speicher, welche die Zugriffzeit auf extern gespeicherte Instruktionen stark verkürzen, und die Analyse der von 'Compilern' benötigten Maschineninstruktionen zur Abbildung von Hochsprachen führte zur Entwicklung von Prozessoren mit wenigen, schnellen, festverdrahteten Instruktionen (RISC). Eine Übersicht zur Entwicklung von CISC- zu RISC-Prozessoren findet man beispielsweise in [Pat85].

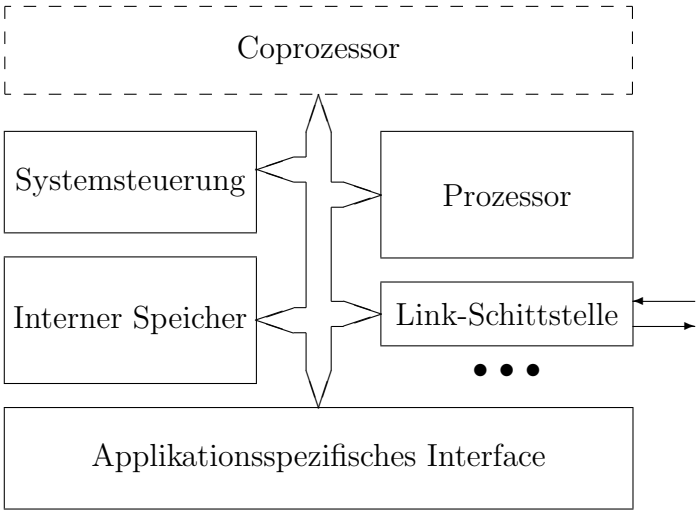

Abbildung 5.2: Blockstruktur eines Prozessors aus der Transputer-Familie.

Kontrollfluß mit Nebenläufigkeit und Kommunikation

Traditionelle Kontrollflußrechner sind das Hardware-Pendant der sequentiellen Programmiersprachen. Der Übergang zur parallelen Programmformulierung (Abschnitt 4) spiegelt sich daher in Veränderungen der Architektur und im Instruktionssatz von Kontrollflußrechnern wider.

Die erste Erweiterung sequentieller Sprachen zur Beschreibung von Parallelität durch Unterprogramme (Abschnitt 4.2.2) wurde noch durch eine — meist in die Betriebssoftware eingebettete — 'Scheduler'-Software zur Verfügung gestellt. Diese Software realisiert einen (quasi-)parallelen Kontrollmechanismus für sequentielle Instruktionssegmente. Ein Kontextwechsel, d.h. ein Wechsel von der Ausführung eines Programmsegmentes zu einem zweiten Segment, dauert etwa so lange wie die Ausführung von 10^2-10^4 Instruktionen. Bei einer Parallelität auf Task-Ebene kann dies noch akzeptiert werden, bei einer Unterstützung feiner Parallelität in der Programmformulierung führen solche Zeiten jedoch zu einer nicht akzeptablen Degradierung der Ausführungszeiten.

Die Zulassung einer parallelen Programmausführung bedarf schon auf einem Einzelprozessor der Kommunikation nebenläufiger Programmteile. Erfolgt die Kommunikation direkt über den gemeinsamen Speicher, kann sie mit dem Datenmechanismus des referenzierten Zugriffs realisiert werden, wobei zur Sicherung der Datenintegrität Methoden zur Synchronisation des Zugriffs (Semaphoren, Monitore, etc.) Anwendung finden müssen. Eine Kommunikation über 'Message-Passing' vermeidet durch das Versenden von Nachrichten die Probleme der Datenintegrität und kann durch den Datenmechanismus des Wertzugriffs realisiert werden. Ebenso wie bei der parallelen Programmausführung entsteht bei der Kommunikation über Software-Schnittstellen ein hoher Verwaltungsaufwand, der nur bei relativ seltenem Datenaustausch nebenläufiger Tasks akzeptiert werden kann.

Die Familie der Transputer [INM88b] unterstützt die Ausführung paralleler Sprachen durch Architektur und Instruktionssatz. Die enge Anlehnung dieser Prozessoren an parallele Sprachen wurde durch die direkte Verwendung von OCCAM als Formalismus beim

Design der Transputer [MS87] erreicht. Auf dieser Basis entstanden verschiedene Prozessorrealisierungen des gleichen Konzepts, welche sich in den verfügbaren Verarbeitungseinheiten, in deren Wortbreite, in der verfügbaren Rechenleistung und in verschiedenen Schnittstellen nach außen unterscheiden [INM88b].

Alle Transputer unterstützen die nebenläufige Programmausführung und die Kommunikation nebenläufiger Prozesse durch Verarbeitungseinheiten und den Instruktionssatz. Damit wird die effiziente Ausführung paralleler Programmformulierungen mit feiner Granularität ermöglicht. Weiterhin erscheint dem Benutzer diese Unterstützung — zumindest bei der Verwendung von Occam — sowohl beim Programmieren eines einzelnen Transputers als auch großer Transputernetze nahezu einheitlich.

Der Start nebenläufiger Prozesse auf einem einzelnen Transputer wird durch spezielle Instruktionen realisiert. Diese bilden mit ihren Mikroprogrammen einen 'Scheduler' [INM87], womit die Notwendigkeit für einen Software-'Scheduler' entfällt. Ein Kontextwechsel zwischen nebenläufigen Programmteilen dauert nur ca. $1\mu s$ [INM88b] und entspricht der Zeit zur Ausführung von etwa 10 Instruktionen.[1] Ein Kontextwechsel ist somit sehr effizient möglich und um Größenordnungen schneller als bei Software-Lösungen. Dies schafft die Grundlage zur effizienten Ausführung paralleler Programme feiner Granularität.

Weiterhin unterstützen Transputer durch spezielle Instruktionen die Kommunikation nebenläufiger Prozesse nach dem in CSP und Occam verwendeten Modell (Abschnitt 4.2.3). Die Implementierung der Kommunikation ist verkoppelt mit dem 'Scheduling' nebenläufiger Prozesse und bewirkt, daß ein erster kommunikationsbereiter Prozeß solange suspendiert wird, bis ein zugehöriger zweiter Prozeß die Kommunikation, und dabei auch den ersten Prozeß, wieder aktiviert. Durch das Suspendieren jedes ersten kommunikationsbereiten Prozesses wird ein Transputer durch deren Wartezeit nicht belastet.

Eine weitere Eigenschaft der Kommunikationsimplementierung auf Transputern ist die gleiche Behandlung von Kanälen zwischen nebenläufigen Prozessen auf dem selben Prozessor und zwischen Prozessen auf unterschiedlichen Prozessoren durch die Software. Der einzige Unterschied besteht in der Adresse der Speicherzelle des Transputers, welche zur Implementierung eines Kanals verwendet wird.

Faßt man das aus dem Kontrollfluß mit Unterstützung von Nebenläufigkeit und Kommunikation entstehende Operationsprinzip eines Einzelprozessors zusammen, so beinhaltet es sowohl das traditionelle Prinzip des Kontrollflusses, ermöglicht aber auch — basierend auf der Software- oder Mikroprogrammebene — eine durch die Verfügbarkeit der Daten gesteuerte Verarbeitung. Diese datengesteuerte Verarbeitung wird als Datenfluß bezeichnet.

Datenfluß

Das Prinzip des Datenflusses kombiniert eine parallele Verfügbarkeit aller Instruktionen eines Programms und damit einen parallelen Kontrollmechanismus mit einem Datenmech-

[1] Die Ausführungszeit einer Instruktion hängt beim Transputer stark von deren jeweiliger Funktion ab, bei 20MHz-Typen wird jedoch üblicherweise eine Leistung von 10MIPS angenommen.

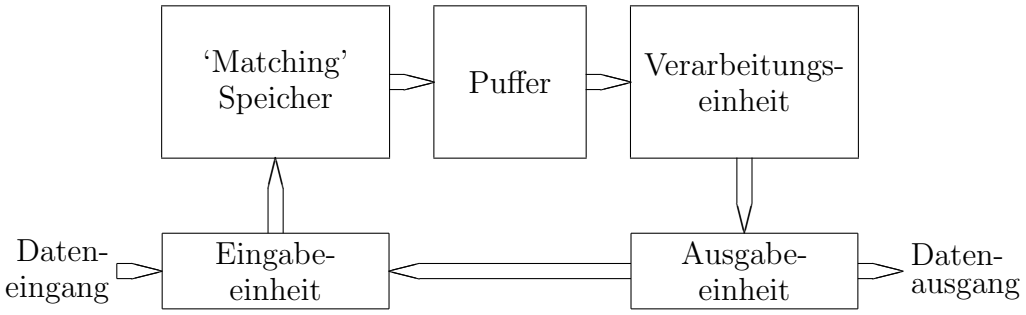

Abbildung 5.3: Blockstruktur eines Datenflußprozessors.

anismus des Wertzugriffs. Dieses Prinzip führte zur Entwicklung verschiedener Datenflußprozessoren, welche zur direkten Ausführung von Datenflußgraphen oder damit korrespondierender Datenflußsprachen (Abschnitt 4.2.1) entworfen wurden [Den79, Tre84, Sri86]. Neben Prototypen für Forschungszwecke ist dieser Prozessortyp auch auf einem Einzelchip zum Aufbau kommerzieller Datenflußsysteme erhältlich [NEC84].

Das Kennzeichen dieser Prozessoren ist eine ringförmige Anordnung von Einheiten zur Verarbeitung von 'activity templates' elementarer Datenflußsprachen. Abbildung 5.3 zeigt eine beispielhafte Anordnung, bestehend aus einem 'Matching'-Speicher, einer Verarbeitungseinheit mit vorgeschaltetem Puffer und Ein- und Ausgabeeinheiten.

Beim Start eines Programms liegen alle den Graphen beschreibenden 'Templates' in dem als 'Matching-Store' bezeichneten Speicher vor. Die 'Templates' enthalten beim Start noch keine gültigen Operanden, somit können die zugehörigen Operationen noch nicht ausgeführt werden. Der Speicher empfängt ankommende Daten, welche jeweils mit einem Zeiger versehen sein müssen, der auf ein zugehöriges 'Template' zeigt. Die Daten werden als Operanden im zugehörigen 'Template' abgespeichert. Dabei überprüft der Speicher, ob alle für die Ausführung der Operation notwendigen Operanden vorhanden sind. Ist dies der Fall, wird das komplette 'Template' dem Speicher entnommen und durch Weitergabe in den Puffer aktiviert.

Die Verarbeitungseinheit entnimmt aktive 'Templates' dem vorgeschalteten Puffer, führt die Instruktion aus und gibt das Ergebnis zusammen mit den im 'Template' gespeicherten Zieladressen der Folge-'Templates' aus.

Die zwischen Verarbeitungseinheit und Speicher angeordneten Ein- und Ausgabeeinheiten dienen als Schnittstelle des Prozessors nach außen. Die Eingabeeinheit empfängt extern ankommende Daten, um diese dem zum Speicher fließenden Datenstrom hinzuzufügen. Die Ausgabeeinheit überprüft die Zeiger der Ergebnisdaten aus der Verarbeitungseinheit und sendet diese — in Abhängigkeit von den Zeigern — zum Speicher oder zum Prozessorausgang.

Eine Programmausführung terminiert, wenn der Speicher keine 'Templates' mehr enthält und auch die Verarbeitungseinheit alle aktiven 'Templates' abgearbeitet hat.

Der Zugriff auf den schnellen 'Matching'-Speicher, welcher passende (engl.: matching) Operanden und Operationen zusammenführt, bedingt einen großen Aufwand bei der Realisierung von Datenflußprozessoren. Hier reicht ein einfacher Schreib- oder Lesezugriff nicht

aus, sondern bei jedem Schreibzugriff muß eine Überprüfung — und bei Vollständigkeit der Operanden ein Lesen — des angesprochenen 'Templates' vorgenommen werden.

Im Vergleich zur Graphenverarbeitung mit sequentiellen Kontrollflußrechnern, wo allein der Kontrollfluß die Ausführung steuerte, wird bei Datenflußprozessoren der Kontroll- und Datenfluß über die 'Templates' verkoppelt. Es besteht somit ein gleichgerichteter Daten- und Kontrollfluß. Vorteil dieser Kopplung ist eine Trennung zwischen dem Speicherzugriff und der Befehlsausführung. Ein 'Template' wird bei Vollständigkeit dem Speicher entnommen und kann dann unabhängig von einem Speicherzugriff verarbeitet werden. Die Anpassung der mittleren Speichergeschwindigkeit an die mittlere Ausführungszeit der Verarbeitungseinheit geschieht durch den zwischengeschalteten Puffer.

Die Kopplung des Kontroll- und Datenflusses bereitet Schwierigkeiten, wenn verschiedene Datensätze mit der gleichen Befehlssequenz bearbeitet werden müssen und man diese Symmetrien ausnutzen möchte. Ein wiederholtes Anwenden der gleichen Instruktionssequenz auf gleichartige Datensätze ist bei der Ausführung statischer Graphen nicht möglich, da 'Templates' nach der Ausführung durch die Ergebniswerte, welche die zugehörigen Folge-'Templates' aktivieren, ersetzt werden.

Zur Überwindung dieser Problematik wurden Datenflußsystemen zur dynamischen Ausführung von Datenflußgraphen entworfen [GKW85, HH89]. Durch Kopieren oder Markieren wird in diesen Systemen ein Reaktivieren benutzter 'Templates' ermöglicht, was die Implementierung dynamischer Kontrollstrukturen wie Schleifen oder Rekursion auch auf Datenflußprozessoren erlaubt.

Datensequenz

Bestimmt die Struktur der Daten bei wiederholter Ausführung gleichartiger Instruktionen eine Programmausführung, so ist es sinnvoll, die Verarbeitung an der Datenstruktur zu orientieren. Dies führt zur Datensequenz-gesteuerten Ausführung. Bei diesem Operationsprinzip spielt der Kontrollmechanismus eine untergeordnete Rolle: Entweder sind die Operationen für alle Datenwerte gleich oder sie ergeben sich implizit aus der Datenreihenfolge.

Bei der Verarbeitung von Vektoren oder höherdimensionalen Datenfeldern mit gleichartiger Operation auf jedem Datenelement können Datensequenzen durch Hardware als geordnete Datenströme generiert und einer Verarbeitungseinheit zugeführt werden. Im Gegensatz zu dem oben beschriebenen Datenflußprinzip findet keine Verkopplung von Daten und Instruktionen statt. Die Operationen sind, falls diese sich überhaupt bei sukzessiven Daten unterscheiden, implizit durch die Datenreihenfolge vorgegeben und werden durch eine einzelne, der Datensequenz zugeordnete Instruktion beschrieben.

Traditionell führte das Operationsprinzip der Datensequenz zur Entwicklung von *Vektoreinheiten* als Ergänzung sequentieller Kontrollflußprozessoren. Die Zuführung der Operanden vom Hauptpeicher zur Vektoreinheit ist dort optimiert, so daß ab einer bestimmten Vektorlänge, welche wunschgemäß sehr klein sein sollte, eine Vektoroperation mit der Vektoreinheit schneller durchgeführt werden kann als mit dem skalaren Prozessor.

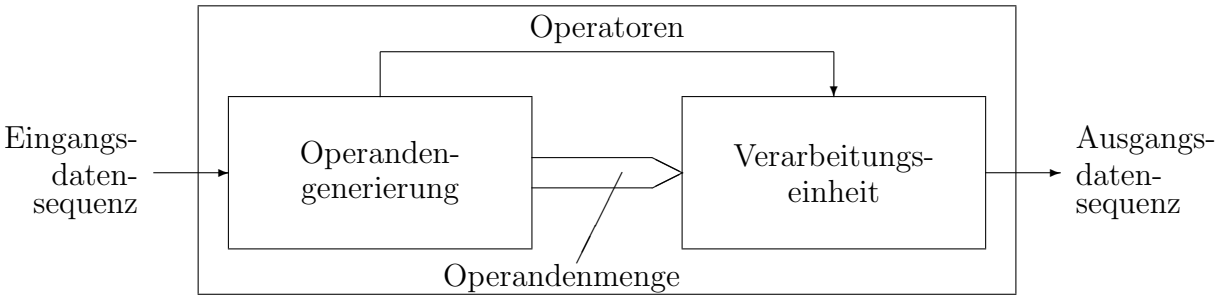

Abbildung 5.4: Blockstruktur eines Pipelineprozessors.

Diese in der Literatur als 'vector breakeven length' bezeichnete Länge liegt bei heutigen Rechnern im Bereich von 1.5 bis ca. 10 Elementen [HJ88].

Zur direkten Ausführung geordneter Datenströme sind *Pipelineprozessoren* geeignet, welche z.B. in der Bild- und Signalverarbeitung eingesetzt werden [Pre83, Ree84, WD84]. Diese Prozessoren arbeiten mit einem festen, oft durch das Prinzip eines Signalgebers vorgegebenen Datenformat. Die vorab bekannte Datenreihenfolge ermöglicht die synchrone Bereitstellung aller für einen Verarbeitungsschritt benötigter Operanden durch eine Hardware. So läßt sich beispielsweise die benötigte Bildumgebung $\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F})$ der lokalen Operationen nach Gleichung 3.17 durch eine adäquate Hardware erzeugen. Diese Operandenmenge steht dann der nachfolgenden Verarbeitungseinheit zur Berechnung eines Ergebnispunktes parallel zur Verfügung.

Abbildung 5.4 zeigt den prinzipiellen Aufbau eines Pipelineprozessors. Dieser besteht aus einer Einheit zur synchronen Generierung benötigter Operandenmengen und aus einer nachfolgenden Verarbeitungseinheit, welche aus den sukzessiven Operandenmengen die Ausgangsdatensequenz errechnet. Falls eine Änderung der Operationen durch die Datensequenz vorgegebenen ist, erfolgt die Auswahl der aktuellen Operation und somit die Generierung eines impliziten Befehlsstroms durch die Einheit zur Operandengenerierung. In der Abbbildung ist diese Möglichkeit durch einen Operatorpfad vom Block der Operandengenerierung zur Verarbeitung gekennzeichnet.

Eine häufig zur lokalen Bildfilterung verwendete zweidimensionale, quadratische Umgebung $\mathcal{U}^{k\times k}$, welche auf einem quadratischen Bildfenster $\mathcal{F}^{k\times k}$ basiert (Gleichung 3.18), kann beispielsweise bei einem zeilensequentiellen Datenstrom durch $k-1$ Schieberegister 'SR' und $(k-1)\cdot k$ Einzelregister 'R' mit der in Abbildung 5.5 für $k=5$ gezeigten Anordnung erzeugt werden. Bei einer Zeilenlänge N werden Schieberegister zur Speicherung von $N-k-1$ Werten benötigt.

Kenngrößen von Pipelineprozessoren sind die Datenrate, mit der sukzessive, zur Verarbeitung benötigte Operanden durch den Block der Operandengenerierung zur Verfügung gestellt werden, sowie die Rate der Verarbeitungseinheit. Beide Werte müssen zur Erzielung einer optimalen Prozessorleistung auf die Rate der Eingangsdaten abgestimmt sein.

Die Einheit zur Generierung der Operandenmenge verbirgt einen lokalen Speicher. Dessen Größe muß mindestens so ausgelegt sein, daß jede Operandenmenge mit den zwischen-

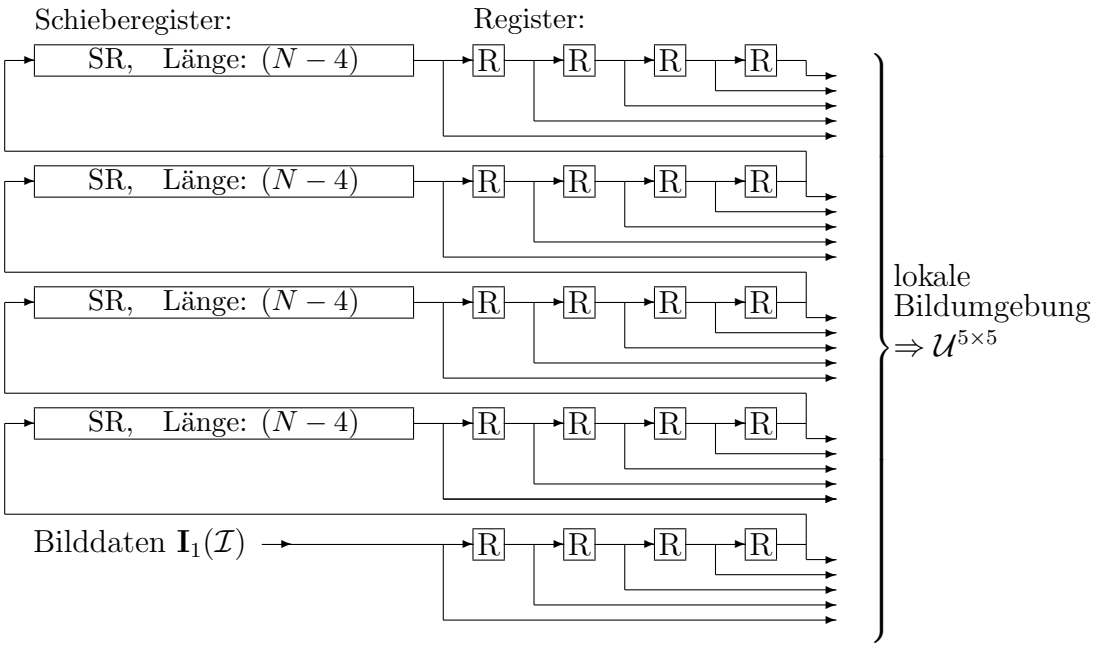

Abbildung 5.5: Erzeugung einer lokalen Bildumgebung $\mathcal{U}^{5\times5}$.

liegenden Daten der Eingangssequenz gespeichert werden kann. Im ungünstigsten Fall, wenn das erste Element einer Sequenz mit dem letzten verknüpft wird, muß die gesamte Sequenz gespeichert werden.

In Abbildung 5.4 werden keine Einheiten zur sequentiellen Erzeugung der Quell- und zur Abspeicherung der Zieldaten gezeigt. Diese sind bei Pipeline-Systemen vom Prozessor getrennt. Mögliche Einheiten sind durch Adreßgeneratoren unterstützte Systemspeicher, die ein effizientes Ein- und Auslesen von Datensequenzen ermöglichen. Weiterhin kann ein Pipelineprozessor als Datenquelle/-senke nach-/vorgeordneter Prozessoren dienen. Dies erlaubt das Kaskadieren der Prozessoren und ermöglicht eine direkte Unterstützung algorithmischer Funktionszerlegungen.

Zum Aufbau von Systemen sind Pipelineprozessoren in verschiedenen Integrationsstufen erhältlich. Diese reichen von der Platinenebene [BMF88] über integrierte Bausteine zur Operandengenerierung und Verarbeitung [LSI90] bis zum vollständigen Pipelineprozessor als integriertem Baustein [INM88a].

Eine dritte Einzelprozessorarchitektur zur Datensequenz-gesteuerten Programmausführung wird durch eine Prozessorfamilie realisiert, welche als *XPuter* [HHR+90] bezeichnet werden. Kernstück dieser Prozessoren (siehe Bild 5.6) ist ein komplexer Adreßgenerator, dort als Datensequenzer bezeichnet, der in vielfältigen Mustern auf einen Datenspeicher zugreift und Datensequenzen einer konfigurierbaren Verarbeitungseinheit (rALU[2]) zuführt. Neben weiteren Reihenfolgen ermöglicht er den Zugriff auf den Datenspeicher in linearer, in zweidimensionaler oder auch in 'Shuffle'-Reihenfolge.

Die mit dem Adreßgenerator aus dem Systemspeicher ausgelesenen Datensequenzen wer-

[2]rALU: reconfigurable Arithmetic Logic Unit

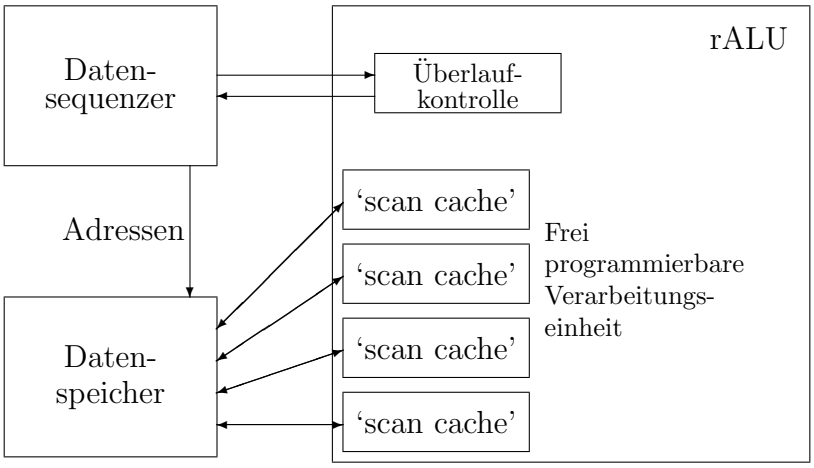

Abbildung 5.6: Blockstruktur zur Prozessorfamilie der XPuter [HHR+90].

den Zwischenspeichern ('scan cache') zugeführt. Diese Zwischenspeicher synchronisieren die Daten und geben sie an eine frei programmierbare Verarbeitungseinheit weiter. Diese Einheit ist während der Verarbeitung einer Sequenz auf eine feste Instruktion eingestellt. Die Programmierung einer Sequenzinstruktion erfolgt bei der vorgeschlagenen Verwendung programmierbarer logischer Bausteine (PLDs) auf Gatter- oder auf Zellebene, sie wird jedoch durch Software-Schichten dem Benutzer verborgen.

5.2 Organisation von Multiprozessorsystemen

Zum Aufbau von Multiprozessorsystemen ergibt sich eine unüberschaubare Anzahl von Möglichkeiten, Prozessoren, Speicher und dazwischenliegende Verbindungsstrukturen anzuordnen. Viele mögliche Anordnungen sind nicht sinnvoll, manche Anordnungen haben sich jedoch — zumindest im Hinblick auf zugeordete Aufgabenstellungen — als besonders brauchbar herauskristallisiert.

In diesem Abschnitt werden Überlegungen zur Prozessorauswahl und Speicherorganisation beim Entwurf von Multiprozessorsystemen ausgeführt. Eine Übersicht über Verbindungsnetzwerke schließt sich im folgenden Abschnitt an.

5.2.1 Prozessorauswahl

Die Auswahl der Prozessoren bestimmt die Rechenleistung und die Funktionalität der Grundoperationen des Systems. Die Auswahlkriterien beginnen mit dem im vorangegangenen Abschnitt angesprochenen Operationsprinzip. Insbesondere stellt sich die Frage, ob alle Prozessoren die gleiche oder unterschiedliche Funktionalität besitzen sollen. Diese Entscheidung führt zu *homogenen* oder *heterogenen* Systemen.

Weiterhin muß die Gesamtanzahl aller Prozessoren sowie die Wortbreite jedes einzelnen Prozessors festgelegt werden. Bei vorgegebener Systemgröße stehen beide Werte zueinan-

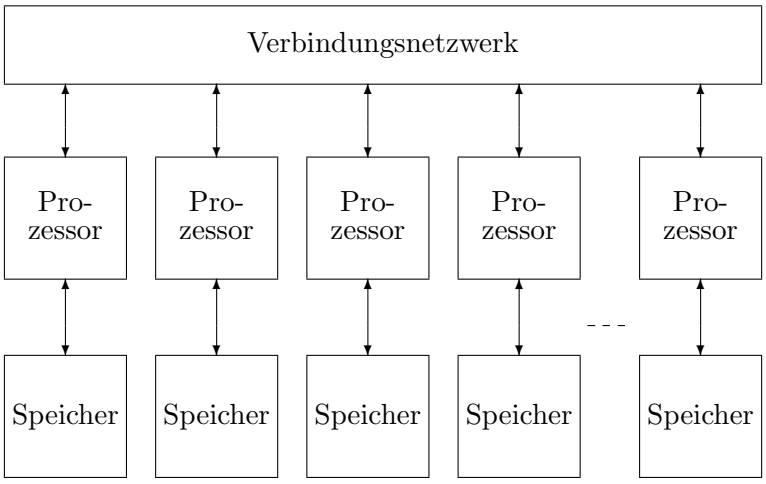

Abbildung 5.7: Lokale Speicherorganisation eines Multiprozessorsystems.

der in Konkurrenz, da mit steigender Wortbreite der Systemaufwand zumindest linear ansteigt.

Bei homogenen, durch Kontrollfluß gesteuerten Systemen besteht ein wichtiges Unterscheidungsmerkmal in der Zuführung der Instruktionen. Man findet die beiden Möglichkeiten, jedem Prozessor eine lokale, eigene Instruktionssequenz zuzuordnen oder alle Prozessoren mit der gleichen Instruktion zu beschicken. Bei unabhängigen Instruktionsströmen für jeden Prozessor ist die Bearbeitung einer unabhängigen Aufgabe auf jedem Prozessor möglich. Ein einheitlicher Befehlsstrom führt zu Feldrechnern, wo nur die Bearbeitung unabhängiger Daten mit einheitlichem Algorithmus möglich ist.

5.2.2 Speicherorganisation

Zur Organisation des Speichers unterscheidet man zwei grundsätzliche Formen: den *lokalen*, verteilten Speicher und den *globalen*, allen Prozessoren gemeinsam zugeordneten Speicher. Die Organisation korrespondiert mit den beiden im vorangegangenen Kapitel (Abschnitt 4.2.2) eingeführten Kommunikationsmodellen nebenläufiger Tasks.

Der Aufbau eines Systems mit *lokalen Speichern*, wo jedem Prozessor ein lokales Speichersegment zugeordnet wird, führt zwangsläufig zu einer verteilten Organisation. Damit ist kein globaler Speicherzugriff möglich und als Kommunikationsmodell nur das 'Message-Passing' anwendbar. Abbildung 5.7 zeigt diese Organisation. Zur vollständigen Realisierung des 'Message-Passing' wird ein Verbindungsnetzwerk zwischen den Prozessoren benötigt. Über dieses Verbindungsnetzwerk werden die Nachrichten ausgetauscht. Die Parameter dieses Netzwerks bestimmen in starkem Maße die Leistungsdaten zur Kommunikation nebenläufiger Tasks und damit die parallele Systemleistung.

Bei Systemen mit *globalem Speicher* hat jeder Prozessor direkten Zugriff auf den Gesamtspeicher. Dieser ist jedoch meist nicht als Einheit realisiert, sondern in unabhängigen

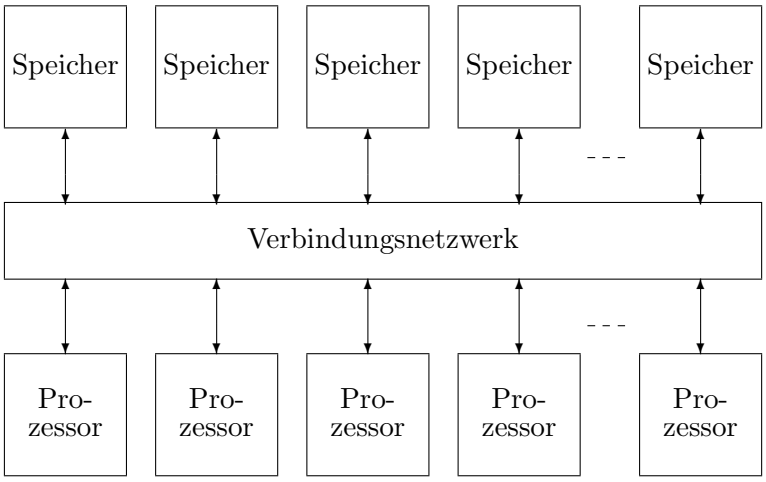

Abbildung 5.8: Organisation eines Multiprozessorsystems mit globalem, segmentiertem Speicher.

Bänken organisiert. Die Segmentierung eines globalen Speichers erhöht die Zugriffbandbreite und die Anzahl der Zugriffschnittstellen. Damit ist eine erste Vorkehrung zur Verminderung von Zugriffkonflikten, wenn mehrere Prozessoren auf den globalen Speicher zugreifen wollen, getroffen.

Wird weiterhin die Speichersegmentierung mit dem Verbindungsnetzwerk zwischen den Prozessoren und Speicherbänken (siehe Abbildung 5.8) abgestimmt, so lassen sich häufig vorkommende, parallele Zugriffmuster aller Prozessoren auf den globalen Speicher finden, welche konfliktfrei durchgeführt werden können. Lawrie zeigt z.B. in [Law75] eine konfliktfreie Organisation zum Zugriff auf Zeilen, Spalten und Diagonalen eines zweidimenisonal organisierten, globalen Speichers auf.

Besteht ein System mit globalem Speicher aus der gleichen Anzahl von Prozessoren und Speicherbänken, so besteht der topologische Unterschied zur lokalen Speicherorganisation in der fehlenden Verbindung zwischen jeweils einem Prozessor und einer dazu korrespondierenden Speicherbank. Faßt man Speicherbänke und Prozessoren der globalen Organisation paarweise als Einheit zusammen, so entsprechen die lokale und globale Organisation einander nahezu. Lediglich der Speicherzugriff ist bei der lokalen Organisation nicht direkt möglich, sondern erfolgt über den vorgeordneten Prozessor, was bezüglich einer Zugrifforganisation und Konfliktauflösung sogar vorteilhaft sein kann.

Somit läßt sich — bei Organisation der Zugriffe — ein globaler Speicher in einem System mit verteilten Speichern bei gleicher Ordnung des zeitlichen Zugriffs simulieren, falls die Äquivalenz beider Verbindungsnetzwerke gewährleistet ist. Die Frage nach einem lokalen oder globalen Speicher wird damit losgelöst von der Hardware-Realisierung, bleibt nur noch auf der Algorithmen- und Software-Ebene bestehen und kann damit durch einen 'Compiler' verborgen werden. Eine weitergehende Untersuchung zur Simulation eines gemeinsamen Speichers in einem System mit lokalen verteilten Speichern findet man in [UW87].

Ein letzter Gesichtspunkt zur Speicherorganisation betrifft die Auswahl der Datenmechanismen, welche beim Zugriff auf die lokalen Speicher oder die Speicherbänke eines globalen Speichers unterstützt werden. Diese Mechanismen müssen mit den Operationsprinzipien der Einzelprozessoren eines Multiprozessorsystems korrespondieren.

5.3 Multiprozessor Verbindungsnetzwerke

Neben dem Operationsprinzip der Einzelprozessoren, der Prozessorauswahl und der Speicherorganisation hat die Auswahl eines geeigneten Verbindungsnetzes zwischen Verarbeitungs- und/oder Speichereinheiten eine entscheidende Bedeutung beim Entwurf eines Multiprozessorsystems. So wie beim traditionellen 'von Neumann'-Rechner der Bus zwischen Prozessor und Speicher einen Flaschenhals bilden kann, können bei Multiprozessorsystemen falsch gewählte Verbindungsnetzwerke weitaus schlimmere Engstellen verursachen und zum entscheidenden Begrenzungsparameter der Systemleistung werden.

Die große Vielfalt von Multiprozessoranordnungen spiegelt sich insbesondere in der Vielfalt der Verbindungsnetzwerke wider. Zur Einordnung und Bewertung von Multiprozessoren und somit auch zum Entwurf neuer Systeme gehört die Analyse dieser Netzwerke. Die folgenden Betrachtungen werden keinen vollständiger Überblick über Verbindungsnetzwerke geben, dazu wird auf die Literatur verwiesen [GE89, Ree87, RLH88, Sie80]. Es sollen hingegen typische Verbindungstopologien aufgegriffen werden, welche — neben allgemeinen Anwendungen — Bedeutung im Bereich der Bildverarbeitung besitzen.

Multiprozessortopologien lassen sich durch einen Graphen $\mathcal{T}$ beschreiben. Die Ecken, durch die Menge $\mathcal{P}$ dargestellt, repräsentieren Einzelprozessoren, Segmente eines globalen Speichers oder — im Falle mehrstufiger Netzwerke — Schalter- und Routingelemente. Zur einfachen Darstellung werden im weiteren die Ecken meist als Prozessoren angesprochen. Die Kanten $\mathcal{N}$ des Graphen beschreiben ein Verbindungsnetzwerk und lassen sich als zweistellige Relation auf der Eckenmenge formulieren:

$$\begin{aligned} \mathcal{T} &= (\mathcal{P}, \mathcal{N}) && (5.1)\\ \mathcal{P} &= \{p_i | i \in I\!N_P\} && (5.2)\\ \mathcal{N} &\subseteq \mathcal{P} \times \mathcal{P}. && (5.3) \end{aligned}$$

Multiprozessortopologien unterscheiden sich im wesentlichen in der Wahl der Teilmenge $\mathcal{N}$ aus der Gesamtmenge der möglichen Verbindungen nach Gleichung 5.3. Die Eckenmenge $\mathcal{P}$ kann lediglich Einschränkungen bezüglich zulässiger Eckenanzahlen (z.B. Einschränkung auf Zweierpotenzanzahlen) aufweisen.

Die Mächtigkeit der Kantenmenge $|\mathcal{N}|$ gibt die Anzahl der Netzwerkverbindungen an. Bei gleichem Aufwand für jede Verbindung ist sie ein Maß für den Realisierungsaufwand und für die maximal zur Verfügung stehende Kommunikationsbandbreite $B_{Max}(\mathcal{T})$ des Netzwerks.

Eine Korrespondenz zwischen der Algorithmen- und Topologiebeschreibung durch Graphen ist an dieser Stelle beabsichtigt. Zur effizienten Implementierung eines Algorithmus auf einem Multiprozessorsystem gilt es, eine möglichst hohe Korrespondenz beider

Graphen zu erzielen. Dabei erfolgt die Zuordnung eines Subgraphen $\mathcal{G}_k$ (Gleichung: 3.10) zu einer Ecke $p_k \in \mathcal{P}$ und eine Abbildung der globalen Kanten $\mathcal{K}^G$ (Gleichung 3.9) auf die Kantenmenge $\mathcal{N}$ der Topologie. Eine Beschränkung auf Zyklenfreiheit wird bei der Topologiebeschreibung nicht vorgenommen, jedoch werden Kommunikationen und somit zugehörige Kanten $(p_i, p_j) \in \mathcal{N}$ zur Beschreibung von Netzwerkverbindungen als gerichtet angesehen.

Es sollen *einstufige* und *mehrstufige* Netzwerke betrachtet werden. Einstufige Netzwerke verbinden Prozessoren direkt untereinander, wobei ein Nachrichtenaustausch zwischen zwei Prozessoren P_i, P_j nicht nur direkt, sondern bei fehlender oder fehlerhafter Verbindung auch indirekt über weitere Prozessoren erfolgen wird.

Mehrstufige Netzwerke werden unterteilt in *schaltbare* und *selbstroutende* Netzwerke. Erstere besitzen im Inneren Schalter zum Einstellen gewünschter Ein-/Ausgangspermutationen. Selbstroutende Netzwerke beinhalten Routingelemente, welche Nachrichten durch zugefügte Zieladressen selbständig zu einem vorgegebenen Ausgang routen. Damit bestimmt die Stellung der Schalter oder die Funktion der Routingelemente die Verbindungspfade zwischen den am Netzwerk angeschlossenen Prozessoren und Speichersegmenten.

5.3.1 Bewertung von Netzwerken

Zur Bewertung verschiedenartiger Netzwerke besteht der Wunsch, Meßgrößen zu ermitteln, welche Eigenschaften eines Netzwerks entkoppelt erfassen und einen einfachen Vergleich ermöglichen. Dazu sind in der Literatur viele Größen definiert worden, bei deren Interpretation jeweils zugrundegelegte Annahmen zu berücksichtigen sind. Im einfachsten Fall beinhalten diese Annahmen keinerlei Informationen über das auszuführende Programm, sondern nehmen eine gleichverteilte Kommunikation jedes Prozessors mit allen übigen Prozessoren des Systems an. Die resultierenden Meßwerte basieren dann allein auf der Topologie eines Verbindungsnetzwerks, sie können für 'worst-case'-Abschätzungen verwendet werden. Berücksichtigt man die Topologie und den Kommunikationsbedarf eines auszuführenden Algorithmus in den Bewertungskriterien, so kommt man zu spezielleren Aussagen, welche einen besseren Netzwerkvergleich für die vorgegebene Aufgabenstellungen ermöglichen. Weitere Anforderungen umfassen die Initialisierung und Terminierung vor und nach dem Programmlauf oder ergeben sich durch die Forderung nach einer modularen Systemtopologie. Die Eignung eines Netzwerks für eine parallele Programmausführung muß an der Berücksichtigung aller jeweils relevanter Anforderungen bewertet werden.

Topologische Meßgrößen

Im folgenden werden einige topologische Meßgrößen hervorgehoben, welche eine erste Einordnung von Netzwerken ermöglichen. Weitere Größen findet man in der Literatur [RLH88, GE89, AJP86, Ree87].

Anzahl der Anschlußpunkte: Die *Anzahl der Anschlußpunkte* r_i, auch als *Portanzahl* oder *'Degree'* bezeichnet, beschreibt die Anzahl der Netzverbindungen[3], welche an einen Prozessor P_i angeschlossen sind. Im Graphen ergibt r_i die Mächtigkeit der Menge aller von einer Ecke p_i wegweisender Kanten:

$$r_i = |\{(p_i, p_k) | (p_i, p_k) \in \mathcal{N},\ p_k \in \mathcal{P}\}| \tag{5.4}$$

Vielfach ist r_i konstant und damit unabhängig von einem Prozessorindex i.

Die Anzahl der Anschlußpunkte ist ein wichtiger Faktor der Kosten eines Prozessors. Weiterhin begrenzt eine fester Wert für r_i die Erweiterbarkeit mancher Netzwerktopologien.

Distanz: Die Distanz $d_{i,j}$ zählt die minimalen Anzahl von Netzverbindungen, welche eine Nachricht bei der Übertragung von Prozessor P_i nach P_j durchlaufen muß. Sie läßt sich im Graphen über die minimale Länge aller positiven Pfade von Ecke p_i zur Ecke p_j definieren:

$$d_{i,j} = \min_{p_i \in \mathcal{P}, p_j \in \mathcal{P}} (|\mathbf{P}^+_{i,j}(\mathcal{T})|) - 1. \tag{5.5}$$

Da zwei benachbarte Ecken eines Pfads jeweils eine Kante des Graphen bilden, entspricht $d_{i,j}$ genau der minimalen Anzahl von Kanten, welche die Ecken p_i und p_j miteinander verbinden. Die Distanz eines Prozessors zu sich selber wird zu 0 angenommen: $d_{i,i} = 0$.

Maximale Distanz, Durchmesser: Die *maximale Distanz* oder der *Durchmesser* eines Netzwerks d_{Max} zählt die Anzahl der Netzverbindungen, welche eine Nachricht im Maximalfall bei der Übertragung zwischen beliebigen Prozessoren P_i und P_j durchlaufen muß. Aus der Graphensicht ergibt sich d_{Max} aus dem Maximum der Distanzen aller Eckenpaare p_i, p_j:

$$d_{Max} = \max_{p_i \in \mathcal{P}, p_j \in \mathcal{P}} (d_{i,j}). \tag{5.6}$$

Die maximale Distanz ist für globale Algorithmen wie z.B. die in Abschnitt 3.2.2 eingeführte Klasse schneller Transformationen, wo ein Ergebniswert von *allen* Eingangswerten abhängt und während der Berechnung keine Datenreduktion auftritt, eine untere Schranke für die Ordnung der Kommunikationszeit. Nimmt man eine Prozessoranzahl P und ein gleiche Kommunikationszeit t_k für jede Kante $(p_i, p_j) \in \mathcal{N}$ an, so benötigt allein die Zuführung der Eingangswerte (bzw. davon abhängiger Zwischenergebnisse) zu einem Prozessor im ungünstigsten Fall die Zeit $t = t_k \cdot d_{Max} = O(d_{Max})$ und begrenzt damit die Ordnung der Ausführungszeit.

Unter der Annahme, daß ein Prozessor gleichzeitig über mehrere Anschlußpunkte Daten ausgeben kann, wird d_{Max} proportional zur Anzahl der Kommunikationsschritte zur Verteilung einer Nachricht im gesamten System. Diese im folgenden noch behandelte Verteiloperation wird in der Literatur als 'Broadcast' bezeichnet.

Mittlere Distanz: Die mittlere Distanz $\bar{d}_i$ bezogen auf einen Prozessor P_i mittelt im einfachen Fall [AJP86] die Entfernungen zwischen diesem Prozessor und den übrigen

[3]Die Definition umfaßt nur die Ausgangsverbindungen, es soll in dieser Arbeit immer eine gleiche Anzahl von Ein- und Ausgängen angenommen werden.

Prozessoren im Netzwerk:

$$\bar{d}_i = \frac{1}{\mathrm{P}-1} \sum_{d=1}^{d_{Max}} d \cdot \Phi(d,i). \tag{5.7}$$

Die Funktion $\Phi(d,i)$ liefert die Anzahl der Prozessoren mit der Distanz d zu Prozessor P_i in einem vorgegebenen Netzwerk zurück. Die Ermittlung der Distanz läßt sich umschreiben in die Summation über die Distanzen aller Ecken zu p_i:

$$\bar{d}_i = \frac{1}{\mathrm{P}-1} \sum_{j=0}^{\mathrm{P}-1} d_{i,j}. \tag{5.8}$$

Bei einem symmetrisches Netzwerk ist der Wert der mittleren Distanz für alle Ecken gleich: $(\bar{d}_i = \bar{d})$, bei asymmetrischen Netzwerken jedoch von der Bezugsecke p_i abhängig.

Hat man Kenntnis über den Kommunikationsbedarf eines Algorithmus A, läßt sich dieser in Gleichung 5.8 berücksichtigen. Bei der Ausführung von A und einem paarweisen Kommunikationsbedarf $a_{i,j}(A)$ zwischen Prozessor P_i und P_j kann die zugehörige Distanz $d_{i,j}$ mit diesem Bedarf gewichtet werden. Ein Kommunikationsbedarf eines Prozessors zu sich selbst soll zu 0 angenommen werden: $a_{i,i}(A) = 0$. Mit der algorithmischen Gewichtung und einer Normierung auf den Gesamtbedarf wird Gleichung 5.8 erweitert und liefert eine von A abhängige mittlere Distanz A:

$$\bar{d}_i(A) = \frac{\sum_{j=0}^{\mathrm{P}-1} d_{i,j} \cdot a_{i,j}(A)}{\sum_{j=0}^{\mathrm{P}-1} a_{i,j}(A)}. \tag{5.9}$$

Damit ist ein erster Vergleich verschiedener Netzwerke bezüglich der Ausführung eines Algorithmus A möglich. Symmetrie oder Asymmetrie der mittleren Distanzen verschiedener Prozessoren ergeben sich nun aufgrund der algorithmischen Ausnutzung eines Netzwerks. So können durch einen asymmetrischen, algorithmischen Kommunikationsbedarf bei einem asymmetrischen Netzwerk wieder symmetrische Distanzen für alle Prozessoren P_i entstehen (Beispiel: Ausführung eines Baumalgorithmus auf einer Baumtopologie). Mögliche Konflikte durch mehrfache Zugehörigkeit einer Kante zu unterschiedlichen Kommunikationspfaden und unterschiedliche Kommunikationszeitpunkte werden in dieser Betrachtung noch nicht berücksichtigt.

Aufgaben und Anforderungen

Verbindungsnetzwerke eines Multiprozessorsystems müssen während einer Programmausführung unterschiedliche Aufgaben unterstützen. Erste Aufgaben betreffen die Initialisierung und Terminierung der parallelen Verarbeitung und beinhalten insbesondere das Laden von Quelldaten vor sowie das Entladen von Ergebnissen nach einer Berechnung. Weiterhin müssen algorithmische Kommunikationsverbindungen während der Berechnung bereitgestellt und die Betriebssoftware-Zugriffe zur Verwaltung des parallelen Systems organisiert werden. Diese drei Aufgabengebiete sollen genauer betrachtet werden:

Laden und Entladen von Daten: Vor dem Start einer Berechnung müssen Quelldaten in einer Initialisierungsphase so in ein Prozessorsystem geladen werden, daß ein Verarbeitungsprogramm diese in einer günstigen Anordnung vorfindet. Weiterhin schließt sich nach dem Programmlauf ein Entladen von Daten an, welche in einer von der Berechnung abhängigen Permutation vorliegen.

In einem Einzelprozessorsystem ist dieser Lade-/Entladevorgang noch vergleichsweise einfach. Die zu verarbeitenden Daten müssen möglichst schnell in den Prozessorspeicher geladen werden, was z.B. direkt durch den Prozessor oder durch spezielle DMA-Einheiten erfolgen kann.

Bei Multiprozessorsystemen wird der Ladevorgang komplexer, denn er hängt, wie aus den Überlegungen in Abschnitt 3.1.1 ersichtlich ist, von einer gewählten Algorithmenzerlegung ab. Alle Quelldaten (Eingangsecken der Subgraphen) müssen vor einer Verarbeitung in die zugehörigen Prozessoren geladen werden. Nach der Verarbeitung muß ein Entladen der Ergebnisdaten (Ausgangsecken) unter Beachtung ihrer Verteilung erfolgen. Für einen in der Bildverarbeitung typischen Datenaustausch zwischen einer einzelnen Datenquelle oder -senke und einem parallelen Prozessorsystem findet man in der Literatur [SS89a] drei elementare Lademodi:

'Broadcast': Werden alle Quelldaten in jeden Prozessor eines parallelen Systems geladen, bezeichnet man diesen Ladevorgang als *'Broadcast'*. Damit werden alle Eingangsecken eines Graphen $\mathcal{G}$ jedem Subgraphen $\mathcal{G}_k$ einer Zerlegung zugeordnet.

'Scattering': Teilt man die Quelldaten auf die P Prozessoren eines Systems auf, wird ein als *'Scattering'* bezeichneter Ladevorgang durchgeführt. Dies entspricht der Aufteilung der Eingangsecken auf die Subgraphen einer Zerlegung $\mathcal{Z}(\mathcal{G})$. Es soll zwischen einer disjunkten und einer überlappenden Aufteilung der Eingangsecken unterschieden werden. Die zweite Aufteilung bedingt ein Kopieren von Quelldaten während des Ladens, gibt dadurch jedoch die Möglichkeit, alle globalen Kanten mit Eingangsecken als Sender zu vermeiden und somit die algorithmische Kommunikation während der Berechnung zu vermindern.

'Gathering': Nach der Ausführung aller Subgraphen liegen Ergebnisdaten in einer durch den Algorithmus bestimmten Reihenfolge (Permutation) verteilt in den parallelen Prozessoren eines Multiprozessorsystems vor. Der als *'Gathering'* bezeichnete Entladevorgang fügt diese verteilten Ergebnisdaten wieder zu einem geordneten Datenobjekt zusammen.

Datenaustausch während der Programmausführung: Zwischen dem Verteilen der Quell- und dem Zusammenfügen der Ergebnisdaten erfolgt die Algorithmen-basierte Programmausführung. Sie stellt den Kern der parallelen Verarbeitung dar. Während dieser Verarbeitung bewirkt eine möglichst direkte Abbildung der algorithmischen Topologie eines Graphen $\mathcal{G}$ auf die Systemtopologie $\mathcal{T}$ eine Minimierung des Kommunikationsaufwands. Bei gegebener Zerlegung eines Datenflußgraphen $\mathcal{Z}(\mathcal{G})$ und Verteilung der Subgraphen $\mathcal{G}_k$ auf die Prozessoren sollte eine möglichst direkte Abbildung der globalen Kanten auf das Verbindungsnetzwerk möglich sein. Dann wird der Routingaufwand zum Senden von Nachrichten in den Kommunikationsschichten minimiert.

Betriebssoftware-Kommunikation: Verbindungspfade zur Betriebssoftware-Kommunikation unterstützen die Transparenz eines Systems aus der Anwendersicht. Eine Verbindungsstruktur in diesem Bereich kann nicht isoliert von einer Betriebssoftware gesehen werden. Daher sollte ein zugehöriges Netzwerk mit seiner Topologie und seinen Hardware-Protokollen die einfache Implementierung komplexer Software-Protokolle durch ein Betriebssystem ermöglichen, welche der Steuerung und Überwachung des parallelen Systems dienen.

Zu den Anforderungen, die sich aus obigen Aufgaben ergeben, kommt die Forderung nach *Modularität* mit hinzu. Die Modularität fordert sinnvollerweise einen linearen Aufwand bei einer Systemerweiterung durch Erhöhung der Prozessoranzahl.

Bei einer vorgegebenen Topologie $\mathcal{T}$ mit der Eckenanzahl $P = |\mathcal{P}|$ als freiem Parameter ist die Mächtigkeit der Kantenmenge $|\mathcal{N}|$ von der Eckenanzahl abhängig:

$$|\mathcal{N}| = f(P). \tag{5.10}$$

Bei modularen Netzwerken muß als notwendige Bedingung zur Begrenzung der Anzahl von Anschlußpunkten r_i und zur Ausgewogenheit des Kommunikations- und Verarbeitungsaufwands ein linearer Zusammenhang in Gleichung 5.10 bestehen. Dies führt mit c als Konstante zur Modularitätsbedingung, wobei ein gleicher Aufwand für jede Netzverbindung vorausgesetzt wird:

$$|\mathcal{N}| \leq c \cdot P. \tag{5.11}$$

Ein sinkender Netzwerkaufwand bei steigender Prozessoranzahl wird nur in sehr seltenen Fällen gegeben sein, daher wird die Modularität üblicherweise die Gleichheit in Gleichung 5.11 ergeben.

5.3.2 Einstufige Netzwerke

Zum Vergleich einstufiger Verbindungsnetze werden der *vollständigen Verbindung* mit optimalen Kommunikationseigenschaften Busverbindungen und typische einstufige Netzwerke gegenübergestellt. Als Busverbindungen werden der *globale Bus* und der *Pipelinebus* betrachtet. Als typische einstufige Netzwerke dienen die *Lineare Verbindung* sowie die *'Mesh'*-, *'Hypercube'*- und *'Perfect Shuffle'*-Verbindungsnetzwerke.

Vollständige Verbindung (VV):
Als Referenz für andere Netzwerke bietet sich die *vollständige Verbindung* an, da jede Ecke dieser Topologie mit allen übrigen Ecken verbunden ist. Damit entspricht die Kantenmenge $\mathcal{N}_{VV}$ genau dem kartesischen Produkt der Eckenmenge:

$$\mathcal{N}_{VV} = \mathcal{P} \times \mathcal{P}. \tag{5.12}$$

Dieses Netzwerk stellt die Obermenge aller einstufigen Netze dar, somit können seine Kenngrößen als maximale obere bzw. untere Schranken angesehen werden:

$$\begin{aligned} d_{Max\,VV} &= \bar{d}_{VV} = 1 \\ r_{VV} &= P. \end{aligned} \tag{5.13}$$

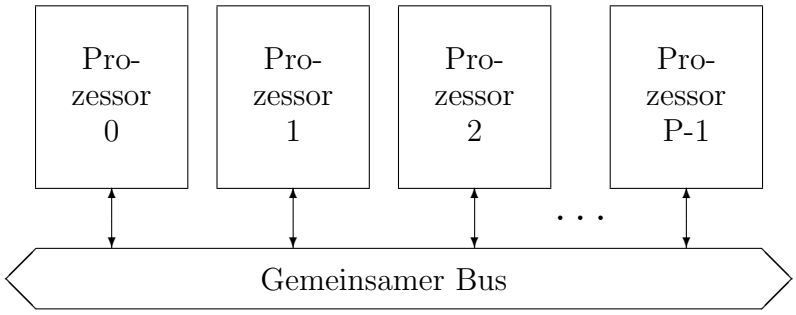

Abbildung 5.9: Gemeinsamer Bus als Verbindung paralleler Prozessoren.

Beim realen Einsatz liegt die Problematik dieses Netzwerks mit seinem quadratisch wachsenden Verbindungsaufwand $|\mathcal{N}|$ klar auf der Hand.

Gemeinsamer Bus:
Die traditionelle Verbindung mehrerer Prozessoren ist der gemeinsame Bus (Abbildung 5.9). Seine Topologie läßt sich durch einen Graphen nach Gleichung 5.1 beschreiben, wenn man den Bus als Systemeinheit mit Verbindungen zu allen Prozessoren interpretiert. Dann entspricht die Topologie des gemeinsamen Busses einer Sterntopologie. Verbindungen zweier Prozessoren werden durch den Buszugriff der angeschlossenen Module im Zeitmultiplex gewährleistet, was sich bei einem symmetrischen Kommunikationsbedarf jedes Prozessors in einer von der Anzahl der Prozessoren umgekehrt proportionalen, mittleren Bandbreite jeder logischen Einzelverbindung niederschlägt. Damit ist ein gemeinsamer Bus als Verbindungsnetzwerk nur bei kleinen Prozessoranzahlen oder geringem Kommunikationsbedarf sinnvoll.

Die direkte Verbindung aller Prozessoren über den Bus bewirkt eine sehr kurze maximale Distanz $d_{Max} = 2$ (Gleichung 5.6) dieser Topologie. Im Gegensatz zur vollständigen Verbindung sind diese kurzen Distanzen jedoch nur im Zeitmultiplex verfügungbar. Nutzbar wird diese kurze Distanz beim schnellen Verteilen eines Datums von einem Sender zu vielen Empfängern ('Broadcast'), wenn ein Prozessor als Sender und alle übrigen gleichzeitig als Empfänger arbeiten. Ein 'Broadcast' ist in diesem Fall in einem einzigen Buszyklus möglich.

Die physikalische Verbindung aller Prozessoren über gemeinsame Signalleitungen begrenzt durch deren elektrische Eigenschaften die Erweiterbarkeit eines globalen Busses. Übliche Prozessoranzahlen liegen im Bereich $P \leq 8 \ldots 16$.

Pipelinebus:
Eine Variation des gemeinsamen Busses ist der *Pipelinebus*. Dieser unterteilt den Gesamtbus durch Pipelinebus-Interfaces in Segmente (Abbildung 5.10). Jedes Pipelinebus-Interface enthält synchron getaktete Register zur Zwischenspeicherung von Busdaten. Dadurch kann ein Pipelinebus in seinem Übertragungsverhalten als Schieberegister angesehen werden.

Die Segmentierung des Busses ermöglicht einen physikalischen Aufbau, der sich modular erweitern läßt. Außer einem gemeinsamen Takt werden nur lokale Verbindungen

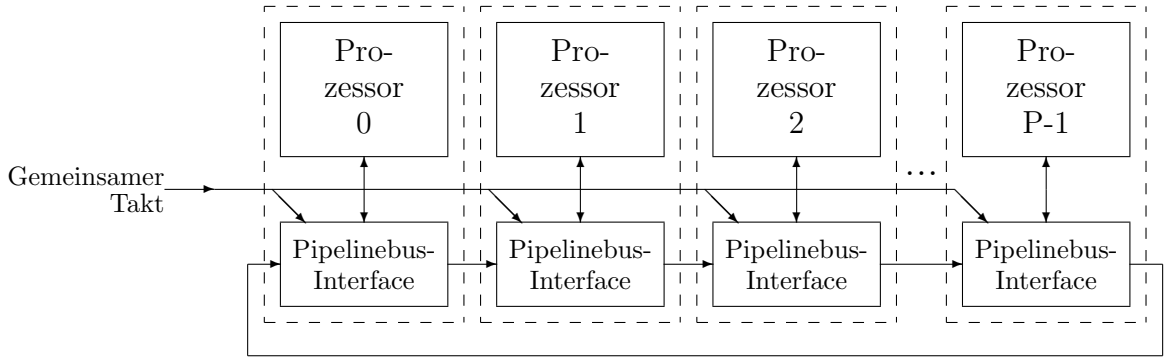

Abbildung 5.10: Der Pipelinebus als Verbindungsnetz paralleler Prozessoren.

benachbarter Prozessoren benötigt. Der Pipelinebus weist eine unidirektionale Übertragungsrichtung auf. Der Aufbau jedes Verbindungspfads zwischen beliebigen Prozessoren wird durch ein ringförmiges Schließen des Busses erreicht.

Die Distanz zweier Prozessoren ist abhängig von deren Abstand auf dem Pipelinebus. Dieser Abstand entspricht im Mittel etwa der halben Prozessoranzahl $\bar{d} \approx P/2$ und nimmt maximal den Wert $d_{Max} = P - 1$ an. Durch eine hohe Taktrate T_0 und durch parallele Übertragungen auf unabhängigen Bussegmenten (lokalen Übertragungen) macht sich dieser lineare Anstieg der Distanz jedoch erst bei hohen Prozessoranzahlen und vielen globalen Kommunikationen bemerkbar.

Die Leistungsfähigkeit des Pipelinebusses ergibt sich durch die Verlagerung der Routingfunktion aus den Prozessoren in die Pipelinebus-Interfaces. Ein Interface empfängt ankommende Daten und sendet sie entweder zum angeschlossenen Prozessor oder zum nächsten Interface in der Pipeline. Dabei ist der Prozessor nur durch das Senden oder Empfangen eigener Daten belastet. Sieht man im Interface einen Modus vor, welcher ein Datum sowohl an den angeschlossenen Prozessor als auch an das nachfolgende Interface sendet, so läßt sich damit ein sehr schnelles 'Broadcast' durchführen.

Das auf dem Pipelinebus übertragene Datenformat und das Routing der Interfaces unterscheidet Pipelinebus-Realisierungen.

Einen Pipelinebus zum freien Austausch von Datenpaketen beschreiben Franke et al. [FHKZ88]. Dieser Bus überträgt Datenpakete mit vorangestellter Zieladresse als Nachrichten und erlaubt das gleichzeitige Versenden vieler Nachrichten. Das Routing der Nachrichten erfolgt dynamisch in den Interfaces nach einer als *'buffer insertion access method'* bezeichneten Strategie:

> Eine vom Prozessor generierte Nachricht wird bei freiem Ausgang durch das Pipelinebus-Interface ausgegeben.
>
> Erreicht eine Nachricht ein Pipelinebus-Interface, wird die Zieladresse inspiziert. Ist das Interface adressiert, gibt es die Nachricht an den Prozessor weiter. Sonst erfolgt die Weitergabe der Nachricht bei freiem Ausgang direkt oder bei belegtem Ausgang über einen Puffer zum Ausgang. Bei der Weitergabe bleibt die Reihenfolge der Nachrichten unbeeinflußt von einer möglichen Pufferung erhalten.

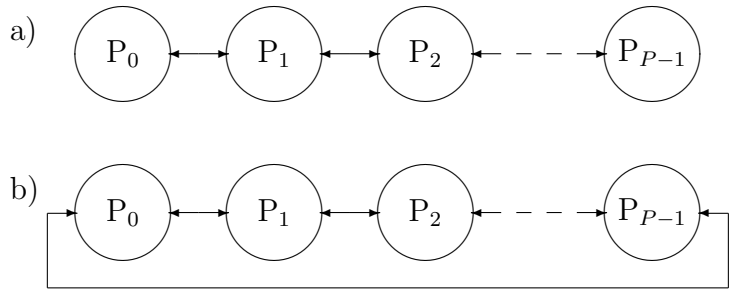

Abbildung 5.11: Lokale Verbindungsnetzwerke: a) Lineare Verbindung, b) Ringnetzwerk.

Die Auflösung von Konflikten erfolgt in dieser Strategie über die zeitlichen Nachrichtenverzögerung durch die Puffer.

Die vorgestellte Strategie sieht kein 'Broadcast' vor, es könnte jedoch durch eine spezielle Adressierung mit dem gleichen linearen Aufwand einer einfachen Datenpaket-Übertragung ermöglicht werden.

Pipelinebusse zur Übertragung strukturierter Massendaten (Vektoren, Felder) findet man — bedingt durch den hohen Kommunikationsbedarf — in leistungsfähigen Bildverarbeitungssystemen [Ree84, Ste83b, MST88]. Bei der Übertragung der Massendaten ist dort der Takt des Busses angepaßt an die Rate, mit der ein Sender Daten generiert. Die Übertragung erfolgt elementweise.

Die Zeit t zum Übertragen eines Datenobjekts errechnet sich aus der Distanz $d_{i,j}$ zwischen einem Sender i und Empfänger j und aus der Größe N des Datenobjekts bei der Taktrate T_0 des Pipelinebusses:

$$t_{Pipelinebus} = (d_{i,j} + N) \cdot T_0. \tag{5.14}$$

Ist die Buslänge und damit die Distanz $d_{i,j}$ klein gegenüber der Größe des Datenobjekts, was z.B. in der Bildverarbeitung bei der Übertragung von Bildmatrizen zutrifft, so wird die Übertragungszeit des Datenobjekts nahezu allein durch N bestimmt.

Das herkömmliche Routing zur Massendatenübertragung ist statisch. Zu einem Verarbeitungszyklus des Systems gehört eine statische, konfliktfreie Einstellung benötigter Verbindungspfade durch die Pipelinebus-Interfaces. Nur zwischen zwei Zyklen kann eine Änderung der Kommunikationspfade vorgenommen werden. Dies erfolgt üblicherweise durch einen übergeordneten Host-Prozessor.

Statisch konfigurierbare Pipelinebusse werden in Systemen mit Pipelineprozessoren eingesetzt, welche nach dem Prinzip der Datensequenz arbeiten (Abschnitt 5.1). Solche Systeme ermöglichen in der Bildverarbeitung durch Anpassung der Bus- und Verarbeitungsraten eine Datenverarbeitung mit Video-Geschwindigkeit.

Lineare Verbindung, Ringnetzwerk:
Prozessoranordnungen mit Verbindungen zu lokalen Nachbarn eignen sich zum Ausführen von Algorithmen mit lokalen Datenabhängigkeiten. Ein lineares Netzwerk und — durch Schließen der linearen Verbindung vom letzten zum ersten Prozessor — ein Ringnetzerk besteht aus solchen lokalen Verbindungen (Abbildung 5.11). Die Topologie ist modular erweiterbar, da die Portanzahl $r_i = 2$ eines Prozessors unabhängig von der Prozessoranzahl P ist.

Probleme bereiten globale Kommunikationsanforderungen zwischen Prozessoren. Die linear mit der Prozessoranzahl wachsende mittlere und maximale Distanz ($\bar{d} \approx P/4$, $d_{Max} = \lfloor P/2 \rfloor$) bedingt eine hohen Routingaufwand bei globalen Kommunikationen und ergibt einen Flaschenhals im System.

'Mesh':
Eine Verallgemeinerung der lokalen Verbindung stellen die *'Mesh'-Verbindungsnetzwerke* dar. Sie werden auch als 'Cube'- oder gelegentlich als 'Hypercube'-Netzwerke bezeichnet. Die Prozessoren dieses Netzwerktyps sind in einer mehrdimensionalen Gitteranordnung organisiert. Mit p-dimensionaler Indizierung der Prozessoren $\mathrm{P}_i,\ i = (i_{p-1}, \ldots, i_0)$ ergibt sich die Topologie einer p-dimensionalen 'Mesh'-Anordnung aus $P = m_{p-1} \cdot m_{p-2} \cdot \ldots \cdot m_0$ Prozessoren zu:

$$\begin{aligned}
\mathcal{T}_{Mesh} &= (\mathcal{P}_{Mesh}, \mathcal{N}_{Mesh}) && (5.15)\\
\mathcal{P}_{Mesh} &= \{p_i | i \in M_{p-1} \times M_{p-2} \times \ldots \times M_0, \quad M_i = I\!N_{m_i}\} && (5.16)\\
\mathcal{N}_{Mesh} &= \{(p_i, p_j) | j = (i_{p-1}, \ldots, i_{k+1}, (i_k + 1) \bmod m_k, i_{k-1}, \ldots, i_0) \vee \\
&\qquad j = (i_{p-1}, \ldots, i_{k+1}, (i_k - 1) \bmod m_k, i_{k-1}, \ldots, i_0), \\
&\qquad p_i \in \mathcal{P}_{Mesh},\ k \in I\!N_p\}. && (5.17)
\end{aligned}$$

Die Kanten aus Gleichung 5.17 verbinden lokal benachbarte Ecken in jeder Dimension. Durch die Modulo-Funktion **mod** erzeugt die obige 'Mesh' Definition einen mehrdimensionalen Torus. Abbildung 5.12 zeigt beispielhaft ein 2−dimensionales 'Mesh'-Netzwerk mit $m_0 = 4$, $m_1 = 3$.

Die maximale Distanz im 'Mesh'-Netzwerk läßt sich durch Aufsummieren der maximalen Distanzen $\lfloor m_i/2 \rfloor$ in jeder Dimension ermitteln:

$$d_{Max\,Mesh} = \sum_{i=0}^{p-1} \lfloor \frac{m_i}{2} \rfloor. \tag{5.18}$$

Sind alle m_i gleich, so gilt $m_i = \sqrt[p]{P}$ und die maximale Distanz errechnet sich zu:

$$d_{Max\,Mesh} = p \lfloor \frac{\sqrt[p]{P}}{2} \rfloor. \tag{5.19}$$

Die Anzahl der Anschlußpunkte jedes Prozessors entspricht der Netzwerkdimension: $r_i = p$ und ist unabhängig von der Gesamtanzahl der Prozessoren P im Netzwerk. Dies erlaubt bei einmal festgelegter Dimension eine modulare Erweiterung des Netzes.

Die 'Mesh'-Topologie ist geeignet für Algorithmen mit p-dimensionalen, lokalen Datenabhängigkeiten. Beispielsweise können die in Abschnitt 3.2.1 betrachteten 2-dimensionalen lokalen Operationen bei einer Datenzerlegung direkt auf eine 2-dimensionalen 'Mesh'-Topologie abgebildet werden.

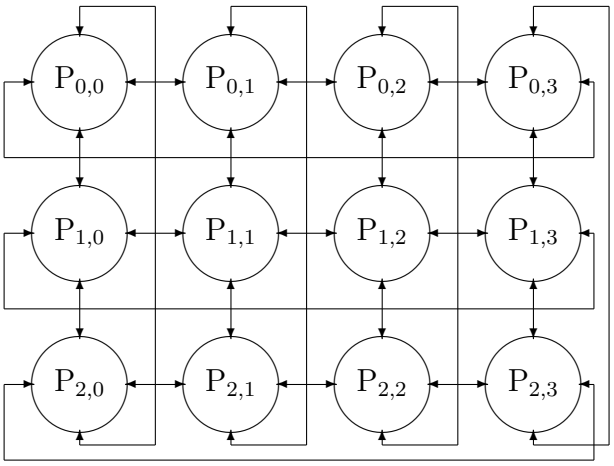

Abbildung 5.12: 2−dimensionales 3×4 'Mesh'-Netzwerk (Torus).

Eine Erweiterung der 'Mesh'-Topologie um eine Broadcast Funktion schlägt Stout [Sto83] zur Reduktion der Rechenzeitordnung vor. Er fordert neben dem Datenaustausch lokal benachbarter Prozessoren ein 'Broadcast', bei dem jeder Prozessor als Sender ausgewählt werden kann, und erreicht damit z.B. eine Reduktion der Ordnung bei der Median- oder Minimumsuche, nicht jedoch beim Sortieren.

Eine weitere als 'Mesh' bezeichnete Verbindungsvariante sieht statt der Ringnetzwerke in jeder Dimension einen gemeinsamen Bus vor [DJ87]. Entsprechend den oben ersichtlichen Unterschieden zwischen globalem Bus und linearer Verbindung muß diese Anordnung als eine Verallgemeinerung des globalen Busses angesehen werden und gehört nicht zu der in diesem Absatz beschriebenen Netzwerkklasse.

'Hypercube':
Wählt man in der Definition der 'Mesh'-Netzwerke (Gleichung 5.16– 5.17) alle m_i zu 2, so erhält man den *'Binary Hypercube'*, der auch als *'Direct Binary n-Cube'* oder *'Boolean Cube'* bezeichnet wird. Er läßt sich übersichtlich mit einer binären Indizierung der Prozessoren $\mathrm{P}_i, i = (i_{p-1}, \ldots, i_i, i_0)_2$ definieren, wobei eine Verbindung zwischen zwei Prozessoren P_i und P_j besteht, wenn die Hammingdistanz $H(i,j)$ genau den Wert 1 annimmt:

$$\begin{aligned} \mathcal{T}_{Hypercube} &= (\mathcal{P}_{Hypercube}, \mathcal{N}_{Hypercube}) && (5.20)\\ \mathcal{P}_{Hypercube} &= \{p_i | i \in I\!N_P, P = 2^p\} && (5.21)\\ \mathcal{N}_{Hypercube} &= \{(p_i, p_j) | j = i \oplus 2^k, \quad p_i \in \mathcal{P}_{Hypercube},\ k \in I\!N_p\}. && (5.22) \end{aligned}$$

Die Distanz $d_{i,j}$ zweier Prozessoren P_i und P_j im Netzwerk ist nach Gleichung 5.22 durch die Hammingdistanz $H(i,j)$ gegeben:

$$d_{i,j\,Hypercube} = H(i,j). \tag{5.23}$$

Bei einem p−dimensionalen 'Hypercube' und somit einem p-stelligen, binären Prozessorindex nimmt diese Distanz maximal den Wert p an:

$$d_{Max\,Hypercube} = \log_2(P) = p. \tag{5.24}$$

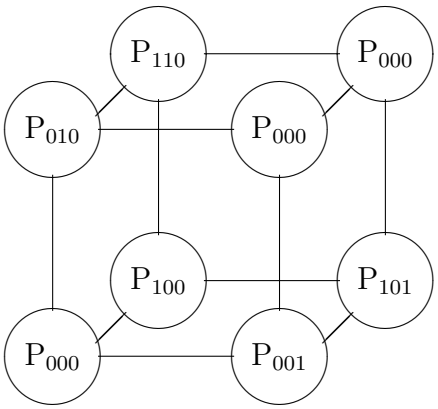

Abbildung 5.13: 'Hypercube'-Verbindungsnetzwerk der Dimension $p = 3$.

Jeder Prozessor eines 'Hypercube'-Systems besitzt Anschlußpunkte r_i zu jeder Dimension der binären Indexmenge, wodurch sich eine logarithmische Abhängigkeit von der Prozessoranzahl P ergibt:

$$r_i = \log_2(P) = p. \tag{5.25}$$

Damit ist bei Verwendung von Prozessoren mit einer festen Anzahl r von Anschlußpunkten die maximale Prozessoranzahl eines 'Hypercube'-Systems $P_{Max\,Hypercube}$ vorgegeben:

$$P_{Max\,Hypercube} = 2^r. \tag{5.26}$$

Ein modularer Ausbau ist somit durch die Anzahl der Anschlußpunkte beschränkt. Durch leichte Modifikationen lassen sich jedoch 'Hypercube'-ähnliche Topologien erzeugen, welche diese Einschränkung zur Modularität nicht mehr aufweisen [RLH88].

Eine tiefergehende Diskussion topologischer Eigenschaften findet man in [SS88]. Dort wird u.a. gezeigt, daß die lokale Verbindung, das Ringnetzwerk und die p-dimensionalen 'Mesh'-Netzwerke direkt in der 'Hypercube'-Topologie enthalten sind. Damit stehen lokale Nachbarschftsverbindungen zur Verfügung und eine direkte Abbildung lokaler Operationen ist bei einer Datenzerlegung möglich.

Die 'Hypercube'-Topologie beinhaltet weiterhin globale Verbindungen. Zur Impementierung der 1. kanonischen Form der globalen Transformationsklasse nach Abschnitt 3.2.2 kann beispielsweise mit der dort vorgenommenen Zerlegung $\mathcal{Z}(\mathcal{G}_G)$ eine Kommunikationsschicht $\mathcal{K}_G^{G,(s)}$ (Gleichung 3.33) direkt auf eine Untermenge von $\mathcal{N}_{Hypercube}$ abgebildet werden. Diese Untermenge umfaßt genau die Verbindungsdimension $p - s$: $\{(p_i, p_j)|j = i \oplus 2^{p-s},\ p_i \in \mathcal{P}_{Hypercube}\}$. Die p Verbindungsdimensionen der 'Hypercube'-Topologie werden damit zur Implementierung der Kommunikationsschichten nur nacheinander benötigt und bleiben die meiste Zeit während der Verarbeitung des Graphen $\mathcal{G}_G$ ungenutzt.

'Perfect-Shuffle' (PS):
'Perfect-Shuffle'-Verbindungsnetzwerke [Sto71], auch häufig als *'Shuffle'* oder *'Shuffle-Exchange'* bezeichnet, bieten globalen Verbindungen durch eine über die 'Shuffle'-Permu-

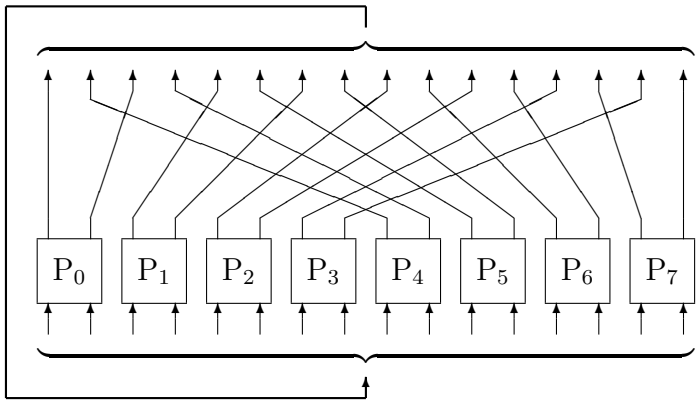

Abbildung 5.14: 'Perfect-Shuffle'-Verbindungsnetzwerk zur Basis 2 für 8 Prozessoren.

tation (Gleichung 3.37) definierte Verbindungstopologie. Diese wird beschrieben mit:

$$\begin{aligned} \mathcal{T}_{PS} &= (\mathcal{P}_{PS}, \mathcal{N}_{PS}) && (5.27)\\ \mathcal{P}_{PS} &= \{p_i | i \in I\!N_P,\ P = B^p, B = 2\} && (5.28)\\ \mathcal{N}_{PS} &= \{(p_i, p_j) | j = \sigma_{p,0}(i) \vee j = \sigma_{p,0}(i) \oplus 1,\ p_i \in \mathcal{P}_{PS}\}. && (5.29) \end{aligned}$$

Ebenso wie bei der 'Hypercube'-Topologie ergibt die maximale Distanz d_{MaxPS} den Logarithmus der Prozessoranzahl:

$$d_{MaxPS} = \log_2(P). \tag{5.30}$$

Zum Aufbau eines 'Shuffle'-Netzwerks benötigt man unabhängig von der Prozessoranzahl $r_i = 2$ Anschlußpunkte an jedem Prozessor. Damit ist eine modulare Erweiterbarkeit dieser Topologie gewährleistet.

Obigem 'Shuffle'-Netzwerk liegt in seiner Kantendefinition über die $\sigma_{p,0}$-Funktion eine Basis $B = 2$ zugrunde. Läßt man eine beliebige Basis $B \in I\!N$ in Gleichung 5.28 zu, so entsteht ein *verallgemeinertes 'Perfect-Shuffle'*-Netzwerk. In [Ree87] wird diese Netzwerkklasse auch als *De Bruijn*-Netzwerk bezeichnet. Auch dieses verallgemeinerte 'Shuffle'-Netzwerk bleibt modular, denn die Anzahl der Anschlußpunkte pro Prozessor bleibt mit $r = B$ konstant und unabhängig von einer Prozessoranzahl P.

Die Definition der 'Shuffle'-Topologie führt zu unidirektionalen Verbindungsleitungen zwischen Prozessoren. Werden bidirektionale Verbindungen benötigt, muß die Kantenmenge $\mathcal{N}_{PS}$ um die inverse 'Shuffle'-Topologie erweitert werden:

$$\begin{aligned} \mathcal{N}_{PS\,bidirektional} = \{(p_i, p_j) | & j = \sigma_{p,0}(i) \vee j = \sigma_{p,0}(i) \oplus 1 \vee \\ & j = \sigma_{p,0}^{-1}(i) \vee j = \sigma_{p,0}^{-1}(i \oplus 1),\ p_i \in \mathcal{P}_{PS}\}. \end{aligned} \tag{5.31}$$

Das entstehende bidirektionale, einstufige Netzwerk eignet sich beispielsweise sehr vorteilhaft zur iterativen Realisierung homogener Formen der im folgenden Abschnitt behandelten mehrstufigen Permutationsnetzwerke, welche in den Verbindungsschichten zwischen Schaltern direkte und inverse 'Shuffle'-Topologien aufweisen.

Parallelrechner mit 'Shuffle'-Verbindungen eignen sich weiterhin zur Implementierung globaler Transformationen. Beispielsweise können die bei der Zerlegung $\mathcal{Z}(\mathcal{G}_{G2})$ entstehenden Kommunikationsschichten $\mathcal{K}_{G2}^{G,(s)}$ (Gleichung 3.47) oder auch die Kommunikationsschichten einer angepaßten Zerlegung von $\mathcal{G}_{G3}$ (Gleichung 3.48) direkt auf die 'Shuffle'-Verbindungen abgebildet werden. Dies ermöglicht eine parallele, auf die Prozessoranzahl P skalierbare Verarbeitung aller Vertreter der in Abschnitt 3.2.2 beschriebenen globalen Transformationen. Mit jeder Kommunikationsschicht werden alle Netzwerkverbindungen genutzt, womit sich eine gleichmäßige und gute Netzwerkauslastung ergibt.

5.3.3 Mehrstufige, schaltbare Netzwerke

Schaltbare Verbindungsnetzwerke ermöglichen die Einstellung vorgebbarer Verbindungen zwischen Ein- und Ausgangsleitungen. Mit der Beschreibung der Ein- und Ausgangsleitungen durch Vektoren $\mathbf{x}$ und $\mathbf{y}$ gleicher Dimension P bildet die elementweise und disjunkte Zuordnung aller Elemente x_i zu y_j eine Permutation $\mathbf{y} = \pi(\mathbf{x})$.

Zur Realisierung vorgegebener Permutationen π sollen mehrstufige, schaltbare Netzwerke angenommen werden, die aus Schalterschichten und dazwischenliegenden Verbindungsschichten mit fester Topologie aufgebaut sind. Die Einstellung einer Permutation erfolgt durch die Schalter. Netzwerke unterscheiden sich in der Anzahl der Verbindungs- und Schalterschichten in Abhängigkeit von der Anzahl P der Anschlußpunkte sowie in der Topologie der Verbindungsschichten und der Funktionalität der Schalter.

Kennzeichen solcher Netzwerke sind die Anzahl `S` der eingebetteten Schalter und die Mächtigkeit der Menge Π einstellbarer Permutationen. Die Anzahl der Schalter dient zur Abschätzung der Netzwerkkosten. Die Mächtigkeit der Permutationsmenge Π ist ein Maß für die Flexibilität. Die höchste Flexibilität ist gegeben, wenn alle Permutationen eingestellt werden können: $\Pi = \Pi_P$. Dann gilt für die Mächtigkeit der Permutationsmenge:

$$|\Pi| = |\Pi_P| = P! \,. \tag{5.32}$$

Ist nur eine Untermenge $\Pi \subset \Pi_P$ einstellbar, so interessiert die Menge der Permutationen, welche ohne Konflikte realisiert werden können, und ob jeder Ausgang bei geeigneten Schalterstellungen von jedem Eingang aus erreichbar ist.

Ist eine Permutation realisierbar, stellt sich die Frage nach dem Aufwand zur Ermittlung der dazu notwendigen Schalterstellungen. Wird nur eine feste Permutation benötigt, kann diese vorab ermittelt und eingestellt werden. Werden nur einige, vordefinierte Permutationen benötigt, können auch diese vorab berechnet werden, zur Laufzeit muß jedoch ein synchrones Umschalten des Netzwerks erfolgen. Werden dynamisch einstellbare Permutationen angestrebt, muß die Ermittlung zugehöriger Schalterstellungen und das synchrone Umschalten zur Laufzeit erfolgen. Dies führt zu einem beachtlichen Laufzeitaufwand neben dem direkt algorithmisch bedingten Aufwand.

Zur Bezeichnung der Netzwerke werden die Anzahl der Schalterschichten als *Stufen* gezählt. Ein dreistufiges, schaltbares Netzwerk besteht damit beispielsweise aus drei Schalterschichten und bis zu vier vor-, zwischen oder nachgeordneten Verbindungsschichten.

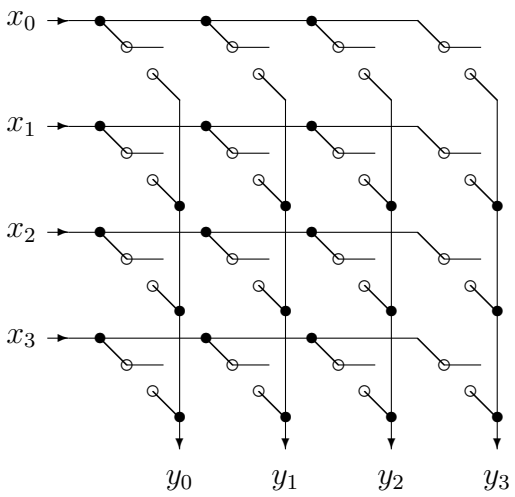

Abbildung 5.15: Kreuzschienenverteiler mit 4 Eingangs- und Ausgangsleitungen.

Kreuzschienenverteiler (KV):
Der *Kreuzschienenverteiler* ist ein universales, mehrstufiges Netz und vergleichbar mit der vollständigen Verbindung einstufiger Netzwerke. Er besteht aus nur einer Schalterschicht und bildet damit ein einstufiges, schaltbares Netz. Jede Eingangsleitung kann über einen Schalter mit jeder beliebige Ausgangsleitung verbunden werden. Damit ist jede Permutation einstellbar, jedoch steigt der Bedarf an Schaltern quadratisch mit der Dimension der Ein- und Ausgangsvektoren:

$$\Pi_{KV} = \Pi_P \tag{5.33}$$
$$\mathtt{S}_{KV} = P^2. \tag{5.34}$$

Zur Vermeidung von Konflikten darf beim Kreuzschienenverteiler nur einer der P Schalter, welche mit einer Ausgangsleitung y_i verbunden sind, geschlossen sein. Denkbar ist jedoch das Aufschalten eines Eingangs an mehrere Ausgangsleitungen, wodurch ein 'Broadcast' ermöglicht wird.

Eine Klasse mehrstufiger, schaltbarer Netzwerke mit $\log_2(P)$ Stufen:
Eine Klasse mehrstufiger, schaltbarer Netzwerke mit jeweils P Ein- und Ausgangsleitungen und $\log_B(P)$, $B = 2$ Stufen erlaubt die Einstellung beliebiger einzelner Ein-/Ausgangsverbindungen.

Abbildung 5.16 zeigt einige Vertreter dieser Netzwerkklasse: Das mit a) gekennzeichnete *'Indirect Binary n-Cube'*-Netzwerk [Pea77] kann iterativ alle Verbindungsdimensionen des einstufigen 'Hypercube'-Netzwerks zur Verfügung stellen. Netzwerk b), welches in seinen Verbindungsschichten jeweils einstufige 'Perfect-Shuffle'-Topologien beinhaltet und somit iterativ auf einem Parallelrechner mit dieser Verbindung simuliert werden kann, wird als *Omega*-Netzwerk bezeichnet. Netzwerk c) zeigt ein von Wu und Feng für topologischen Untersuchungen benutztes Referenznetzwerk [WF80a, WF80b], welches dort als *'Baseline'*- oder *'Reverse-Exchange'*-Netzwerk bezeichnet wird. Netzwerk d) zeigt schließlich einen Vertreter der schaltbaren *Banyan*-Netzwerke [JM88].

Vielfältige Arbeiten beschäftigen sich mit den Eigenschaften dieser Netze [Law75] und

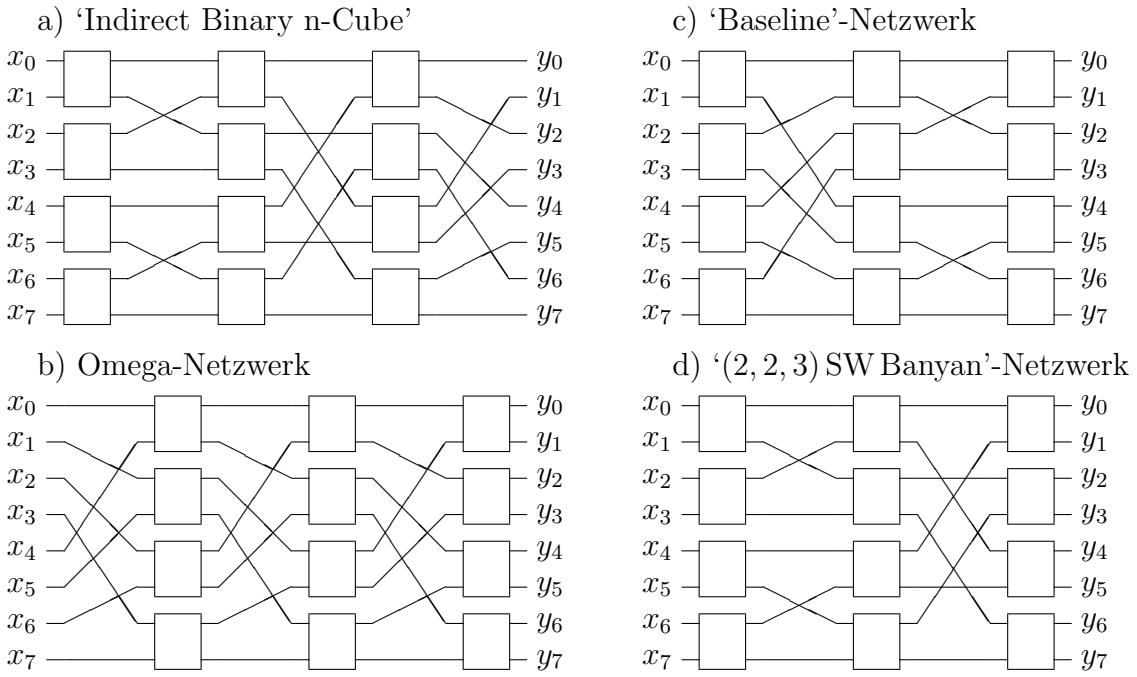

Abbildung 5.16: Beipiele $\log_2(P)$-stufiger, schaltbarer Netzwerke.

weisen topologische Äquivalenzen der Klassenmitglieder nach. Agrawal [Agr83] definiert eine 'Buddy Property', welche als Konstruktions- oder Identifikationsregel dieser Netzwerkklasse dienen kann. Wu und Feng [WF80a, WF80b] zeigen topologische und funktionale Äquivalenzen unterschiedlicher Repräsentanten auf, wobei zur topologischen Äquivalenz ein Permutieren der Ein- und Ausgangsleitungen zur Erzielung einer gleichen Funktion erlaubt ist, bei der funktionalen Äquivalenz jedoch ein identisches Ein-/Ausgangsverhalten zweier Netzwerke gefordert wird.

Für die Permutationsmenge und die Schalteranzahl gilt:

$$\Pi_{\log_2(P)} \subset \Pi_P \tag{5.35}$$

$$\mathtt{S}_{\log_2(P)} = \frac{1}{2} P \cdot \log_2(P). \tag{5.36}$$

Die Menge $\Pi_{\log_2(P)}$ möglicher Permutationen bildet eine Untermenge der vollständigen Permutationsmenge. In [WF80b] sind die erreichbaren Permutationen für das dort als Referenz dienende 'Baseline'-Netzwerk hergeleitet, welche sich durch die Äquivalenzen auf die übrigen Netzwerke übertragen lassen. In [Law75] und [Ste83a] findet man die entsprechenden Permutationsmengen des Omega Netzwerks.

Die Schalter dieser Netzwerkklasse sind — im Gegensatz zum Kreuzschienenverteiler — in Elementen mit je 2 Ein- und Ausgängen organisiert. Gleichung 5.36 zählt solche Schaltelemente, wobei jedes Element durch 4 Schalter realisiert werden kann. Zum Permutieren der beiden Eingangsleitungen werden die Elemente über einen Steuereingang auf *Durchgang* oder *Vertauschen* geschaltet (Abbildung 5.17 a),b)). Soll weiterhin ein 'Broadcast'

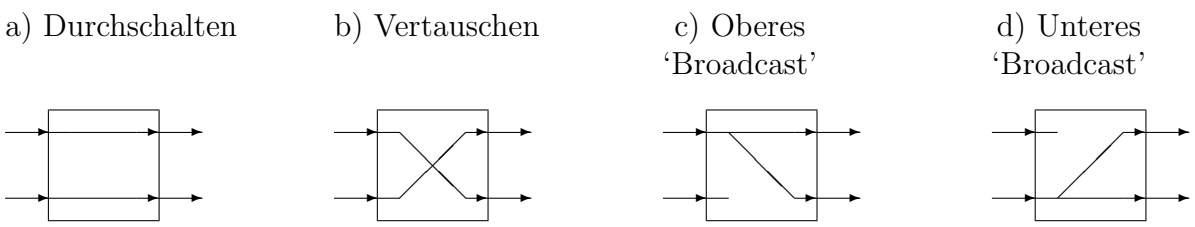

Abbildung 5.17: Zustände eines Schaltelements mehrstufiger, schaltbarer Netzwerke.

möglich sein, müssen zwei weitere Schaltmöglichkeiten, ein *oberes* und ein *unteres 'Broadcast'* [Law75] (Abbildung 5.17 c),d)), vorgesehen werden.

Die Realisierung der 'Broadcast'-Funktion wird anschaulich, wenn man alle Pfade von einem Eingang x_i zu jedem Ausgang y_j betrachtet. Diese Pfade spannen einen binären Baum auf, in dem die Ecken durch Schaltelemente und die Kanten durch Leitungen der Verbindungsschichten gebildet werden. Abbildung 5.18 zeigt diesen eingebetteten Baum in einem Omega-Netzwerk.

Mit den Schaltmöglichkeiten c) und d) aus Abbildung 5.17 ist die Datenverteilung des 'Broadcast' aus der Baumtopologie direkt ersichtlich.

Eine Verallgemeinerung dieser Netzwerkklasse für unterschiedliche Anzahlen von Ein- und Ausgängen unter Verwendung höherdimensionaler Schalterelemente wird in [BA83] vorgeschlagen. Weiterhin findet man in [PL83] eine Verallgemeinerung der Omega-Netzwerke mit Berücksichtigung von Fehlertoleranzen.

Permutationsnetzwerke (PN)
Mehrstufige Permutationsnetzwerke kombinieren einen niedrigen Schalter- und Verbindungsaufwand mit der Möglichkeit, die vollständige Permutationsmenge Π_P zu realisieren. Abbildung 5.19 zeigt die rekursive Struktur eines Permutationsnetzwerks: Hinter einer Schalterschicht werden alle Netzwerkeingänge durch eine inverse 'Shuffle'-Verbindungsstufe auf zwei Permutationsnetzwerke halber Dimension aufgeteilt. Eine 'Shuffle'-Stufe führt die Ausgänge der beiden Netzwerke wieder über eine Schalterstufe zusammen, deren Ausgangsleitungen die Netzwerkausgänge bilden.

Waksman beschreibt in [Wak68] einen Routing-Algorithmus, mit welchem die Schalterstellungen für dieses Netzwerk zur Realisierung einer frei vorgebbaren Permutation π ermittelt werden können.

Die Auflösung der rekursiven Netzwerkstruktur führt an der Eingangsseite zu der in Abbildung 5.16 c) dargestellten 'Baseline'-Topologie. An Ausgangsseite entsteht analog die dazu inverse Topologie. Somit läßt sich ein Permutationsnetzwerk, wie auch Wu in [WF80b] ausführt, durch Kombination eines 'Baseline'- und inversen 'Baseline'-Netzwerks (oder dazu äquivalenter Kombinationen paarweise inverser, $\log_2(P)$-stufiger Netzwerke) realisieren. Die in der Mitte aufeinandertreffenden, zueinander inversen Verbindungsschichten δ und δ^{-1} (Abbildung 5.20) kompensieren einander. Damit treffen zwei Schalterschichten direkt aufeinander, wovon eine Schicht fest eingestellt und damit ganz weggelassen werden kann.

Die Realisierung eines Permutationsnetzwerks aus paarweise inversen, $\log_2(P)$-stufigen Netzwerken mit homogenen Verbindungsschichten ist iterativ auf einem Multiprozes-

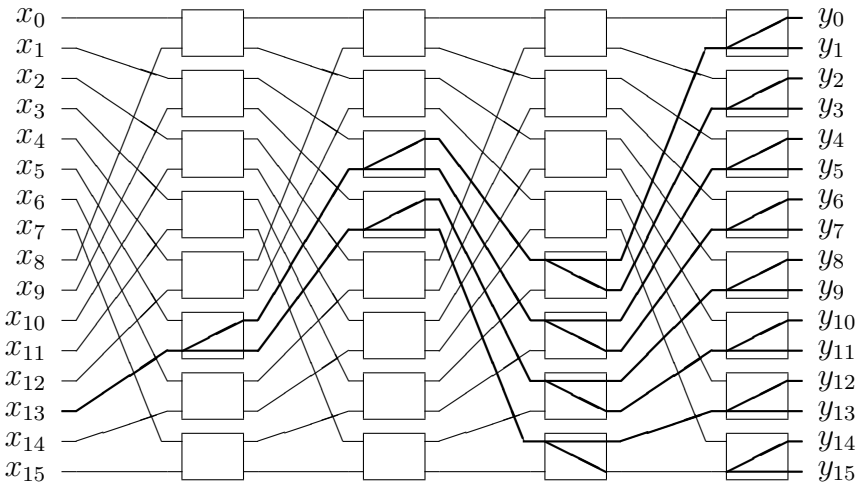

Abbildung 5.18: Eingebetteter Baum in einem Omega-Netzwerk zur Realisierung der 'Broadcast'-Funktion.

sorsystem mit korrespondierendem einstufigem Verbindungsnetzwerk möglich. Dies trifft für ein Permutationsnetzwerk zu, welches aus inversem Omega- und Omega-Netzwerk zusammengesetzt ist. Dessen iterative Realisierung benötigt ein Multiprozessorsystem mit einstufiger, bidirektionaler 'Shuffle'-Topologie, da diese Topologie die benötigten, homogenen Verbindungsschichten direkt und invers enthält (Gleichung 5.31).

Trotz einer höheren Anzahl von Schalter- und Verbindungsschichten der Permutationsnetzwerke gegenüber den $\log_2(P)$-stufigen Netzwerken bleibt die reduzierte Ordnung gegenüber dem Kreuzschienenverteiler erhalten. Die erreichbare Permutationsmenge Π_{PN} wird zur vollständigen Permutationsmenge:

$$\Pi_{PN} = \Pi_P \tag{5.37}$$

$$\mathtt{S}_{PN} = P \cdot (\log_2(P) - \frac{1}{2}). \tag{5.38}$$

Die Ermittlung der Schalterstellungen $\mathtt{S}_{PN}$ einer gegebenen Permutation π wird jedoch zur aufwendigen Aufgabe und kann bei einer dynamischen Rekonfigurierung während der Programmausführung zur Degradierung der Rechenzeitordnung führen. Dies begründet die Suche nach effizienten Routingalgorithmen und nach häufig gebrauchten, einfach einstellbaren Permutationen [Soo83, Sez87].

5.3.4 Mehrstufige, selbstroutende Netzwerke

Die Problematik des Routings mehstufiger, schaltbarer Netzwerke wird in selbstroutenden Netzwerken umgangen, indem die Schalter durch Routingelemente (siehe Abbildung 5.21) ersetzt und Datenpakete mit vorangestellter Zieladresse gesendet werden. In diesen *mehrstufigen, selbstroutenden Netzwerken* geschieht das Routing lokal in den Routingelementen. Der Aufbau einer Ein-/Ausgangsverbindung erfolgt dynamisch für jedes Datenpaket ('Packet switching' [Ree87]).

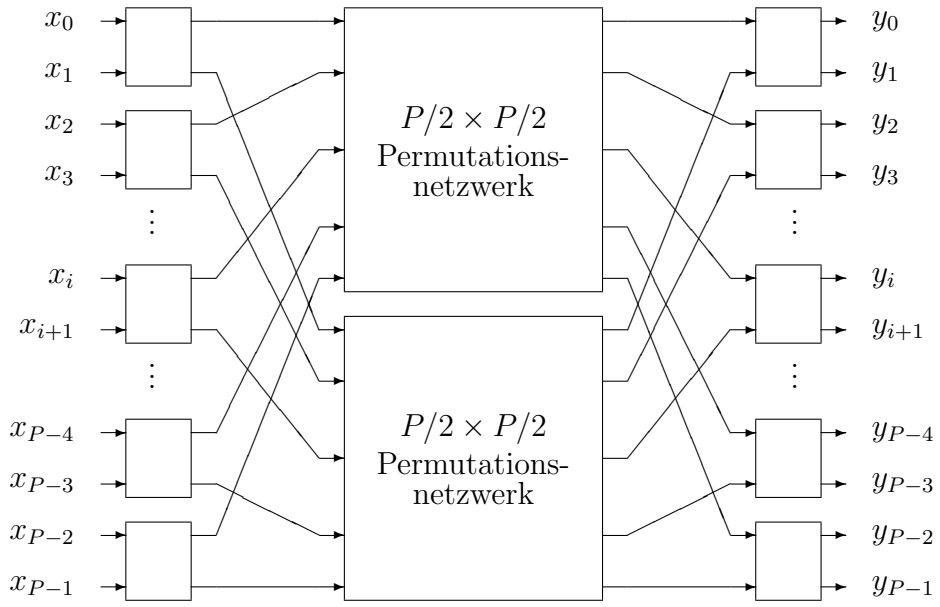

Abbildung 5.19: Rekursive Darstellung eines mehrstufigen Permutationsnetzwerks.

Die daraus resultierende Routingstrategie im Kreuzschienenverteiler ist trivial: Erreicht ein Datenpaket den Schalter mit angeschlossener Zielleitung, muß dieser das Paket in die Ausgangsleitung umleiten.

In den $\log_2(P)$-stufigen Netzen führt das Auswerten sukzessiver Bits der binären Zieladresse j einer Ausgangsleitung y_j in den Routingelementen zu einem einfacher Algorithmus [WF80a]: Ist das ausgewählte Bit auf 0 gesetzt, wird das Datenpaket an Ausgang 0 weitergeleitet, was der Schalterstellung "Durchreichen" (Abbildung 5.17) entspricht. Anderenfalls erfolgt die Ausgabe des Datenpakets über Ausgang 1, in Analogie zur Schalterstellung "Vertauschen". Dieser Algorithmus benötigt zum Routen allein die Zieladresse j einer Ausgangsleitung und ist unabhängig vom Index i der Eingangsleitung x_i.

Mit der Verlagerung des Routings in die Schaltelemente wird eine globale Netzwerksicht aufgegeben. Dadurch kann eine globale Konfliktauflösung beim Aufbau mehrerer Datenpfade zwischen Ein-/Ausgangspaaren (x_i, y_j), wie sie beim Routen schaltbarer Permutationsnetzwerke möglich ist, nicht mehr erreicht werden. Die Konfliktauflösung geschieht nun durch Zwischenspeicherung der Daten in den Routingelementen mit einer konfliktfreien Ausgabe im Zeitmultiplex. Zur Speicherung dienen Puffer, bei deren Überlauf jedoch ein Datenverlust einntritt. Die zur Vermeidung von Verlusten notwendige Puffergröße hängt von der gewählten Netzwerktopologie, den verwendeten Ein-/Ausgangspfaden und der Übertragungstatistik der Daten auf diesen Pfaden ab.

Durch die Datenpufferung zur Konfliktauflösung ist die Übertragungszeit eines Datenpakets im Netzwerk nicht mehr fest, sondern wird abhängig vom Sendezeitpunkt. Andere Pakete und somit der Netzwerkzustand zum Sendezeitpunkt beeinflussen die Übertragung. Existieren mehrere Pfade zum Aufbau einer Verbindung zwischen einem Ein-/Ausgangspaar (x_i, y_j), kann durch die Verzögerung zeitlicher Konfliktauflösungen ein

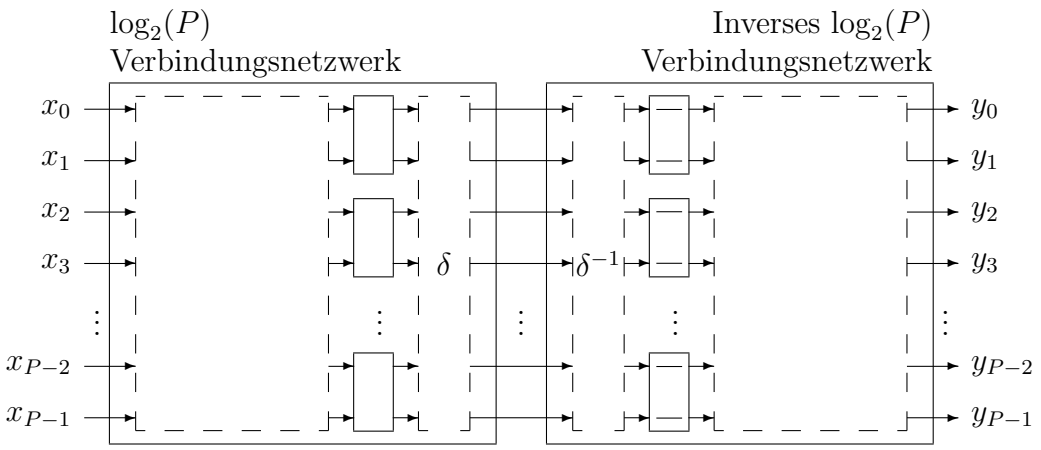

Abbildung 5.20: Konstruktion eines Permutationsnetwerks aus zwei paarweise inversen $\log_2(P)$-stufigen Netzwerken.

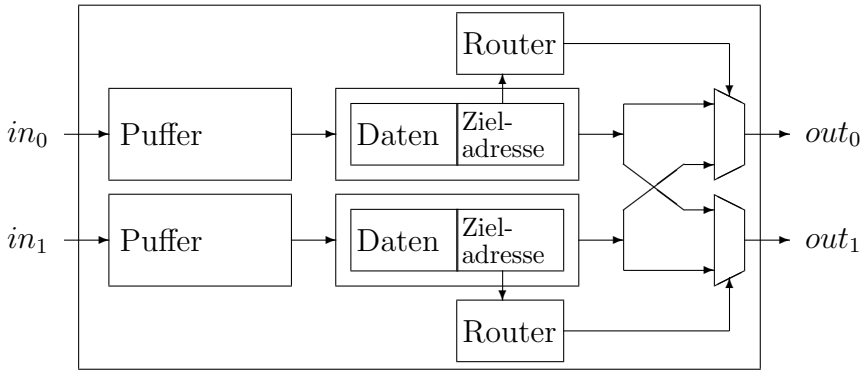

Abbildung 5.21: Routingelement für mehrstufige, selbstroutende Verbindungsnetzwerke.

später gesendetes Paket ein früheres Paket auf einem anderen Pfad im Netz überholen. Zur Gewährung der Datensynchronität müssen also bei selbstroutenden Netzwerken besondere Vorkehrungen getroffen werden.

Kapitel 6

Einordnung und Bewertung paralleler Rechner

Die vielfältigen Prinzipien und Möglichkeiten des Rechnerentwurfs führen zur Notwendigkeit, unterschiedliche Rechnersysteme zu vergleichen. Der Vergleich setzt an zwei Gesichtspunkten an: dem *strukturellen Vergleich* und dem *Leistungsvergleich*.

Der strukturelle Vergleich, d.h. der Vergleich von Rechnersystemen bezüglich ihrer Organisation, führt zur Klassifikation. Hier besteht der Wunsch, z.B. in Analogie zur Botanik, eine Taxonomie aufzubauen, welche eine Einordnung eines Systems und darauf basierend in gewissen Grenzen seine Bewertung ermöglicht.

Der Leistungsvergleich unterschiedlicher Systeme geschieht auf verschiedenen Ebenen. Zunächst können absolute Meßgrößen ermittelt werden, welche auf Hardware-bestimmten Werten oder auf Ausführungszeiten spezieller Algorithmen basieren. Allgemeinere Aussagen erhält man durch parametrisierbare Leistungsmodelle, welche Aussagen über die parallele Ausführung von Algorithmen bei verschiedenen Systemkonfigurationen erlauben.

6.1 Klassifikation von Rechnern

Die Rechnerklassifikation dient zur Einordnung der vielfältigen seriellen und parallelen Systeme. Die Intension — entsprechend dem von Carl Linné im 18. Jahrhundert entwickelten System zur Klassifikation von Pflanzen und Organismen [SNSS83] — eine Taxonomie zur Klassifikation von Computer Architekturen zu entwickeln, führte zu verschiedenen Schemata, welche durch die fortschreitende, technische Entwicklung erweitert und verändert werden.

Taxonomie nach Flynn
Eine erste, immer wieder aufgegriffene Klassifikation von Flynn [Fly66] basiert auf der Anzahl von Instruktionen und Daten, welche während eines Instruktionsschritts einem Rechnersystem zugeführt werden. Er beschreibt vier Rechnerklassen: SISD, SIMD, MISD und MIMD[1], welche sich in der Zuführung von Daten und Instruktionen unterscheiden.

Einzelprozessoren bilden in dieser Taxonomie die Klasse der SISD-Maschinen. Die interne Architektur, welche auf den unterschiedlichen Operationsprinzipien basiert (Abschnitt 5.1), wird nicht beachtet.

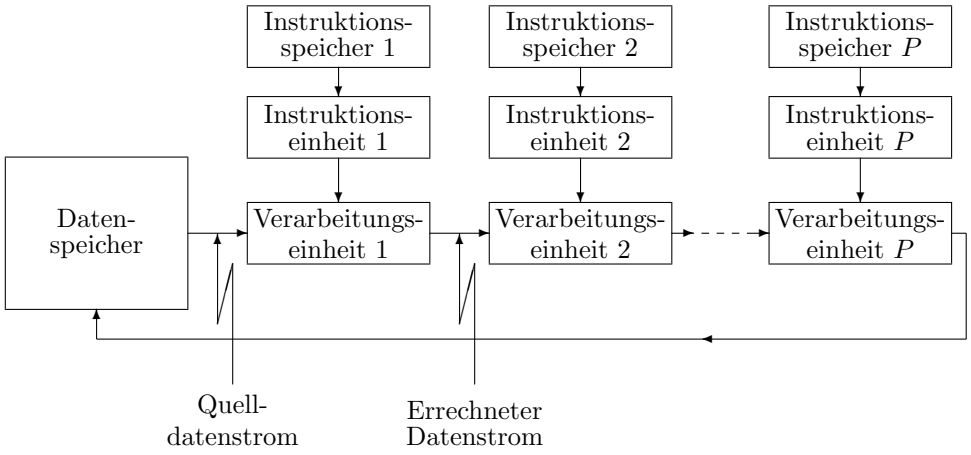

Abbildung 6.1: MISD-System nach Flynn [Fly66].

Die Klasse der SIMD-Rechner beschreibt Multiprozessorsysteme, wo alle Prozessoren den gleichen Instruktionsstrom ausführen. Die früheren Feldrechner (z.B. Illiac IV) aber auch modere Superrechner wie die 'Connection Machine' [Hil85] mit sehr hoher Prozessoranzahl gehören dieser Klasse an. Anwendung findet dieser Rechnertypus besonders im Bereich der Massendatenverarbeitung, wenn viele gleichartige Operandenmengen mit der identischen Instruktionssequenz verarbeitet werden können.

Als MISD Beispiel führt Flynn ein Pipeline-System an, mit dem ein Eingangsdatenstrom gemäß einer Funktionszerlegung verarbeitet wird (siehe Abbildung 6.1). Alle Prozessoren bearbeiten den gleichen, sich jedoch sukzessiv mit jedem Prozessor verändernden Datenstrom. MISD-Rechner zeigen ein einheitliches Erscheinungsbild. Im Gegensatz zur SIMD- und MIMD-Klasse ergibt sich die Systemvielfalt nicht durch verschiedene globale Topologien, sondern durch unterschiedliche Funktionalitäten der Einzelprozessoren. In einem zweiten MISD-Beispiel von Flynn wird allen Prozessoren des Systems der identische Datenstrom zugeführt. Diese Struktur hat jedoch bisher beim Multiprozessorentwurf kaum Beachtung gefunden.

Die MIMD-Klasse umfaßt alle Rechnersysteme aus eigenständigen Prozessoren. Jeder Prozessor führt einen eigenen, unabhängigen Instruktionsstrom aus. Flynns Taxonomie ermöglicht keine Differenzierung zwischen homogenen, d.h. aus gleichartigen Prozessoren aufgebauten, und heterogenen Systemen. Auch wird die Topologie der Verbindungsnetze nicht berücksichtigt.

Strukturelle oder hierarchische Taxonomie
Die große Vielfalt verfügbarer Systeme spiegelt sich in Taxonomien wider, welche die Systeme klassifizieren und in eine hierarchische Baumstruktur einordnen. Dieses Vorgehen kommt der Analogie zur Botanik am nächsten. Hockney und Jesshope [HJ88] beschreiben eine strukturelle Taxonomie, in der sie, beginnend mit der Oberklasse 'Computer', sukzessiv eine Verfeinerung gemäß unterschiedlicher Systemmerkmale vornehmen. Abbildung 6.2 zeigt die Spitze ihres Taxonomiebaumes.

Algebraische Taxonomie

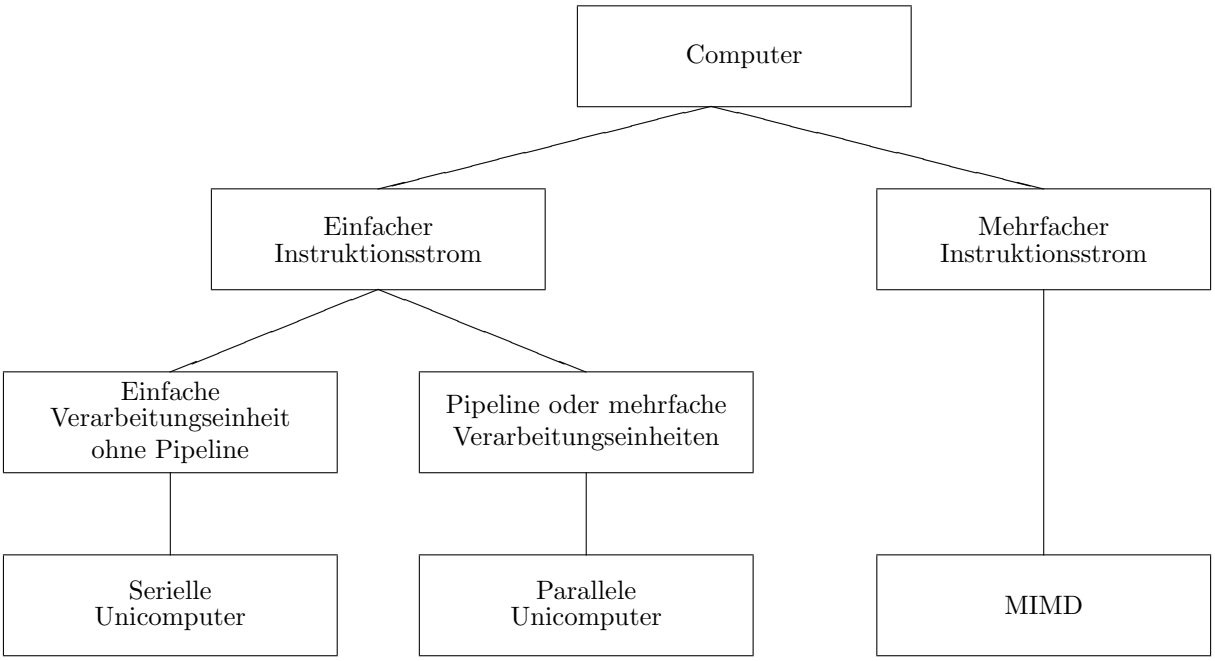

Abbildung 6.2: Hierarchischer Taxonomiebaum nach Hockney und Jesshope [HJ88].

Einen formaleren Weg schlagen Entwickler algebraischer Taxonomien ein. Mit algebraischen Notationen sollen Strukturen und Eigenschaften eines Rechnersystems auf eine "übersichtliche Formel" gebracht werden. Mit wenigen Operanden und Operationen wird eine Beschreibung — vergleichbar mit einer chemischen Formel — angestrebt.

Beispiele algebraischer Taxonomien sind das ECS (Erlangen Classification Scheme) [Hän77], die ASN ('Algebraic-style structural notation') [HJ88] sowie eine von Dasgupta vorgeschlagene hierarchische Syntax [Das90].

In der ASN Notation werden beispielsweise Operanden, welche stellvertretend für die Einheiten eines Rechnersystemes (Speicher, Instruktionswerk, ALU, usw.) stehen, über Operatoren miteinander in Zusammenhang gebracht. Die Operatoren beschreiben die Struktur und Art der Interaktion zwischen den Operanden und beinhalten somit auch die Beschreibung der Verbindungsnetzwerke. In ASN wird beispielsweise ein serieller von Neumann Computer 'C' durch die Formel:

$$C = I[E - M] \tag{6.1}$$

definiert. Der Term '$I[E - M]$' beschreibt eine mit einem Speicher 'M' verbundene Verarbeitungseinheit 'E', welche durch ein Instruktionswerk 'I' gesteuert werden. Der Operator '$-$' repräsentiert die Verbindung zwischen Verarbeitungseinheit und Speicher und somit den Systembus. Das Klammerpaar '[ ]' ordnet dem Instruktionswerk 'I' die Steuerung des '$E - M$' Paares zu.

Mit dem n-fachen Replikatoroperator '$n\{\,\}$' und einer hochgestellten Netzwerkbeschreibung zwischen parallelen Einheiten lassen sich beispielsweise zwei Systeme aus n über

'Shuffle'-Netzwerk verbundenen Einzelprozessoren wie folgt ausdrücken:

$$C(SIMD\,Shuffle) = I[n\{E-M\}^{(Shuffle)}] \quad (6.2)$$
$$C(MIMD\,Shuffle) = n\{I[E-M]\}^{(Shuffle)}. \quad (6.3)$$

Gleichung 6.2 steht für ein SIMD System aus n Prozessoren '$E-M$', welche alle über das gleiche Instruktionswerk 'I' gesteuert werden. In Gleichung 6.3 wird im Gegensatz dazu ein MIMD System gleicher Topologie, jedoch aus unabhängigen Prozessoren mit jeweils eigenem Instruktionswerk beschrieben.

Dieser kurze Anriß der ASN-Syntax soll die Beschreibungsmöglichkeiten einer algebraischen Klassifizierung paralleler Rechnersysteme andeuten. Die Entwicklung dieser Taxonomieformen ist noch in starkem Fluß. Daher bleibt abzuwarten, welche Möglichkeiten sich in Zukunft aus der algebraischen Klassifikation von Systemen für Systementwickler und -anwender bezüglich einer Transparenz und Vergleichbarkeit von Rechnern ergeben werden.

6.2 Bewertung der Systemleistung

6.2.1 Leistungsmessung bei Einzelprozessoren

MIPS, MOPS und MFLOPS:
Zur Leistungsangabe von Einzelprozessoren findet man in der Literatur Meßgrößen, welche die Anzahl ausführbarer Operationen in einem festen Zeitintervall zählen. Die bekanntesten Meßgrößen sind MIPS, MOPS oder MFLOPS [Jes87]:

MIPS (Mega Instructions Per Second) zählt die Anzahl der Instruktionen, welche ein Prozessor pro Sekunde ausführen kann. Der erzielte Meßwert ist direkt vom Instruktionssatz eines Prozessors abhängig.

MOPS (Mega Operations Per Second) zählt die Anzahl der Operationen, welche ein Prozessor pro Sekunde ausführen kann, wobei zusammen mit einem erzielten Meßwert die zugrundegelegte Operation genau beschrieben werden muß. Beispielsweise ist ein Multiplizier-/Addierzyklus eine vielfach wiederholte Kernoperation von Algorithmen zur Signalverarbeitung, sie kann dort als Meßoperation dienen.

MFLOPS (Mega Floating Point Operations Per Second) zählt die Anzahl der Fließkommaoperationen pro Sekunde, wobei auch hier das verwendete Zahlenformat und die getestete arithmetische Operation zu spezifizieren ist.

Die Meßgröße MIPS liefert einen prozessorabhängigen und damit nur bedingt aussagekräftigen Wert zum Vergleich verschiedener Prozessoren. Die Meßgrößen MOPS und MFLOPS ergeben zwar bessere Vergleichswerte für die Leistungsfähigkeit der Verarbeitungseinheiten, erlauben jedoch nur bedingt Vorhersagen zur Leistung einer Programmimplementierung. Ein Faktor ≥ 3 zwischen maximaler und erzielter Leistung liegen durchaus im üblichen Rahmen.

Typische Algorithmen
Leistungswerte, welche einen aussagekräftigereren Vergleich unterschiedlicher Prozessoren

für bestimmte Aufgabenstellungen erlauben, erhält man durch die Ausführung typischer Algorithmen. Häufig verwendete Algorithmen sind [Jes87]:

- Matrix Multiplikation
- Matrix Inversion
- Schnelle Fourier Transformation (FFT).

Wie auch bei den obigen Leistungsgrößen beeinflussen die Randbedingungen der Messung (Problemgröße, Implementierungssprache, Zahlenrepräsentation, Algorithmentyp, verwendete 'Compiler' etc.) die erzielten Meßwerte. Nur mit Angabe der Randbedingungen erlauben erzielte Meßwerte einen Prozessorvergleich.

Programme zur Leistungsmessung ('Benchmarks')
Um bei der Leistungsmessung unabhängig von speziellen Algorithmen zu werden und Prozessoren mit unterschiedlichen Programmanforderungen zu testen, wurden Anwenderprogramme analysiert und statistisch bezüglich der verwendeten Operationen ausgewertet. Solche Ergebnisse dienten als Grundlage zur Entwicklung spezieller Meßprogramme. Die *'Wheatstone'*- und *'Dhrystone'*-Algorithmen sind Beispiele solcher Meßprogramme.

6.2.2 Leistung von Multiprozessorsystemen

Beim Übergang vom Einzelprozessor zu Multiprozessorsystemen ergibt die Summe aus den Einzelleistungen selten eine aussagekräftige Leistungsangabe. Neben den Einzelleistungen bestimmt die Zerlegung eines implementierten Algorithmus (Kapitel 3) die resultierende Leistung eines parallelen Systems. Durch die Zerlegung erfolgt eine Aufteilung des Bedarfs an Rechenleistung, jedoch entsteht ein zusätzlicher Bedarf zur Synchronisation und Kommunikation der verteilten Programmsegmente.

Zeitmodelle
Der Einfluß einer Zerlegung spiegelt sich in der Ausführungszeit von Algorithmen wider. Abhängig von den Randbedingungen einer Implementierung werden verschiedene Modelle zur Beschreibung der Ausführungszeit $t(N, P)$ vorgeschlagen. Dabei wird in den folgenden Modellen ein homogenes, paralleles System aus P Prozessoren und ein Algorithmus der Dimension N vorausgesetzt.

Ein erstes grundlegendes Modell formuliert Amdahl, indem er die für einen Algorithmus benötigte sequentielle Rechenzeit in einen seriellen Anteil $t_S(N)$ und einen parallelen Anteil $t_P(N)$ zerlegt. Bei der parallelen Implementierung muß der serielle Anteil auf jedem Prozessor ausgeführt werden, der parallele Anteil läßt sich hingegen auf P Prozessoren aufteilen:

$$t_{Amdahl}(N, P) = t_S(N) + \frac{t_P(N)}{P}. \tag{6.4}$$

Eine Erweiterung dieses Modells nehmen Flatt und Kennedy [FK89] vor, indem sie Gleichung 6.4 um einen Synchronisations- und Kommunikationsterm $t_0(N, P)$ erweitern,

welcher die Berücksichtigung von Verbindungsnetzwerken im Zeitmodell ermöglicht. Für einen Einzelprozessor $P = 1$ entfällt dieser Term:

$$t_{Flatt}(N,P) = t_S(N) + \frac{t_P(N)}{P} + t_0(N,P) \tag{6.5}$$

mit:

$$t_0(N,1) = 0. \tag{6.6}$$

Ein ähnliches Laufzeitmodell findet man bei Brochard [Bro89], wobei dort die Differenzierung zwischen seriellem und parallelem Anteil nicht vorgenommen, der Zeitanteil t_0 hingegen in einen Kommunikations- und Kontrollterm aufgespalten wird.

Ein Modell von Gustafson [Gus88] wurde für SIMD-Systeme mit sehr hoher Prozessoranzahl aufgestellt. Es nimmt für massiv numerische Probleme eine feste Laufzeit $t_{Gustafson}(N,P) = const$ an. Eine höhere Prozessoranzahl ermöglicht dann die Bearbeitung eines Algorithmus mit erweiterter Problemgröße $N(P)$, so daß diese von der Prozessoranzahl abhängig wird:

$$t_{Gustafson}(N,P) = t_S + t_P(N(P)) = const \tag{6.7}$$

$$t_{Gustafson}(N,1) = t_S + P \cdot t_P(N(P)). \tag{6.8}$$

Die konstante Zeit der parallelen Ausführung wird zur Referenz statt der sequentiellen Zeit in den obigen Modellen. Dadurch wird die hypothetische Ausführungszeit auf einem Einzelprozessor in diesem Modell abhängig von der Prozessoranzahl P.

Zur Beschreibung verfügbarer Prozessoren reichen lineare Modelle zur Berücksichtigung von Rechen- und Kommunikationszeiten nicht aus. Moderne Prozessoren, wie z.B. die in Abschnitt 5.1 vorgestellte Transputerfamilie, erlauben ein paralleles Rechnen und Kommunizieren. Nimmt man eine Programmzerlegung in p Schichten mit unabhängigen Kommunikations- und Rechenanteilen an, ergibt sich die Ausführungszeit $t_{Schicht\,i}$ einer Schicht i aus dem Maximum der Rechen- $t_{R\,i}$ und Kommunikationszeit $t_{K\,i}$:

$$t_{Schicht\,i} = \max(t_{R\,i}(N), t_{K\,i}(N)). \tag{6.9}$$

Damit berechnet sich die nichtlineare Gesamtlaufzeit t_{NL} zu:

$$t_{NL} = \sum_{i=1}^{p} t_{Schicht\,i}. \tag{6.10}$$

'Speedup'

Basierend auf den Zeitmodellen wird der *'Speedup'* S definiert, welcher die Zeit der Algorithmenausführung auf einem Einzelprozessor ($P = 1$) ins Verhältnis setzt zur parallelen Ausführung:

$$S(N,P) := \frac{T(N,1)}{T(N,P)}. \tag{6.11}$$

Wählt man für $T(N,1)$ den besten bekannten seriellen Algorithmus eines Problemes, so wird $S(N,P)$ als *absoluter 'Speedup'* bezeichnet, handelt es sich hingegen um die auf $P = 1$ skalierte Form von $T(N,P)$, erhält man den *relativen 'Speedup'* [Bro89].

Auf die Frage nach der oberen Schranke S_{Max} des 'Speedup' findet man unterschiedliche Antworten. Sie reichen von Beispielen mit superlinearem Verhalten [Par86], (d.h. $O(S_{Max}) > P$) über lineares Verhalten ($O(S_{Max}) = P$) bis, aufgrund der räumlichen 3−dimensionalen Ausdehnung von Rechnern und der damit verbundenen Laufzeiten, hinunter zu $O(S_{Max}) = \sqrt[3]{P}$ [Vit88].

Effizienz
Der auf die Prozessoranzahl normierte 'Speedup' führt zur Effizienz $E(N, P)$:

$$E(N, P) = \frac{S(N, P)}{P}. \tag{6.12}$$

Mit der Annahme eines linear beschränkten 'Speedup' nimmt die Effizienz bei optimaler Systemauslastung den Wert $E_{Max} = 1$ an, sonst liegen die Werte für $E(N, P)$ im Intervall $[0, 1]$.

Die Effizienz ermöglicht durch die Normierung mit P einen übersichtlichen Systemvergleich bei unterschiedlichen Prozessoranzahlen. Sie wird daher in Kapitel 10 für erzielte Meßwerte bei der Implementierung paralleler Algorithmen ermittelt.

Kapitel 7

Ein paralleler Bildverarbeitungsrechner mit schneller Pipeline

Leistungsfähige Bildverarbeitungssysteme zu Vorverarbeitung sind heutzutage meist aus Prozessoren aufgebaut, welche durch das Operationsprinzip des Datenflusses oder der Datensequenz gesteuert werden [Gun90, BMF88]. Diese Systeme beinhalten Pipelinebusse verschiedener Realisierungen als Kommunikationsstruktur zur Übertragung der zu verarbeitenden, regulär strukturierten Bilddaten. Bei Abstimmung des Pipelinebus-Taktes und der Verarbeitungssgeschwindigkeit der datengesteuerten Prozessoren mit der Datengenerierungsrate der Bildgeber ist eine direkte Verarbeitung aufgenommener Bilder möglich. Die Synchronisierung der Prozessoren auf die Bilddaten erfolgt implizit durch die datengesteuerte Verarbeitung.

Im Bereich der Bildvorverarbeitung bietet sich die Pipeline-Verarbeitung aufgrund der klar strukturierten, oft auf einer lokalen Umgebung arbeitenden Algorithmen direkt an. Beim Übergang in die folgenden Verarbeitungsschichten der Bildauswertung (Abbildung 2.1) werden jedoch die Algorithmen komplexer und erlauben damit nicht mehr die direkte Abbildung auf eine einfach strukturierte Pipelineprozessor-Anordnung.

Die Anwendung von Zerlegungsprinzipien, wie z.B. die in Abschnitt 3.2.2 vorgestellte Zerlegung globaler Transformationen, bedingt komplexere Systemstrukturen. Geeignet sind parallele Multiprozessorsysteme, deren Verbindungsnetzwerke die inhärente Topologie eines jeweiligen Algorithmus unterstützen. Bei Verwendung von Kontrollflußprozessoren erhält man weiterhin eine sehr große Flexibilität in der Programmierung, da deren freie Programmierung am weitesten entwickelt ist und dafür die leistungsfähigsten Software-Werkzeuge zur Verfügung stehen.

Die Anlehnung an die physikalisch durch Bildgeber erzeugte sequentielle Datenstruktur führte zusammen mit der gewünschten Flexibilität in den Verarbeitungsstufen zur Merkmalsextraktion und Klassifikation beim Entwurf des Harburger Parallelen Bildverarbeitungssystems [Lan89], im folgenden verkürzt als Harburger System bezeichnet, zur Kombination von Pipeline- und Parallel-Architektur.

Aus globaler Sicht zeigt das System die in Abbildung 7.1 dargestellte Pipeline-Struktur, welche sich an dem Schichtenmodell von Abbildung 2.1 orientiert. Jede Schicht wird durch

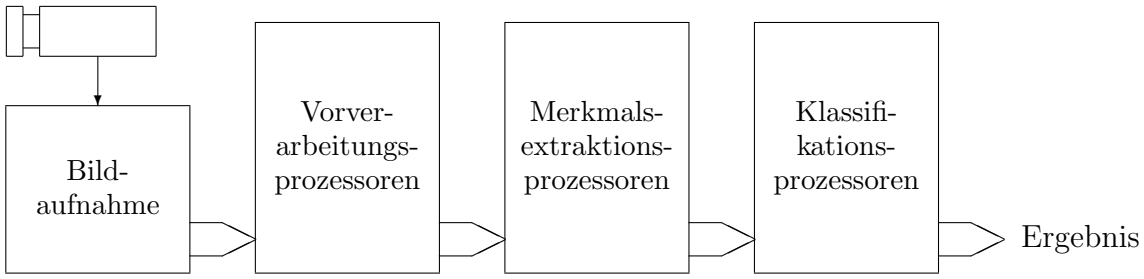

Abbildung 7.1: Globale Pipeline-Organisation des HARBURGER SYSTEMS.

einen korrespondierenden Pipelineprozessor realisiert. Dabei ist an dieser Stelle noch nichts über die innere, lokale Struktur der dargestellten Prozessoren angenommen. Es handelt sich vielmehr um Funktionsblöcke mit einem an die Aufgabenstellung angepaßten Ein-/Ausgangsverhalten.

Erst die lokale Sicht in diese Pipeline-Blöcke zeigt die an jeweilige Algorithmenklassen angepaßte innere Struktur. Im Vorverarbeitungsbereich bietet sich der Einsatz herkömmlicher Pipelineprozessoren an. Dabei kann bei möglicher Funktionszerlegung die Realisierung der Vorverarbeitungsaufgabe durch viele, in Reihe geschaltete Einzelprozessoren erfolgen.

Der globale Block zur Merkmalsextraktion beinhaltet Multiprozessorsysteme mit einstellbarer Verbindungstopologie, welche hierarchisch der Datenflußtopologie untergeordnet sind. Diese Multiprozessorsubsysteme stellen die in dieser Schicht notwendige Flexibilität durch die Möglichkeit der parallelen Programmierung zur Verfügung und erlauben die Modellierung unterschiedlichster Ein-/Ausgangsverhalten auf der Datenflußebene. Weiterhin wird der effiziente Übergang zwischen der fest strukturierten Bilddatenrepräsentation und einer algorithmenabhängig verteilten Datenrepräsentation in einem Subsystem durch eine funktionale Erweiterung des Pipelinebus-Konzepts ermöglicht. Dazu verbindet ein in Kapitel 8 beschriebener Pipelinebus die Verarbeitungsblöcke und stellt schnelle Ein-/Ausgabeoperationen zum Verteilen ('Scattering', 'Broadcast') und Zusammenfügen ('Gathering') von Daten (Abschnitt 5.3.1) zur Verfügung.

Die Klassifikationsaufgabe benötigt meist deutlich weniger Rechenleistung als die vorgeordneten Schichten. Daher ist im HARBURGER SYSTEM keine spezielle Architekturunterstützung für diese Aufgabe vorgesehen. In vielen Fällen wird die Klassifikation vom Systemhost oder der Merkmalsstufe mit erledigt werden. Ist für aufwendige Klassifikationsaufgaben ein eigener, globaler Prozessor erforderlich, besitzt die für die Merkmalsextraktion vorgeschlagene lokale Multiprozessortopologie eine so große Flexibilität, daß sie ebenfalls als innere Struktur eines globalen Klassifikationsprozessors verwendet werden kann.

7.1 Systemübersicht

Die Systemübersicht in Abbildung 7.2 zeigt die Realisierung der globalen Prozessorfunktionen nach Abbildung 7.1 im HARBURGER SYSTEM. Neben einem Funktionsblock zur

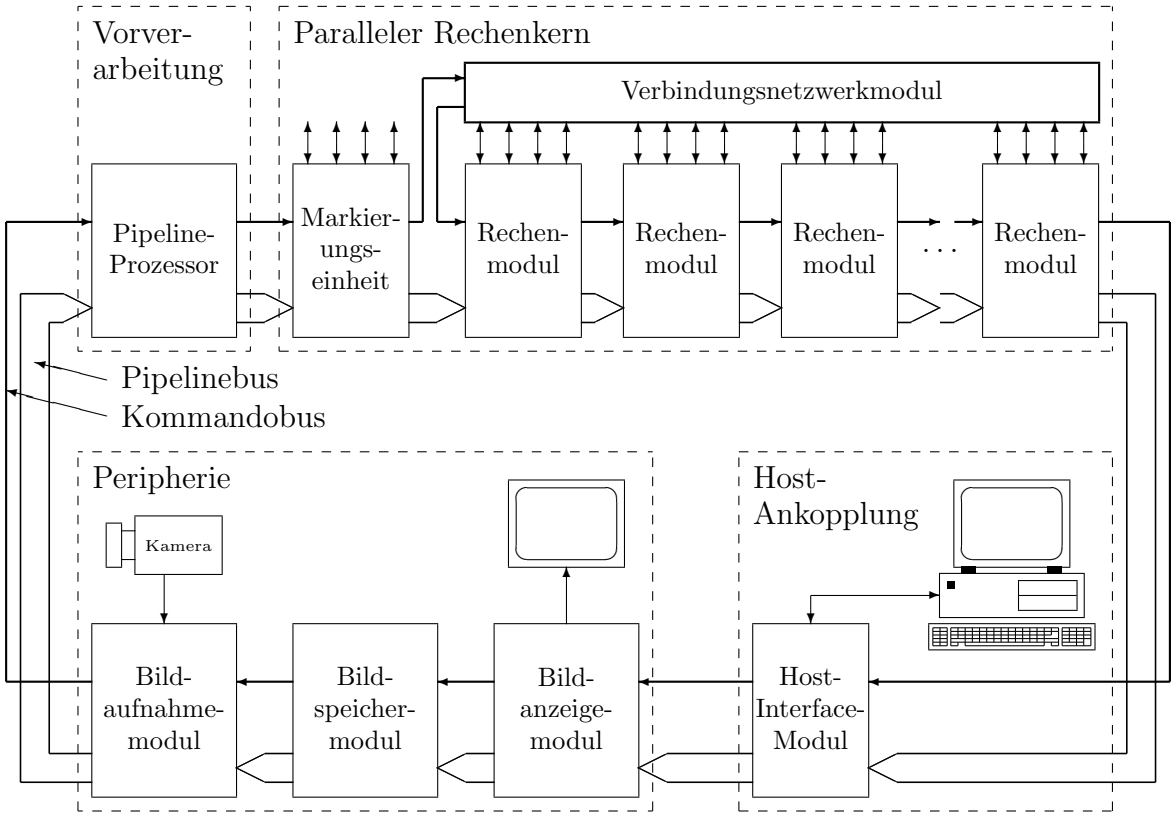

Abbildung 7.2: Systemübersicht des HARBURGER PARALLELEN BILDVERARBEITUNGSSYSTEMS.

Vorverarbeitung sowie einem parallelen Rechenkern zur Merkmalsextraktion und gegebenenfalls zur Klassifikation finden sich in dieser Übersicht Peripherieeinheiten zur Bildaufnahme, -speicherung und -anzeige sowie die Ankopplung an einen übergeordneten Host. Die Abbildung zeigt einen einfachen Systemausbau, in dem jeder Funktionsblock nur einfach vorhanden ist. Größere, leistungsfähigere Systeme können durch mehrfache gleichartige Funktionsblöcke, welche beispielsweise im Pipeline-Betrieb eine algorithmische Funktionszerlegung unterstützen, aufgebaut werden.

Der schnelle Pipelinebus ist in allen Systemen der vorgestellten Klasse die globale Basis für den Transport von Massendaten. Er ermöglicht deren flexible Zuführung zu den einzelnen Verarbeitungsstufen. Dieser Bus bildet die Kommunikationsverbindung zum Laden und Entladen der Bildmassendaten, so daß alle Module, welche während einer Verarbeitung Zugriff auf die Bildinformation benötigen, an diesen Bus angeschlossen sein müssen. Dazu beinhalten solche Module ein intelligentes Pipelinebus-Interface (Siehe Abschnitt 8.4), welches unabhängig von der übrigen Modul-Hardware das Busprotokoll realisiert und die Schnittstellen zum vor- und nachgeordneten Modul zur Verfügung stellt. Der Pipelinebus ist in der Abbildung im Ring geschlossen. Damit ist eine iterative Datenverarbeitung unter Verwendung aller Funktionseinheiten möglich.

7.1.1 Funktionseinheiten des Systems

Die Funktionen einzelner Module des HARBURGER SYSTEMS können nicht isoliert gesehen werden. Vielmehr ist zum Verständnis der lokalen Funktionalität ein Verständnis globaler Systemfunktionen erforderlich. Diese werden bei der Betrachtung von Funktionseinheiten des Systems, welche mehrere Einzelmodule umfassen können, deutlich.

Peripherie
Einen ersten Funktionsblock bildet die *Peripherie* zur Bildaufnahme, -speicherung und -wiedergabe. Die Bildgenerierung erfolgt durch Bildaufnahme- oder Bildspeichermodule, welche ein Bild z.B. zeilensequentiell auf den Pipelinebus ausgeben. Entsprechend erfolgt der Empfang eines verarbeiteten Bildes durch Bildspeicher- oder Bildanzeigemodule, welche einen Datenstrom vom Bus aufnehmen und in gewünschter Art abspeichern. Die Module dieses Peripherieblocks entsprechen in der Funktionalität den Einheiten herkömmlicher Systeme.

Vorverarbeitung
Eine zweite Funktionseinheit umfaßt die *Vorverarbeitung*. Hier finden die unterschiedlichen Konzepte zum Aufbau von Pipeline-Systemen zur Bildvorverarbeitung Anwendung und können in die Architektur des HARBURGER SYSTEMS eingefügt werden. Vielfach werden diese Konzepte schon durch integrierte Bausteine unterstützt. Der Aufbau von Vorverarbeitungssystemen ist jedoch weiterhin ein interessantes Gebiet, wo durch die Synergie aus Technologie und Architektur viele neue Entwicklungen zu erkennen sind. In der vorliegenden Arbeit werden diese Entwicklungen nicht vertieft.

Paralleler Rechenkern
Das tiefere Interesse dieser Arbeit gilt der nächsten Funktionseinheit, einem *parallelen Rechenkern*, welcher aus P Rechenmodulen, einem Verbindungsnetzwerk und einer vorgeschalteten Markierungseinheit aufgebaut ist. Dieser Kern dient als flexibles Multiprozessorsystem zur Ausführung von Bildverarbeitungsalgorithmen der höheren Verarbeitungsschichten. Der Rechenkern zeigt nach außen hin Datenflußverhalten, ist Bestandteil der globalen Verarbeitungspipeline und ermöglicht ein schnelles Zu- und Abführen von Daten.

Intern im Rechenkern bilden die Rechenmodule ein eingebettetes Multiprozessorsystem, welches über den Pipelinebus durch verschiedene Lade- und Entlademodi unterstützt wird. Diese Unterstützung basiert, wie in Kapitel 8 noch detailliert ausgeführt wird, auf einer funktionalen Erweiterung des Pipelinebusses gegenüber herkömmlichen Realisierungen. Spezielle Adressierungsarten werden auf dem Bus durch zusätzliche, parallel zu den Daten übertragene Kontrollinformationen ermöglicht. Das Erzeugen und das Hinzufügen dieser Informationen zu einem in den parallelen Rechenkern hineinfließenden Datenstrom ist Aufgabe der vorgeschalteten Markierungseinheit.

Die physikalische Verbindungstopologie des eingebetteten Multiprozessorsystems kann mit dem Verbindungsnetzwerkmodul eingestellt und damit an die algorithmische Topologie eines parallelen Programms angepaßt werden. Bei der Verwendung modularer Netzwerke ist ein modularer Ausbau der Prozessoranzahl P, angepaßt an eine notwendige Verarbeitungsleistung, möglich. In den Rechenmodulen werden als Prozessoren Transputer

eingesetzt. Diese ermöglichen eine einfache Realisierung von Netzwerken mit Portanzahl $r \leq 4$ und bieten eine gute Unterstützung zur Kommunikation. Mit der beschränkten Portanzahl ist die Einstellung der im vorangegangenen Kapitel vorgestellten 'Mesh'- oder 'Shuffle'-Topologien mit theoretisch beliebigen Prozessoranzahlen, nur eingeschränkt durch Hardware-Randbedingungen, möglich. Weiterhin kann ein 'Hypercube'-Netzwerk mit bis zu $P = 2^4$ Prozessoren eingestellt werden, durch modifizierte Hypercube Topologien (z.B.'Cube Connected Cycles' Netzwerke) lassen sich auch höherdimensionale, Hypercube-ähnliche Netzwerke mit größerer Prozessoranzahl unterstützen.

Der Einsatz von Transputern als Prozessoren im parallelen Rechenkern bietet weitere Vorteile. Wie oben ausgeführt, muß ein Rechenkern in der globalen Systemsicht Datenflußverhalten zeigen. Dazu ist eine Steuerung des globalen Programmablaufs durch die zu verarbeitenden Daten erforderlich. Die Prozessorfamilie der Transputer unterstützt durch das Operationsprinzip des 'Kontrollflusses mit Nebenläufigkeit und Kommunikation' (Abschnitt 5.1) sowohl den traditionellen, flexiblen Kontrollfluß als auch ein Datenflußverhalten, welches den Programmablauf auf die Verfügbarkeit von Daten triggert.

Host-Ankopplung
Eine *Host-Ankopplung* ermöglicht den Zugriff eines Steuerrechners sowohl auf den Kommandobus als auch auf den Pipelinebus des HARBURGER SYSTEMS.

Über den Kommandobus erfolgt die Kommunikation mit einer auf die Systemmodule verteilten Betriebssoftware. Durch den Zugriff des Host auf diesen Bus ist die zentrale Steuerung des Systems durch diesen Rechner möglich.

Der Pipelinebus-Zugriff erlaubt die Integration des Host-Rechners in die Verarbeitungspipeline. Damit können komplexe Berechnungen mit geringen Anforderungen an die Rechenleistung und Datenrate dort lokal ausgeführt werden. Weiterhin ist die Simulation noch nicht vorhandener Funktionseinheiten auf dem Host möglich.

7.1.2 Rechenleistung

Die maximal verfügbare Rechenleistung des HARBURGER SYSTEMS ergibt sich aus der Summe der Rechenleistungen der Funktionsblöcke zur Realisierung der Bildverarbeitungsschichten.

Im Vorverarbeitungsbereich ist diese Leistung durch eine auf der Funktionszerlegung basierenden Pipelinisierung einstellbar. Durch die Ringstruktur des Pipelinebusses kann eine Verarbeitung mit einer massiv Hardware-unterstützten Vorverarbeitungspipeline in einem Durchlauf oder mit geringerem Hardware-Aufwand in mehreren Iterationen durchgeführt werden.

Die maximale Rechenleistung eines parallelen Rechenkerns wird bestimmt durch die Anzahl der verwendeten Rechenmodule sowie die Leistung der Einzelprozessoren. Durch die Auswahl dieser Prozessoren läßt sich ein Rechenkern an bestimmte Aufgabengebiete anpassen, so eignen sich z.B. Signalprozessoren besonders zur Ausführung von Signalverarbeitungsalgorithmen. Reicht die verfügbare Rechenleistung der Einzelprozessoren nicht aus, läßt sie sich durch zugeordnete Coprozessoren (Vektoreinheiten, Arithmetikprozessoren, etc.) noch steigern. Im HARBURGER SYSTEM wurden Transputer als Einzelprozessoren ausgewählt. Insbesondere der Typ `T800` bietet durch einen eingebauten

Arithmetikprozessor eine hohe Rechenleistung bei der Fest- und Gleitkommarechnung (10 MIPS, 1,5 MFLOPS mittlere Raten beim 20 MHz-Typ [INM88b]).

7.1.3 Kommunikationsleistung

Die nutzbare Kommunikationsleistung nimmt bei großen Prozessoranzahlen den entscheidenden Einfluß auf die Leistungsdaten eines Systems. Neben der Bandbreite ist der Entwurf der Topologie und Funktionalität der Kommunikationsverbindungen entscheidend zur Vermeidung von Systemengpässen. Nur mit einer Kommunikation, welche an eine Aufgabe angepaßt ist, läßt sich eine hohe maximale Rechenleistung ausnutzen.

Um keine Engstellen bei Kommunikation im HARBURGER SYSTEMS entstehen zu lassen, werden in Anlehnung an die Aufgaben und Anforderungen an ein Multiprozessorsystem (Abschnitt 5.3.1) drei Kommunikationsstrukturen im System vorgesehen:

Kommandobus: Der Kommandobus dient zur Kommunikation einer verteilten Betriebssoftware und überträgt strukturierte Datenblöcke. Diese Blöcke werden gemäß der Funktionalität eines Zielmoduls interpretiert. Bei der Übertragung ist die Realisierung eines komlexen Protokolls, welches deutlich über der Hardware-Schicht liegt, entscheidender als die Übertragungsgeschwindigkeit. Diese sollte zwar möglichst hoch sein, besitzt jedoch nicht den Stellenwert wie bei den folgenden beiden Kommunikationsstrukturen.

Der Kommandobus des HARBURGER SYSTEMS ist durch einen zyklisch geschlossenen Ringbus realisiert, wobei zur physikalischen Verbindung Transputer-Links Verwendung finden. Teilweise basieren diese Link-Verbindungen, um schnelle Transputer-Links für algorithmische Aufgaben frei zu halten, auf zusätzlichen, über Unterbrechungsroutinen (Interrupt) gesteuerten Link-Adaptoren. Dadurch ergibt sich eine Übertragungsrate von ca. 37 $kBytes/s$. Die Funktionen und Protokolle des Busses werden durch lokale Betriebssoftware-Prozesse auf jedem Modul realisiert (Kapitel 9).

Pipelinebus: Der schnelle Pipelinebus des HARBURGER SYSTEMS dient in der globalen Systemsicht als Schnittstelle zwischen den einzelnen Verarbeitungsschichten. Seine Funktionalität ist pixelorientiert und somit fest an die Bildrepräsentation angelehnt.

Durch die Erweiterung seiner Funktionalität zur Realisierung von Lade- und Entlademodi (Abschnitt 8) unterstützt er zusätzlich das in einen Rechenkern eingebettete Multiprozessorsystem und damit die Realisierung von Bildverarbeitungsschichten höherer Ebene.

Der bisherige Hardware-Aufbau weist, angepaßt an Standard Schwarz/Weiß-Video-Raten, eine Datenübertragungsrate von 10 $MBytes/s$ bei einer Datenwortbreite von einem Byte auf. Mit heutiger Technologie ist es jedoch möglich, die Übertragungsrate durch Erhöhung des Pipeline-Taktes und der Wortbreite um einen Gesamtfaktor von ca. 10 zu steigern.

Verbindungsnetzwerk: Das Verbindungsnetzwerkmodul bildet beim eingebetteten Multiprozessorsystem eines parallelen Rechenkerns die algorithmische Verbindung. Im HARBURGER SYSTEM basiert dieses Netzwerk auf Transputer-Links. Die

maximale Übertragungsbandbreite des Netzwerks wächst mit jedem zusätzlich im Kern eingesetzten Rechenmodul, die Nutzbarkeit dieser Bandbreite resultiert aus der Abstimmung zwischen Netzwerktopologie und Algorithmen (Abschnitt 5.3). Bei einer angenommenen bidirektionalen Bandbreite jedes Transputer-Links von $B_{Link} = 2.35\,MBytes/s$ [INM88b] ergibt sich bei P Prozessoren und 4 Links pro Prozessor die maximale Gesamtbandbreite B des Netzwerks zu:

$$B_{Max} = \frac{1}{2} P \cdot 4 B_{Link} = P \cdot 9.4\,MBytes/s. \tag{7.1}$$

Diese Kommunikationsleistung steht bei Transputern durch die eigenständige Link-Hardware nahezu parallel neben der Rechenleistung zur Verfügung.

7.2 Module des Systems

Trotz der zunächst erscheinenden Modulvielfalt des HARBURGER SYSTEMS lassen sich die von Peripheriegeräten unhabhängigen Module mit nur wenigen, einheitlich aufgebauten Einheiten realisieren. Die Anpassung an die jeweiligen Aufgaben wird bei geeignet entworfener Hardware durch Treibersoftware und Firmware[1] vorgenommen. Durch solch einen modularen Aufbau lassen sich Markierungseinheit, Rechenmodul sowie Bildspeichermodul mit der gleichen Hardware realisieren.

Auch im Vorverarbeitungsbereich geht die Entwicklung in Richtung einer Modularisierung. Zum Aufbau von Pipelineprozessoren sind Einheiten zur Operandengenerierung sowie verschiedene Verarbeitungseinheiten als VLSI-Bausteine verfügbar [LSI90]. Diese können zur Realisierung benötigter Funktionen kombiniert werden. Eine andere Möglichkeit zum Aufbau modularer Vorverarbeitungseinheiten basiert auf dem Einsatz eines SIMD Zeilenprozessorsystems, welches in einer Ringtopologie parallel eine komplette Bildzeile bearbeitet. Eine Funktionsanpassung kann durch systemnahe Bibliotheksprogramme erfolgen. Mit heutiger VLSI Technologie und einem Design mit 1-*Bit*-Prozessoren ist die Realisierung eines 512-Prozessorsystems auf einem einzelnen Chip möglich [CCJ90]. In dieser Arbeit werden diese Möglichkeiten der Vorverarbeitung nicht weiter verfolgt.

Zur Anpassung von Peripheriegeräten (z.B. Kamera, Monitor) sind spezielle Module erforderlich, welche benötigte gerätespezifische Schnittstellen zur Verfügung stellen. Beim Entwurf solcher Module für das HARBURGER SYSTEM können modulare Teileinheiten wie z.B. das Pipelinebus-Interface eingefügt werden, so daß teilweise auch hier eine Modularität erzielt wird.

Zum Nachweis der vorgestellten Funktionalität sowie als Basis zur Implementierung unterschiedlicher Bildverarbeitungsalgorithmen wurde an der TU Hamburg-Harburg ein Prototyp des HARBURGER SYSTEMS aufgebaut. Dieser Prototyp zeigt die Funktionsfähigkeit

[1] Mit *Firmware* soll im folgenden alle Software bezeichnet werden, welche auf einem Modul nicht vom Prozessor ausgeführt, sondern zur Programmierung von Hardware-Einheiten verwendet wird. Somit gehört beispielsweise die Software, welche die Funktionen logischer Bausteine (PALs, GALs, EPLDs, LCAs usw.) beschreibt, zur Firmware eines Systems und auch Speicherinhalte schneller PROM-Bausteinen zur Steuerung endlicher Automaten. Treiberprogramme zur Software Ansteuerung von Hardware-Einheiten durch einen Modulprozessor, gelegentlich auch als Firmware angesprochen, sollen hingegen als *Treibersoftware* bezeichnet werden.

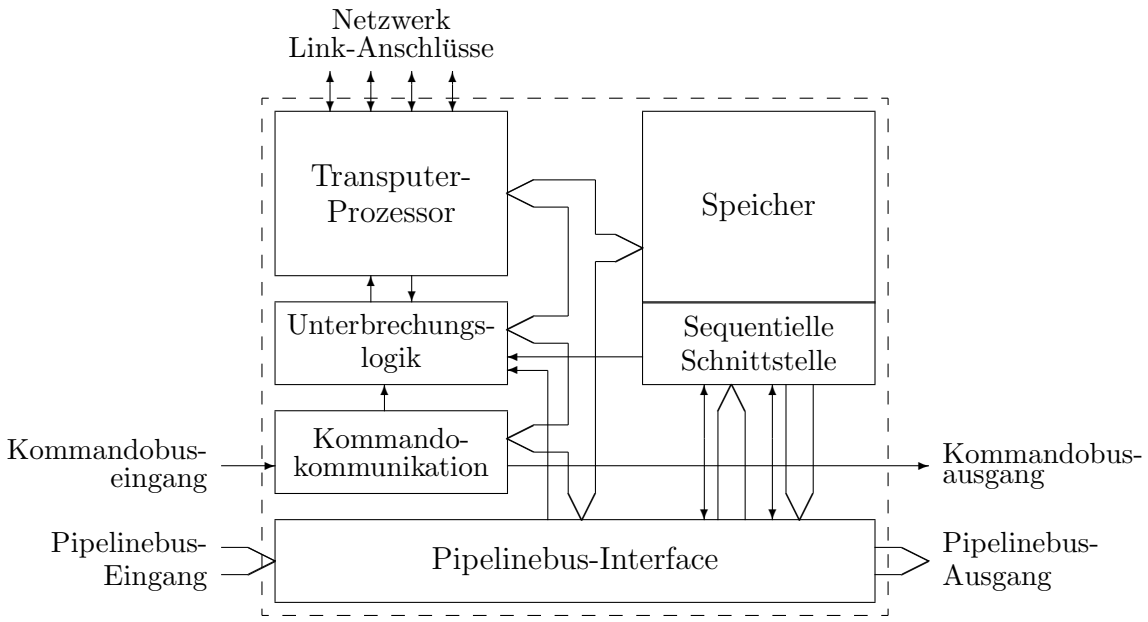

Abbildung 7.3: Blockstruktur eines Transputer-Moduls.

des vorgestellten Konzepts auf. Weiterhin wurden mit diesem System die in Kapitel 10 beschriebenen Implementierungen und Leistungsmessungen durchgeführt.

Bei der folgenden Vorstellung der Blockstrukturen bisher realisierter Module des Prototypen ist zur Erläuterung der Hardware-Funktionen häufig ein Verweis auf die zum Betrieb notwendige Firm- und Betriebssoftware nötig. Dies resultiert aus der engen Verzahnung dieser drei Ebenen im HARBURGER SYSTEM, wo erst die Kombination aus Hard-, Firm- und Betriebssoftware die gewünschte, für Anwenderprozesse einfach handhabbare Funktionalität ergibt.

7.2.1 Transputer-Modul

Bildspeicher, Markierungseinheit und Rechenmodul des HARBURGER SYSTEMS können durch die gleiche Hardware realisiert werden. Das dazu entworfene Transputer-Modul erhält seine unterschiedliche Funktionalität durch Firmware und eine lokale Betriebssoftware.

Abbildung 7.3 zeigt ein Transputer-Modul. Es weist eine funktionale Schichtung auf. Die oberste Schicht bilden Transputer und Speicher. Diese Schicht ist der Zielbereich für die Implementierung von Programmen. Bei Rechenmodulen wird in diesem Bereich die Implementierung der Algorithmen durchgeführt. Eine Markierungseinheit führt hier die Programme zur Generierung von Markierungsinformation aus und speichert Felder mit dieser Information. Die Nutzung dieses Modulteils steht unter der Kontrolle einer lokalen Betriebssoftware (Kapitel 9), welche die Ausführung von Anwenderprogrammen erlaubt und problemorientierte, Hardware-unabhängige Schnittstellen zum Ansprechen der weiteren Modulschichten bereitstellt.

Die mittlere Schicht des Transputer-Moduls besteht aus Hardware zur Kommandokom-

munikation, aus Unterbrechungslogik und einer sequentieller Speicherschnittstelle. Sie wird von der lokalen Betriebssoftware über zugeordnete Treiber verwaltet.

Die Kommandokommunikation bildet physikalisch den Kommandobus zur Systemsteuerung. Diese Hardware beinhaltet im wesentlichen zusätzliche Link-Schnittstellen, um die primären 4 Transputer-Links für algorithmische Belange frei zu halten.

Die Unterbrechungslogik unterstützt die Behandlung dynamischer Ereignisse (Interrupts), welche bei der Kommandokommunikation, beim Transfer von Datenblöcken zwischen der Speicherschnittstelle und dem Pipelinebus-Interface sowie in diesem Interface selbst auftreten können.

Die sequentielle Speicherschnittstelle ermöglicht den schnellen Transfer von Datenströmen vom Pipelinebus-Eingang zum Speicher des Moduls sowie von diesem Speicher zum Pipelinebus-Ausgang jeweils über das Pipelinebus-Interface. Diese Schnittstelle bestimmt neben dem Pipelinebus-Takt die Ein- und Ausgansbandbreite zum Laden und Entladen eines Moduls.

Eine erste Realisierung dieser sequentiellen Schnittstelle sah eine durch einen 'Block-Move'-Befehl des Transputers unterstützte Transfereinheit vor, welche jedoch noch Rechenzeit des Transputer für einen Datenkopiervorgang zwischen Speicher und Transfereinheit benötigt und bei sehr hohen Datenraten noch eine Engstelle bildet. Die Auflösung dieser Engstelle durch Verwendung einer DMA-Einheit hätte an dieser Stelle zwar einen prozessorunabhängigen, schnellen Datentransfer erlaubt, jedoch einen hohen Realisierungsaufwand bedeutet und den Prozessorzugriff auf den Speicher während eines Transfers stark behindert.

In einer zweiten Realisierung des Transputer-Moduls [Lut90] wurden, um die angesprochene Engstelle zu entfernen, Video-RAM Bausteine zum Aufbau des Speichers verwendet. Diese Bausteine beinhalten zwei weitgehend unabhängige Zugriffschnittstellen. Die erste Schnittstelle stellt den herkömmlichen, frei addressierbaren Adreß-/Datenbus zur Verfügung, die zweite Schnittstelle ermöglicht dazu parallel einen schnellen sequentiellen Zugriff auf ganze Speicherzeilen. Durch die Anbindung des Transputers über die erste und des Pipelinebus-Interface über die zweite Schnittstelle an den Speicher wird eine weitgehende Entkopplung zwischen dem Prozessorzugriff und dem Zugriff zum Laden und Entladen erreicht.

In der untersten Schicht, dem Pipelinebus-Interface, werden die benötigten Lade- und Entlademodi realisiert. Es wird in seiner Struktur in Kapitel 8 vorgestellt.

7.2.2 Bildaufnahmemodul

Zum Einlesen von Bilddaten in das Harburger System wurde ein Bildaufnahmemodul entworfen, welches in seiner Struktur für unterschiedliche Bild- oder Datengeber geeignet ist. Abbildung 7.4 zeigt das Blockschaltbild der realisierten Hardware zum Anschluß von Standard-Video-Signalgebern (z.B. Kamera).

Der Transputer-Prozessor samt Speicher dient zur Steuerung des Moduls. Dort wird die angepaßte, lokale Betriebssoftware ausgeführt. Eine Verarbeitung digitalisierter Video-Daten durch diesen Transputer ist nicht vorgesehen. Zwei Links dienen als

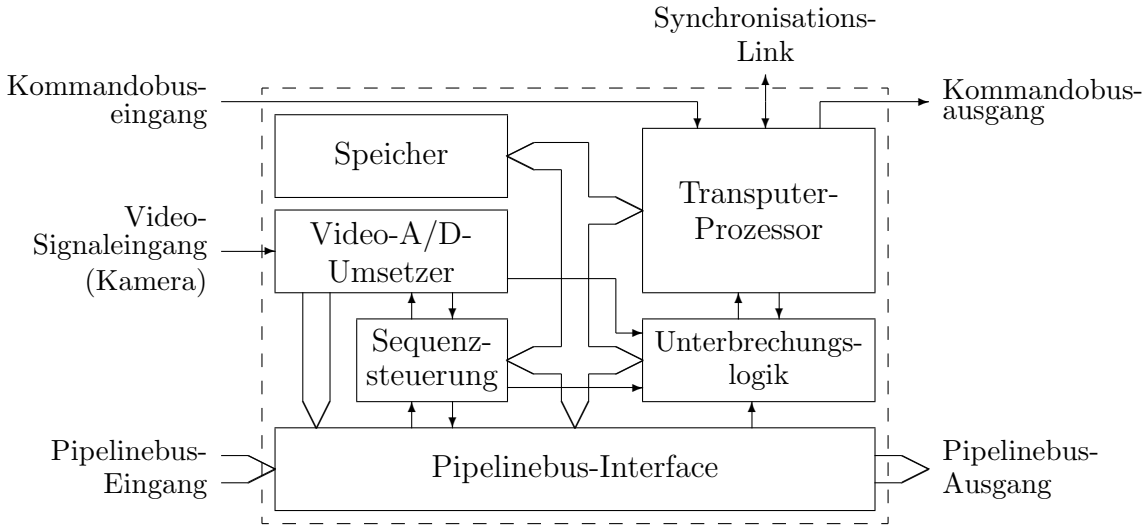

Abbildung 7.4: Blockstruktur eines Bildaufnahmemoduls.

Kommandobusein- und -ausgang. Ein dritter Link erlaubt die Synchronisierung der Bildaufnahme auf eine externe Anforderung. Dieser Link kann mit einem Modul, welches Information über das globale zeitliche Verhalten des Systems besitzt, verbunden werden (z.B. mit einer Markierungseinheit), um von dort aus die Bildaufnahme zu triggern.

Über eine Unterbrechungslogik reagiert der Prozessor auf dynamische Ereignisse. Viele dieser Ereignisse dienen der Steuerung interner Abläufe bei der Bildaufnahme und sind nach außen hin nicht transparent. Sie beinhalten die Reaktion auf Zeilen- und Bildsynchronsignale. In diese Abläufe ist die Maskierung eines aufgenommenen Bildes mit einstellbarem Fenster eingearbeitet, wodurch selektiv ein Teilbild aus einem Gesamtbild entnommen werden kann. Weiterhin ist eine einfache, zeilenweise Markierung des Bildes statt mit einer Markierungseinheit auch schon mit diesem Bildaufnahmemodul möglich.

Zum Start einer Bildaufnahme dient ein vom Pipelinebus-Interface generiertes Ereignis, welches die Ankunft eines speziellen Steuerdatums auf dem Bus signalisiert. Über dieses Ereignis kann, alternativ zum oben angesprochenen Synchronisations-Link, die Triggerung einer Bildaufnahme erfolgen.

Mit der Sequenzsteuerung werden Signale, welche den Datentransfer vom Umsetzer zum Pipelinebus-Interface steuern, zeitlich synchron mit dem Bildgeber erzeugt. Im Zusammenspiel dieses Funktionsblocks mit der Unterbrechungslogik und den Treiberprozessen der lokalen Betriebssoftware erfolgt die einstellbare Maskierung und Markierung des Eingangsbildes.

Der Datenpfad vom Bildgeber über Video-A/D-Umsetzer und Pipelinebus-Interface zum Pipelinebus-Ausgang beinhaltet keinen Bildspeicher. Lediglich im Pipelinebus-Interface kann eine Datenpufferung, z.B. über einen FIFO-Speicher, zur Anpassung des mittleren Pipelinebus- und Umsetzertaktes vorgenommen werden. Da Video-Bildgeber in einem festen Zeitraster arbeiten und somit, zur Vermeidung von Verlusten, Daten direkt auf den Pipelinebus ausgegeben werden müssen, führt die Sequenzsteuerung eine Überprüfung der Echtzeitbedingung durch. Sobald ein fehlerhaftes globales Zeitverhalten

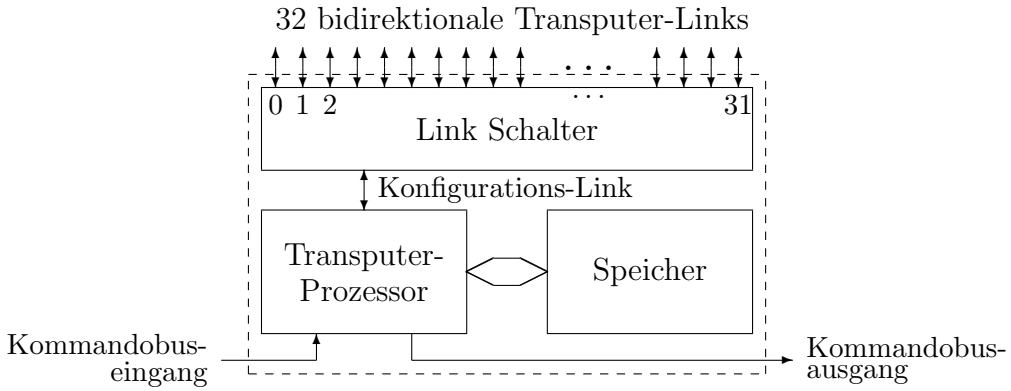

Abbildung 7.5: Verbindungsnetzwerkmodul zur Software-Konfigurierung von Verbindungsnetzwerken.

einen unerwünschten Datenverlust verursacht, generiert die lokale Betriebssoftware eine Meldung und sendet diese über den Kommandobus an den Systemhost. Von dort aus kann dann geeignet auf dieses Fehlerereignis reagiert werden.

Der Entwurf eines Video-A/D-Unsetzers wird durch den verwendeten Bildgeber bestimmt. Zum Aufbau von Video-Umsetzern sind hochintegrierte Bausteine oder Module [Ana88, Boo90] erhältlich, welche den Entwurf dieser Einheiten mit nur wenigen Bauteilen ermöglichen.

Das Pipelinebus-Interface muß während der Bildaufnahme ein schnelles Entladen digitalisierter Bilddaten unterstützen. Ein Laden von Datenströmen wird nicht benötigt. Lediglich der Empfang einzelner Steuerdaten zur Synchronisation der Bildaufnahme über die Unterbrechungslogik ist vorgesehen, welche ein zugehöriges Triggerereignis auslösen.

7.2.3 Verbindungsnetzwerkmodul

Das Verbindungsnetzwerkmodul des Harburger Systems [Mil89] ermöglicht eine durch Software einstellbare Verbindungstopologie zwischen den Rechenmodulen eines Rechenkerns. Es ist in den Kommandobus zur Systemsteuerung integriert. Die Einstellung einer gewünschten Netzwerktopologie erfolgt global durch den Host. Die auf Prozessor und Speicher installierte lokale Betriebssoftware erlaubt das Laden und Aktivieren dieser Topologien über modulspezifische Kommandos.

Die Einstellung einer Topologie geschieht durch Programmierung eines kommerziell verfügbaren 'Link-Switch'-Schalterbausteins [INM88b]. Dieser Baustein erlaubt als 32×32 einstufiges, schaltbares Netzwerk die Permutation der 32 Link-Eingängen auf die Ausgänge. Beim Anschluß aller Transputer-Links an dieses Verbindungsnetzwerkmodul können somit bis zu 8 Rechenmodule eines Rechenkerns in einer gewünschten, jedoch durch die 4 Links eines Rechenmoduls begrenzten Topologie verbunden werden. Eine Erhöhung der Rechenmodulanzahl ist durch Reduzierung der an das Netzwerk angeschlossenen Links oder durch Kaskadierung mehrerer Schalterbausteine möglich, z.B. als mehrstufiges, schaltbares Verbindungsnetzwerk (Abschnitt 5.3.3).

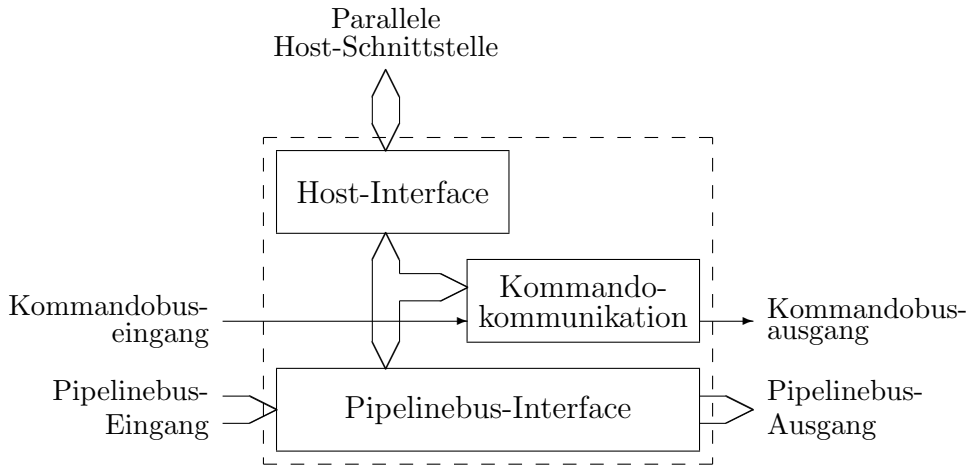

Abbildung 7.6: Modul zur Ankopplung eines Host-Rechners an das HARBURGER SYSTEM.

7.2.4 Host-Interface-Modul

Um das HARBURGER SYSTEM für einen übergeordeten Host-Rechner transparent zu machen, wurde ein Interface-Modul zu dessen Ankopplung entworfen (Abbildung 7.6). Eine universell anpaßbare, parallele Schnittstelle des Interface gibt dem Host Zugriff auf den Kommando- und den Pipelinebus des Systems.

Damit kann der Host über den Kommandobusausgang funktionsspezifische Kommandos zur Systemsteuerung an die lokalen, modulangepaßten Betriebssoftware-Prozesse senden. Weiterhin erreichen dort generierte Statusinformationen den Host über den Kommandobuseingang, so daß dieser in geeigneter Weise reagieren kann.

Das Pipelinebus-Interface dieses Moduls emöglicht dem Host-Rechner den direkten Zugriff auf den schnellen Pipelinebus. Durch diesen Zugiff ist die Einbindung des Host in die Verarbeitungspipeline möglich, wobei diese Einbindung nur bei einer Host-Verarbeitung mit geringem Rechenzeitbedarf oder zu Testzwecken sinnvoll ist.

Ein spezieller, durch Firmware gesteuerter Testmodus des Interface ermöglicht die Überprüfung von Busaktivitäten. In diesem Modus können vorbeifließende Daten (bei großen Datenmengen gegebenenfalls durch Reduzierung der Busdatenrate) mit einer geeigneten Testsoftware vom Host protokolliert werden. Während der Implementierungsphase von Algorithmen bietet diese Möglichkeit — neben den in den lokalen Betriebssoftware-Prozessen eingebauten Hilfen — Unterstützung bei der Fehlersuche und erhöht die Transparenz des Systems.

7.2.5 Prototypsystem

Zur Verifikation der beschriebenen Systemfunktionen sowie zur ersten Implementierung von Algorithmen wurde ein Prototyp des HARBURGER SYSTEMS aufgebaut. Abbildung 7.7 zeigt einen Überblick.

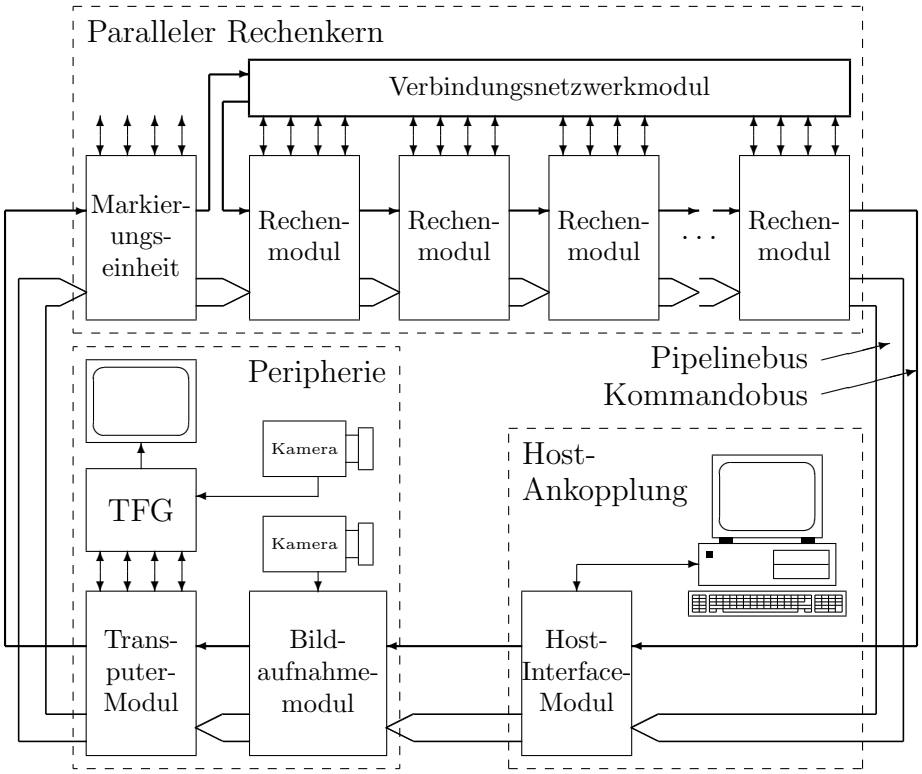

Abbildung 7.7: Übersicht des realisierten Prototypen.

Der Rechenkern besteht aus 8 Rechenmodulen, welche noch die in Abschnitt 7.2.1 angesprochene sequentielle Speicherschnittstelle mit eingeschränkter Übertragungsbandbreite beinhalten. Die Rechenmodule sind mit ihren Links über ein Verbindungsnetzwerkmodul verbunden. Als Markierungseinheit wurde ein Transputer-Modul mit schnellem Video-RAM-Speicher als Labormuster aufgebaut, welches obige Einschränkung zur Übertragungsbandbreite nicht mehr aufweist.

Die summierte Rechenleistung der 8 Rechenmodule des Rechenkerns ergibt bei Verwendung von $20\,MHz$ `T800`-Transputern [INM88b] die mittleren Werte von $8 \cdot 10$ MIPS= 80 MIPS und $8 \cdot 1,5$ MFLOPS= 12 MFLOPS.

Eine erste Peripherieeinheit wurde durch Link-Kopplung eines kommerziell erhältlichen Transputer-'Frame-Grabber' (TFG) mit einem Transputer-Modul realisert. Zwar erreicht diese Kombination nur ca. 50% der Übertragungsrate spezieller Aufnahme- und Wiedergabemodule, jedoch stehen damit die Funktionen der Bildaufnahme und -wiedergabe einschließlich einer Datenankopplung an den Pipelinebus für Implementierungen zur Verfügung. Die Steuerung dieser Funktionen übernimmt ein Anwenderprogramm auf dem Transputer-Modul, welches Bilddatenblöcke zum und vom TFG transferieren und die Bildaufnahme auf dem TFG stimulieren kann. Durch das Anwenderprogramm können weiterhin Daten markiert auf den Pipelinebus ausgegeben und somit einfache Markierungsaufgaben mit reduzierter Geschwindigkeit durchgeführt werden.

Beim Ausbau der Peripherie wurde das in Abschnitt 7.2.2 beschriebene Bildaufnahmemodul als Bildgeber in das System integriert. Damit ist im Prototyp die Generierung von

Bilddaten mit zeilenweiser Markierung mit voller Video-Geschwindigkeit möglich.

Die Host-Ankopplung ist in der beschriebenen Funktionalität im Prototypen vorhanden. Es wurde eine Kopplung mit einem IBM-PC als Host realisiert.

Kapitel 8

Ein schneller Pipelinebus

In den bisherigen Kapiteln wurde mehrfach die Notwendigkeit zum schnellen Verschieben, Laden und Entladen von Daten in parallelen Systemen herausgestellt. Der schnelle Pipelinebus des HARBURGER SYSTEMS ermöglicht diese Aufgabe mit Video-Geschwindigkeit. Neben dem Aufbau von Vorverarbeitungspipelines erlaubt dieser Bus innerhalb des in einen Rechenkern eingebetteten Multiprozessorsystems das schnelle Aufteilen und Zusammenfügen von Daten. Die funktionale Erweiterung gegenüber herkömmlichen Pipelinebus-Konzepten ermöglicht ein Pipeline-unterstütztes 'Scattering', 'Broadcast' und 'Gathering'.

8.1 Token-unterstützte Adressierung

Die Pipelinebus Erweiterung basiert auf einer Datenmarkierung durch Kontroll- und Adreßfelder. Jedem Datum (Data) wird ein Kontroll- (Ctrl) und Adreßteil (Addr) zugefügt. Das damit entstehende Tripel soll im folgenden als Token T bezeichnet werden:

$$T = ((Ctrl), (Addr), (Data)). \tag{8.1}$$

Token werden parallel auf dem Pipelinebus übertragen. Die Auswertung der enthaltenen Markierungen muß, um die Übertragungsgeschwindigkeit des Busses nicht zu beeinträchtigen, in jedem Modul mit der vollen Busgeschwindigkeit geschehen.

In Korrespondenz zu den drei Lademodi 'Scattering', 'Broadcast' und 'Gathering' werden drei Werte für den Kontrollteil (Ctrl) definiert und mit den Synonymen *'Single'*, *'All'* und *'Empty'* bezeichnet. Ein vierter Wert mit Synonym *'None'* dient zur Markierung leerer Token und ermöglicht das Auffüllen von Lücken auf dem Bus während der Übertragung. Durch diese Tokentypen werden in Verbindung mit dem Adreßteil (Addr) die angesprochenen Lademodi realisiert. Die Interpretation und Verarbeitung der Typen ist abhängig von einer Modulfunktion und muß bei gleicher Hardware z.B. auf einer Markierungseinheit anders erfolgen als auf einem Rechenmodul.

Die Bezeichnung der Token orientiert sich an der Übertragung eines Einzeldatums:

'Single': Ein Token vom Typ *'Single'* dient der Übertragung eines Einzeldatums von einem Quell- zu einem Zielmodul. Der Adreßteil spezifiziert das Zielmodul auf dem Pipelinebus, welches das Token empfängt und vom Bus entfernt.

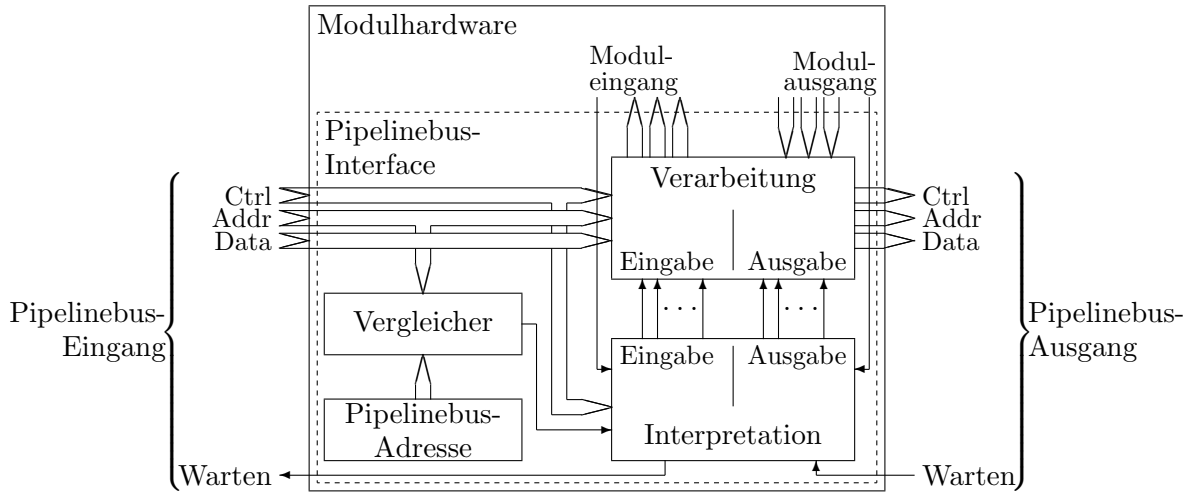

Abbildung 8.1: Funktionseinheiten des Pipelinebus-Interface.

'All': Ein Token vom Typ *'All'* wird zum Versenden eines Datums von einem Quellmodul zu vielen Zielmodulen verwendet. Ein Zielmodul nimmt ein *'All'* Token auf. Weiterhin wird das Token dann vom Bus entfernt, wenn es durch den Adreßteil ausgewählt wird, anderenfalls hingegen zum nachfolgenden Modul in der Pipeline weitergegeben. Somit spezifiziert der Adreßteil das letzte Zielmodul in der Pipeline.

'Empty': Token des Typs *'Empty'* enthalten kein gültiges Datum, sie werden vielmehr zur Stimulation versandt, um über den Bus das Modul, welches durch den Adreßteil spezifiziert ist, zur Ausgabe eines Tokens zu veranlassen.

'None': Token des Typs *'None'* füllen den leeren Bus auf.

Zur Interpretation und Verarbeitung von Token besitzen alle an den schnellen Pipelinebus angeschlossenen Module des Harburger Systems ein Interface, welches synchron zum Bustakt Token empfängt, verarbeitet und ausgibt (Abbildung 8.1). Zur Identifikation eines Moduls ist dort eine Pipelinebus-Adresse gespeichert, welche mit dem Adreßteil jedes empfangenen Tokens verglichen wird. Das Vergleichsergebnis steuert in Verbindung mit dem im Kontrollteil kodierten Typ die Interpretation und Verarbeitung eines ankommenden Tokens.

Im Pipelinebus-Interface erfolgt, in Übereinstimmung mit der bei digitalen Verarbeitungsautomaten üblichen Aufteilung in Verarbeitungs- und Steuerteil, eine Aufgabentrennung durch einen Verarbeitungs- und Interpretationsblock. Dies ermöglicht bei programmierbarer Realisierung des Interpretationblocks eine funktionale Anpassung des Interface an modulabhängige Tokeninterpretationen. Je nach Funktionalität steuert so der Interpretationsblock unterschiedliche, Token-abhängige Möglichkeiten der Verarbeitung.

Eine Synchronisierung auf die vom Modul generierten Daten und auf die Verarbeitungsrate des nachfolgenden Moduls erfolgt ebenfalls unter Kontrolle des Interpretationsblocks. Die Synchronisierung vom Modul generierter Token ermöglichen noch zu beschreibende *Attribute*. Die Anpassung einer nachfolgenden, langsamen Verarbeitung

kann ein Nachfolgemodul durch ein *Warte*-Signal vornehmen. Bei dessen Aktivierung wird synchron zum Bustakt die Ausgabedauer eines Tokens verlängert.

Zur Verarbeitung ankommender Token gemäß der zu realisierenden Lademodi stellt der Verarbeitungsblock unterschiedliche Verarbeitungsmöglichkeiten, im folgenden als *Aktionen* bezeichnet, zur Verfügung. Gesteuert von einer modulabhängigen Interpretation wählt der Interpretationsblock notwendige Aktionen daraus aus.

Die Aktionen der Verarbeitungseinheit teilen sich in Eingabe- und Ausgabeaktionen auf. Beide Aktionen werden simultan ausgeführt und müssen aufeinander synchronisiert werden. Es besteht weiterhin ein inhaltlicher Zusammenhang zwischen beiden. So ist eine Ausgabeaktion, welche ein empfangenes Eingangstoken verarbeitet, nur dann sinnvoll, wenn die zugehörige Eingabeaktion die benötigten Daten zur Verfügung stellt. In der unten beschriebenen Notation[1] benötigen beispielsweise die Ausgabeaktionen (Weitergabe), (Stimulation) und (Markierung) die Bereitstellung eines Eingangstokens durch die Eingabeaktion (Bearbeitung). Wird ein Eingangstoken nicht durch die Aktion (Bearbeitung) zur Verfügung gestellt, sollte auf der Ausgangsseite die Aktion (Ignorierung) durchgeführt werden.

Eingabeaktionen: Beim Empfang eines Tokens kann jede der folgenden Eingabeaktionen — auch in Kombination — durchgeführt werden:

(Aufnahme): Ein ankommendes Token wird an den Moduleingang weitergereicht. Ist der Moduleingang belegt, muß durch Aktivierung des Wartesignals 'Warten' das Token vom vorgeordneten Modul in der Pipeline für einen weiteren Buszyklus angefordert werden.

(Bearbeitung): Ein ankommendes Token wird im Verarbeitungsblock zur Bearbeitung durch eine Ausgabeaktion gespeichert, falls eine noch anstehende Ausgabeaktion im nächsten Buszyklus abgeschlossen sein wird. Anderenfalls muß durch Aktivierung des Wartesignals das Token vom vorgeordneten Modul in der Pipeline für einen weiteren Buszyklus angefordert werden.

(—): Keine Eingabeaktion wird durchgeführt.

Ausgabeaktionen: Zur Erzeugung eines Ausgangstokens muß genau eine der folgenden Ausgabeaktionen ausgewählt werden:

(Weitergabe): Ein ankommendes Token wird durch den Block der Verarbeitung über den Pipelinebus-Ausgang an ein nachfolgendes Modul weitergereicht. Die Aktion terminiert im gleichen Buszyklus, wenn das nachfolgende Modul in der Pipeline kein Wartesignal aktiviert.

(Stimulation): Ein ankommendes Token wird durch ein vom Modul generiertes, für die Stimulation attributiertes Token ersetzt. Das so erzeugte Token wird über den Pipelinebus-Ausgang an ein nachfolgendes Modul ausgegeben. Die Aktion terminiert im gleichen Buszyklus, wenn das benötigte, attributierte Token vorliegt und auch das nachfolgende Modul in der Pipeline kein Wartesignal aktiviert.

[1] Die Bezeichnung von Aktionen, Attributen und Signalen erfolgt in den Definitionen des Pipelinebus-Interface durch verbale, in Klammern eingeschlossene Beschreibungen. In der Hardware des Interface werden die Beschreibungen auf physikalische Signale abgebildet, welche die Verarbeitung steuern.

(Markierung): Kontroll- und Adreßteil eines ankommenden Tokens werden durch vom Modul generierte, für die Markierung attributierte Werte ersetzt. Das damit neu entstandene Token aus altem Datenteil und neuem Adreß- und Kontrollteil wird über den Pipelinebus-Ausgang ausgegeben. Die Aktion terminiert im gleichen Buszyklus, wenn das benötigte, attributierte Token vorliegt und auch das nachfolgende Modul in der Pipeline kein Wartesignal aktiviert.

(Ignorierung): Ein ankommendes Token wird ignoriert.

Ausgabeattribute: Zur Datenausgabe über den Modulausgang erzeugt ein Modul Token und fügt diesen ein Ausgabeattribut hinzu. Dieses Attribut bestimmt, wie der Interpretationsblock die Ausgabe des Tokens durch den Block der Verarbeitung vornimmt:

(frei ausgeben): Ein vom Modul generiertes Token soll frei und ohne Stimulation durch ein Eingangstoken über den Pipelinebus-Ausgang ausgegeben werden. Dies kann während der Aktion (Ignorierung) und auch während wartender Aktionen (Stimulation) oder (Markierung) erfolgen. In letzteren beiden Fällen terminieren diese Aktionen nicht, sondern der Interpretationsblock nimmt zwischenzeitlich einen Ausgabezustand an, kehrt jedoch nach Abschluß der freien Ausgabe zur wartenden Aktion zurück.

(stimuliert ausgeben): Ein vom Modul generiertes Token muß durch die Ausgabeaktion (Stimulation) synchron in einen Tokenstrom des Pipelinebusses eingefügt werden.

(einfach markieren): Ein vom Modul generierter Kontroll- und Adreßteil muß zur Markierung eines empfangenen Tokens durch obige Aktion (Markierung) verwendet werden.

(dupliziert markieren): Ein vom Modul generierter Kontroll- und Adreßteil muß zur Markierung eines empfangenen Tokens durch obige Aktion (Markierung) verwendet werden. Dabei terminiert diese Aktion nicht, sondern das empfangene Token wird dupliziert, so daß eine weitere Markierung des gleichen Tokens vorgenommen wird.

(Modulausgang leer): Es liegen noch keine Tokendaten am Modulausgang vor. Eine gegebenenfalls wartende Ausgabeaktion muß auf die Generierung von Tokendaten durch das Modul warten.

Entscheidend bei der Erzeugung von Ausgangstoken durch den Block der Verarbeitung ist das zeitlich synchronisierte Zusammenführen von Ausgabeaktionen und korrespondierend attributierter, vom Modul generierter Tokendaten durch den Interpretationsblock. Eine Synchronisierung muß beispielsweise zwischen der Aktion (Markierung) und generierten Tokendaten mit dem Attribut (einfach markieren) oder (dupliziert markieren) erfolgen. Würde während dieser Synchronisation ein nicht zusammengehöriges Aktion/Attribut-Paar auftreten, führte dies zum Blockieren des Ausgangs. Ein Beispiel wäre das Zusammentreffen der Aktion (Markierung) mit einem (stimuliert ausgeben)-attributierten Token.

Solche Blockierungen werden im folgenden dadurch vermieden, daß bei einer modulabhängigen Tokeninterpretation entweder die Aktion (Markierung) oder die Aktion (Stimulation) sowie die jeweils zugehörigen Ausgabeattribute zugelassen werden, nicht

Tabelle 8.1: Definition der Pipelinebus-Funktionalität eines Rechenmoduls.

Token-typ:	Adreßvergleich: (Addr)=(Pipelinebus-Adresse)	Adreßvergleich: (Addr)≠(Pipelinebus-Adresse)
'Single'	(Aufnahme) (Ignorierung)	(Bearbeitung) (Weitergabe)
'All'	(Aufnahme) (Ignorierung)	(Aufnahme)∧(Bearbeitung) (Weitergabe)
'Empty'	(Bearbeitung) (Stimulation)	(Bearbeitung) (Weitergabe)
'None'	(—) (Ignorierung)	(—) (Ignorierung)

Attribute der Ausgangstoken: (stimuliert ausgeben)
(Modulausgang leer)

Tabelle 8.2: Definition der Pipelinebus-Funktionalität einer Markierungseinheit.

Token-typ:	Adreßvergleich: (Addr)=(Pipelinebus-Adresse)	Adreßvergleich: (Addr)≠(Pipelinebus-Adresse)
'Single'	(Bearbeitung) (Markierung)	(Bearbeitung) (Weitergabe)
'All'	(Bearbeitung) (Weitergabe)	(Bearbeitung) (Weitergabe)
'Empty'	(Bearbeitung) (Weitergabe)	(Bearbeitung) (Weitergabe)
'None'	(—) (Ignorierung)	(—) (Ignorierung)

Attribute der Ausgangstoken: (einfach markieren)
(dupliziert markieren)
(frei ausgeben)
(Modulausgang leer)

jedoch beide Aktionen. Für die Realisierung der Multiprozessorlademodi stellt dies, wie im folgenden noch deutlich wird, keine Einschränkung dar.

8.2 Betriebsarten des Pipelinebus-Interface

Mit der Definition der Verarbeitungsaktionen kann eine modulspezifische Zuordnung der Aktionen zu den Eingangstoken eines Transputer-Moduls erfolgen. In Abhängigkeit der Systemfunktion müssen Tabellen zur Umsetzung von Eingangstoken in Aktionen erstellt werden, welche in die Firmware des Interpretationsblocks zu übertragen sind und dort die Pipelinebus-Funktionalität festlegen. Die Tabellen 8.1 bis 8.3 zeigen entsprechende Tabellen für Rechenmodul, Markierungseinheit und Speichermodul. Die Tabellen stellen beispielhafte Vorschläge für die Tokenverarbeitung dar und können speziellen Anforderun-

Tabelle 8.3: Definition der Pipelinebus-Funktionalität eines Speichermoduls.

Token-typ:	Adreßvergleich: (Addr)=(Pipelinebus-Adresse)	Adreßvergleich: (Addr)≠(Pipelinebus-Adresse)
'Single'	(Aufnahme) (Ignorierung)	(Bearbeitung) (Weitergabe)
'All'	(Aufnahme) (Ignorierung)	(Aufnahme)∧(Bearbeitung) (Weitergabe)
'Empty'	(Bearbeitung) (Weitergabe)	(Bearbeitung) (Weitergabe)
'None'	(—) (Ignorierung)	(—) (Ignorierung)

Attribute der Ausgangstoken: (frei ausgeben)
(Modulausgang leer)

gen angepaßt werden. Die hier festgelegte Funktionalität ist auf die Realisierung der geforderten Multiprozessorlademodi abgestimmt.

Für jede Kombination des Tokentyps mit dem Ergebnis des Adreßvergleichs sind in den Tabellen übereinander die auszulösenden Eingabe- und Ausgabeaktionen angeführt. Weiterhin sind unter den Tabellen die erlaubten Ausgabeattribute angeführt.

Tabelle 8.1 zeigt die Interpretation der ankommenden Token durch ein Rechenmodul. Token vom Typ *'Single'* werden bei Adreßgleichheit aufgenommen, *'All'* Token dienen der gleichzeitigen Aufnahme und Weitergabe von Daten und *'Empty'* Token stimulieren bei Adreßgleichheit die Ausgabe eines vom Modul generierten Tokens. Durch Attribute ist nur die stimulierte Datenausgabe unterstützt.

Tabelle 8.2 beschreibt die Interpretation ankommender Token durch eine Markierungseinheit. Token vom Typ *'Single'* werden bei Adreßgleichheit mit einem neuen, vom Modul generierten Kontroll- und Adreßteil markiert. Die Markierung ist sowohl einfach als auch dupliziert möglich. Alle übrigen Tokenkonstellationen werden weitergegeben oder ignoriert. Bleibt der Pipelinebus-Ausgang durch die Interpretation der Eingangstoken ungenutzt, so können vom Modul generierte, mit dem Attribut (frei ausgeben) versehene Token ausgegeben werden. Die Markierungseinheit benötigt diese Möglichkeit zur freien Ausgabe von Stimulationstoken des Typs *'Empty'* und unterstützt damit, wie noch ausgeführt wird, das schnelle 'Gathering' als Entlademodus.

Die in Tabelle 8.3 ausgeführte Interpretation eines Speichermoduls erlaubt den Empfang adressierter Token und die freie Datenausgabe ohne Stimulation.

8.3 Realisierung von Multiprozessorlademodi mit dem schnellen Pipelinebus

Die eingeführte Pipelinebus-Funktionalität der Module des Harburger Systems ermöglicht die Definition der notwendigen Multiprozessorlademodi für den Rechenkern. Eine

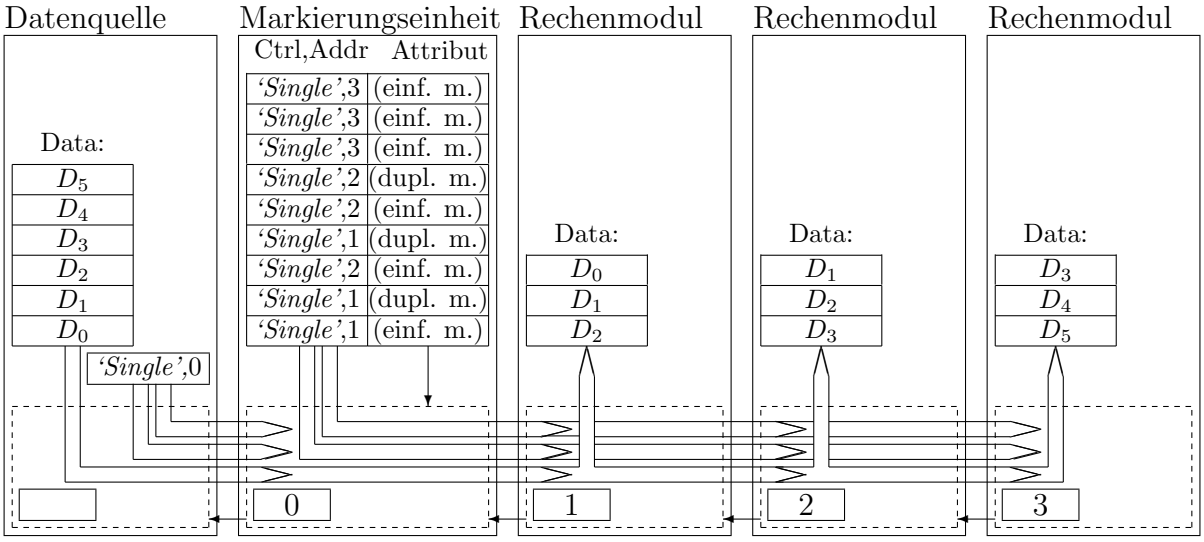

Abbildung 8.2: Verteilung von Daten mittels 'Scattering' im HARBURGER SYSTEM.

Anordnung aus Bildquelle, Markierungseinheit und nachfolgenden Rechenmodulen wird zur Beschreibung der Lademodi 'Scattering' und 'Broadcast' vorausgesetzt. Zur Beschreibung des 'Gathering', dem Zusammenfügen der Ergebnisdaten, wird eine Anordnung aus Markierungseinheit, nachfolgenden Rechenmodulen und einer Datensenke verwendet.

Datenquelle kann ein Bildspeicher- oder Bildaufnahmemodul sein, wobei die Pipelinebus-Funktionalität der Bildaufnahme dem Ausgabeverhalten eines Bildspeichers entspricht. Gleiches gilt für ein Anzeigemodul als Datensenke, welches in seinem Pipelinebus-Verhalten mit der Eingangsseite eines Bildspeichermoduls korrespondiert. Weiterhin erlaubt der modulare Aufbau in Systemen mit mehreren Rechenkernen, vor- und nachgeordnete Kerne als Datenquelle oder -senke eines lokalen Abschnitts der Verarbeitungspipeline anzusehen.

8.3.1 'Scattering', 'Broadcast' und 'Gathering'

Lademodus 'Scattering'

Das schnelle Aufteilen vom Quellmodul generierter Daten ('Scattering') erfolgt im HARBURGER SYSTEM durch Markierung der Quelltoken. Die Datenquelle generiert Token vom Typ *'Single'* und der Pipelinebus-Adresse der Markierungseinheit. Dieser Tokenstrom wird im Pipelinebus-Interface der Markierungseinheit mit dem Strom dort generierter Kontroll- und Adreßinformationen zusammengeführt. Der Markierungsstrom beinhaltet die Verteilungsinformation, welche im Fall des 'Scattering' jedem Quelldatum den Kontrollteil *'Single'* und die Adresse des gewünschten Zielrechenmoduls zuweist.

Die Information über die Aufteilung der Daten auf die Rechenmodule muß konzentriert in der Markierungseinheit vorliegen, dann kann diese die Markierungen durchführen. Die Ermittlung der Markierungen wird meist vorab und lokal auf dieser Einheit erfolgen, eine verteilte Ermittlung mit anschließendem Ladevorgang auf diese Einheit ist jedoch auch denkbar.

Beim disjunkten Verteilen entspricht die Anzahl der vom Quellmodul generierten Datenwerte der Anzahl der Markierungen, welche dann alle dem Pipelinebus-Interface der Markierungseinheit mit dem Attribut (einfach markieren) zugeführt werden.

Zum überlappten Verteilen können Markierungen mit dem Attribut (dupliziert markieren) in den Markierungsstrom eingefügt werden. Diese bewirken ein Duplizieren der Quelldaten im Pipelinebus-Interface der Markierungseinheit, wodurch das gleiche Datum mit mehreren Markierungen versehen wird.

Abbildung 8.2 zeigt den Vorgang des überlappten 'Scattering' beispielhaft für einen Quelldatensatz aus den 6 Elementen $D_0, \ldots, D_5$. Quellmodul und Markierungseinheit zeigen die Quell- und Markierungsdaten *vor* dem Verteilungsvorgang; in den drei Rechenmodulen sind die verteilten Daten *nach* Ablauf der Verteilung dargestellt.

Die gestrichelt gezeichneten Boxen in jedem Modul sollen das Pipelinebus-Interface darstellen. In der linken unteren Ecke dieser Interfaces ist jeweils die Pipelinebus-Adresse angedeutet. Im beispielhaften Ablauf sendet die Datenquelle die Daten sequentiell mit Kontrollteil *'Single'* und dem Adreßteil 0, welcher mit der Pipelinebus-Adresse der Markierungseinheit übereinstimmt, auf den Pipelinebus. Über den Bus gelangt der Tokenstrom zur Markierungseinheit. Dem ersten Token mit dem Datum D_0 wird der Kontroll- und Adreßteil *'Single'*,1 hinzugefügt. Das resultierende Token $T_A = (\textit{'Single'}, 1, D_0)$ gelangt über den Pipelinebus zum ersten Rechenmodul, wo es aufgrund der Adreßgleichheit empfangen und vom Bus entfernt wird.

Das zweite Datum D_1 wird im Beispiel an das erste und an an das zweite Rechenmodul gesandt. Dies wird durch die beiden Markierungen *'Single'*,1 mit dem Attribut (dupliziert markieren) und *'Single'*,2 mit dem Attribut (einfach markieren) an zweiter und dritter Stelle des Markierungsstroms ausgedrückt. Die erste der beiden Markierungen erzeugt ein Token $T_B = (\textit{'Single'}, 1, D_1)$ und dupliziert das Datum. Die folgende Markierung erzeugt das Token $T_C = (\textit{'Single'}, 2, D_1)$. Das Datum D_1 wird somit durch die Duplizierung sowohl an das erste als auch an das zweite Rechenmodul adressiert.

Die Markierung der folgenden Daten geschieht in analoger Weise, wobei D_2 und D_3 ebenfalls dupliziert werden. Das Markieren der Quelldaten in der spezifizierten Weise führt schließlich zur dargestellten Datenverteilung auf den Rechenmodulen.

Beim Ladevorgang des 'Scattering' und einer etwa gleichmäßigen Aufteilung der Quelldaten auf P Rechenmodule ist die mittlere Eingangsdatenrate jedes Moduls etwa um den Faktor $1/P$ niedriger als die Quelldatenrate. Gleichmäßige Aufteilungen erlauben somit die Verwendung von Rechenmodulen mit reduzierter Eingangsdatenrate, ohne Wartezyklen auf dem Bus auszulösen, falls eine kurzfristige Datenaufnahme konsekutiver Werte mit der vollen Busdatenrate (z.B. durch eine FIFO-Pufferung) gewährleistet wird. Die Markierungseinheit muß hingegen die Markierungsaufgabe mit der vollen Rate der Datenquelle durchführen.

Lademodus 'Broadcast'

Zur Realisierung des 'Broadcast'-Lademodus wird die gleiche Anordnung wie beim 'Scattering' verwendet. Die Datenquelle adressiert die Quelltoken an die Markierungseinheit

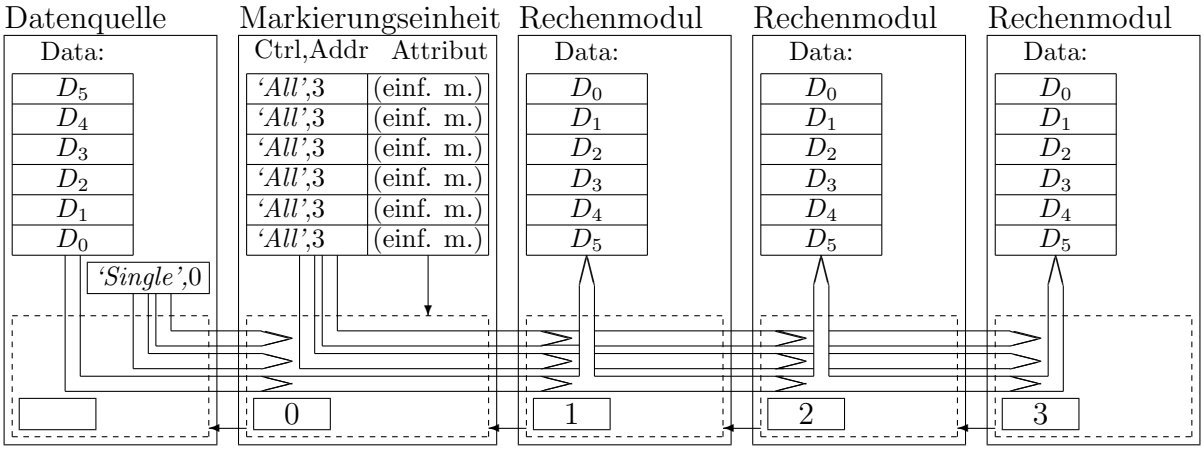

Abbildung 8.3: Verteilung von Daten mittels 'Broadcast' im HARBURGER SYSTEM.

mittels *'Single'*-Markierung. Diese generiert einen Markierungsstrom, bei dem alle Kontrollteile eine *'All'*-Kennung erhalten und alle Adreßteile mit der Pipelinebus-Adresse des letzten Rechenmoduls in der Pipeline, welches die Daten empfangen soll, übereinstimmen. Die Attribute aller Markierungen werden auf den Wert (einfach markieren) gesetzt. Dieser Markierungsstrom ersetzt die Kontroll- und Adreßteile des Tokenstroms der Datenquelle. Die nachfolgenden Rechenmodule empfangen den Strom dieser *'All'*-markierten Datentoken und reichen ihn gleichzeitig auf dem Bus weiter. Nur das letzte, durch den Adreßteil spezifizierte Modul empfängt den Strom und entfernt ihn vom Bus, da seine Pipelinebus-Adresse mit den Tokenadressen übereinstimmt.

Abbildung 8.3 zeigt einen Ladevorgang mittels 'Broadcast'. Die Datenquelle adressiert die Quelldaten mit *'Single'*-Markierung und Adreßteil 0 an die Markierungseinheit, welche in die Token dann eine *'All'*-Markierung und die Adresse 3 des letzten Rechenmoduls einträgt. Durch den beschriebenen Ladevorgang empfangen alle Rechenmodule den vollständigen Quelldatensatz, hier im Beispiel die 6 Datenwerte $D_0, \ldots, D_5$.

Die Abbildung zeigt in der Datenquelle und Markierungseinheit die Daten und Markierungen *vor* dem Lade- und Markierungsvorgang. Die in den Rechenmodulen dargestellten Datenwerte liegen erst *nach* dem 'Broadcast' vor.

Beim Ladevorgang des 'Broadcast' muß, falls keine Wartezyklen auf dem Bus ausgelöst werden sollen, jede Einheit ankommende Token auf dem Pipelinebus mit der vollen Quelldatenrate verarbeiten. Insbesondere ist diese Forderung bei Verwendung von Quellmodulen mit Echtzeitbedingungen (z.B. Bildaufnahme mit Kamera) zu beachten. Bei Verwendung von Transputer-Modulen, wo der Datentransfer zwischen Pipelinebus-Interface und Modulspeicher durch Hardware-Einheiten und Video-RAM unterstützt wird (Abschnitt 7.2.1), ist die Erfüllung dieser Forderung möglich.

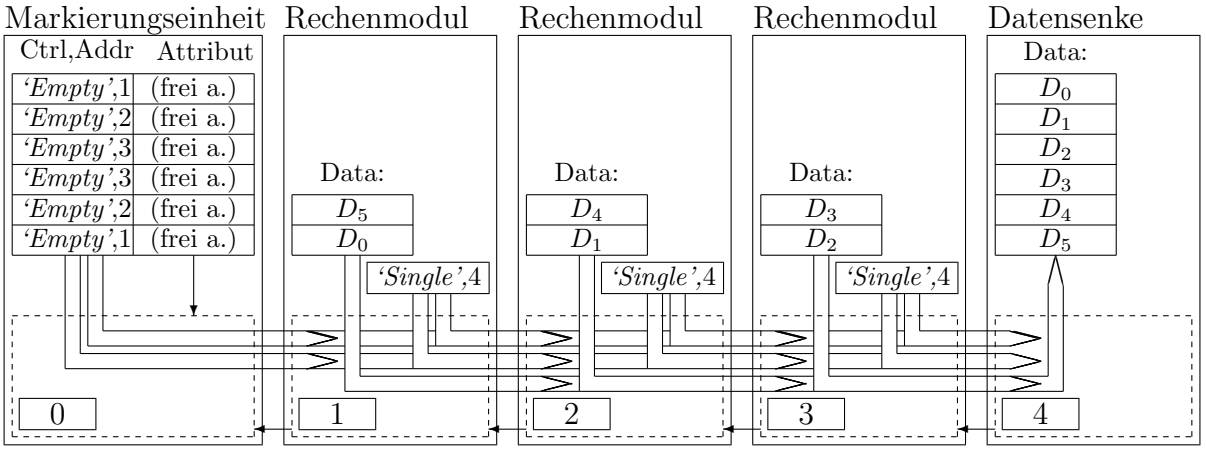

Abbildung 8.4: Zusammenfassen von verteilten Ergebnisdaten mittels 'Gathering' im HARBURGER SYSTEM.

Entlademodus 'Gathering'

Beim Zusammenfügen errechneter Ergebnisdaten, beim 'Gathering' im HARBURGER SYSTEM wird — in Analogie zum 'Scattering' — davon ausgegangen, daß die Information über die Datenverteilung explizit bekannt ist und in der Markierungseinheit vorliegt. Ebenso wie beim 'Scattering' ist damit jedoch keine Einschränkung bezüglich der Ermittlung dieser Information getroffen.

Die Markierungseinheit stimuliert das Zusammenfügen verteilter Ergebnisse der Rechenmodule. Dazu generiert sie einen Tokenstrom mit *'Empty'*-Kontrollteilen und mit Adreßteilen, welche die Auslesereihenfolge der Rechenmodule spezifizieren. Die Datenteile der Token des Stroms sind unbenutzt. Die Ausgabe des Tokenstroms auf den schnellen Pipelinebus erfolgt mit dem Attribut (frei ausgeben) und benötigt somit keine Stimulation. Erreicht ein Token das adressierte Rechenmodul, so wird dieses Modul stimuliert, das Token durch ein Token mit Attribut (stimuliert ausgeben), welches ein Ergebnisdatum enthält, zu ersetzen. Somit dient der Markierungsstrom als leerer Datenrahmen, der beim Vorbeifließen die Rechenmodule stimuliert, diesen mit den lokal ermittelten Ergebnisdaten aufzufüllen. Hinter dem letzten Rechenmodul ist der Strom bei korrekter Korrespondenz zwischen Stimulation und Ergebnisdaten vollständig ersetzt und kann wohlgeordnet einer Datensenke zugeführt werden.

Abbildung 8.4 zeigt ein Beispiel für das 'Gathering'. Der Ablauf wird durch die Markierungseinheit gestartet, welche nun am Anfang der Pipeline angeordnet ist. Im Beispiel liegen in dieser Einheit *vor* dem Start des Entladevorgangs 6 Token mit dem Kontrollteil *'Empty'* vor. Durch den Adreßteil 1 stimuliert das erste Token das erste Rechenmodul zur Ausgabe des Tokens mit dem Datum D_0. Das zweite Token stimuliert das zweite Rechenmodul, und entsprechend werden alle Ergebnisdaten $D_0, \ldots, D_5$ stimuliert ausgelesen und als *'Single'* Token mit Adreßteil 4 an die Datensenke adressiert. Die Verteilung der Ergebnisdaten auf den Rechenmodulen *vor* dem Start und die Datenreihenfolge in der Datensenke *nach* Abschluß des Entladevorgangs kann der Abbildung entnommen werden.

8.3.2 Modularität und Ladezeit

Die beschriebenen Lade- und Entladevorgänge weisen keine Einschränkung bezüglich der Rechenmodulanzahl auf und können modular beim Systemausbau angepaßt werden. Eine Einschränkung ergibt sich zunächst nur durch die Wortbreite des Adreßteils (Addr) der Pipelinebus-Token. Reicht der Zahlenbereich des Adreßteils bei hohen Modulanzahlen im System nicht mehr aus (im HARBURGER SYSTEM sind 6-Bit vorgesehen), schafft eine verteilte Markierung durch Verwendung mehrerer, hierarchisch organisierter Markierungseinheiten Abhilfe.

Die Zeit t_p zum Laden oder Entladen der Rechenmodule über den Pipelinebus hängt von deren Anzahl P ab. Bei steigender Modulanzahl vergrößert sich die Länge der Ladepipeline und damit der zeitliche Abstand des letzten Rechenmoduls von der Datenquelle. Soll ein Datenobjekt der Größe N in die Rechenmodule geladen werden, so benötigt dies bei einem Pipelinebus-Takt T_0 die Zeit t_p:[2]:

$$t_p \;=\; (P+2)\cdot T_0 + N\cdot T_0. \tag{8.2}$$

Diese Zeit ergibt sich aus dem Abstand $P+2$ der Datenquelle zum letzten Rechenmodul und der Anzahl der Zeittakte N zum Ausgeben der Daten. Die durch den Abstand vorgegebene Initialisierungszeit der Pipeline kann bei großen Datenobjekten, wenn $N \gg (P+2)$ gilt, vernachlässigt werden. Bei Bilddaten (z.B. $N = 2^{18}$ bei 512×512 Bildern) ist diese Bedingung meistens erfüllt.

8.4 Hardware-Struktur und Steuerung des Pipelinebus-Interface

Die Struktur der Pipelinebus-Interfaces im Harburger System (siehe Abbildung 8.5) zeigt neben der in Abbildung 8.1 schon eingeführten horizontalen Schichtung (Interpretations- und Verarbeitungsblock) eine vertikale Schichtung. Diese vertikale Schichtung umfaßt eine Eingangs-, Ausgangs- und Warteschicht. In der Eingangsschicht werden die Eingabeaktionen abgearbeitet, die Ausgangsschicht führt empfangene und generierte Token über die Ausgabeaktionen und Attribute zusammen. Die anschließende Warteschicht gewährleistet die sofortige Reaktion auf die Warteanforderung eines nachgeordneten, langsamen Moduls.

Die vertikale Schichtung teilt den Interpretationsblock in eine Ein- und Ausgangssteuerung. In beiden Blöcken verbergen sich endliche Automaten, deren Firmware die Interpretationstabellen zur Umsetzung der Token auf Aktionen sowie Zustandsdiagramme zur Ausführung der Aktionen beinhalten. In den folgenden Unterabschnitten wird noch detailliert auf diese Firmware eingegangen.

[2]Die Ladezeit t_p kann sich im Falle des 'Scattering' um bis zu $P \cdot T_0$ verkürzen, wenn die P letzten Datenwerte auf vordere Rechenmodule der Pipeline geladen werden. Für die folgenden Überlegungen soll der langsamste Fall, das letzte Datum wird auf das letzte Modul der Pipeline geladen, angenommen werden.

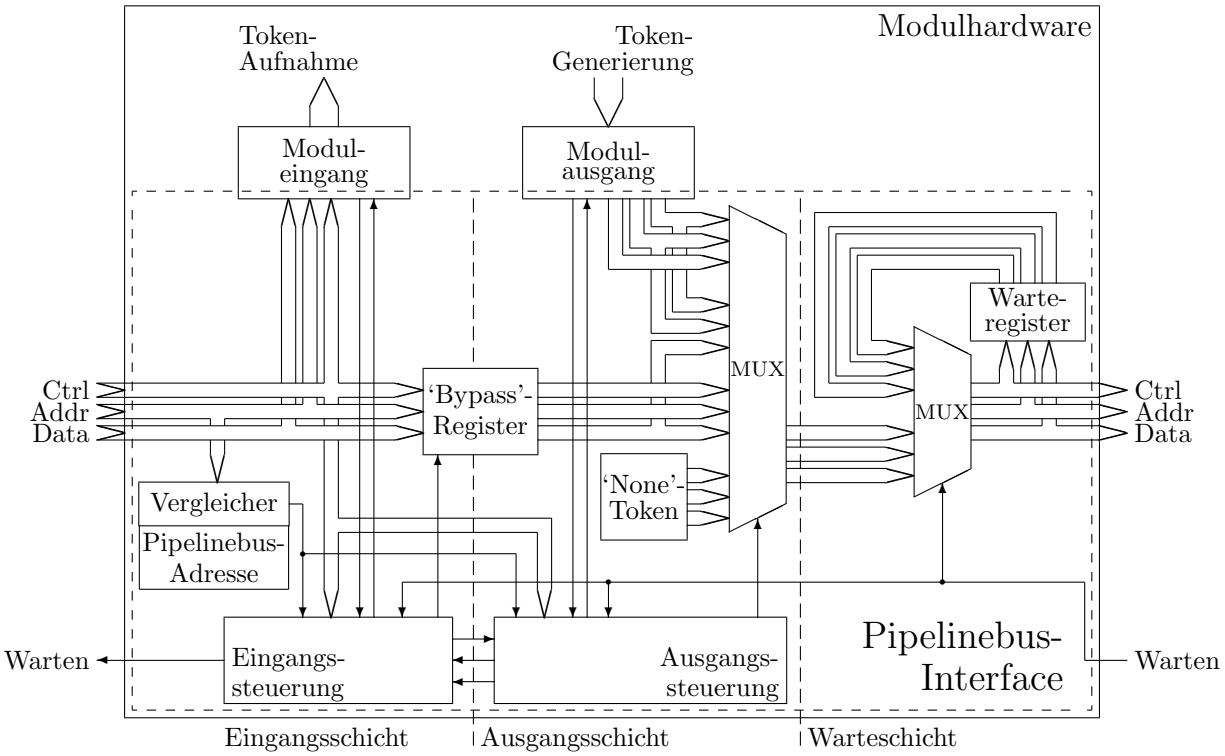

Abbildung 8.5: Übersicht der Hardware-Struktur eines Pipelinebus-Interface.

Der Verarbeitungsblock beinhaltet ein 'Bypass'-Register[3] als Übergang von der Ein- auf die Ausgangsschicht. In der Ausgangsschicht ermöglicht ein erster Multiplexer (MUX) die Auswahl des auszugebenden Tokens. Dazu steht neben dem empfangenen und generierten Token sowie deren Kombination ein leeres Token (*'None'*) an den Eingängen des Multiplexers zur Verfügung.

Das nachfolgende Modul der Pipeline kann durch Aktivieren des Wartesignales mit dem Multiplexer der Warteschicht das alte Token des vorangegangenen Pipeline-Zyklus für einen weiteren Zyklus auswählen, was in den Steuerungen der Ein- und Ausgangsschicht bei der Bearbeitung des aktuellen Zyklus natürlich beachtet werden muß. Die Speicherung des alten Tokens erfolgt im *Warteregister*. Da der Warteregistereingang an den Ausgang des Multiplexers der Wartestufe angeschlossen ist und somit im Wartefall der alte Wert immer wieder nachgeladen wird, kann ein nachfolgendes Modul das Wartesignal beliebig lange aktivieren und damit die Ausgabe eines Tokens verzögern, ohne daß Busdaten verlorengehen.

[3] An Register ist in diesem und den folgenden Blockschaltbildern kein Taktsignal angezeichnet, es wird immer implizit eine Taktung mit dem Pipelinebus-Takt T_0 vorausgesetzt. Eine Steuerleitung, welche an ein Register führt, soll ein Freigabesignal ('Enable') zur Qualifizierung des Taktes T_0 darstellen. Weiterhin werden alle Register als D-Typ Register angenommen, deren Ladevorgang, falls freigegeben, immer synchron mit dem Übergang in den nächsten Zyklus erfolgt.

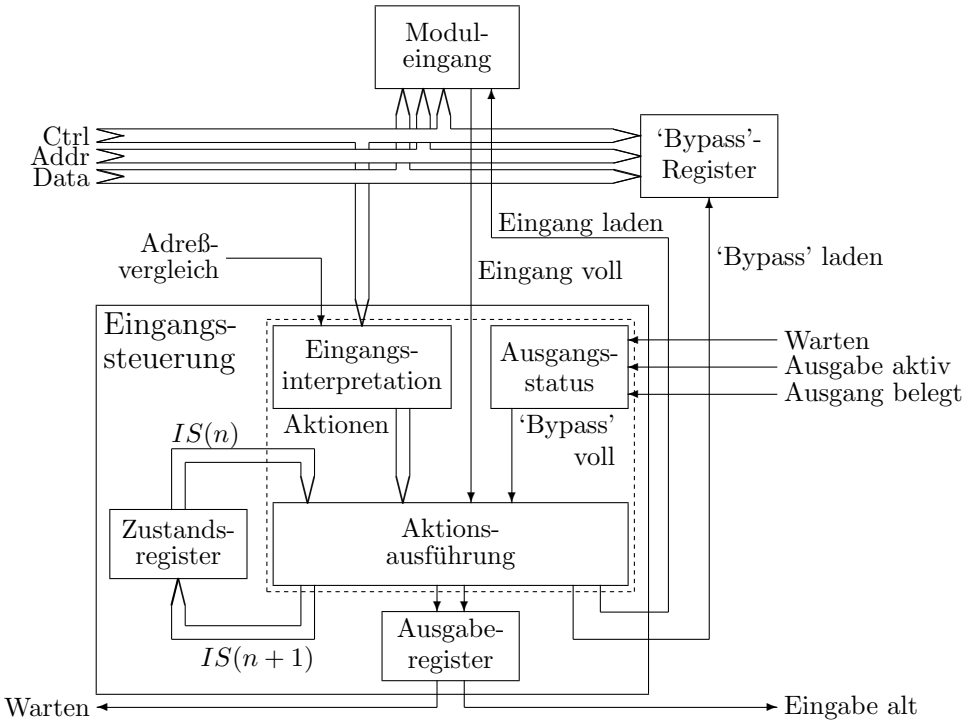

Abbildung 8.6: Eingangsschicht des Pipelinebus-Interface.

8.4.1 Steuerung der Eingangsschicht

Zur detaillierten Beschreibung der Eingangsseite des Pipelinebus-Interface ist in Abbildung 8.6 die Eingangsschicht hervorgehoben. Insbesondere wird die Eingangssteuerung in ihrer vollen funktionalen Auflösung gezeigt.

Von den Token des Pipelinebus-Eingangs wird der Kontrollteil und das Ergebnis des Adreßvergleichs an einen Block der Steuerung zur Eingangsinterpretation geführt. Dort ist eine Tabelle der Eingabeaktionen abgespeichert. Einige exemplarische Eingangstabellen sind in den Tabellen 8.1 bis 8.3 enthalten. Der Block zur Eingangsinterpretation gibt die von den Token abgeleiteten Eingabeaktionen an die Aktionsausführung weiter.

Der Zustand des Moduleingangs wird über das Signal (Eingang voll) an den Block der Aktionsausführung gegeben. Das Signal kann aktiv oder inaktiv[4] sein. Im zweiten Fall kann ein ankommendes Token von der Aktionsausführung durch Aktivieren des (Eingang laden)-Signals in den Eingang geladen werden.

Der Zustand ('Bypass' voll) des 'Bypass'-Registers muß aus den drei Signalen (Ausgang belegt), (Ausgabe aktiv) und (Warten) der Ausgangs- und Warteschicht berechnet werden:

$$(\text{'Bypass' voll}) = ((\text{Ausgabe aktiv}) \wedge (\text{Warten})) \vee (\text{Ausgang belegt}). \qquad (8.3)$$

Das Signal (Ausgabe aktiv) wird von der Ausgangsschicht aktiviert, wenn eine Tokenausgabe an ein nachfolgendes Modul durchgeführt wird. Diese Ausgabe wird im aktuellen

[4]Der aktive Zustand wird im folgenden durch ein Signal direkt ausgedrückt: z.B. (Eingang voll). Der inaktive Zustand wird durch Negierung des Signals ausgedrückt: z.B. $\overline{\text{(Eingang voll)}}$

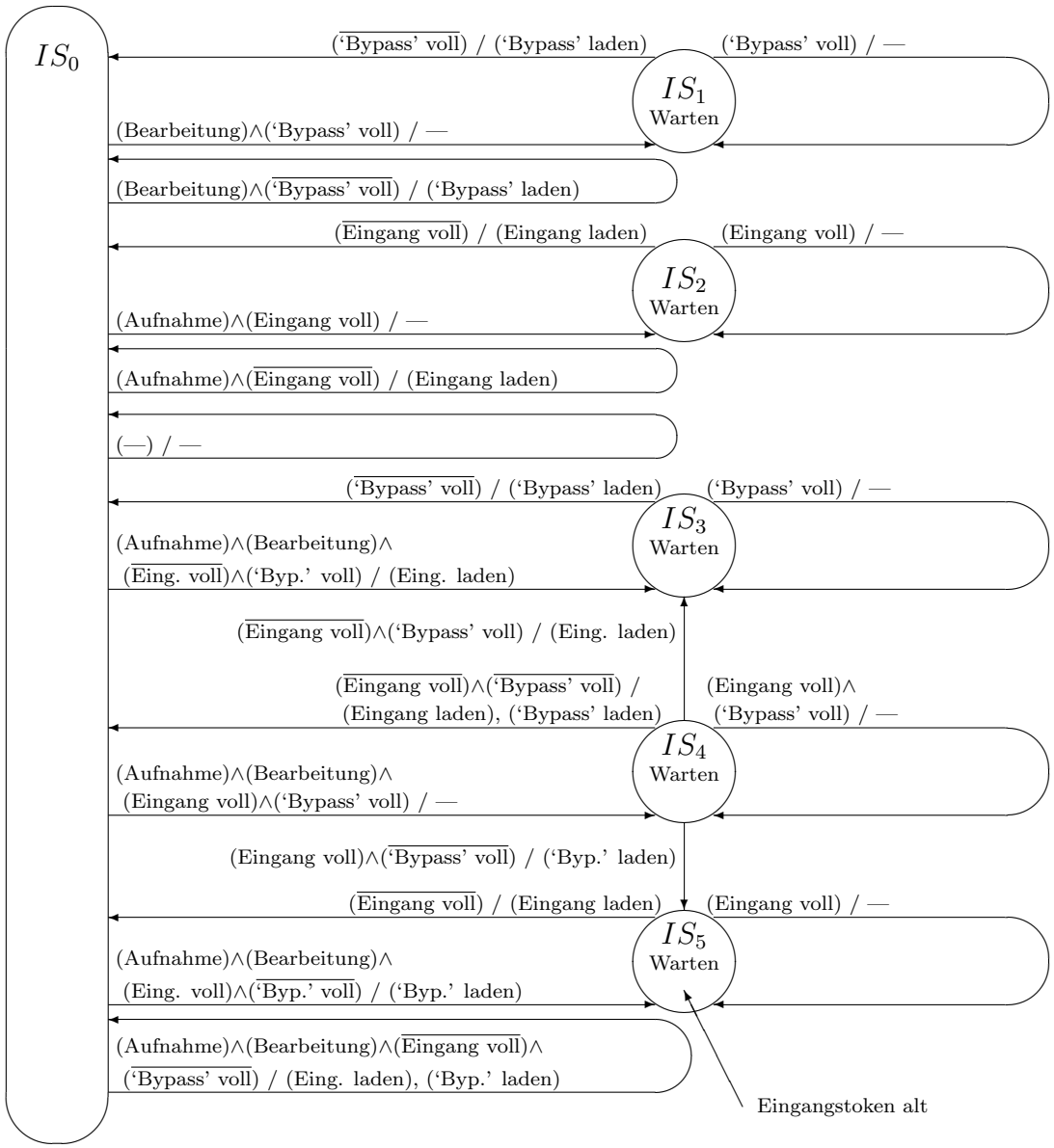

Abbildung 8.7: Zustandsdiagramm des endlichen Automaten zur Ausführung aller möglichen Eingabeaktionen (zur Notation siehe Fußnote auf Seite 105).

Buszyklus abgeschlossen, falls kein (Warten)-Signal des nachfolgenden Moduls aktiviert ist. Das Signal (Ausgang belegt) zeigt, daß die Ausgangsschicht mit der Ausführung einer Aktion beschäftigt ist, welche im nächsten Buszyklus nicht abgeschlossen sein wird. Wann die Aktivierung der Signale (Ausgabe aktiv) und (Ausgang belegt) erfolgt, wird durch die Steuerung der Ausgangsschicht spezifiziert.

Die eingangsseitige Verknüpfung der drei Signale der Ausgangsseite zu dem interessierenden Signal ('Bypass' voll) hat zeitliche Gründe. Der Block des Ausgangsstatus kann mit der Eingangsinterpretation, und der Aktionsausführung, da keiner dieser Blöcke innere Rückkopplungen besitzt, zu einem in Abbildung 8.6 gestrichelt dargestellten Block zusammengefaßt und für schnelle Anwendungen als einstufiges Netz realisiert werden.

Im Prototypen des HARBURGER SYSTEMS ist dieser Gesamtblock als schneller Speicher realisiert, dessen Inhalt aus den drei Unterblöcken errechnet wird.

Ausführung der Eingabeaktionen
Die Ausführung der Eingabeaktionen erfolgt durch einen endlichen Automaten, welcher durch den Block der Aktionsausführung sowie durch ein Zustands- und Ausgaberegister gebildet wird. Die Firmware dieses Automaten basiert direkt auf den Eingabeaktionen und nur indirekt über die Eingangsinterpretation auf den Eingangstoken. Sie kann daher einheitlich für alle Pipelinebus-Interfaces erstellt werden. Eine unterschiedliche Funktionalität unterscheidet nur die Eingangsinterpretationstabellen. Das benötigte Zustandsdiagramm[5] zur Ausführung der in Abschnitt 8.1 beschriebenen der Eingabeaktionen zeigt Abbildung 8.7.

Den zentrale Zustand des Diagramms bildet IS_0. Solange der Moduleingang und das 'Bypass'-Register zur Datenaufnahme bereit sind, wird dieser Zustand beim Übergang in einen neuen Taktzyklus nicht verlassen. Die Ausführung unterschiedlicher Eingabeaktionen spiegelt sich nur in der Verwendung unterschiedlicher Übergangskanten von IS_0 auf sich selber wider, welche dann die benötigen Freigabesignale (Eingang laden) und ('Bypass' laden) aktivieren. Erst durch eine Aktion (Aufnahme) bei vollem Moduleingang oder durch eine Aktion (Bearbeitung) bei belegtem 'Bypass'-Register wird der Zustand IS_0 verlassen und einer der Wartezustände $\{IS_1, \ldots, IS_5\}$ eingenommen.

IS_1 dient zum Warten auf Freigabe des 'Bypass'-Registers bei der Ausführung der Aktion (Bearbeitung). IS_2 dient entsprechend zum Warten auf Freigabe des Eingangs bei der Aktion (Aufnahme). IS_3 bis IS_5 koordinieren den Warteprozeß beim gleichzeitigen Auftreten der Aktionen (Aufnahme) und (Bearbeitung). Dabei wird im Zustand IS_5 durch das Signal (Eingangstoken alt) zur Ausgangsschicht gemeldet, daß ein am Eingang anliegendes Token schon über das 'Bypass'-Register weitergereicht wurde, wegen eines vollen Moduleingangs aber noch in der Eingangsschicht gehalten werden muß.

Während der Zustände IS_1 bis IS_5 aktiviert die Eingangssteuerung das Signal (Warten), woraufhin ein vorgeordnetes Modul das Token des vorangegangenen Taktzyklus - wiederholt ausgeben muß. Bei den Übergängen zum Zustand IS_0 sowie bei den Übergängen $IS_4 \rightarrow IS_3$ und $IS_4 \rightarrow IS_5$ werden die Freigabesignale (Eingang laden) oder ('Bypass' laden) zum Laden des Moduleingangs oder 'Bypass'-Registers aktiviert.

8.4.2 Steuerung der Ausgangsschicht

Eine detaillierte Übersicht der Ausgangsschicht des Pipelinebus-Interface zeigt Abbildung 8.8. Diese Schicht führt vom Modul generierte Token mit den Token zusammen, welche durch die Eingangssteuerung über die Aktion (Bearbeiten) an die Ausgangsschicht weitergereicht werden. Ein Multiplexer (MUX) kann alternativ den Modulausgang,

[5] In der Darstellung der Zustandsdiagramme werden Zustandsübergänge mit der zugehörigen Übergangsbedingung und, getrennt durch einen Schrägstrich '/', mit aktiven Signalen während des Übergangs gekennzeichnet: (Bedingung)/(Aktive Signale). Diese Bedingungen und aktive Signale besitzen nur während des zeitlichen Übergangsintervalls einen gültigen Wert.
Zustände selbst sind mit einem Zustandsnamen und den aktiven Signalen gekennzeichnet, welche während des gesamten Taktzyklus gültig sind und sich nur beim Übergang in einen Folgezustand ändern.

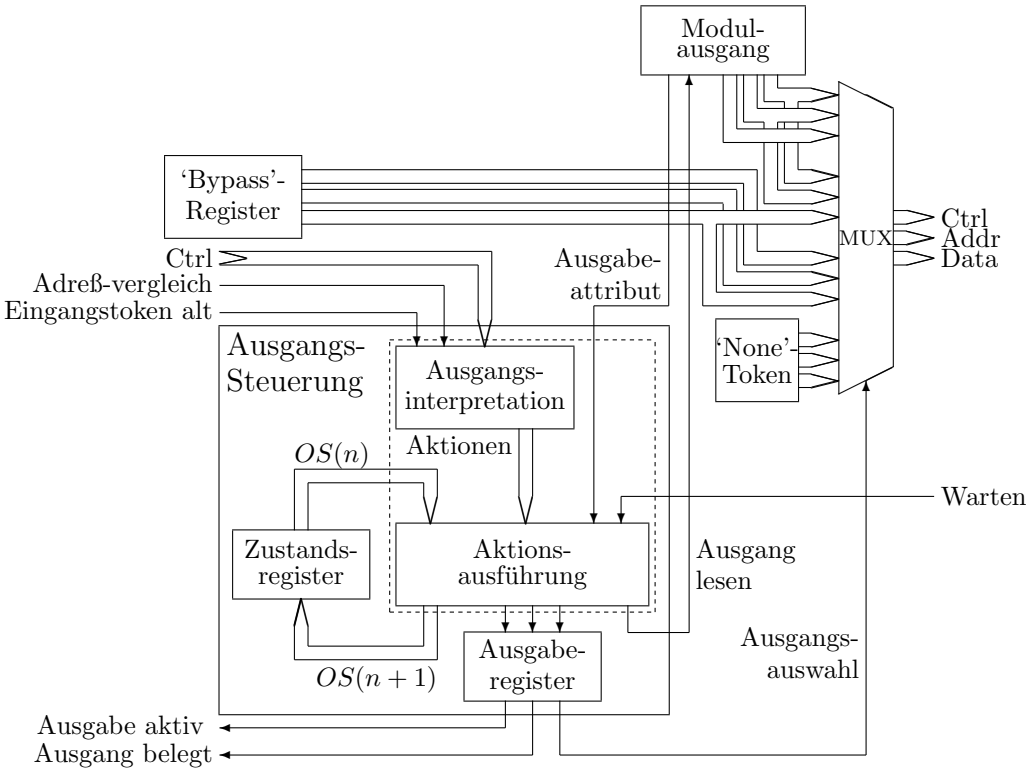

Abbildung 8.8: Ausgangsschicht des schnellen Pipelinebus-Interface.

das 'Bypass'-Register, ein *'None'*-Token oder den Kontroll- und Adreßteil des Modulausgangs kombiniert mit dem Datenteil des 'Bypass'-Register ausgeben. In jedem Buszyklus muß genau eine der vier Multiplexeralternativen ausgewählt werden, damit immer ein definierter Wert am Pipelinebus-Ausgang anliegt.

Die Steuerung der Ausgabe erfolgt durch Interpretation des Kontrollteils (Ctrl) und des Adreßvergleichs der Token am Pipelinebus-Eingang. Die Interpretation der Eingangswerte ermöglicht eine schnelle Reaktion auf ankommende Token durch die Ausgangsschicht. Wenn z.B. die Eingangssteuerung ein Token zur weiteren Bearbeitung in das 'Bypass'-Register lädt, sind die zu einer Ausgabeaktion gehörigen Steuersignale durch die Ausgangssteuerung schon bestimmt und werden aktiviert. Ist das von der Eingangssteuerung generierte Statussignal (Eingangstoken alt) aktiv, werden die anliegenden Werte ignoriert und die Aktion (Ignorierung) aufgerufen.

Aus der Ausgangsinterpretation gewonnene Aktionen werden, wie auch schon in der Eingangsschicht, an die Aktionsausführung weitergegeben. Diese arbeitet als endlicher Automat und führt ebenfalls eine von der Interpretation unabhängige, direkt auf die Aktionen gestützte Verarbeitung durch. Damit beschränkt sich auch in der Ausgangsschicht die Implementierung einer neuen Pipelinebus-Funktionalität auf die Installation einer neuen Interpretationstabelle (z.B. Tabellen 8.1-8.3). Die Ausgangsinterpretation und die Aktionsausführung lassen sich zu einem einzelnen Block zusammenfassen. Dies ist durch den gestrichelt dargestellten Block innerhalb der Ausgangssteuerung angedeutet. In den bisherigen Modulrealisierungen des HARBURGER SYSTEMS wird dieser Block als schneller Speicher realisiert, dessen Inhalt aus den beiden logischen Blöcken errechnet wird.

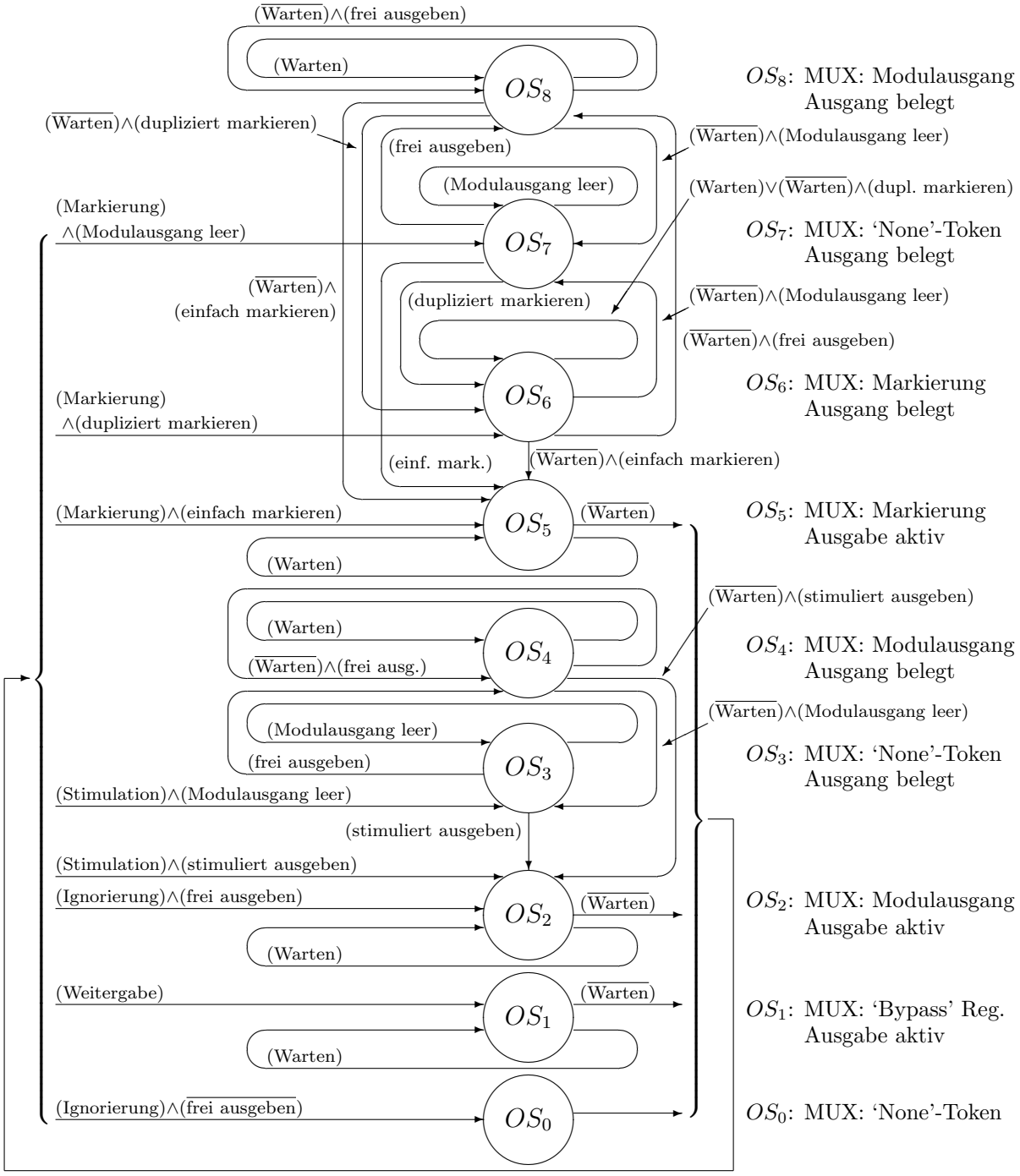

Abbildung 8.9: Zustandsdiagramm zur Aktionsausführung der Ausgangsschicht.

Ausführung der Ausgabeaktionen
Die Ausführung der Ausgabeaktionen unter Beachtung der Attribute am Modulausgang zeigt das Zustandsdiagramm in Abbildung 8.9. Die Aktionsausführung erfolgt in diesem Diagramm von links nach rechts. Die linken Kanten zeigen die möglichen Übergänge zum Starten von Aktionen. Sie führen, in Abhängigkeit von Ausgabeaktionen und Attributen, in einen Zustand aus der Menge $\mathcal{OS}_S = \{OS_0, OS_1, OS_2, OS_3, OS_5, OS_6, OS_7\}$. Die aus den Zuständen der Menge $\mathcal{OS}_T = \{OS_0, OS_1, OS_2, OS_5, \}$ nach rechts wegführenden, mit ($\overline{\text{Warten}}$) markierten Kanten[6] terminieren eine Aktion und bewirken durch die Rückführung auf die linke Seite den Start der Folgeaktion. Jede Terminierungskante kann mit jeder Startkante zu einer Übergangskante kombiniert werden, was im Diagramm zu $|\mathcal{OS}_S \times \mathcal{OS}_T| = 28$ Kanten führt. Diese Kanten sind nicht explizit dargestellt, sondern durch die Klammerung angedeutet. Die Bedingung einer Kante zum Übergang in eine neue Aktion ergibt sich aus der Terminierungsbedingung ($\overline{\text{Warten}}$) und der neuen Startbedingung.

Beispielsweise wird beim Start der Aktion (Weitergabe) der Zustand OS_1 eingenommen. Dieser Zustand wählt am Ausgabemultiplexer das 'Bypass'-Register aus und aktiviert das Signal (Ausgabe aktiv) für die Eingangssteuerung. Der Zustand wird verlassen, wenn das Signal (Warten) während eines Taktzyklus inaktiv bleibt und damit die Aktion (Weitergabe) erfolgreich durchgeführt werden konnte.

Den unterschiedlichen Zuständen sind verschiedene Aufgaben zugeordnet. So wird im Zustand OS_3 bei anstehender Aktion (Stimulation) auf zugehörig attributierte Tokendaten vom Modulausgang gewartet . Erscheinen während dieser Wartezeit Tokendaten mit dem Attribut (frei ausgeben), so wird deren Ausgabe über den Zustand OS_4 eingefügt und danach wieder der Wartezustand OS_3 eingenommen. Erst durch ein (stimuliert ausgeben)-attributiertes Token kann die Aktion (Stimulation) über den Zustand OS_2 terminieren.

Die Zustände OS_5 bis OS_8 führen die Aktion (Markierung) aus. In OS_6 wird die Markierung mit einem (dupliziert markieren)-attributierten Token durchgeführt, wobei durch Aktivierung des (Ausgang belegt)-Signals das Datentoken im 'Bypass'-Register erhalten bleibt. OS_7 wartet auf Ausgabetoken bei leerem Modulausgang und OS_8 fügt die Ausgabe eines (frei ausgeben)-attributierten Tokens in die Ausführung der Aktion ein. Die Aktion (Markierung) kann nur durch den Zustand OS_5 verlassen werden, in dem die Markierung des Datentokens durch ein Token mit dem Attribut (einfach markieren) erfolgt.

Die Funktionen der verbleibenden Zustände sollten aus dem Diagramm klar ersichtlich sein.

8.5 Weitere Anmerkungen zum Pipelinebus-Interface

In den obigen Abschnitten wird die Anpassung des Pipelinebus-Interface an die Erfordernisse unterschiedlicher Module zur Realisierung der Lademodi 'Scattering', 'Broadcast' und 'Gathering' aufgezeigt. Die Einführung modulunabhängiger Aktionen reduziert

[6] Die aus dem Zustand OS_0 wegführende Kante ist nicht mit ($\overline{\text{Warten}}$) markiert, terminiert aber auch eine Ausgabeaktion und wird hier mitbetrachtet.

diese Anpassung auf die Erstellung modulspezifischer Ein- und Ausgabetabellen. Die Aktionsausführungen aller Module, realisiert durch verkoppelte, endliche Automaten, können damit durch identische Zustandsdiagramme beschrieben werden.

Durch diese einfache Anpassung wird die Berücksichtigung applikationsspezifischer Anforderungen durch ein Pipelinebus-Interface auf der Ebene der Interpretationstabellen denkbar. Ein möglicher weiterer Lademodus könnte z.B. einem laufenden Datenstrom markierte Werte entnehmen, gleichzeitig aber auch den vollständigen Datenstrom für eine überlappende Verarbeitung weiterreichen.

Die Anpassung der Aktionsausführungen an Applikationen wird deutlich problematischer und ist zur Realisierung von Multiprozessorlademodi kaum notwendig. Sollen trotzdem Änderungen vorgenommen werden, müssen Vorkehrungen zur Vermeidung von Busblockierungen, welche in der jetzigen Zustandsbeschreibung enthalten sind, in eine neue Beschreibung mit einfließen. Diese Vorkehrungen umfassen eine geeignete Verkopplung der Ein- und Ausgangsschichten, die Beachtung von Warteanforderungen nachfolgender Module sowie eine Prioritätenvorgabe beim Schalten des Ausgangsschichtmultiplexers.

Kapitel 9

Eine parallele Betriebssoftware

9.1 Anforderungen

Ein paralleles Rechnersystem wie das HARBURGER SYSTEM benötigt zur anwenderfreundlichen Nutzung eine verteilte Betriebssoftware. Diese Software muß die problemorientierte Nutzung spezieller Peripheriemodule, aber auch die freie Programmierung des im Rechenkern eingebetteten MIMD-Systems unterstützen.

Eine breite Erfüllung allgemeiner Forderungen, welche bei der parallelen Programmierung auftreten, führt zum aufwendigen Einsatz eines verteilten, parallelen Betriebssystems. Für viele Anwendungen reicht jedoch eine Untermenge der vollen Betriebssystemfunktionalität aus, was den Entwurf einer zum Betrieb notwendigen Software vereinfacht. Die verteilte Betriebssoftware des HARBURGER SYSTEMS [Lan90b] bildet solch eine Untermenge. Die Aufgaben dieser Software umfassen:

- Abfrage von Statusinformation,
- Reaktion auf Fehlerereignisse mit Meldung zum übergeordeten Host,
- Starten von Anwenderprozessen auf Systemmodulen (Scheduling),
- Organisation von Ein-/Ausgabemöglichkeiten für Anwenderprozesse,
- Verwaltung von Hardware-Schnittstellen der Module über Treiberprozesse, so daß Anwenderprozesse auf problemorientierte Schnittstellen zugreifen können.

Auf ein verteiltes File-System, welches jedem Anwenderprozeß freien Zugriff auf Dateien gibt, wurde verzichtet.

Die parallele Organisation der Aufgaben bedingt einen Informationsaustausch zwischen den verteilten Betriebssoftware-Prozessen der Systemmodule und benötigt somit eine Unterstützung zur Kommunikation. Im HARBURGER SYSTEM ist eine parallele Organisation der Betriebssoftware mit zentralistischer Steuerung durch den Host vorgesehen.

9.2 Globale Organisation der verteilten Betriebssoftware

Die globale Organisation der verteilten Betriebssoftware umfaßt — in Analogie zum algorithmischen Systementwurf — die Kommunikation der Systemmodule (einschließlich der Auswahl einer geeigneten Verbindungstopologie) und die Festlegung der Modulfunktionalitäten aus der Betriebssoftware-Sicht.

Die Kommunikation muß die Formulierung und Übertragung komplexer Kommandoprotokolle unterstützen, dann können flexible, an Modulfunktionen angepaßte Kommandos definiert werden, welche den Aufruf dieser Funktionen problemorientiert ermöglichen.

Die benötigte Bandbreite der Kommunikation weist im Vergleich zur algorithmischen Kommunikation geringere Anforderungen auf, darf jedoch nicht unterbewertet werden.

Im Harburger System wird die Betriebssoftware-Kommunikation durch den Kommandobus (Abbildung 7.2) unterstützt, der physikalisch aus Transputer-Links aufgebaut ist. Durch Verwendung interruptgesteuerter Link-Adaptoren ist die Übertragungsbandbreite dieses Busses auf ca. $37\,kBytes/s$ beschränkt. Als Topologie der Busverbindung wurde eine Ring gewählt (Abbildung 9.1). Er benötigt nur 2 Ports pro Modul und bewirkt einen einfachen Routingalgorithmus zum Versenden von Kommandos. Bei großen Systemen kann diese Topologie nicht beibehalten werden, da mit der Modulanzahl der Netzwerkdurchmesser linear wächst und zum Flaschenhals wird. Dann muß eine andere, z.B. hierarchisch strukturierte Topologie die Betriebssoftware unterstützen und insbesondere in deren Routingprozesse eingearbeitet werden.

Zur Kommunikation dient ein Protokoll aus vier Feldern:

$$[\textit{(Modulindex),(Kommandolänge),(Funktion), (Parameter)}]$$

Es beinhaltet die zwei universellen Feldern (Modulindex) und (Kommando Länge). Diese Felder ermöglichen jedem Modul durch die Betriebssoftware-Prozesse 'In' und 'Out' ein vom Kommandoinhalt unabhängiges Routen. Das Feld (Modulindex) adressiert das Zielmodul des Kommandos, wobei die Moduladressen beim Systemstart festzulegen sind. Das Feld (Kommandolänge) enthält die Gesamtlänge des Kommandos, so daß ein Weiterrouten fremder Kommandos ohne Kenntnis der Funktion und Parameteranzahl möglich ist. Die speziellen Felder (Funktion) und (Parameter) werden vom adressierten Modul in Abhängigkeit der eigenen Funktionalität interpretiert.

Mit der Betriebssoftware erhält der Anwender Unterstützung beim Laden und Ausführen von Programmen. Insbesondere die freie Programmierung der in Abschnitt 7.2.1 vorgestellten Transputer-Module wird erleichtert. Ein Anwenderprozeß kann über lokale, Hardware-unabhängige Schnittstellen die Ressourcen eines Moduls (Pipelinebus-Eingabe und -Ausgabe, dynamische Verwaltung des Systemspeichers, virtuelles Terminal als 'Debug'-Hilfe, etc.) problemorientiert nutzen.

Das Konfigurieren eines globalen, parallelen Programms zur Startzeit [Pou90] wird durch die Übergabe von Laufzeitparametern ermöglicht. Dies schafft gegenüber einer Konfigurierung zur Übersetzungszeit die Basis zur Realisierung skalierbarer Anwenderprozesse, z.B. bezüglich der Prozessoranzahl und der Problemgröße. Eine Konfigurierung zur

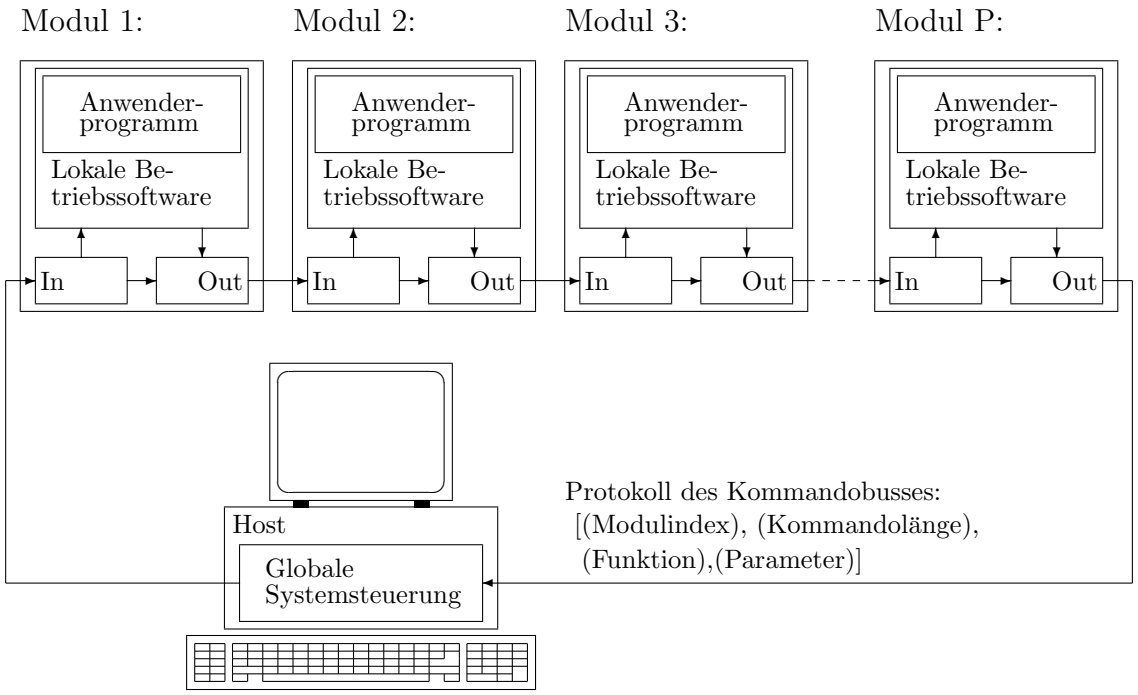

Abbildung 9.1: Globale Organisation der Betriebssoftware des HARBURGER SYSTEMS.

Laufzeit, welche ein dynamisches Verteilen der Rechenlast während der Ausführung vornimmt, ist in dieser Software hingegen nicht vorgesehen.

Weiterhin ermöglicht die Betriebssoftware das Synchronisieren des Systems und die Abfrage von Statusinformation einzelner Module durch den Host. Peripheriemodule (z.B. Bildaufnahme, Verbindungsnetzwerk) sind durch angepaßte Kommandos unterstützt, so daß sie durch den Host problemorientiert aktiviert werden können.

9.3 Lokale Betriebssoftware eines Transputer-Moduls

Beispielhaft für andere Systemmodule wird die Struktur der lokale Betriebssoftware eines Transputer-Moduls aufgezeigt. Die parallele Strukturierung der globalen Betriebssoftware, welche sich zwangsläufig durch die örtliche Verteilung auf die Systemmodule ergibt, wird in dieser Software lokal weiter unterstützt. Die lokale Funktionalität entsteht durch unabhängige, nebenläufige Prozesse, deren Abhängigkeiten zueinander nur über Kommunikationsverbindungen hergestellt werden. Dies führt beim Transputer-Modul[1] zu der in Abbildung 9.2 dargestellten lokalen, parallelen Strukturierung. Sie ist in der Sprache OCCAM (Abschnitt 4.2.3) nebenläufig formuliert, wobei lokale Kommunikationsverbindungen auf Software-Links abgebildet sind.

[1] An dieser Stelle wird nicht zwischen den unterschiedlichen Funktionalitäten eines Transputer-Moduls unterschieden. Ein Rechenmodul, eine Markierungseinheit oder ein Bildspeicher lassen sich weitgehend mit der gleichen Prozeßstruktur der lokalen Betriebssoftware betreiben.

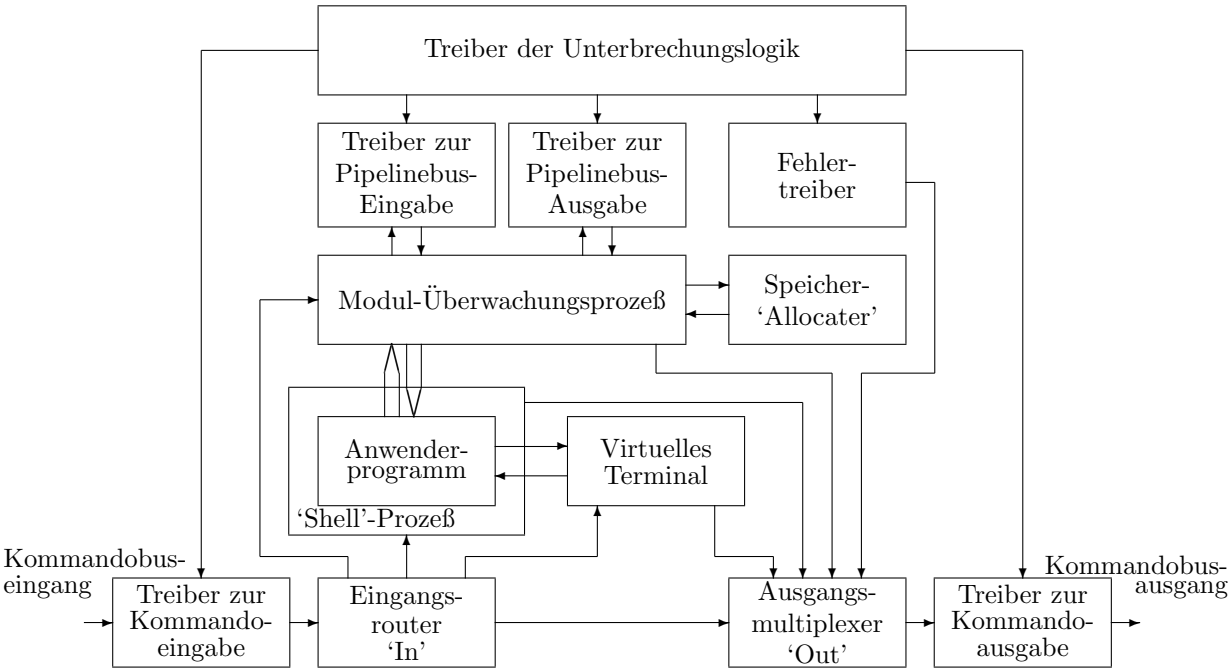

Abbildung 9.2: Parallele Prozeßstruktur der lokalen Betriebssoftware eines Transputer-Moduls.

Im oberen Bereich der Abbildung 9.2 und beim Kommandobuseingang und -ausgang sind Hardware-spezifische Treiberprozesse eingebunden, welche zugehörige Hardware-Einheiten (Abbildung 7.3) unterstützen. In direkter Korrespondenz zur physikalisch parallel vorliegenden Hardware sind diese Treiber, ermöglicht durch die nebenläufige Programmformulierung in Occam, (quasi-)parallel den Hardware-Einheiten zugeordnet.

Die *Treiber zur Kommandoeingabe und -ausgabe* steuern die Hardware zur Betriebssoftware-Kommunikation. Die *Treiber zur Pipelinebus-Eingabe und -Ausgabe* steuern den Datentransfer zwischen dem Pipelinebus-Interface und der sequentiellen Speicherschnittstelle. Ein *Fehlertreiber* wird durch erkannte Hardware-Fehler auf dem Modul aktiviert. Ein *Treiber der Unterbrechungslogik* übernimmt die Kontrolle dynamischer Hardware-Ereignisse (Interrupts). Er ist den anderen Treibern übergeordnet, wertet dynamische Ereignisse aus und meldet diese über Software-Links an zugehörige Treiberprozesse weiter.

Eingebettet in die Umgebung der Hardware-spezifischen Treiber sind Prozesse, welche dem Modul die gewünschte Anwendersicht verleihen. Diese umfassen in der derzeitigen Software einen 'Shell'-Prozeß, ein virtuelles Terminal und einen Speicher-'Allocater'.

Der *'Shell'-Prozeß* dient zum Ausführen der Anwenderprozesse. Ein Anwenderprozeß wird initialisiert und, nach der Übergabe notwendiger Schnittstellen und Parameter, eingebettet in den 'Shell'-Prozeß gestartet. Beim Terminieren des Anwenderprozesses wird der 'Shell'-Prozeß wieder aktiv.

Das *virtuelle Terminal* stellt lokal eine Bildschirm- und Tastaturschnittstelle zur Verfügung, welche zunächst auf dem Modul simuliert, aber durch Kommandos vom Host mit dessen physikalischer Ein- und Ausgabe verbunden werden kann. Dieses virtuelle Terminal

gibt dem Anwender Systemtransparenz bis hinunter auf die Modulebene, so daß dieser bei Bedarf vom Host aus jedes Modul erreichen und generierte Kontrollausgaben eines lokalen Anwenderprozesses verifizieren kann.

Der *Speicher-'Allocater'* dient zur dynamischen Verwaltung des Systemspeichers. Ein Anwenderprozeß kann Speicher zum Aufbau dynamischer Datenobjekte anfordern und nach Verwendung wieder freigeben.

Zur oben geforderten Abfrage von Statusinformation beinhaltet die lokale Betriebssoftware einen *Prozeß zur Modulüberwachung*. Der Anwender hat keinen direkten Zugriff auf diesen Prozeß, jedoch werden dort seine Zugriffe auf die Ressourcen des Moduls überwacht. Der Überwachungsstatus kann über Kommandos vom Host abgefragt werden. Er beinhaltet z.B. Informationen über Probleme beim Allozieren von Speicher oder über hängende Anforderungen einer Pipelinebus-Eingabe oder -Ausgabe aufgrund fehlerhaft unausgewogener globaler Kommunikationen. Weiterhin ermöglicht dieser Prozeß dem Host den Zugriff auf interne Modulregister (z.B. Pipelinebus-Adresse) und damit ein Konfigurieren des Systems.

Ein *Eingangsrouter 'In'* interpretiert am Kommandobus eintreffende Kommandos. Er vergleicht das (Modulindex)-Feld des Kommandoprotokolls (Abbildung 9.1) mit einer eigenen, während der Systeminitialisierung ermittelten Adresse. Bei Gleichheit routet er die nachfolgenden Funktions- und Parameterfelder zum Zielprozeß innerhalb der lokalen Betriebssoftware. Bei Ungleichhheit wird das vollständige Kommando zum *Ausgangsmultiplexer 'Out'* weitergereicht. Dieser kombiniert von der Eingangsseite weitergereichte Kommandos und von lokalen Betriebssoftware-Prozessen generierte, für den Host bestimmte Statuskommandos zu einem Ausgabestrom und leitet diesen Strom an den Treiber zur Kommandoausgabe weiter.

Eine weitergehende Beschreibung der Betriebssoftware eines Transputer-Moduls, welche das Software-Interface der Anwenderprozesse und die implementierten Host-Kommandos umfaßt, findet man in [Lan90b].

Kapitel 10

Parallele Implementierung von Algorithmen

Die folgenden Beispiele zur Implementierung von Algorithmen auf dem HARBURGER SYSTEM zeigen die Funktionsfähigkeit des vorgestellten Systemkonzepts auf. Die Implementierungen erfolgten auf dem in Abschnitt 7.2.5 vorgestellten Prototypen mit der verteilten Betriebssoftware nach Kapitel 9. Beginnend mit Messungen der Ladezeit im System steigen die Systemanforderungen aufeinanderfolgend betrachteter Algorithmen.

Im ersten Abschnitt wird die Datenübertragung auf dem schnellen Pipelinebus untersucht. Diese Messungen geben Aufschluß über die Leistungsparameter des Busses aus der Sicht der Anwenderprogramme.

Der zweite Abschnitt beschreibt die Implementierung einer parallelen Bildrotation. Dieser Algorithmus benötigt ein schnelles, verteiltes Laden und Entladen von Bildern, die algorithmische Verarbeitung erfolgt jedoch lokal ohne Kommunikation.

Die Implementierung der zweidimensionalen Fouriertransformation stellt Anforderungen an ein schnelles, verteiltes Laden und Entladen von Daten und benötigt während der algorithmischen Verarbeitung einen hohen, globalen Kommunikationsbedarf.

Als weiteres Beispiel zur parallelen Implementierung eines komplexen Algorithmus wird im letzten Abschnitt ein Programm zur Bewegungsschätzung in Bildfolgen vorgestellt, welches neben den Ladeanforderungen ebenfalls einen globalen, algorithmischen Kommunikationsbedarf besitzt.

Die Implementierungssprache aller Algorithmen ist OCCAM und es werden keine Assembler-Routinen zur Laufzeitoptimierung eingebunden. Für die Zeitmessungen werden die durch OCCAM unterstützten internen Zähler des Transputers verwendet. Damit können detaillierte interne Zeitmessungen durch Software-Aufrufe in die Programme eingefügt werden.

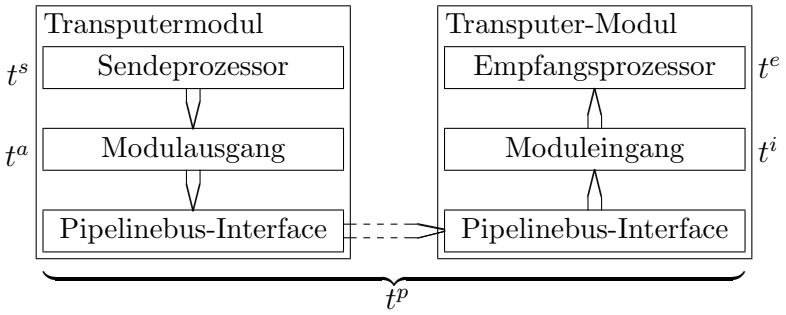

Abbildung 10.1: Anordnung zur Analyse der Datenübertragung zwischen zwei Transputer-Modulen.

10.1 Übertragung von Daten auf dem schnellen Pipelinebus

Zum Laden und Entladen von Daten wird ein Datenblock von einem Sender- zu einem Empfängermodul über den schnellen Pipelinebus verschoben. Für die Betrachtung der Übertragung ist es dabei unerheblich, ob ein Sender oder Empfänger aus einem einzelnen Modul besteht oder z.B. durch einen Rechenkern repräsentiert wird. Die Übertragungzeiten zwischen zwei Einzelmodulen entsprechen nahezu den Zeiten zur Realisierung der drei Lademodi 'Broadcast', 'Scattering' und 'Gathering'. Besonders bei großen Datenblöcken ist die Abhängigkeit der Pipelinebus-Übertragung von der Prozessoranzahl P vernachlässigbar (Abschnitt 8.3.2). Als Ladezeiten können daher die beispielhaft für zwei Transputer-Module ermittelten Übertragungszeiten verwendet werden.

Abbildung 10.1 zeigt die zur Zeitermittlung verwendete Anordnung. Durch Austausch des Sende- und Empfangsprozessors durch andere Sende- oder Empfangseinheiten läßt sich die Anordnung und das im folgenden aufgestellte Zeitmodell direkt für andere Systemmodule verwenden.

Die Übertragung kann in 5 Schichten zerlegt werden, woraus 5 Zeiten resultieren. Aus diesen Einzelzeiten setzt sich die Gesamtübertragungszeit t zusammen:

t^s: Will ein Anwenderprogramm einen Datenblock ausgeben, so verstreicht eine Latenzzeit, bis die Ausgabe des Blocks in den Modulausgang erfolgt. Diese Latenzzeit wird durch die lokale Betriebssoftware (Abbildung 9.2) und dort insbesondere durch den Modulüberwachungsprozeß und den Treiber zur Pipelinebus-Ausgabe verursacht. Während der Datenübertragung müssen die Daten aus dem Speicher zum Pipelinebus-Interface transferiert werden. Die Zeit t^s setzt sich aus der Latenz- und der Transferzeit zusammen.

t^a: Der Modulausgang bewirkt eine Verzögerung t^a für Daten, z.B. durch FIFOs zur Datenratenanpassung oder durch Synchronisierungen zwischen dem Sendeprozessor und dem Pipelinebus-Interface.

t^p: Die Zeit t^p beschreibt die Datenübertragung auf dem schnellen Pipelinebus. Bei der Ermittlung dieser Zeit muß beachtet werden, ob zur Übertragung der Bus exklusiv zur Verfügung steht oder andere unabhängige Übertragungen den Bus belasten.

Tabelle 10.1: Parameter der Übertragungszeit, gemessen mit Transputer-Modulen des Prototypen des HARBURGER SYSTEMS [Lan90a].

	Initialisierung	Übertragung
t^s	$113\,\mu s$	$152\,ns$
t^a	$25 \ldots 60\,ns$	$35 \ldots 70\,ns$
t^p	$P \cdot 100\,ns$	$100\,ns$
t^i	$25 \ldots 60\,ns$	$35 \ldots 70\,ns$
t^e	$109\,\mu s$	$139\,ns$
t	$222\,\mu s$	$152\,ns$

Im folgenden soll eine exklusive Busbenutzung angenommen werden. Damit kann Gleichung 8.2 (Seite 101) zur Berechnung von t^p verwendet werden.

t^i: Die Zeit t^i beschreibt in direkter Analogie zu t^a die Verzögerung des Moduleingangs auf der Empfängerseite. Beide Werte entsprechen einander.

t^e: Die Zeit t^e zum Aktivieren und Lesen des Moduleingangs durch einen Anwenderprozeß auf der Empfängerseite über die lokale Betriebssoftware korrespondiert mit obiger Zeit t^s der Senderseite.

Bedingt durch den internen Aufbau der einzelnen Schichten kann für die zugehörigen Zeiten t^s, t^a, t^p, t^i, t^e und damit auch für die gesamte Übertragungszeit t eines Datenblocks der Größe N ein lineares Zeitmodell aus einer Initialisierungszeit t_0 und der Übertragungszeit $N \cdot t_{\ddot{u}}$ angesetzt werden:

$$t = t_0 + N \cdot t_{\ddot{u}} \tag{10.1}$$

$$t_0 = t_0^s + t_0^a + t_0^p + t_0^i + t_0^e \tag{10.2}$$

$$t_{\ddot{u}} = \max(t_{\ddot{u}}^s, t_{\ddot{u}}^a, t_{\ddot{u}}^p, t_{\ddot{u}}^i, t_{\ddot{u}}^e) \tag{10.3}$$

Messungen der unterschiedlichen Parameter an der ersten Version des Transputer-Moduls [Lan90a] bestätigen das angesetzte lineare Zeitmodell und ergeben sehr gute Übereinstimmungen mit den aufgrund des Aufbaus erwarteten Parametern. Die Ergebnisse der Messungen sind in Tabelle 10.1 zusammengestellt und führen zu den Parametern t_0 und $t_{\ddot{u}}$ für die Gesamtübertragungszeit:

$$t \approx 222\,\mu s + N \cdot 152\,ns \qquad \Rightarrow \qquad \begin{array}{l} t_0 \approx 222\,\mu s \\ t_{\ddot{u}} \approx 152\,ns \end{array} \tag{10.4}$$

Der zweite Entwurf des Transputer-Moduls [Lut90] (Abschnitt 7.2.1) paßt die Zeit $t_{\ddot{u}}$ durch Verbesserung der Zeiten $t_{\ddot{u}}^s$ und $t_{\ddot{u}}^e$ an die Zykluszeit $t_{\ddot{u}}^p = 100\,ns$ des Pipelinebusses an. In diesem Entwurf wird die Zykluszeit $t_{\ddot{u}}^p$ des Pipelinebusses zur begrenzenden Größe, so daß dort das Erhöhen der Busfrequenz zur Steigerung der Übertragungsrate notwendig ist.

Die für t_0 ermittelten Werte zeigen, daß diese Latenzzeit hauptsächlich durch die Treiber und den Überwachungsprozeß der lokalen Betriebssoftware verursacht wird. Die Reduktion dieser Zeit ist nur durch Reduktion von Software-Funktionalität möglich, so daß z.B. eine Überwachung und somit Systemtransparenz gegen eine Verbesserung der Latenzzeit ausgetauscht werden kann.

10.2 Bildrotation

Die Bildrotation ist ein Beispiel für die Koordinatentransformation $\mathbf{I}_2(\mathcal{I}) = \mathbf{I}_1(\mathbf{t}(\mathcal{I}, \mathbf{T}))$ eines Bildes $\mathbf{I}_1$ nach $\mathbf{I}_2$. Die Transformation $\mathbf{t}$ bildet die zweidimensionale Indexmenge $\mathcal{I} = I_0 \times I_1 = \{\, \mathbf{i} \mid \mathbf{i} = (i_0, i_1),\ i_0 \in I_0,\ i_1 \in I_1\}$, welche die räumliche Topologie eines Bildes beschreibt, auf sich selbst ab. Zur Ermittlung eines Ergebnispunktes $\mathbf{I}_2(\mathbf{j}) \in \mathbf{I}_2(\mathcal{I})$ muß bei nicht spezifizierten Parametern $\mathbf{T}$ wegen des unbekannten Quellindex $\mathbf{i} = \mathbf{t}(\mathbf{j}, \mathbf{T})$ das gesamte Ausgangsbild $\mathbf{I}_1(\mathcal{I})$ vorgehalten werden:

$$\mathbf{I}_2(\mathbf{j}) = \mathbf{I}_1(\mathbf{i}) = \mathbf{I}_1(\mathbf{t}(\mathbf{j}, \mathbf{T})) \tag{10.5}$$

Bei Verwendung diskreter Indexmengen (z.B. $\mathcal{I} = I\!N_N \times I\!N_N$) kann die exakte Berechnung der Transformation einen Indexwert $\mathbf{i}^* = \mathbf{t}(\mathbf{j}, \mathbf{T})$ ergeben, welcher nicht zur Indexmenge gehört: $\mathbf{i}^* \notin \mathcal{I}$. Für diesen Fall sind zwei Lösungswege üblich: Der erste, als *'Nächster Nachbar' Interpolation* bekannte Weg rundet den ermittelten Indexwert $\mathbf{i}^*$ auf den Wert der Indexmenge $\mathcal{I}$ mit dem geringsten Abstand[1]. Ein zweiter Weg interpoliert einen Punkt in einem hypothetischen Bild $\mathbf{I}_1^*(\mathbf{i}^*)$. Diesem hypothetischen Bild liegt eine Indexmenge $\mathcal{I}^* \supset \mathcal{I}$ zugrunde, welche den ermittelten Index beinhaltet: $\mathbf{i}^* = \mathbf{t}(\mathbf{j}, \mathbf{T}) \in \mathcal{I}^*$. Die Interpolation dieses Bildpunktes erfolgt über eine Umgebung $\mathcal{U}_{\mathbf{i}^*}(\mathbf{I}_1(\mathcal{I}), \mathcal{F})$.[2] Bei einer zweidimensionalen, diskreten Indexmenge (z.B. $I\!N \times I\!N$) und einer zweidimensionalen, linearen Interpolation 1.Ordnung über die Umgebung eines 2×2 Fensters $\mathcal{F}$ führt dieses Vorgehen zur *Bilinearen Interpolation*.

Obige Betrachtungen zur Interpolation gelten, wenn $\mathbf{i}^*$ zwischen Punkten der Indexmenge $\mathcal{I}$ liegt. Bei endlichen Indexmengen wie z.B. $I\!N_N \times I\!N_N$ kann jedoch ein errechneter Index weit entfernt liegen. Dann muß eine Randbehandlung z.B. durch Annahme eines konstanten Wertes für $\mathbf{I}_1(\mathbf{i}^*)$ durchgeführt werden (Siehe dazu auch Fußnote auf Seite 19).

Die Bildrotation ergibt als Transformation $\mathbf{t}$ die Multiplikation des zweidimensionalen Zielindex $\mathbf{j} = (j_0, j_1)$ mit einer Drehmatrix $\mathbf{A}(\varphi)$. Als Drehpunkt werde der Ursprung $(0, 0)$ angenommen. Dann läßt sich ein rotiertes Bild $\mathbf{I}_2(\mathcal{I})$, welches aus einem Bild $\mathbf{I}_1(\mathcal{I})$ durch Rotation um den Winkel φ entsteht, wie folgt beschreiben:

$$\mathbf{I}_2(\mathcal{I}) = \{\mathbf{I}_2(\mathbf{j}) \mid \mathbf{I}_2(\mathbf{j}) = \mathbf{I}_1(\mathbf{t}(\mathbf{j}, \varphi)) = \mathbf{I}_1(\mathbf{A}(\varphi) \cdot \mathbf{j}),\ \mathbf{j} \in \mathcal{I}\} \tag{10.6}$$

Mit der Matrix $\mathbf{A}(\varphi)$:

$$\mathbf{A}(\varphi) = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \tag{10.7}$$

Implementierung:
Die Implementierung der Bildrotation auf dem HARBURGER SYSTEM berechnet aus einem $N \times N$ Quellbild $\mathbf{I}_1(\mathcal{I})$ nach Gleichung 10.6 ein Bild $\mathbf{I}_2(\mathcal{I})$ gleicher Größe. Die Ergebnisse des Produkts aus Zielindex $\mathbf{j}$ und Drehmatrix $\mathbf{A}(\varphi)$ liegen aufgrund der trigonometrischen Funktionen nicht immer im diskreten Indexraum $I\!N_N \times I\!N_N$. Es wird daher eine 'Nächster

[1] Bezüglich einer dem Bild angepaßten Metrik.

[2] Die Definition der Umgebung muß an dieser Stelle gegenüber Gleichung 3.17 erweitert werden, da hier der Bezugspunkt $\mathbf{i}^*$ nicht zu der Indexmenge $\mathcal{I}$, aus der die Umgebungspunkte entnommen werden, gehört.

Nachbar'-Interpolation für zwischenliegende Indexpunkte durchgeführt und ein konstanter Wert für Randpunkte des Bildes $\mathbf{I}_1(\mathcal{I})$ gewählt.

Zur Parallelisierung des Algorithmus wird die Indexmenge des Ergebnisbildes $\mathbf{I}_2$ segmentiert. Jedem der P Prozessoren wird die Berechnung eines gleichgroßen Indexsegments zugewiesen. Es erfolgt eine Zerlegung in P horizontale Streifen, so daß jeder Prozessor nur ein Segment der Größe $N \times (N/P)$ bearbeiten muß. Um als Parameter jeden beliebigen Winkel φ vorgeben zu können und einen einfachen Zugriff auf das Quellbild $\mathbf{I}_1$ zu gewährleisten, wird dieses Bild komplett auf jeden Prozessor geladen.

Der Ablauf des Algorithmus umfaßt die Aufnahme und -verteilung des Quellbildes $\mathbf{I}_1$. Dazu wird ein 'Broadcast' (Seite 98) dieses Bildes an alle Rechenmodule mittels Markierung der Daten durch einen Kontrollteil *'All'* durchgeführt. Daran schließt sich die verteilte Berechnung des gedrehten Bildes $\mathbf{I}_2$ an. Nach der Berechnung erfolgt die Datenausgabe durch 'Gathering' (Seite 100), wobei die Rechenmodule gemäß der gewählten horizontalen Segmentierung durch Token mit Kontrollteil *'Empty'* zur Datenausgabe stimuliert werden und das gedrehte Bild in der damit vorgegebenen Reihenfolge ausgeben.

Der implementierte Algorithmus rotiert zu Zeitpunkten T_k aufgenommene Quellbilder $\mathbf{I}_1(T_k), k \in I\!N$ in Sequenz. Eine kurze Zykluszeit ΔT_k wird durch ein 'Pipelining' auf Programmebene erreicht, wobei alle möglichen parallelen Programmsegmente auch parallel ausgeführt werden. Abbildung 10.2 zeigt das Diagramm des realisierten 'Pipelining' auf der Programmebene. Horizontal sind parallel ablaufende Programmteile dargestellt, die vertikale Richtung zeigt den zeitlichen, sequentiellen Ablauf. Dieser zeitliche Ablauf ist in Segmente ΔT_k aufgeteilt, welche jeweils einen Rotationszyklus umfassen. Während der Zyklen ΔT_0 bis ΔT_2 erfolgt die Initialisierung der Pipeline-Verarbeitung. Erst im Zyklus ΔT_3 werden alle zur Ermittlung eines rotierten Bildes notwendigen Programmteile parallel ausgeführt. Das Diagramm läßt sich durch Duplizieren dieses dritten Zyklus bei Aktualisierung der Zeitindizes beliebig fortsetzten.

Das dargestellte Diagramm bezieht sich auf das in Abschnitt 7.2.5 vorgestellte Prototypsystem, welches bei der Rotation ohne Markierungseinheit betrieben wird. Die Markierungsaufgabe verteilt sich daher auf das Bildaufnahmemodul, welches die zum 'Broadcast' notwendige *'All'*-Markierung generiert, und auf das TFG/Transputer-Modul, welches die Stimulation zur Bildaufnahme und zur Ausgabe des Ergebnisbildes vornimmt. Weiterhin übernimmt das TFG/Transputer-Modul auch die Bilddarstellung.

Mit gestrichelt dargestellten Boxen ist im Diagramm die Verarbeitung eines Bildes, welche sich von der Aufnahme bis zur Darstellung über 4 Berechnungszyklen ΔT_i erstreckt, markiert. Die ineinander verschachtelten Sequenzen zur Verarbeitung sukzessiver Bilder ergeben das 'Pipelining' auf der Programmebene.

Mit dieser Pipeline-Programmausführung wurden die in Abbildung 10.3 dargestellten Zeiten für einen Rotationszyklus nach der Initialisierung gemessen. Die Werte zeigen gemittelte Ausführungszeiten über verschiedene Winkel φ aus dem gesamten Intervall $[0 \ldots 2 \cdot \pi)$ und umfassen einen kompletten, mit ΔT_3 in Abbildung 10.2 beispielhaft dargestellten Rotationszyklus ΔT_i. Lediglich die vorgesehene Synchronisation auf ein zum Programmablauf asynchrones Videosignal bei der Bildaufnahme, welches einen nichtlinearen Einfluß auf die Messungen genommen hätte, wurde umgangen.

Einen Vergleich der Meßwerte ermöglicht Abbildung 10.4. Dort sind die Effizienzen (Gleichung 6.12) des parallelen Rotationsalgorithmus dargestellt, welche mit der Bildgröße

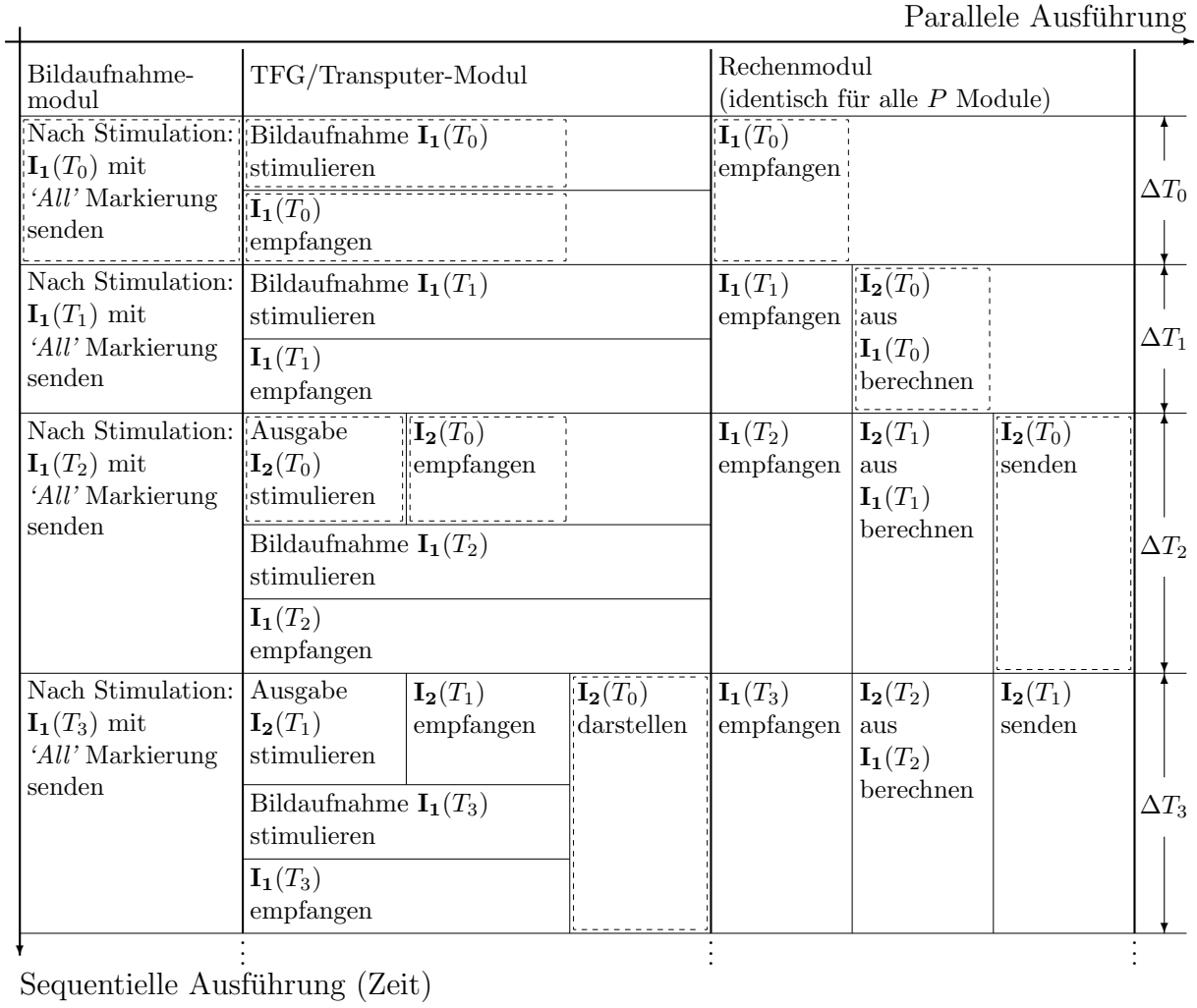

Abbildung 10.2: Diagramm zur sequentiell/parallelen Ausführung der Bildrotation mittels Programm-'Pipelining' auf dem HARBURGER SYSTEM.

$N \times N$ normiert wurden. Die Effizienzen sind auf einem Einzelprozessorsystem nahezu gleich und ergeben die mit 1 skalierte Bezugsgröße. Mit steigender Prozessoranzahl nimmt die Effizienz ab, besonders bei kleinen Datenobjekten (Bildgröße 64×64). Dieser Effizienzabfall resultiert in der Initialisierung einer Übertragung (Gleichung 10.4), welche bei kleinen Datenblöcken gegenüber der Berechnungszeit nicht mehr vernachlässigbar ist. Weiterhin wird der Effizienzabfall durch die noch simulierte Markierungseinheit im Prototypsystem verstärkt. Dies gilt insbesondere beim Auslesen rotierter Bilder bei zunehmender Segmentierung aufgrund höherer Prozessoranzahlen. Hier ist in einem vollständig ausgebauten System noch eine Verbesserung zu erwarten.

10.3 Schnelle Fouriertransformation

Die schnelle Fouriertransformation (FFT) [RG75] steht an dieser Stelle beispielhaft für die in Abschnitt 3.2.2 beschriebene Klasse schneller Transformationen. Die Ausführung dieser Transformationen kann auf verschiedenen, zueinander äquivalenten Topologien erfolgen.

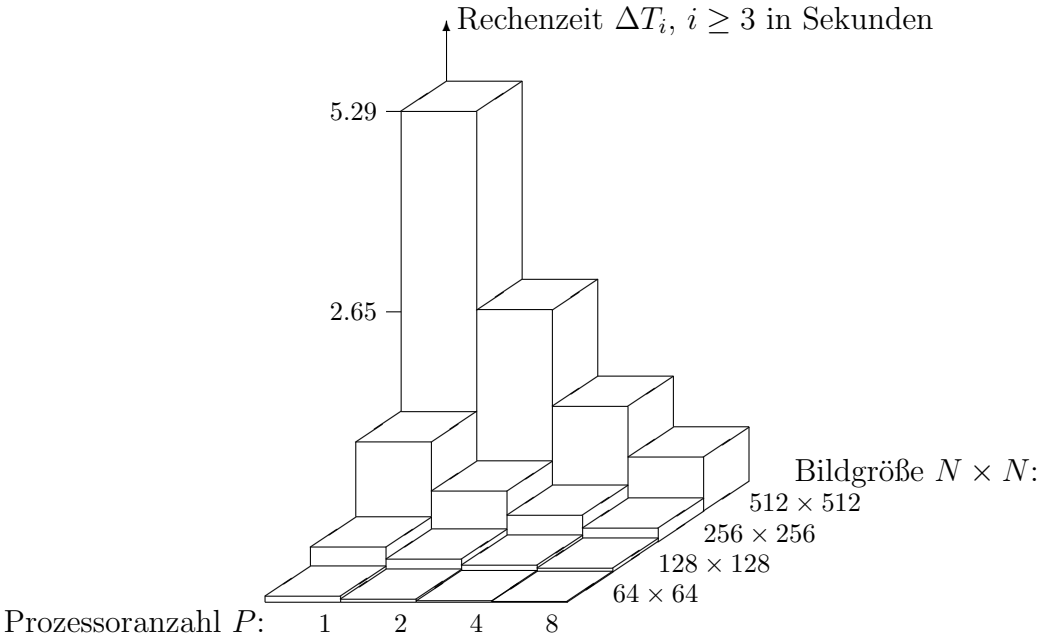

Abbildung 10.3: Rechenzeiten zur Bildrotation.

Die Implementierung der FFT auf dem HARBURGER SYSTEM basiert spaltenweise auf dem skalierbaren, zweistufigen Graphen $\mathcal{G}_{G3}(q)$ (Gleichung 3.48) und zeilenweise auf dem Graphen $\mathcal{G}_G$ der 1. kanonischen Form (Gleichung 3.27). Die Anpassung an die Anzahl der eingesetzten Prozessoren (Reskalierung) erfolgt zur Ladezeit des Programmes durch die Aufteilung der Bildspalten in P Segmente und deren Zuordnung zu den Rechenmodulen. Zur Minimierung der Kommunikation bei gutem 'Load-Balancing' wird $\mathcal{G}_{G3}(q)$ mit $q = \log_2(P) + 1$ skaliert. Horizontal werden komplette Zeilen einem Rechenmodul zugeordnet.

Die Ausführung der FFT erfolgt mit P Rechenmodule und dem TFG/Transputer-Modul des Prototypsystems (Abbildung 7.7). Die Rechenmodule realisieren eine parallele, zweidimensionale $N \times M$ FFT, das TFG/Transputer-Modul führt die Bildaufnahme und -wiedergabe sowie die Markierungsaufgaben aus. Abbildung 10.5 zeigt einen Zyklus zur Berechnung der FFT eines Bildes. Nebenläufige Programmblöcke sind im Diagramm horizontal angeordnet. In vertikaler Richtung ist der sequentielle Programmablauf dargestellt.

Das TFG/Transputer-Modul führt zyklisch die Bildaufnahme, das Laden und Entladen der P Rechenmodule und die Anzeige eines transformierten Bildes aus. Bei der Berechnung der FFT mit der verwendeten Topologie tritt zeilen- und spaltenweise ein 'Bit-Reverse' auf. Die Auflösung dieses 'Bit-Reverse' wird mit der vorgegebenen Verteilung der Bilder in eine globale und lokale Auflösung zerlegt. Die globale Auflösung ist in den Adressen der Token beim 'Gathering' zum Entladen des Ergebnisbildes aus den P Rechenmodulen berücksichtigt und benötigt zur Laufzeit keinen Aufwand. Die lokale Auflösung erfolgt in jedem Berechnungszyklus verteilt auf den Rechenmodulen (Abbildung 10.5).

Die Programmausführung des TFG/Transputer-Moduls weist ein 'Pipelining' auf. Im aktuellen Berechnungszyklus erfolgt die Aufnahme eines Bildes für den folgenden Zyklus

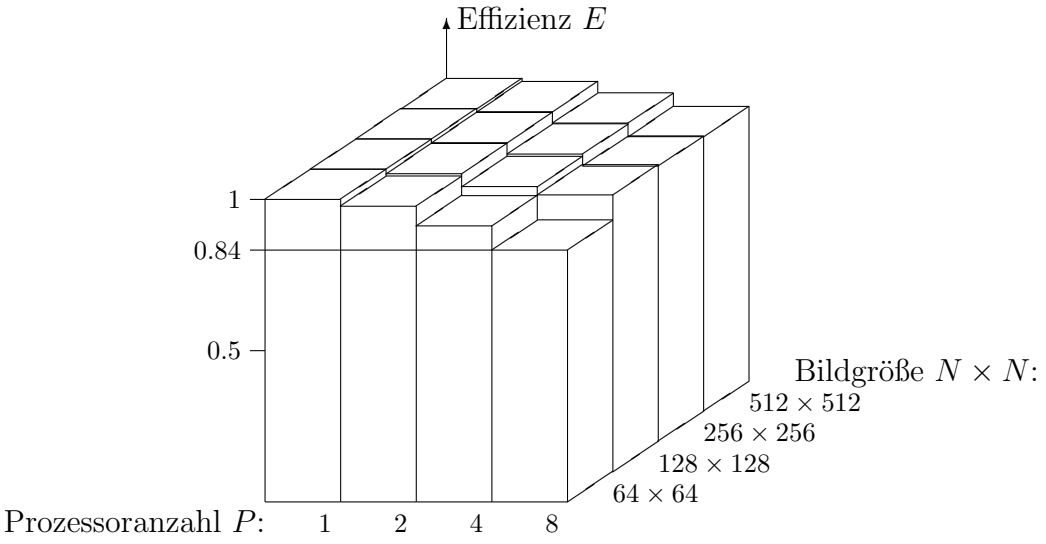

Abbildung 10.4: Effizienzen bei der Ausführung der Bildrotation.

sowie die Darstellung des im vorhergehenden Zyklus transformierten Bildes. Dadurch kann — bei genügend langer Berechnungszeit der FFT auf den Rechenmodulen — die Bildaufnahme und -anzeige verdeckt erfolgen und nimmt somit keinen Einfluß auf die Zykluszeit ΔT_{FFT_Z} der FFT-Ausführung.

Die zweidimensionale FFT-Ausführung auf den Rechenmodulen erfolgt durch Spaltengefolgt von Zeilentransformationen. Die Schicht der Eingangsecken und die darauffolgende erste Kommunikationsschicht wird durch den Ladevorgang ('Scattering') realisiert, wobei die Integration der Kommunikation keinerlei Laufzeitaufwand verursacht. Für die Spaltentransformationen des Graphen $\mathcal{G}_{G3}$ verbleiben damit $\log_2(P)$ parallele Berechnungschichten mit nachfolgenden Kommunikationsschichten und $\log_2(M/P)$ lokale Berechnungsschichten ohne Kommunikationen. Die Zeilentransformationen werden, da alle Daten lokal vorliegen, als 'In-Place' Algorithmus nach der 1. kanonischen Form $\mathcal{G}_G$ (Gleichung 3.27) berechnet.

Durch Programm-'Pipelining' mit dem Ladevorgang überlappt werden, sobald benötigte Daten vorhanden sind, zeilenweise auf den Rechenmodulen die Berechnungen der Spaltenoperationen $f_i^{(1)}$ der ersten Schicht und die nachgeordneten Kommunikationen der Schicht 2 ausgeführt (Abbildung 10.5). Die Berechnungen der weiteren $\log_2(P) - 1$ parallelen Spaltenschichten erfolgen überlappt mit den nachfolgenden Kommunikationsschichten. Anschließend kann die Ausführung der verbleibenden $\log_2(M/P)$ Spalten- und aller $\log_2(N)$ Zeilenschichten lokal auf jedem Rechenmodul und ohne Kommunikation erfolgen. Die Auflösung des lokalen 'Bit-Reverse' am Ende des Berechnungszyklus wird schließlich überlappt mit der Bildausgabe durchgeführt.

Die Messungen der Rechenzeiten $\Delta T_{FFT_Z}(P, N, M)$ eines FFT-Zyklus bei zyklisch aufeinanderfolgenden Bildtransformation sind in Abbildung 10.6 zusammengefaßt. Eine Markierung zeigt den Zeitanteil ΔT_{FFT_I} (unterer Teil der Zeitsäulen), der zur Berechnung von $\log_2(M) - 1$ Spalten- und der $\log_2(N)$ Zeilenschichten auf den Rechenmodulen benötigt wird. Die Differenz (oberer Teil der Zeitsäulen) ergibt die Zeit des Bildempfangs

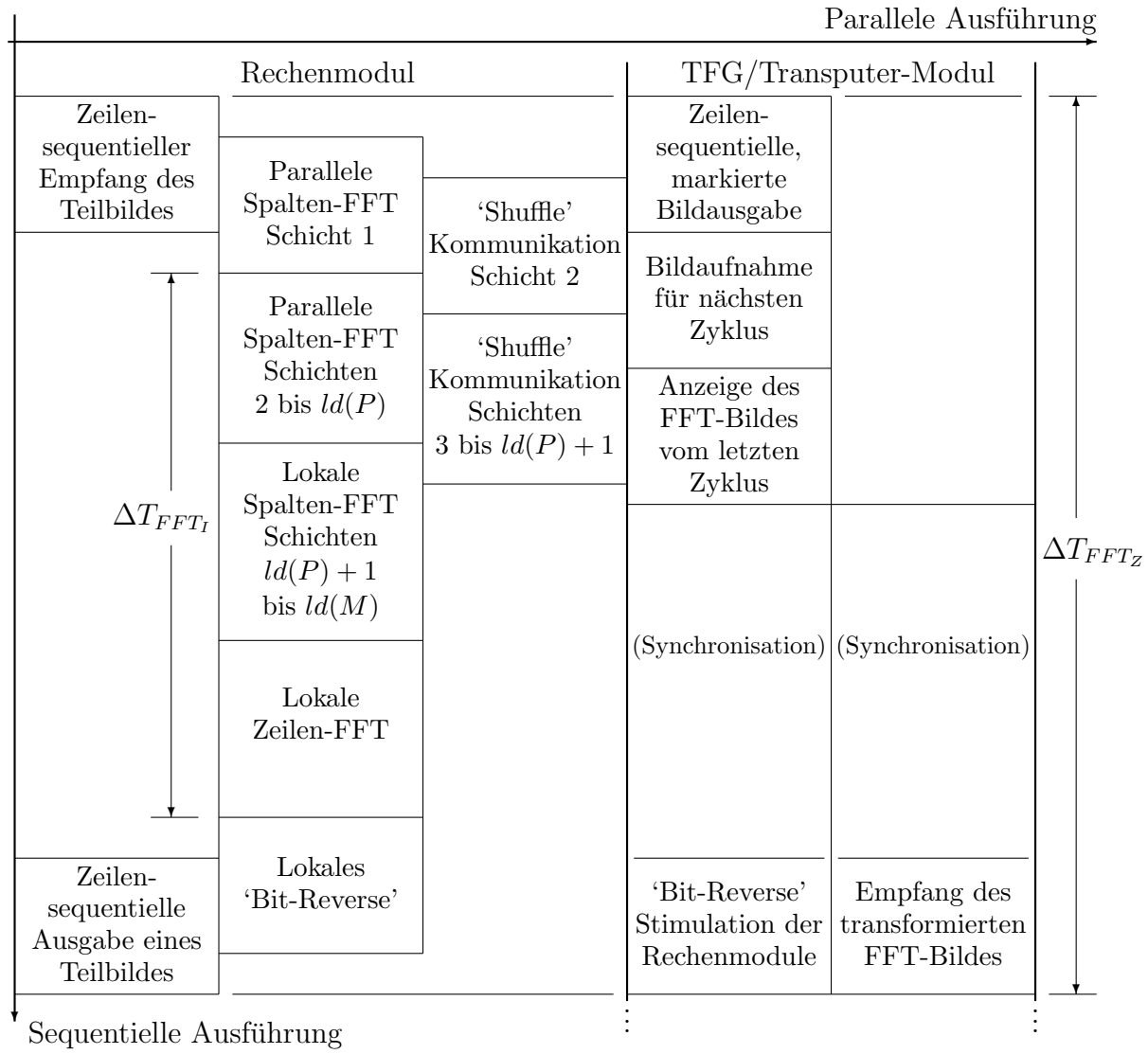

Abbildung 10.5: Diagramm zur parallel/sequentiellen Ausführung der zweidimensionalen, schnellen Fouriertransformation auf dem Prototypen des HARBURGER SYSTEMS.

mit der überlappten Berechnung der ersten Spaltenschicht sowie der Bildausgabe mit dem überlappten lokalen 'Bit-Reverse'.

Einen übersichtlichen Vergleich der Meßwerte ermöglicht die Darstellung der Effizienz. Abbildung 10.7 zeigt die aus den Zykluszeiten ΔT_{FFT_Z} ermittelten Effizienzen über der Prozessoranzahl und Bildgröße. Zur Ermittlung werden die Zeiten ΔT_{FFT_Z} mit der Prozessoranzahl P und der von der Bildgröße abhängigen Gesamtanzahl $N \cdot M \cdot \log_2(N \cdot M)$ durchzuführender Operationen $f_i^{(k)}$ normiert. Dies ergibt eine hypothetische Ausführungszeit einer einzelnen Operation $f_i^{(k)}$ für die durch P, N und M unterschiedenen Konfigurationen. Bezieht man diese normierten Ausführungszeiten auf die beste hypothetische Ausführungszeit (bei den durchgeführten Messungen: $P_0 = 2$ Prozessoren, $N_0 \times M_0 = 256 \times 256$), so erhält man relative Effizienzen bezüglich dieser besten gemesse-

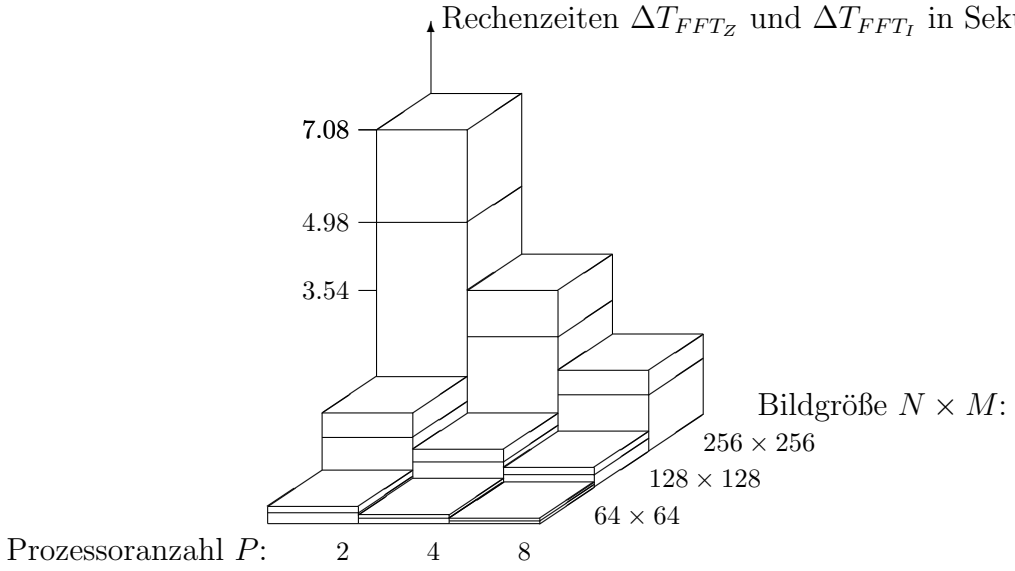

Abbildung 10.6: Ausführungszeiten ΔT_{FFT_Z} zur Berechnung eines vollständigen, zweidimensionalen FFT-Zyklus und interne Zeiten ΔT_{FFT_I} der Rechenmodule (unterer Teil der Zeitsäulen) zur Berechnung von $\log_2(M) - 1$ Spalten- und der $\log_2(N)$ Zeilenschichten.

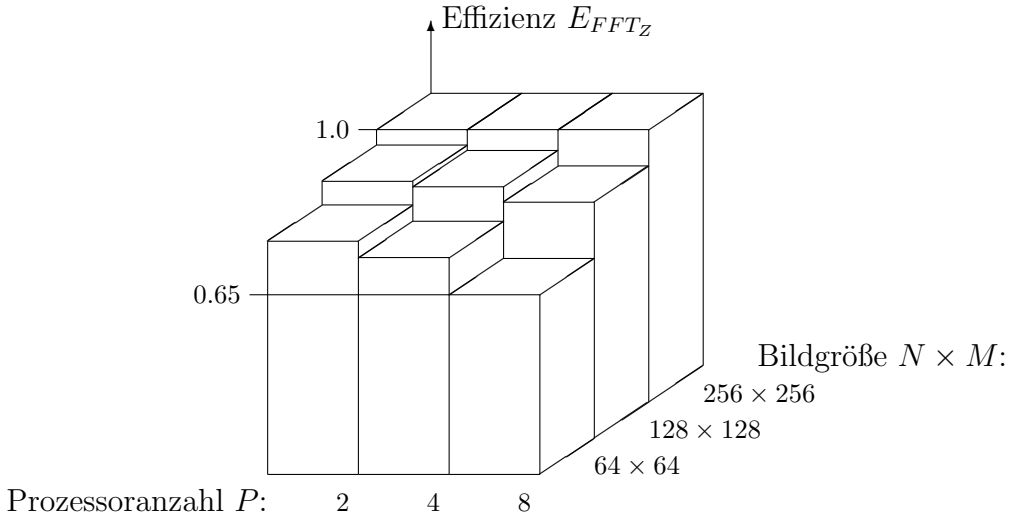

Abbildung 10.7: Normierte Effizienzen der zyklisch wiederholten Berechnung der zweidimensionalen FFT.

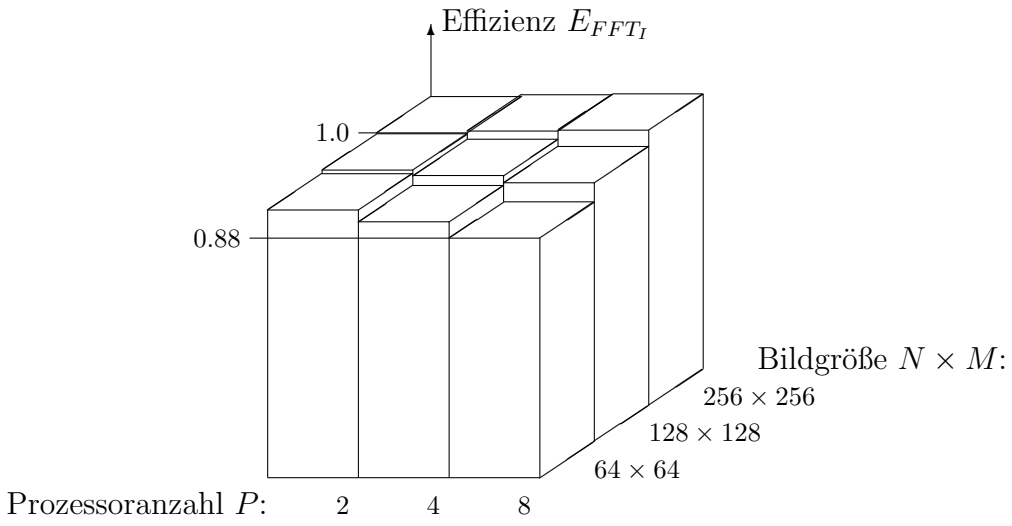

Abbildung 10.8: Erzielte Effizienzen zur Berechnung von $\log_2(M) - 1$ Spalten- und $\log_2(N)$ Zeilenschichten der FFT.

nen Konfiguration:

$$E_{FFT_Z} = \frac{\frac{\Delta T_{FFT_Z} \cdot P_0}{N_0 \cdot M_0 \cdot \log_2(N_0 \cdot M_0)}}{\frac{\Delta T_{FFT_Z} \cdot P}{N \cdot M \cdot \log_2(N \cdot M)}} \tag{10.8}$$

Das Ergebnis (Abbildung 10.7) zeigt bei einer Bildgröße 256×256 Punkten eine weitgehend konstante Effizienz bei Veränderung der Prozessoranzahl. Durch das überlappte Laden und Kommunizieren sowie durch die effiziente Übertragung großer Datenblöcke bestimmt dort nahezu allein die Berechnungsordnung der FFT die Effizienz. Bei abnehmender Bildgröße nimmt der Lade- und Kommunikationseinfluß zu und die Initialisierungszeiten bewirken einen, auch von der Prozessoranzahl abhängigen, Effizienzabfall.

Bestätigt wird dieses Ergebnis durch die internen Transformationszeiten der Rechenmodule ΔT_{FFT_I}, welche keine Lade-, 'Bit Reverse'- und Entladezeiten enthalten. Diese Zeiten zur Berechnung der Spalten- und Zeilenschichten (ohne der mit dem Datenempfang überlappten Spaltenschicht 1) führen bei einer Normierung entsprechend Gleichung 10.8 (mit $P_0 = 2$, $N_0 \times M_0 = 256 \times 256$) zu den in Abbildung 10.8 dargestellten Effizienzen.

Bemerkenswert ist der leichte Effizienzanstieg bei 256×256 Bildern mit steigender Transputeranzahl. Dies ist auf eine effiziente Berechnung der paralleler Spaltenschichten zurückzuführen, deren Anzahl logarithmisch mit der Prozessoranzahl wächst. Die effiziente Berechnung dieser parallelen Schichten wird durch eine einfache Adressierung der Daten und durch die Überlappung der Kommunikation mit der Berechnung ermöglicht. Durch die Hardware-Unterstützung der Transputer zur Kommunikation und durch die Anwendung von Programm-'Pipelining' wird die Kommunikation hinter der Berechnung verborgen.

Bei einer Erhöhung der Prozessoranzahl P ist aufgrund der verdeckten Kommunikationen hinter den parallelen Berechnungsschichten und der effizienten Lademöglichkeiten

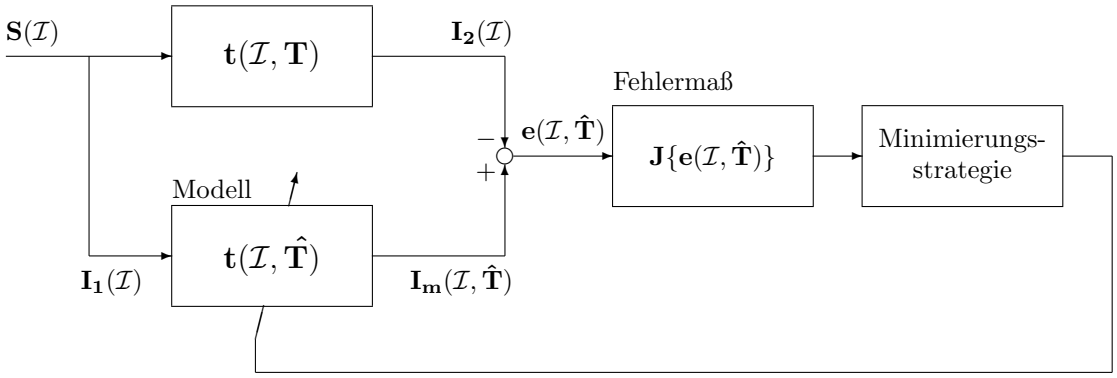

Abbildung 10.9: Blockstruktur der schnellen Bewegungsschätzung (Nach [Die88]).

des HARBURGER SYSTEMS bei genügend großen Bildern zunächst ein Erhalt der hohen Effizienz zu erwarten.

10.4 Schnelle Bewegungsschätzung

Die Implementierung einer schnellen Bewegungsschätzung in Bildfolgen zeigt die Möglichkeiten des HARBURGER SYSTEMS zur Implementierung komplexer Programmpakete auf.

Diehl hat in [Die88] schnelle Algorithmen zur Schätzung von Bewegungsparametern ebener Objekte im dreidimensionalen Raum vorgestellt. Die Bewegung eines ebenen Objektes $\mathbf{S}(\mathcal{I})$, $\mathcal{I} = I_0 \times I_1$ kann durch eine geometrische Transformation $\mathbf{t}(\mathcal{I}, \mathbf{T})$ mit den Parametern $\mathbf{T}$ beschrieben werden. Im einfachen Fall verbirgt sich hinter $\mathbf{t}(\mathcal{I}, \mathbf{T})$ eine translatorische Verschiebung, Abschnitt 10.2 behandelte die Rotation als Transformation, weiterhin können komplexere Bewegungsmodelle wie z.B. die affine Abbildung damit ausgedrückt werden.

Zwei sukzessive Bilder $\mathbf{I}_1(\mathcal{I})$ und $\mathbf{I}_2(\mathcal{I})$ des Objektes $\mathbf{S}(\mathcal{I})$ lassen sich (unter Vernachlässigung von Rauschen und Ausschnittbegrenzung bei der Bildaufnahme) durch $\mathbf{t}(\mathcal{I}, \mathbf{T})$ beschreiben:

$$\mathbf{I}_1(\mathcal{I}) = \mathbf{S}(\mathcal{I}) \tag{10.9}$$

$$\mathbf{I}_2(\mathcal{I}) = \mathbf{S}(\mathbf{t}(\mathcal{I}, \mathbf{T})) \tag{10.10}$$

Nach Festlegung eines Bewegungsmodelles für die Transformation $\mathbf{t}(\mathcal{I})$ kann diese allein durch den zugehörigen Parametervektor $\mathbf{T}$ angegeben werden. Im Falle der translatorischen Verschiebung beinhaltet dieser Vektor z.B. die zwei Offsetwerte, mit der Komplexität eines Modelles nimmt die Dimension des Parametervektors zu. Damit kann eine verkürzte Schreibweise für $\mathbf{I}_2$ verwendet werden:

$$\mathbf{I}_2(\mathcal{I}) = \mathbf{S}(\mathcal{I}, \mathbf{T}) \tag{10.11}$$

Zur Schätzung des unbekannten Parametervektors $\mathbf{T}$ setzt Diehl einen iterativen Algorithmus an, dessen Blockstruktur Abbildung 10.9 zeigt. Mit einem geschätzten Param-

etervektor $\hat{\mathbf{T}}$ wird ein Modellbild $\mathbf{I}_m(\mathcal{I}, \hat{\mathbf{T}})$ — basierend auf dem Ausgangsbild $\mathbf{I}_1(\mathcal{I})$ — errechnet:

$$\mathbf{I}_m(\mathcal{I}, \hat{\mathbf{T}}) = \mathbf{S}(\mathcal{I}, \hat{\mathbf{T}}) \tag{10.12}$$

Die Differenz:

$$\mathbf{e}(\mathcal{I}, \hat{\mathbf{T}}) = \mathbf{I}_2(\mathcal{I}) - \mathbf{I}_m(\mathcal{I}, \hat{\mathbf{T}}) \tag{10.13}$$

zwischen dem transformierten Bild $\mathbf{I}_2(\mathcal{I})$ und dem Modellbild $\mathbf{I}_m(\mathcal{I}, \hat{\mathbf{T}})$ wird durch ein Fehlermaß $\mathbf{J}\{\mathbf{e}(\mathcal{I}, \hat{\mathbf{T}})\}$ gewichtet, welches beim Erreichen des optimalen, geschätzten Parametervektors $\hat{\mathbf{T}}^*$ minimal wird.

Zum Modifizieren des Schätzwertes stellt Diehl verschiedene Minimierungsstrategien vor, insbesondere untersucht er Newton-basierte Algorithmen wie das modifizierte Newton-Raphson- [BM79] und das Quasi-Newton-Verfahren.

Gewähltes Modell:
Bei der Implementierung der Bewegungsschätzung auf dem HARBURGER SYSTEM [LB90] wurde ein Bewegungsmodell gewählt, welches auf der affinen Abbildung basiert:

$$\begin{aligned} \mathbf{t}(\mathbf{i}) &= \mathbf{A} \cdot \mathbf{i} + \mathbf{d} \\ &= \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \cdot \mathbf{i} + \begin{pmatrix} d_0 \\ d_1 \end{pmatrix} \end{aligned} \tag{10.14}$$

Dieses Modell führt zu einem 6-dimensionalen Parametervektor $\mathbf{T}$:

$$\mathbf{T} = \{a_{00}, a_{01}, a_{10}, a_{11}, d_0, d_1\} \tag{10.15}$$

Als Fehlermaß dient der Erwartungswert des Modellfehlerquadrats:

$$\mathbf{E}\{(\mathbf{I}_2(\mathcal{I}) - \mathbf{I}_m(\mathcal{I}, \hat{\mathbf{T}}))^2\} \tag{10.16}$$

Dieser Erwartungswert kann, da nur die beiden Bilder $\mathbf{I}_1(\mathcal{I})$ und $\mathbf{I}_2(\mathcal{I})$ zur Bestimmung von $\hat{\mathbf{T}}$ verfügbar sind, nicht als Scharmittelwert gebildet werden. Seine Berechnung erfolgt daher durch Summation über den betrachteten Bildausschnitt.

Zur Verstellung des geschätzten Parametervektors wird als Minimierungsstrategie ein von Diehl vorgeschlagenes, kombiniertes dreistufiges Verfahren, welches auf den Newton-Raphson- und Quasi-Newton-Algorithmen basiert, verwendet. Dabei erfolgt in der ersten Stufe nur die Schätzung der Translationsparameter d_0 und d_1, in den weiteren Stufen, wenn diese Parameter schon in der Nähe ihres Optimums liegt, werden alle Parameter geschätzt.

Implementierung:
Der größte Rechenaufwand in der Iterationsschleife des Algorithmus (Abbildung 10.9) tritt bei der Bestimmung des Modellbildes $\mathbf{I}_m(\mathcal{I}, \hat{\mathbf{T}})$ sowie der anschließenden Differenz- und Erwartungswertbildung auf, da diese Schritte über die gesamte Bilddimension auszuführen sind. Die übrigen Berechnungen werden nur noch auf Vektoren und Matrizen der Dimension des Parametervektors, im Beispiel Dimension 6, durchgeführt.

Die Parallelisierung auf dem HARBURGER SYSTEM umfaßt nur die Bildoperationen. Dazu wird eine Segmentierung der Indexmenge $\mathcal{I}$ in nahezu gleiche Teile vorgenommen und

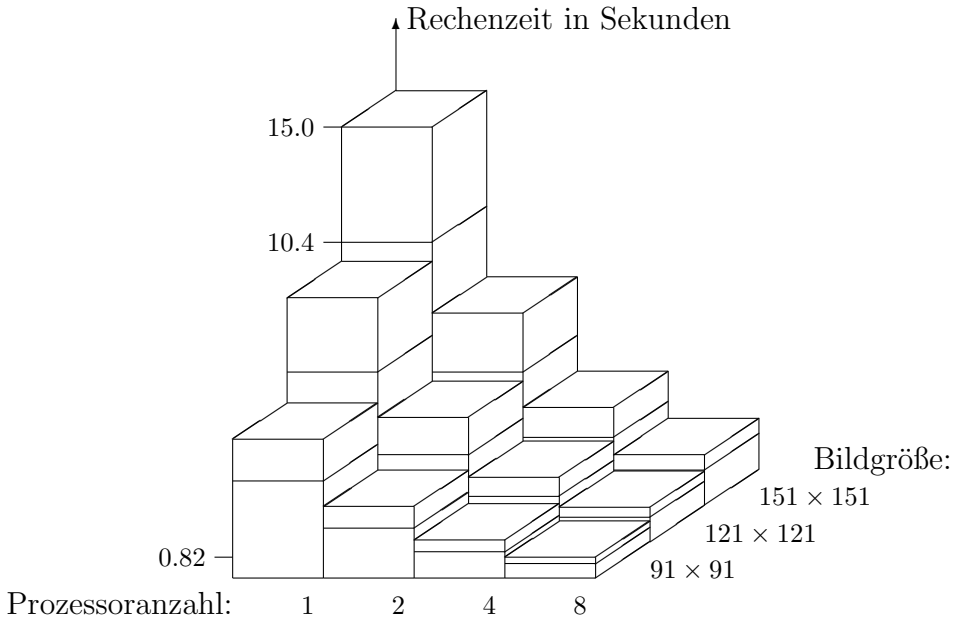

Abbildung 10.10: Rechenzeiten zur schnellen Bewegungsschätzung in Bildfolgen. Oberer Bereich: Initialisierung. Unterer Bereich: 10 Iterationsschritte.

jedem beteiligten Prozessor ein Segment zugeordnet, auf welchem er die oben angeführten Bildoperationen durchführt. Zur Minimierung der algorithmischen Kommunikation erhält jeder Prozessor das vollständige Bild $\mathbf{I}_1(\mathcal{I})$ mittels 'Broadcast' beim Ladevorgang. Damit kann eine lokale Berechnung des Modellbildes $\mathbf{I}_m(\mathcal{I}, \hat{\mathbf{T}})$ und der nachfolgenden Differenz mit $\mathbf{I}_1(\mathcal{I})$ im zugeordneten Segment erfolgen.

Die Berechnung des Erwartungswertes nach Gleichung 10.16 durch Summation über den gesamten Bildausschnitt erfolgt zunächst ebenfalls lokal über den Bildsegmenten. Die ermittelten Teilerwartungswerte der Prozessoren werden dann durch eine globale Transformation mit der Addition als Operator aller $f_i^{(s)}$ zusammengefaßt und auf die Prozessoren verteilt (Abschnitt 3.2.2, Gleichung 3.30). Die Ausführung erfolgt auf einer 'Perfect Shuffle'-Verbindungstopologie in $\log_2(P)$ Schritten.

Die anschließende Minimierungsstrategie zur Modifizierung des geschätzten Parametervektors $\hat{\mathbf{T}}$ wird redundant auf jedem Prozessor ausgeführt.

Mit der vorgestellten Parallelisierung wurden Rechenzeiten zur Initialisierung und zur Ausführung von 10 Iterationsschritten ermittelt (Abbildung 10.10). Die Initialisierung umfaßt das Einlesen des gedrehten Bildes $\mathbf{I}_2(\mathcal{I})$ sowie die Vorabberechnung der Hesse-Matrix im Optimum und des ersten Gradientenvektors vom Gütekriterium [Die88]. In jedem Iterationsschritt wird parallel der Gradientenvektor aktualisiert und redundant auf jedem Prozessor die Aktualisierung der übrigen Größen vorgenommen.

Die Parallelisierung nur bezüglich der Indexmenge schlägt sich in der Effizienz der Implementierung nieder (Abbildung 10.11). Bei der Ausführung des Algorithmus auf einem einzelnen Prozessor ist nur ein geringer Effizienzabfall bei Reduzierung der Bildgröße festzustellen. Bei Erhöhung der Prozessoranzahl sinkt bei konstanter Bildgröße die Ef-

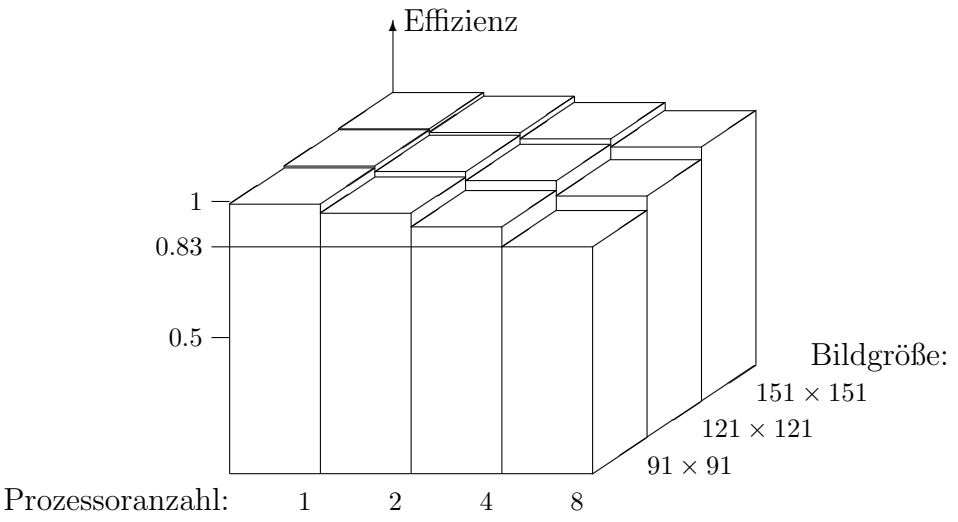

Abbildung 10.11: Effizienzen der iterativen Schätzung von Bewegungsparametern mit affinem Modell und Parallelisierung bezüglich der Bildoperationen.

fizienz durch die redundante Realisierung der Minimierungsstrategie. Insbesondere bei kleinen Bildern wirkt sich der von der Bildgröße unabhängige Minimierungsaufwand stärker aus, da dort sein relativer Anteil am Gesamtaufwand am größten ist.

Die im Vergleich zur Bilddimension niedrige, durch die Dimension des Parametervektors $\mathbf{T}$ vorgegebene Dimension der Berechnungen zur Minimierungsstrategie macht die modulare Parallelisierung dieses Schätzteiles bei großen Prozessoranzahlen aufwendig. Die Eliminierung dieser redundanten Berechnungen wird, da der Hauptanteil des Rechenaufwands schon bei den betrachteten kleinen Bildgrößen durch die Bildoperationen verursacht wird, erst bei einer deutlichen Erhöhung der Prozessoranzahl sinnvoll.

Kapitel 11

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der digitalen Verarbeitung von Bildern in den Verarbeitungsschichten von der Aufnahme bis zum Ergebnis. Diese Schichten der Bildverarbeitung implizieren eine Pipeline-Architektur für ein Rechnersystem. Weiterhin werden die Implementierungsschritte eines vorliegenden Algorithmus bis hin zu seiner Ausführung auf einer Hardware betrachtet. Diese Schritte bedingen bei einer leistungsfähigen und modularen Ausführung des Algorithmus eine Parallelisierung, deren Formulierung und anschließend die Abbildung auf eine Parallel-Architektur. In dieser Arbeit werden beide Architekturen kombiniert und damit eine Rechnerarchitektur hergeleitet, welche die Vorteile der Pipeline zum schnellen Zu- und Abführen von Bilddaten mit der Flexibilität und Leistungsfähigkeit moderner Parallelrechner verbindet.

Der erste Teil der Arbeit (Kapitel 2-6) betrachtet die Implementierungsschritte zur parallelen Ausführung von Algorithmen. In Kapitel 2 werden Anforderungen an Rechnersysteme zur Bildverarbeitung aufgezeigt. In den Kapiteln 3-5 werden die Implementierungsschritte näher untersucht. Der erste Schritt, die Parallelisierung, bedingt die Zerlegung eines Algorithmus. Dazu wird in Kapitel 3 eine Beschreibung durch Graphen eingeführt und mit deren Hilfe die modulare, skalierbare Zerlegung zweier wichtiger Algorithmenklassen der Bildverarbeitung, der lokalen Operationen und der globalen Transformationen, ausgeführt. In Kapitel 4 wird der zweite Schritt, die parallelen Programmformulierung, betrachtet. Dabei wird aufgezeigt, wie eine durch Zerlegung extrahierte Parallelität in der parallelen Programmformulierung berücksichtigt werden kann. Kapitel 5 umfaßt den dritten Schritt, die Ausführung eines parallelen Programms. Dazu werden Operationsprinzipien von Einzelprozessoren, Organisationen von Prozessoren und Speichern zur Bildung von Multiprozessorsystemen und Topologien relevanter Verbindungsnetzwerke im Hinblick auf Bildverarbeitungsaufgaben betrachtet. Kapitel 6 beinhandelt die Klassifikation von Rechnersystemen sowie die Leistungsbewertung der Algorithmenimplementierungen auf Einzel- und Multiprozessorsystemen.

Der zweite Teil der Arbeit (Kapitel 7-10) zeigt das Konzept und die Realisierung eines flexiblen, modularen Bildverarbeitungssystems auf. In Kapitel 7 wird das Systemkonzept des Harburger Parallelen Bildverarbeitungssystems, welches die Verbindung zwischen der Pipeline- und Parallel-Architektur herstellt, und der im Rahmen dieser Arbeit realisierte Prototyp vorgestellt. In Kapitel 8 erfolgt die detaillierte Beschreibung des schnelle Pipelinebusses, der im System zum Transport und Verteilen der Bilddaten vorge-

sehen ist. Dabei werden seine Möglichkeiten zur Unterstützung paralleler Multiprozessorsysteme durch schnelle Lademodi hervorgehoben. Kapitel 9 enthält eine Übersicht der für den Systembetrieb notwendigen parallelen Betriebssoftware. Als Nachweis zur Funktionalität des vorgeschlagenen Systemkonzepts werden schließlich in Kapitel 10 Implementierungen beispielhafter Algorithmen aufgezeigt und die erzielten Leistungswerte diskutiert.

Literaturverzeichnis

[Agr83] Dharma P. Agrawal. Graph Theoretical Analysis and Design of Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-32(7):637–648, July 1983.

[AJP86] Dharma P. Agrawal, Virendra K. Janakiram und Girish C. Pathak. Evaluating the Performance of Multicomputer Configurations. *Computer*, Seite 23–37, May 1986.

[Ana88] Analog Devices Inc. *Data Conversion Products Databook*, April 1988.

[ASBH90] Klaus Arbter, Wesley E. Snyder, Hans Burkhardt und Gerd Hirzinger. Application of Affine-Invariant Fourier Descriptors to Recognition of 3-D Objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):640–647, July 1990.

[BA83] Laxmi N. Bhuyan und Dharma P. Agrawal. Design and Performance of Generalized Interconnection Networks. *IEEE Transactions on Computers*, C-32(12):1081–1090, December 1983.

[BB85] Hans Burkhardt und Lineu C. Barbosa. Contributions to the Application of the Viterbi Algorithm. *IEEE Transactions on Information Theory*, IT-31(5):626–634, September 1985.

[BLN90] Hans Burkhardt, Bernhard Lang und Michael Nölle. Aspects of Parallel Image Processing Algorithms and Structures. In H. Burkhardt, Y. Neuvo und J.C. Simon, Editoren, *From Pixels to Features II, Parallelism in Image Processing, ESPRIT BRA 3035 Workshop, Bonas*, September 1990.

[BM79] H. Burkhardt und H. Moll. A modified Newton-Raphson-search for the model adaptiv identification of delays. In R. Isermann, Editor, *Identification and System Parameter Estimation*, Seite 1279–1286. Pergamon Press, 1979.

[BMF88] BMFT Cooperative Project. *Family of Fast Image Processing Systems*. Bericht zur Präsentation, Brüssel, 18. November 1988.

[Boo90] Booktree Corporation, San Diego. *Product Selection Guide Autumn 1990*, 1990.

[BR84] M. Banahan und A. Rutter. *UNIX*. Hanser Verlag, 1984.

[Bro89] Luigi Brochard. Efficiency of some parallel numerical algorithms on distributed systems. *Parallel Computing*, (12):21–44, 1989.

[BT89] Dimitri P. Bertsekas und John N. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods*. Prentice Hall, 1989.

[Bur79] Hans Burkhardt. *Transformationen zur lageinvarianten Merkmalgewinnung*, Nummer 7 von *VDI-Fortschritt-Bericht, Reihe 10: Angewandte Informatik*. VDI Verlag, Düsseldorf, 1979.

[Bur89] Hans Burkhardt. Methoden der Mustererkennung. Vorlesung im Hauptstudium Elektrotechnik an der TU Hamburg-Harburg, 1989. Sommersemester.

[CCJ90] Gen-Huey Chen, Maw-Sheng Chern und Jin Hwang Jang. Pipeline Architectures for Dynamic Programming Algorithms. *Parallel Computing*, 13:111–117, 1990.

[Das90] Subrata Dasgupta. A Hierarchical Taxonomic System for Computer Architectures. *IEEE Computer*, 23(3):64–74, March 1990.

[Den79] Jack B. Dennis. The Varieties of Data Flow Computers. In *Proceedings of the First International Conference on Distributed Computing Systems*, Seite 430–439. IEEE, 1979.

[Die88] Norbert Diehl. *Methoden zur allgemeinen Bewegungsschätzung in Bildfolgen*, Nummer 92 von *VDI-Fortschritt-Bericht, Reihe 10: Informatik, Kommunikationstechnik*. VDI-Verlag, 1988.

[Din89] Anne Dinning. A Survey of Synchronization Methods for Parallel Computers. *IEEE Computer*, 22(7):66–77, July 1989.

[DJ87] P. W. Dowd und K. Jabbour. Performance Evaluation of Spanning Multiaccess Channel Hypercube Interconnection Network. *IEE Proceedings*, 134(6):295–302, November 1987.

[FHKZ88] Bernd Franke, Ralf Harneit, Axel Kern und Christoph Zeidler. The Pipeline Bus: An Interconnection Network for Multiprocessor Systems. *Parallel Computing*, 7:403–412, 1988.

[FK89] Horace P. Flatt und Ken Kennedy. Performance of Parallel Processors. *Parallel Computing*, 12:1–20, 1989.

[Fly66] Michael J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[Fu82] K. S. Fu. Introduction to Pattern Processing. In K. S. Fu und Tadao Ichikawa, Editoren, *Special Computer Architectures for Pattern Processing*, Kapitel 3, Seite 35–63. CRC Press, Boca Raton, Florida, 1982.

[GE89] Ran Ginosar und David Egozi. Topological Comparison of Perfect Shuffle and Hypercube. *International Journal of Parallel Programming*, 18(1):37–68, 1989.

[GKB87] John Gurd, Chris Kirkham und Wim Böhm. The Manchester Dataflow Computing System. In J. J. Dongarra, Editor, *Experimental Parallel Computing Architectures*, Seite 177–219. North Holland, 1987.

[GKW85] J. R. Gurd, C. C. Kirkham und I. Watson. The Manchester Prototype Dataflow Computer. *Communications of th ACM*, 28(1):34–52, January 1985.

[Gun90] Anton Gunzinger. *Synchroner Datenflußrechner zur Echtzeitbildverarbeitung.* Verlag der Fachvereine, Zürich, 1990.

[Gus88] John L. Gustafson. Reevaluating Amdahl's Law. *Communication of the ACM*, 31(5):532–533, May 1988.

[Hän77] Wolfgang Händler. The Impact of Classification Schemes on Computer Architecture. In *Proceedings of the 1977 International Conference on Parallel Processing*, Seite 7–15. IEEE, 1977.

[HH89] F. Hutner und R. Holzner. Architektur, Programmierung und Leistungsbewertung des MIT-Datenflußrechners. *Informatik Spektrum*, (12):147–157, 1989.

[HHR+90] R. W. Hartenstein, A. G. Hirschbiel, M. Riedmüller, K. Schmidt und M. Weber. Automatic Synthesis of Cheap Hardware Accelerators for Image Processing and Digital Signal Processing. In *12. DAGM-Symposium Mustererkennung, Oberkochen*, Seite 404–417, Berlin Heidelberg, 1990. Springer Verlag.

[Hil85] W. Daniel Hillis. *The Connection Machine.* Artificial Intelligence. The MIT Press, 1985.

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

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HsF83] Kai Hwang und King sun Fu. Integrated Computer Architecture for Image Processing and Database Management. *Computer*, Seite 51–60, January 1983.

[INM87] INMOS Limited, Bristol, UK. *The transputer instruction set – a compiler writers' guide*, February 1987.

[INM88a] INMOS Limited, Bristol, UK. *IMS A110 Image and Signal Processing Subsystem*, June 1988.

[INM88b] INMOS Limited, Bristol, UK. *The Transputer Data Book*, First Edition, November 1988.

[Int87] Intel Corporation. *Programmable Logic Handbook*, 1987.

[Jes87] Chris Jesshope, Editor. *Parallel Processing, State of the Art Report 15:4.* Pergamon Infotech Limited, 1987.

[JM88] Roy M. Jenevein und Thomas Mookken. Traffic Analysis of Rectangular SW-Banyan Networks. In *1988 IEEE Symposium on Computer Architecture*, Seite 333–342. IEEE, 1988.

[Lan89] Bernhard Lang. Ein paralleles Transputersystem zur digitalen Bildverarbeitung mit schneller Pipelinekopplung. In *11. DAGM-Symposium Mustererkennung, Hamburg*, Seite 372–379, Berlin Heidelberg, 1989. Springer Verlag.

[Lan90a] Bernhard Lang. Leistungsmessung am Parallelen Transputersystem der TUHH. Interner Bericht, Technische Informatik I, TU Hamburg-Harburg, November 1990.

[Lan90b] Bernhard Lang. Paralleler Bildverarbeitungsrechner auf Transputerbasis, Programmier Anleitung. Interner Bericht, Technische Informatik I, TU Hamburg-Harburg, Januar 1990.

[Law75] Duncan H. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, December 1975.

[LB90] Bernhard Lang und Hans Burkhardt. A parallel transputer system with fast pipeline interconnection. In *Applications of Digital Image Processing XIII, Proceedings: San Diego, 10-13 July 1990*, Bellingham, Washington, 1990. SPIE.

[LE89] C.-E. Liedtke und M. Ender. *Wissensbasierte Bildverarbeitung*, Nummer 19 von *Nachrichtentechnik*. Springer Verlag, 1989.

[Lel90] Wm Leler. Linda Meets Unix. *IEEE Computer*, 23(2):43–54, February 1990.

[LSI90] LSI Logic Corporation. *Digital Signal Processing (DSP) Databook*, June 1990.

[Ltd89] Perhelion Software Ltd., Editor. *The Helios Operating System*. Prentice Hall, 1989.

[Lut90] Elmar Lutter. Entwicklung und Implementierung einer Markierungseinheit für ein paralleles Transputer-Bildverarbeitungssystem. Diplomarbeit, Technische Informatik I, TU Hamburg-Harburg, 1990.

[MG89] C. McCreary und H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32(9):1073–1078, September 1989.

[Mil89] Stefan Milde. Aufbau eines 'Link-Switch'-Modul für ein paralleles Transputer-Bildverarbeitungs-System. Diplomarbeit, Technische Informatik I, TU Hamburg-Harburg, 1989.

[MS87] David May und R. Sheperd. The INMOS Transputer. In Chris Jesshope, Editor, *Parallel Processing, State of the Art Report 15:4*, Kapitel 6, Seite 71–92. Pergamon Infotech Limited, 1987.

[MST88] G. Thiesing M. S. Tatari, W. Melchert. Architektur und Programmierkonzept für "Familie schneller Bildverarbeitungsrechner". In *Mustererkennung 1988, 10. DAGM-Symposium Zürich*, Seite 68–75. Springer Verlag, September 1988.

[NEC84] NEC Electronics. *Product Description Image Pipelined Processor μPD7281D*, Edition v2.0, November 1984.

[Nie83] Heinrich Niemann. *Klassifikation von Mustern*. Springer Verlag, 1983.

[Par86] D. Parkinson. Parallel efficiency can be greater than unity. *Parallel Computing*, (3):261–262, 1986.

[Par90] Parsec Developments, Leiden. *Par.C System, User's Manual and Library Reference, V 1.3*, 5th Edition, June 1990.

[Pat85] David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[Pea77] Marshall C. Pease. The Indirect Binary N-Cube Microprocessor Array. *IEEE Transactions on Computers*, C-26(5):458–473, May 1977.

[PL83] Krishnan Padmanabhan und Duncan H. Lawrie. A Class of Redundant Path Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-32(12):1099–1108, December 1983.

[PM87] Dick Pountain und David May. *A tutorial introduction to* Occam *programming, Including language definition*. INMOS Limited, Bristol, UK, March 1987.

[Pou90] Dick Pountain. Configuring Parallel Programs, Part 2. *BYTE*, Seite 327–334, January 1990.

[Pre83] Kendal Preston, Jr. Cellular Logic Computers for Pattern Recognition. *Computer*, Seite 36–47, January 1983.

[Ree84] Anthony P. Reeves. SURVEY Parallel Computer Architectures for Image Processing. *Computer Vision, Graphics and Image Processing*, 25:68–88, 1984.

[Ree87] Daniel A. Reed. *Multicomputer Networks*. Scientific Computation. The MIT Press, 1987.

[RG75] Lawrence R. Rabiner und Bernard Gold. *Theory and Application of Digital Signal Processing*. Prentice Hall, Englewood Cliffs, 1975.

[RLH88] U. Raabe, M. Lobjinski und M. Horn. Verbindungsstrukturen für Multiprozessoren. *Informatik Spektrum*, (11):195–206, 1988.

[Sch72] Helmut Schönfelder. *Fernsehtechnik*. Justus von Liebig Verlag, Darmstadt, 1972.

[Ser82] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, 1982.

[Sez87] André Seznec. A New Interconnection Network for SIMD Computers: The Sigma Network. *IEEE Transactions on Computers*, C-36(7):794–801, July 1987.

[Sie80] Howard Jay Siegel. The Theory Underlying the Partitioning of Permutation Networks. *IEEE Transactions on Computers*, C-29(9):791–801, September 1980.

[SNSS83] E. Strasburger, F. Noll, H. Schenk und A. F. W. Schimper. *Lehrbuch der Botanik*. Gustav Fischer Verlag, Stuttgart, New York, 32. Ausgabe, 1983.

[Soo83] A. K. Sood. Design of Multistage Interconnection Networks. *IEE Proceedings*, 130(4):109–115, July 1983.

[Sri86] Vason P. Srini. An Architectural Comparison of Dataflow Systems. *IEEE Computer*, Seite 68–88, March 1986.

[SS88] Youcef Saad und Martin H. Schultz. Topological Properties of Hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, July 1988.

[SS89a] Youcef Saad und Martin H. Schultz. Data Communication in Parallel Architectures. *Parallel Computing*, (11):131–150, 1989.

[SS89b] J. Schabernack und A. Schütte. Occam2 und Ada – Eine Gegenüberstellung. *Informatik Spektrum*, 12:3–18, 1989.

[Ste83a] David Steinberg. Invariant Properties of the Shuffle Exchange and a Simplified Cost-Effective Version of the Omega Network. *IEEE Transactions on Computers*, C-32(5):444–450, May 1983.

[Ste83b] Stanley R. Sternberg. Biomedical Image Processing. *Computer*, Seite 22–34, January 1983.

[Sto71] Harold S. Stone. Parallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, February 1971.

[Sto83] Quentin F. Stout. Mesh-Connected Computers with Broadcasting. *IEEE Transactions on Computers*, C-32(9):826–830, September 1983.

[Sto90] Harold S. Stone. *High-Performance Computer Architecture*. Electrical and Computer Engineering. Addison Wesley, Second Edition, 1990.

[Tos89] Toshiba Corporation. *MOS Memory Products Data Book*, 1989.

[Tre84] Philip C. Treleaven. Decentralized computer architecture. In J. Tiberghien, Editor, *New Computer Architectures*, Seite 1–58. Academic Press, 1984.

[UW87] Eli Upfal und Avi Wigderson. How to Share Memory in a Distributed System. *Journal of the Association for Computing Machinery*, 34(1):116–127, January 1987.

[Veg84] Stephen R. Vegdahl. A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Transactions on Computers*, C-33(12):1050–1071, December 1984.

[Vit88] Paul M. B. Vitányi. Locality, Communication and Interconnect Length in Multicomputers. *SIAM Journal of Computing*, 17(4):659–672, August 1988.

[Wah84] F. Wahl. *Digitale Bildsignalverarbeitung*. Springer Verlag, 1984.

[Wak68] Abraham Waksman. A Permutation Network. *Journal of the ACM*, 15(1):159–163, January 1968.

[WD84] Erling H. Wold und Alvin M. Despain. Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementations. *IEEE Transactions on Computers*, C-33(5):414–426, May 1984.

[WF80a] Chuan-Lin Wu und Tse-Yun Feng. On a Class of Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-29(8):694–702, August 1980.

[WF80b] Chuan-Lin Wu und Tse-Yun Feng. The Reverse-Exchange Interconnection Network. *IEEE Transactions on Computers*, C-29(9):801–811, September 1980.

[Wir85] Niklaus Wirth. *Programmieren in Modula-2*. Springer Verlag, 1985.

[Xil88] Xilinx Inc. *The Programmable Gate Array Data Book*, 1988.

Anhang A

Verwendete Symbole und Operatoren

A.1 Symbole

A	Symbol für einen Algorithmus
$a_{i,j}(A)$	Kommunikationsbedarf eines Algorithmus A zwischen Prozessoren P_i und P_j
$(Addr)$	Adreßteil eines Tokens T
B	Bandbreite
$(\ldots)_B$	Zahlenrepräsentation zur Basis B
$C_k^{(s)}$	Kommunikationsoverhead zur Definition der Granularität
$(Ctrl)$	Kontrollteil eines Tokens T
$d_{i,j}$	Topologiemaß: Distanz zwischen den Ecken p_i und p_j
d_{Max}	Topologiemaß: Maximale Distanz eines Netzwerks
$\bar{d}_i$	Topologiemaß: Mittlere Distanz aller Ecken zu p_i
$(Data)$	Datum eines Tokens T
$\mathcal{E}$	Eckenmenge des Graphen $\mathcal{G}$
$\mathcal{E}^{(s)}$	Berechnungsschicht einer Zerlegung $\mathcal{Z}(\mathcal{G})$
e_i	Element der Eckenmenge $e_i \in \mathcal{E}$
$E(N,P)$	Effizienz bei der parallelen Implementierung von Algorithmen
$\mathbf{E}\{\ldots\}$	Erwartungswert
$\mathbf{e}(\mathcal{I},\mathbf{T})$	Schätzfehler bei der schnellen Bewegungsschätzung
$\mathcal{F}$	Fenstermenge zur Umgebungsdefinition
$\mathbf{f}$	Element der Fentermenge $\mathbf{f} \in \mathcal{F}$
f	Frequenz (Kap. 2)
f	Symbol für eine Funktion
$f_i^{(s)}$	Funktionsecke der Graphen globaler Transformationen
$G_k^{(s)}$	Granularitätsmaß
$\mathcal{G}$	Algorithmengraph
$\mathcal{G}_k$	Subgraph von $\mathcal{G}$
$\mathcal{G}_G$, $\mathcal{G}_{G2}$, $\mathcal{G}_{G3}$	Graphen globaler Transformationen
H	Horizontalauflösung eines Video-Bildes in Pixel

$\mathcal{I}$	Indexmenge
$\mathbf{i}$	Indexwert $\mathbf{i} \in \mathcal{I}$
$\mathbf{I}_1(\mathcal{I})$, $\mathbf{I}_2(\mathcal{I})$	Bilder über der Indexmenge $\mathcal{I}$
i, j	Indizes
$\mathbf{J}\{\mathbf{e}\}$	Fehlermaß zur Gewichtung des Schätzfehlers $\mathbf{e}$
$\mathcal{K}$	Kantenmenge des Graphen $\mathcal{G}$
$\mathcal{K}^G$	Globale Kantenmenge einer Zerlegung $\mathcal{Z}(\mathcal{G})$
$\mathcal{K}^{G,(s)}$	Kommunikationsschicht einer Zerlegung $\mathcal{Z}(\mathcal{G})$
$k_{i,j}$	Element der Kantenmenge $k_{ij} \in \mathcal{K}$
$\mathcal{N}$	Kantenmenge einer Topologie $\mathcal{T}$ (Verbindungsnetzwerk)
$\mathbb{N}$	Menge der natürlichen Zahlen $\{0, 1, 2, 3, \ldots\}$
$\mathbb{N}_k$	Endliche Menge ganzer Zahlen: $\{0, 1, \ldots, k-2, k-1\}$
N	Problemgröße eines Algorithmus, Dimension von Datenvektoren, etc.
n	Logarithmus von N
$O(A)$	Rechenzeitordnung zur Ausführung eines Algorithmus A
$\mathcal{P}$	Eckenmenge einer Topologie $\mathcal{T}$ (Prozessormenge)
p_i	Ecke eines Topologiegraphen $p_i \in \mathcal{P}$
P	Prozessoranzahl
P_i	Bezeichnung für einen Prozessor, $i \in \mathbb{N}_P$
p	Logarithmus der Prozessoranzahl
$\mathbf{P}(\mathcal{G})$	Pfad des Graphen $\mathcal{G}$
$\mathbf{P}^+(\mathcal{G})$	Positiver Pfad des Graphen $\mathcal{G}$
$\mathbf{p}$	Fester Indexwert $\mathbf{p} \in \mathcal{I}$
R	Datenrate
m	Index der Berechnungsschicht bei der Zerlegung von Datenflußgraphen
r_i	Anzahl der Anschlußpunkte eines Prozessors P_i
S	Seitenverhältnis eines Video-Bildes
$\mathtt{S}$	Schalter von schaltbaren Verbindungsnetzwerken
$S(N, P)$	Speedup bei der parallelen Implementierung von Algorithmen
$\mathbf{S}(\mathcal{I})$	Objekt über dem Indexraum $\mathcal{I}$
$\mathbf{S}(\mathcal{I}, \mathbf{T})$	Bewegtes Objekt über dem Indexraum $\mathcal{I}$
s	Schichtindex bei der Zerlegung von Datenflußgraphen
$\mathcal{T}$	Topologie zur Beschreibung eines Multiprozessorsystems
$T(\mathbf{x})$	Globale Transformation
$\mathbf{t}(\mathcal{I}, \mathbf{T})$	Koordinatentransformation auf der Indexmenge $\mathcal{I}$ mit Parametervektor $\mathbf{T}$
$T(\mathcal{G})$	Tiefe eines Graphen $\mathcal{G}$
$\mathbf{T}$	Parametervektor zur Koordinatentransformation
$\hat{\mathbf{T}}$	Geschätzter Parametervektor
$\hat{\mathbf{T}}^*$	Geschätzter Parametervektor im Optimum
T, T_0	Taktperioden
T	Token $T = ((Ctrl), (Addr), (Data))$
t	Laufzeit, Ausführungszeit
t_K	Kommunikationszeit
t_{NL}	Nichtlineare Ausführungszeit
t_P	Paralleler Anteil der Ausführungszeit

t_R	Rechenzeit
t_S	Serieller Anteil der Ausführungszeit
$t_{ü}$	Übertragungszeit
t_0	Kommunikations- und Synchronisationszeit (Kap. 6)
t_0	Initialisierungszeit (Kap. 10)
$\mathcal{U}_{\mathbf{i}}(\mathbf{I}_1(\mathcal{I}), \mathcal{F})$	Umgebungsmenge zum Index $\mathbf{i}$ aus Bild $\mathbf{I}_1$ mit Fenstermenge $\mathcal{F}$
V	Vertikalauflösung eines Video-Bildes in Zeilen
$V(e_i)$	Vorgänger einer Ecke e_i des Graphen $\mathcal{G}$
$\mathcal{W}$	Menge von Gewichten zur lokalen Bildfilterung
W	Bildwechselfrequenz
$\mathbf{x}$, $\mathbf{y}$	Vektoren
$\tilde{\mathbf{x}}$	Transformierter Vektor
$\mathcal{Z}(\mathcal{G})$	Zerlegung eines Graphen $\mathcal{G}$
ΔT	Verarbeitungs Zyklus bei der sukzessiven Algorithmen-Ausführung
$\Phi(d, i)$	Anzahl der Ecken mit Distanz d zur Ecke p_i einer Topologie $\mathcal{T}$
$\pi(\mathbf{x})$	Permutation der Elemente des Vektors $\mathbf{x}$
Π	Permutationsmenge
Π_P	Vollständige Permutationsmenge

A.2 Operatoren

$\lfloor a \rfloor$	Größte ganze Zahl $\leq a \in I\!R$ (Gaußklammern)
$i \bmod j$	Modulo-Division zweier ganzer Zahlen
$H(i, j)$	Hamming Distanz zweier binärcodierter ganzer Zahlen
$i \wedge j$	Bitweise UND-Verknüpfung bei binärcodierten ganzen Zahlen, Logische UND-Verknüpfung bei boolschen Operanden
$i \vee j$	Bitweise ODER-Verknüpfung bei binärcodierten ganzen Zahlen, Logische ODER-Verknüpfung bei boolschen Operanden
$i \oplus j$	Bitweise EXKLUSIV-ODER-Verknüpfung zweier binärcodierter ganzer Zahlen
$\mathbf{I}_1(\mathcal{I}) \ominus \mathcal{F}$	Morphologischer Operator der Erosion
$\mathbf{I}_1(\mathcal{I}) * \mathcal{W}$	Faltungsoperator
$f_2 \circ f_1$	Komposition zweier Funktionen
$\exists$	"Es existiert"
$\forall$	"Für alle"
$\Rightarrow$	"Daraus folgt"
$\{\ldots\}$	Mengenklammern
$\mathcal{E}_1 \cup \mathcal{E}_2$	Vereinigung zweier Mengen
$\mathcal{E}_1 \cap \mathcal{E}_2$	Durchschnitt zweier Mengen
$\mathcal{E}_1 \backslash \mathcal{E}_2$	Mengendifferenz
$\mathcal{E}_1 \times \mathcal{E}_2$	Kreuzprodukt zweier Mengen
$\lvert\mathcal{E}\rvert$	Mächtigkeit einer Menge
$\lvert\mathbf{P}(\mathcal{G})\rvert$	Länge eines Pfades

Tabellarischer Lebenslauf

Name:	Bernhard Lang
Geburtsdatum, Ort:	15. 9. 1958, Groß-Rechtenbach
Familienstand:	ledig
Staatsangehörigkeit:	deutsch

Ausbildung:

1964 - 1968:	Grundschule in Groß-Rechtenbach.
1968 - 1977:	Gymnasium in Wetzlar, 1977 Abitur mit naturwissenschaftlichem Schwerpunkt.
1977 - 1983:	Studium der Elektrotechnik an der Technischen Hochschule in Darmstadt, 1979 Abschluß des Vordiploms, ab 1979 Fachrichtung Regelungstechnik, 1983 Abschluß des Studiums mit Diplom.

Beruflicher Werdegang:

1983 - 1986:	Ernst Leitz Wetzlar GmbH, Wetzlar: Position als Entwicklungsingenieur, Entwurf und Entwicklung von Hard- und Software zur Realisierung von Algorithmen im Bereich der Hochleistungs-Bildanalyse.
1986 - heute:	Technische Universität Hamburg-Harburg, Technische Informatik: Position als wissenschaftlicher Mitarbeiter. Forschung im Bereich Bild- und Parallelverarbeitung, Entwurf eines Parallelrechners für Bildverarbeitungsalgorithmen höherer Ebene.