Parallele Matrizenmultiplikation

Monographie

Volker Zerbe

Technische Universität Ilmenau

Vorwort

Die Matrizenmultiplikation ist die Basisoperation der linearen Algebra und somit auch einer Vielzahl wichtiger Anwendungen, wie Mission Level Design Automation, Computergraphik und Robotik. Zudem stellt die Matrizenmultiplikation meist auch den größten Anteil an den genannten Anwendungen. Viele Probleme können auf die Matrizenmultiplikation reduziert und optimal gelöst werden, wenn die Matrizenmultiplikation selbst optimal gelöst wird. Effiziente Algorithmen für die Matrizenmultiplikation sind deshalb von hohem praktischen Interesse. Eine Darstellung von Grundprinzipien, Entwurfsmöglichkeiten und Realisierungen insbesondere von parallelen Algorithmen, die das Leistungspotential innovativer Rechnerarchitekturen erschließen, ist notwendig und wichtig. Es ist nicht das vordergründige Ziel, eine möglichst vollständige Sammlung der bisher entwickelten parallelen Algorithmen für die Matrizenmultiplikation zu liefern, vielmehr soll eine umfassende Gegenüberstellung von repräsentativen sequentiellen Algorithmen und originären parallelen Algorithmen aufgezeigt und ihr Bezug zur Rechnerarchitektur verdeutlicht werden. Somit stellt die Arbeit eine sinnvolle Erweiterung zu Pan [70] dar. Besonderer Wert ist auf die Darstellung von Algorithmen gelegt, die nicht nur von theoretischem Interesse sind, sondern auch in der Praxis eine Bedeutung haben. Leider sind viele der asymptotisch "besten" Algorithmen schwer zu implementieren. Der Mehraufwand ist derart groß, so daß sie nur von begrenztem praktischen Nutzen sind.

Die vorliegende Arbeit ist als Arrondierung dessen zu verstehen, was der Autor in den letzten 8 Jahren an Forschungsergebnissen auf dem Gebiet erzielt hat. Diese eigenen Ergebnisse wurden mit Grundlagen und Ergebnissen anderer angereichert, so daß sich eine geschlossene Darstellung ergibt. Eine Übersicht über die Struktur der Arbeit zeigt Abb. 1.

Im ersten Kapitel werden die wesentlichen Grundlagen betrachtet. Eine zentrale Stelle nehmen dabei die Punkte parallele Architekturen und parallele Programmierung ein. Das Wechselspiel der Kategorien Architektur, Programmiersprache und Algorithmus ist für die Entwicklung der Informatik und seiner Anwendungsgebiete von entscheidender Bedeutung. Neue Rechnerarchitekturen erfordern neue Algorithmen um das Leistungspotential voll auszuschöpfen. Notwendig ist zudem eine Programmiersprache, die eine effiziente Implementierung möglich macht. Es folgt eine kurze Darstellung von Konzepten für den Entwurf sowohl

sequentieller als auch paralleler Algorithmen. Die Vorstellung eines Begriffsinstrumentariums für die Komplexitätsanalyse und Bewertung der Leistungsfähigkeit paralleler Algorithmen schließt das Kapitel ab.

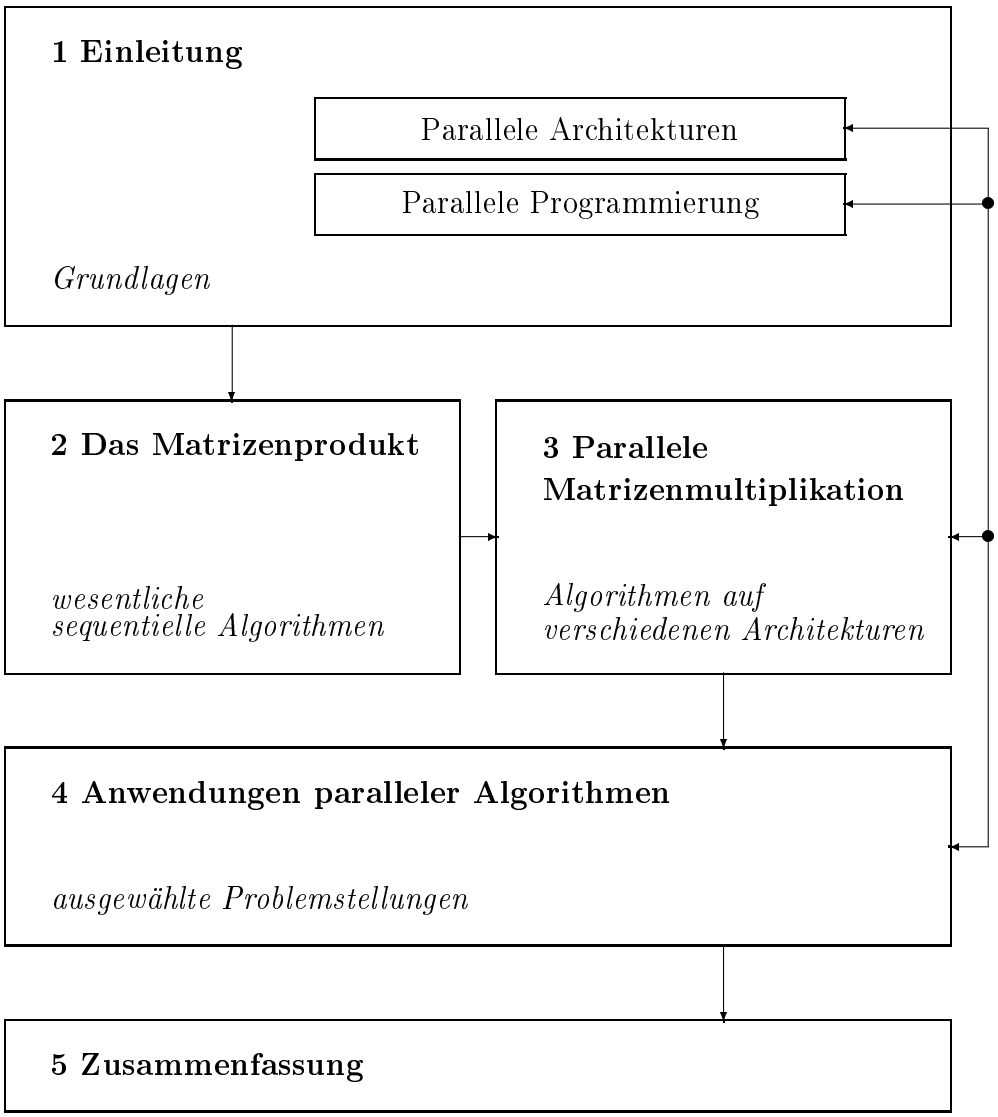


Abbildung 1: Struktur der Arbeit

Verschiedene wesentliche sequentielle Algorithmen für die Matrizenmultiplikation werden in Kapitel 2 betrachtet. Im Vordergrund stehen die Algorithmen nach Strassen und Winograd. Sie haben die größte Bedeutung unter den sequentiellen Algorithmen neben dem "naiven" Multiplikationsalgorithmus aus implementierungstechnischer Sicht erlangt.

Kapitel 3 ist der parallelen Matrizenmultiplikation gewidmet. Der Schwerpunkt liegt nun nicht mehr nur auf der Matrizenmultiplikation, sondern auf der gemeinsamen Betrachtung der Kategorien Architektur und Algorithmus.

In Kapitel 4 werden ausgewählte Problemstellungen aus Anwendungen paralleler Algorithmen behandelt. Die Inhalte der einzelnen Abschnitte resultieren bis auf wenige Ausnahmen aus der Forschungsarbeit der Gruppe des Autors.

Die Zusammenfassung im Kapitel 5 rundet die Arbeit ab. Im Literaturverzeichnis sind nur jene Originalarbeiten aufgeführt, aus denen Begriffsbildungen, Anregungen oder Ergebnisse für die eigene Arbeit entnommen wurden.

An dieser Stelle möchte ich mich besonders herzlich bei meiner Familie bedanken, die mich während der Entstehung der Arbeit stets ermutigte und unterstützte. Allen Kolleginnen und Kollegen des Institutes Theoretische und Technische Informatik danke ich für die angenehme Arbeitsatmosphäre.

Volker Zerbe

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Einführung 1
1.2 Parallele Architekturen 4
1.3 Parallele Programmierung 10
1.4 Entwurfskonzepte 13
1.5 Kenngrößen paralleler Algorithmen 19
1.6 Darstellung der Algorithmen 24

2 Das Matrizenprodukt **27**

2.1 Eine Einführung 27
2.2 Winograd-Algorithmus 29
2.3 Strassen-Algorithmus 31
2.4 Bilineare Algorithmen 35
2.5 Kettenmultiplikation von Matrizen 40
2.6 Geschichte 44

3 Parallele Matrizenmultiplikation **47**

3.1 Lineares Feld 47
3.2 Gitter 49
3.3 Torus 53
3.4 Netz von Bäumen 56
3.5 Hypercube 59
3.6 Systolisches Feld 63

4 Anwendungen paralleler Algorithmen **71**

4.1 Matrizeninversion und biomagnetische Felder 71
4.2 Neuronale Netze auf einer Prozessorfarm 80
4.3 Matrizenmultiplikation auf einem Transputernetz 86

4.4 Fourier und Multiplikation von Polynomen 90
4.5 Numerische Resultate 94
4.6 Standard-Bibliotheken 98

5 Zusammenfassung **105**
5.1 Speedup und Scaleup 106
5.2 Algorithmus und Architektur 107

Literaturverzeichnis **109**

Abbildungsverzeichnis

1 Struktur der Arbeit ii

1.1 Ausführungszeit in Abhängigkeit von der Anzahl der Prozessoren für eine konstante Problemgröße 3
1.2 Klassifikation nach Flynn 5
1.3 MIMD-Klassifikation, a) verteilter Speicher, b) gemeinsamer Speicher 5
1.4 a) Ring, b) Hypercube, c) Ring eingebettet im Hypercube 6
1.5 a) Gitter, b) Torus 7
1.6 Butterfly-Graph der Dimension $d = 3$ 7
1.7 a) Binärbäum, b) Systolische Felder 8
1.8 a) Elementare Schalterzustände, b) Banyan-Netz 9
1.9 Beispiel fork/join und Kontrollfluß 12
1.10 Beispiel cobegin/coend und Kontrollfluß 13
1.11 Beispiel pardo/parend und Kontrollfluß 13
1.12 Broadcasting in einem 2D bzw. 3D Gitter 18
1.13 Informationsfluß 20
1.14 Parallelitätsprofil für einen Teile und Herrsche Algorithmus 23

2.1 Zeitkomplexität in Abhängigkeit von der Matrixgröße für den Strassen und "naiven" Algorithmus 35
2.2 Vergleich der Zeitkomplexitäten in Abhänigkeit von der Matrixgröße für die Algorithmen $a(n_k)$, $b(n_k)$ und "naiv" 40
2.3 $m[i, j]$ bzw. $s[i, j]$ Tabellen der Kettenmultiplikation von Matrizen für $n = 6$ 43
2.4 Entwicklung des Exponenten 44

3.1 "Schulmethode" zur Lösung der Multiplikation zweier Integerwerte, $7 \cdot 3 = 21$. 47

3.2 Datenfluß am Beispiel $n = 3$ in einem linearen Feld doppelter Länge, [65] 49
3.3 Multiplikation von $a = 3 = [011]$ und $b = 7 = [111]$, [65] 50
3.4 Berechnung des Matrix-Vektor-Produktes $\vec{y} = \mathcal{A}\vec{x}$ auf einem linearen Feld für $n = 4$ 51
3.5 Berechnung des Matrix-Matrix-Produktes $\mathcal{AB}$ auf einem Gitter für $n = 4$ 51
3.6 Schnappschuß des Datenflusses zum 5. Schritt 52
3.7 Matrix-Vektor-Multiplikation auf einem 4-Prozessor-Ring 53
3.8 p^2 Torus-Architektur 54
3.9 Ein $(2 \times 2 \times 2)$ Netz von Bäumen. Die Kreise stellen die 8 Original Knoten dar. Die verbleibenden 12 Knoten sind die Wurzelknoten der Bäume in den 3 Ausbreitungen. 56
3.10 Matrix-Vektor-Produkt auf einem Netz von Bäumen. Die Baumstruktur ist aus Übersichtlichkeit nur teilweise angedeutet. 57
3.11 Multiplikation zweier Matrizen vom Typ $(2, 2)$ auf einem $(2 \times 2 \times 2)$ Netz von Bäumen. 58
3.12 Broadcasting der Matrizen $\mathcal{A}$, $\mathcal{B}$ 61
3.13 Spaltenverteilung 61
3.14 Zeilenverteilung 62
3.15 Systolischer Algorithmus zur Multiplikation von Bandmatrizen . . 63
3.16 Ein lineares systolisches Array 64
3.17 Multiplikation einer Matrix mit einem Vektor mit Hilfe eines systolischen Feldes 66
3.18 Systolisches Array $(n = 4)$, incl. Basiszelle für die Winograd-Multiplikation 67

4.1 Mapping 73
4.2 Speedup in Abhängigkeit von der Problemgröße mit $p = 8$ Prozessoren vernetzt als Ring, Gitter und Hypercube 75
4.3 Speedup in Abhängigkeit von der Problemgröße für eine unterschiedliche Anzahl von Prozessoren vernetzt in einem Ring 76
4.4 Farmer-Prinzip 81
4.5 Prozeßstruktur 82
4.6 Lastverteilung bei der Berechnung mit 80 Prozessoren 85
4.7 single Prozeßstruktur 87
4.8 multi Prozeßstruktur 87
4.9 Master, Agent und Slave 88

4.10 Abhängigkeit des Speedup von der Matrixgröße 89
4.11 Graphische Darstellung zur effizienten Berechnung von Polynomen. ω_{2n-1} ist $(2n-1)$-te Einheitswurzel. 90
4.12 Eine Butterfly Operation. 93
4.13 Kombinatorische FFT für $n = 8$, s. a. [21] 94
4.14 Rechenzeit in Abhängigkeit von der Matrixgröße, SC 800 96
4.15 Rechenzeit in Abhängigkeit von der Matrixgröße, Pentium III . . 96
4.16 Verteilung der Teiloperationen 97
4.17 Rechenzeit in Abhängigkeit von der Matrixgröße, Multicluster . . 98
4.18 ScaLAPACK Softwarehierarchie 100

Tabellenverzeichnis

1.1 Parameter und statische Topologien 8
1.2 Parameter und dynamische Topologien 9
1.3 Broadcasting 17
1.4 Ausführungszeit des Broadcasting für einige Topologien 18

2.1 Strassen und "naiver" Algorithmus, Vergleich der Zeitkomplexitäten 34

3.1 Operationen der Winogradzellen 67
3.2 Phase 1 68
3.3 Phase 2 68

4.1 Ergebnisse durch parallele Berechnung des Herzfeldes 77
4.2 Meßwerte der Laufzeit 85
4.3 Meßwerte der Laufzeit in Clock-Ticks, wobei ein Clock-Tick $\frac{1}{15625}$s entspricht. 89
4.4 Laufzeitmessungen für den "naiven" und Strassen Algorithmus . . 95
4.5 Rechendichte für BLAS 102

Kapitel 1

Einleitung

1.1 Einführung

Die Attraktivität der Parallelverarbeitung beruht auf der Vorstellung, daß ein Algorithmus $a(n)$, n ist dabei das Maß für die Problemgröße, für einen Parallelrechner mit p Prozessoren entwickelt, p mal schneller ausgeführt werden kann als auf einem Einprozessorsystem. Das läßt zu der Annahme verleiten, daß bei hinreichend großem p die Programmlaufzeit t beliebig verkleinert werden könnte.

$$t_p(a(n)) = \lim_{p \to \infty} \frac{t(a(n))}{p} = 0 \tag{1.1}$$

Daß dies nicht zu erwarten ist, bedarf keiner umfassenden Erläuterung.

Voraussetzung für eine Leistungssteigerung ist natürlich die Parallelisierbarkeit des Problems. Während beispielsweise die Matrizenaddition wegen der Unabhängigkeit der einzelnen Elemente leicht parallelisierbar ist, existieren Problemstellungen, die sich nicht parallelisieren lassen. Das heißt, selbst mit einer großen Anzahl von Prozessoren lassen sich die notwendigen Berechnungen nicht schneller ausführen als mit einem einzigen Prozessor. Ein einfaches Beispiel dieser Art ist in [95] angegeben. Man berechne die Potenz $y = x^{2^n}$ mit $n \in \mathbb{N}, x \in \mathbb{R}$. Sequentielle Algorithmen berechnen y durch fortlaufende Quadrierung in n Schritten. Da aber im ersten Schritt höchstens x^2, im zweiten Schritt höchstens x^4 usw. gebildet werden kann, ist durch den Einsatz weiterer Prozessoren die Berechnung nicht grundsätzlich schneller auszuführen. Der folgende Abschnitt illustriert an einem simplifizierten Beispiel die Problematik der Parallelisierung.

Beispiel 1.1 *Man stelle sich vor, ein auf einem Tisch befindlicher Stapel eines Rommé-Kartenspiels, bestehend aus 104 Karten (Joker mal ausgenommen), sei so schnell wie möglich zu sortieren und als Stapel wieder auf dem Tisch abzulegen.*

Ein naheliegender Zugang zur Lösung der Aufgabe ist gegeben durch das Teile

und Herrsche Konzept, s. a. Abschnitt 1.4. Dazu können wir folgenden Algorithmus realisieren:

1. Teile die 104 Karten nach schwarz und rot.
2. Teile die schwarzen Karten nach Kreuz und Pik und die roten Karten nach Herz und Karo.
3. Teile die 4 Stapel (Kreuz, Pik, Herz, Karo) nach den Farben der Kartenrückseite.
4. Sortiere die verbleibenden 13 Karten in den 8 Stapeln der Größe nach. Für das Sortieren von nur 13 Karten bietet sich der Insertion-Sort Algorithmus [21] an.
5. Lege alle sortierten Stapel übereinander.

Durch das Teilen der 104 Karten in jeweils 8 unabhängige Stapel ist dieser Algorithmus leicht parallelisierbar. So können im 3. Schritt beispielsweise die 4 Stapel gleichzeitig bearbeitet werden. In einer geselligen Runde von 8 gleichsam fingerfertigen Personen haben wir viele Stunden mit folgendem Ergebnis [1] sortiert, Abb. 1.1.
Im Idealfall würde dies bedeuten, daß bei p Prozessoren (Personen) die Zeit zum Lösen eines Problems $\frac{t}{p}$ betragen würde, wenn auf einem Prozessor die Zeit t benötigt wird. Dieser Idealfall wird allerdings nur in den seltensten Fällen erreicht, nämlich dann, wenn die auf die einzelnen Prozessoren verteilten Teilaufgaben genau gleichgroß und wenn sie unabhängig von den Teilaufgaben anderer Prozessoren bearbeitet werden können. Eine ganz ausgeglichene Verteilung der Teilaufgaben ist oft nur schwer realisierbar. Meist hängt die Bearbeitung eines Teilproblems auch vom Ergebnis einer Teilaufgabe in einem anderen Prozessor ab, s. o. simplifiziertes Beispiel. Dies bedeutet, daß zwischen Prozessoren Kommunikation stattfinden muß, die Zeit benötigt. Je größer die Anzahl der am Problem arbeitenden Prozessoren ist, desto größer ist auch der Kommunikationsaufwand. Außerdem muß jeder empfangende Prozessor u. U. warten, wenn die Information nicht rechtzeitig bereitgestellt werden kann. Man sagt daher auch, die Prozessoren würden synchronisiert. Wichtig für parallele Algorithmen sind also

- möglichst geringe Kommunikation,
- möglichst wenig Synchronisation und
- eine gleichmäßige Lastverteilung.

[1] Saxonia Weltrekorde: Ralf Laue sortierte ein gemischtes Kartenspiel, bestehend aus 52 Karten, in 53,2s.

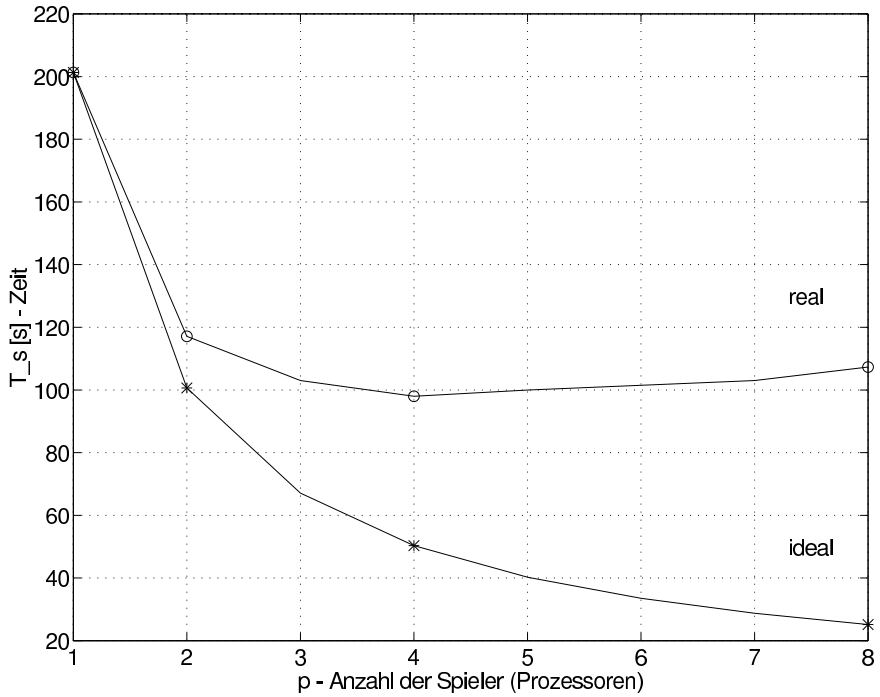


Abbildung 1.1: Ausführungszeit in Abhängigkeit von der Anzahl der Prozessoren für eine konstante Problemgröße

Das Beispiel zeigt zudem noch, daß es offensichtlich eine optimale Anzahl an Prozessoren gibt, die gemeinsam an einem Problem der Größe n (Sortieren von $n = 104$ Karten) arbeiten.

Da wir die Betrachtung über parallele Algorithmen zu Beginn weitgehend losgelöst von aktuellen Rechnerarchitekturen anstellen wollen, legen wir einen hypothetischen Rechner zu Grunde [85]. Dieses System besteht aus beliebig vielen Prozessoren. Jeder Prozessor kann zu jedem Zeitpunkt eine beliebige Operation aus $O = \{+, -, \times, \div\}$ in einem Schritt für die Dauer einer Zeiteinheit ausführen. Es werden zwei Arten von Schritten unterschieden:

- arithmetisch/logische (Operationen) Schritte und
- Datentransport (Routing) Schritte.

Die Laufzeit eines parallelen Programms erhält man also durch die Summe der

Anzahl der zwei Schrittarten oder der Anzahl der benötigten Zeiteinheiten. Jeder Algorithmus entwickelt auf solchem Rechner seine maximale Leistung, weil die Ursachen für Ineffizienz eliminiert sind. Das heißt, Zeitverluste durch Kommunikation sind gleich null, Zugriffskonflikte jeglicher Art treten nicht auf und es sind immer genügend Prozessoren verfügbar. Obwohl der Rechner niemals gebaut werden kann, stellt er ein nützliches Konzept für die Leistungsbewertung von Algorithmen dar.

1.2 Parallele Architekturen

Ohne an dieser Stelle auf Einzelheiten von Parallelprozessoren einzugehen, soll eine Übersicht über die Kategorien paralleler Architekturen gegeben werden. Flynn veröffentlichte [32, 33] eine zunächst grobe Klassifizierung, die bis heute jedoch ihre Bedeutung nicht verloren hat. Ausgehend von der Tatsache, daß ein Rechner Folgen von Operationen auf Folgen von Daten ausführt, unterscheidet man Befehlsströme und Datenströme. Je nach Ein- bzw. Vielfachheit der Ströme gelangt man durch Kombination zu 4 Klassen. Flynn unterscheidet folgende Kategorien:

- SISD - single instruction single data (konventionelle von Neumann Rechner)
- SIMD - single instruction multiple data (Feld- Vektorrechner)
- MISD - multiple instruction single data (leer)
- MIMD - multiple instruction multiple data (Multiprozessorssysteme, Multirechnersysteme)

Abb. 1.2 illustriert die verschiedenen Ansätze in graphischer Form.

Da die Klasse MISD im allgemeinen als leer angesehen wird [54, 44], das Arbeiten verschiedener Prozessoren auf ein und demselben Datum erscheint nicht als sinnvoll, gliedern sich Parallelrechner in die Klassen SIMD und MIMD. Im Falle des SIMD-Prinzips wird genau ein Befehl jeweils gleichzeitig auf einer Vielzahl von Daten ausgeführt. Bei einer MIMD-Architektur hingegen können auf einer Vielzahl von Daten mehrere Befehle jeweils gleichzeitig ausgeführt werden.

Ein weiteres wesentliches Ordnungskriterium ist die Organisation des Speichers. Greifen alle Prozessoren auf einen gemeinsamen Speicher zu, so spricht man von einem Parallelrechner mit shared memory. Besitzt dagegen jeder Prozessor seinen eigenen Speicher, so spricht man von einem Parallelrechner mit distributed memory. Beide Architekturklassen sind in Abb. 1.3 grafisch dargestellt.

Dem Verbindungsnetz kommen dabei unterschiedliche Hauptaufgaben zu. Während es bei Parallelrechnern mit shared memory hilft, das Problem der Zugriffs-

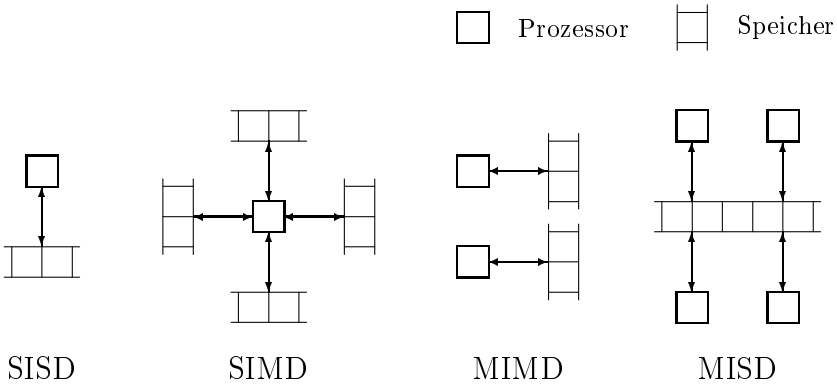


Abbildung 1.2: Klassifikation nach Flynn

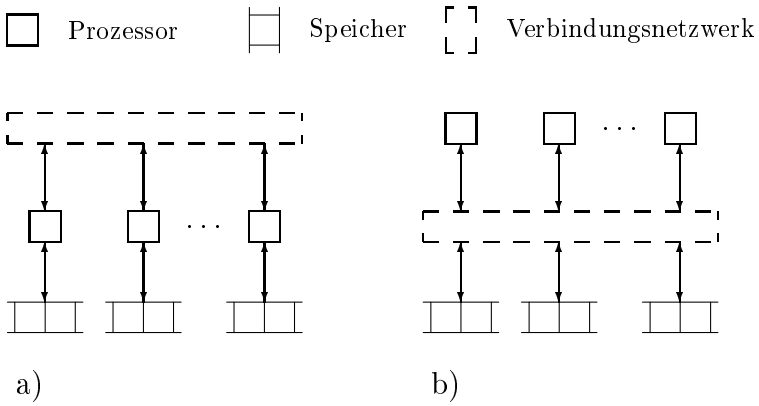


Abbildung 1.3: MIMD-Klassifikation, a) verteilter Speicher, b) gemeinsamer Speicher

konflikte zu mildern, entscheidet es bei Parallelrechnern mit distributed memory über die Effizienz des Datenaustausches. Zur Bewertung von Verbindungsnetzen werden eine Reihe von Kriterien wie Erweiterbarkeit, Leistung (Bandbreite: Anzahl der Daten pro Zeiteinheit, Latenz: Anzahl der Zeiteinheiten pro Datum), Kosten, Zuverlässigkeit und Funktionalität herangezogen, s. a. [47, 44].

Sind Komponenten des Rechnersystems fest miteinander verbunden, so heißt das Netz statisch. Die Topologie bestimmt so die Charakteristik des Netzes. Ein erstes Kriterium für die jeweilige Topologie ist der Knotengrad. Er ist entweder konstant oder von der Größe des Netzes abhängig und gibt die Anzahl der direkten Nachbarn des Knotens an. Ein geringer und konstanter Knotengrad begünstigt demnach niedrigere Kosten. Der ATOLL-chip als Netzwerkinterface, entwickelt an der Uni Mannheim [53], verfügt über 4 bidirektionale Linkports und erlaubt

den Anschluß von bis zu 4 Nachbarn.

Die Struktur mit der geringsten Konnektivität (Verhältnis von Kanten zu Knoten) ist der Ring. Mit einem konstanten Knotengrad von 2 ist der Ring kostengünstig zu realisieren, hat jedoch einen linear mit der Knotenzahl p wachsenden Durchmesser. Unter dem Durchmesser eines Netzes versteht man die maximale Länge für die Verbindung zweier Knoten. Für die Ringarchitektur ist zu bemerken, daß sie sich in die meisten der anderen Strukturen einbetten läßt. Ein Algorithmus, der für einen Parallelrechner mit Ringarchitektur entwickelt wurde, kann deshalb relativ leicht auf fast alle der anderen Architekturen übertragen werden und eignet sich auch für Clusterarchitekturen. Eine sehr populäre Architektur stellt der Hypercube dar. Zwei Prozessoren sind genau dann miteinander verbunden, wenn sich die Binärdarstellungen ihrer Prozessornummern in genau einer Stelle unterscheiden. Die Anzahl p der Prozessoren ist in diesem Fall eine Zweierpotenz. In Abb. 1.4 sind Ring und Hypercube sowie die Einbettung einer Ringarchitektur in einen Hypercube der Dimension 3 dargestellt. Der Ring wurde dabei so eingebettet, daß im Ring benachbarte Prozessoren auch im Hypercube direkt verbunden sind. Dies läßt sich für einen Hypercube beliebiger Dimension mit Hilfe der Gray-Codierung erreichen.

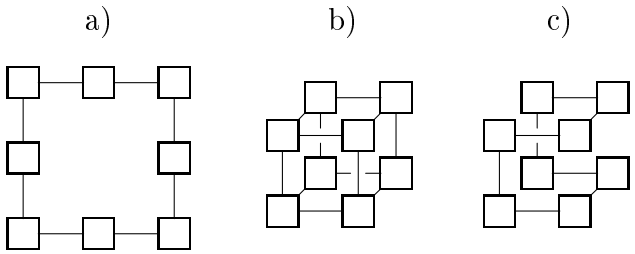


Abbildung 1.4: a) Ring, b) Hypercube, c) Ring eingebettet im Hypercube

Eine sehr verbreitete Topologie ist die des zweidimensionalen Gitter oder des zweidimensionalen Torus, s. Abb. 1.5. Der Knotengrad ist konstant 4, von den Randknoten des Gitter abgesehen. Gitterstrukturen eignen sich für alle Probleme, bei denen ein zweidimensionaler Datenbereich gegeben ist und partitioniert bearbeitet werden kann.

Heiß [41] führt weiter aus, daß auch Gitter und Tori höherer Dimension $d > 2$ verschaltet werden können. Von besonderer Bedeutung sind 3D Architekturen, die sich an unsere dreidimensionale Welt anlehnen. Anwendungen wie Wettervorhersage, Klimamodelle, Aerodynamik, etc. beruhen auf einer diskreten Zerlegung des dreidimensionalen Raumes und der parallelen Berechnung physikalischer Größen mit lokal begrenzter Kommunikation. Der bereits erwähnte Hypercube ist auf Grund des günstigen Durchmessers für Probleme geeignet, die nicht lokale

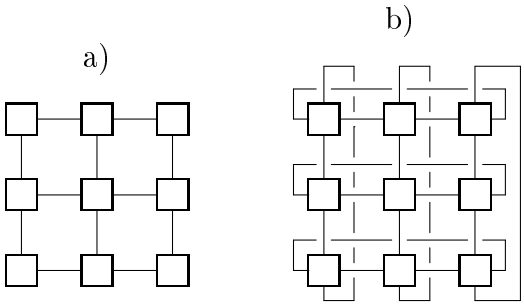


Abbildung 1.5: a) Gitter, b) Torus

Kommunikation erfordern. Aufgrund seiner Struktur sind Erweiterungen nur in Zweierpotenzen möglich. Nachteilig wirkt sich der mit der Dimension wachsende Knotengrad aus. Exemplarisch sei der Butterfly-Graph in Abb. 1.6 genannt, der eine Knotenzahl $n = d \cdot 2^d$ und wegen des Knotengrades von 4 einen Durchmesser $d + \lfloor \frac{d}{2} \rfloor$ besitzt. [2]

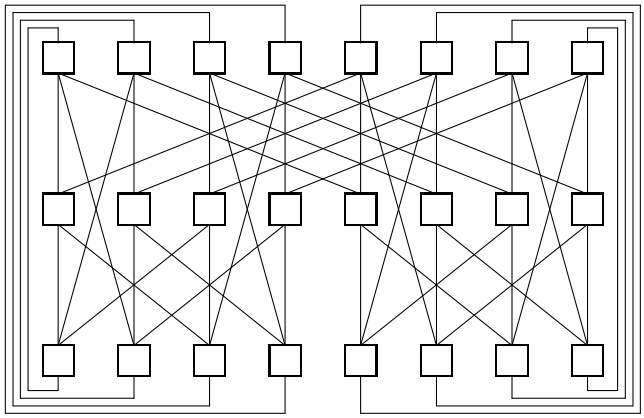


Abbildung 1.6: Butterfly-Graph der Dimension $d = 3$

Neben allgemeinen Kosten/Leistungsverhältnissen können bestimmte Topologien auch wegen ihrer Eignung für bestimmte Algorithmenklassen ausgewählt werden. Im Idealfall spiegelt die Netzwerkstruktur gerade die Kommunikationsstruktur des parallelen Programms wieder. Binärbäume eignen sich etwa für sogenannte Teile und Herrsche Algorithmen. Der Durchmesser wächst nur logarithmisch mit der Knotenzahl, während der Knotengrad maximal 3 ist. Algorithmen, deren

[2] $\lceil x \rceil$, kleinste ganze Zahl η, so daß $\eta \geq x$ bzw. $\lfloor x \rfloor$, größte ganze Zahl η, so daß $\eta \leq x$

Funktionsweise auf rythmischen Strömen von Daten durch Prozessorfelder beruhen, eignen sich besonders gut für die vollständige Hardware-Implementierung mit der VLSI[3]-Technologie. Algorithmen, die auf einem hexagonalen systolischen Feld Abb. 1.7 implementiert werden, werden systolische Algorithmen genannt.

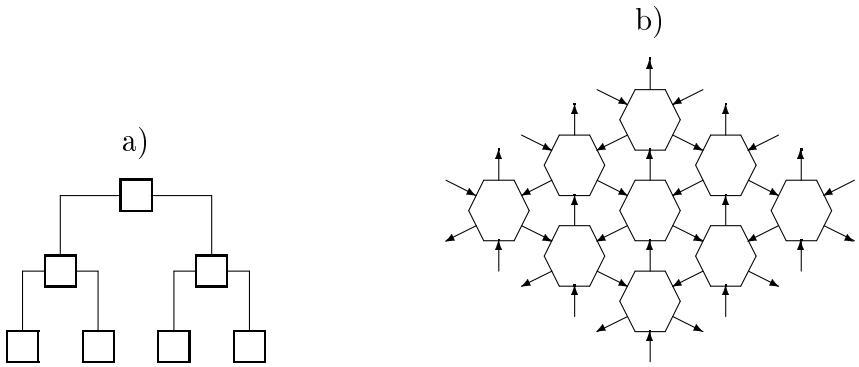


Abbildung 1.7: a) Binärbäum, b) Systolische Felder

Eine Zusammenfassung der Parameter der skizzierten Topologien zeigt Tabelle 1.1.

Topologie	Knotengrad	Durchmesser	Kommentar
Ring	2	$\lfloor \frac{N}{2} \rfloor$	N Knoten
Hypercube	n	n	N Knoten mit $n = \log N$
2D-Gitter	4	$2(r-1)$	$r \times r$ Gitter mit $r = \sqrt{N}$
2D-Torus	4	$2\lfloor \frac{r}{2} \rfloor$	$r \times r$ Torus mit $r = \sqrt{N}$
Binärbaum	3	$2(h-1)$	Höhe des Baumes mit $h = \lceil \log N \rceil$

Tabelle 1.1: Parameter und statische Topologien

Neben den bisher betrachteten statischen Verbindungsnetzen wird noch nach dynamischen unterschieden.

Für einen groben Überblick lassen sich dynamische Verbindungsnetze in die Klassen busartige Netze, Schaltermatrizen und mehrstufige Netze unterteilen. Busse bilden die kostengünstige Alternative. Die Anzahl der anschließbaren Einheiten

[3]VLSI - very large scale integration

ist jedoch bedingt durch auftretende Buskonflikte begrenzt. Ein Beispiel für Buskopplungen sind Multiprozessorsysteme, bedingt eben durch Buskonflikte meist mit moderaten Prozessorzahlen. Die im Vergleich dazu aufwendigste ist die Schaltermatrix, die jeden Anschluß mit jedem verbinden kann. Als Kompromißlösung bieten sich hierarchische Lösungen. Mit Hilfe von Grundelementen, meist vier Schalterzustände s. Abb. 1.8, können mehrstufige Netze gebildet werden. Der quadratische Aufwand kann auf $\frac{n}{2}\log n$ reduziert werden, auf Kosten einer Verzögerung in der Übertragungszeit. Im Unterschied zu statischen Netzen ist die Distanz bzw. Latenz zwischen zwei beliebigen Prozessoren gleich, wächst jedoch logaritmisch mit der Knotenzahl. Abb. 1.8 zeigt exemplarisch ein Banyan-Netz, welches der Algorithmusstruktur der Fast-Fourier-Transformation entspricht. Es existiert genau ein Pfad zwischen einem Eingang und einem Ausgang.

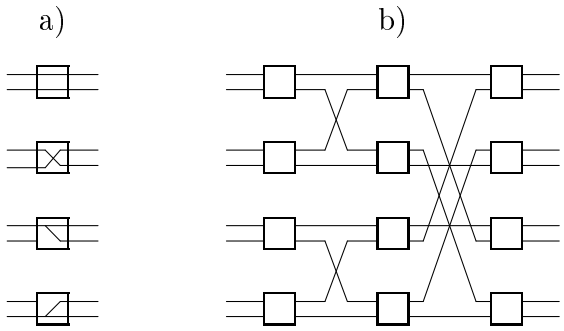


Abbildung 1.8: a) Elementare Schalterzustände, b) Banyan-Netz

Tabelle 1.2 zeigt die drei Klassen dynamischer Verbindungsnetze mit ihren wesentlichen Eigenschaften

	busartige Netze	mehrstufige Netze	Schaltermatrix
Latenz	1	$\log n$	1
Bandbreite	$\frac{1}{n}$	1	1
Kosten	n	$n \log n$	n^2

Tabelle 1.2: Parameter und dynamische Topologien

Im Gegensatz zu den Rechnerarchitekturen ist ein vergleichsweise geringer Aufwand getrieben worden, um Programmiersprachen neu bzw. weiter zu entwickeln.

1.3 Parallele Programmierung

In diesem Abschnitt geht es um Aspekte der parallelen Programmierung, um die Frage, wie gegebene Probleme für eine parallele Lösung aufbereitet und letztlich formuliert werden müssen. Ideal wäre eine Umgebung, in der alle Aspekte der Parallelität dem Programmierer verborgen bleiben, die also eine effiziente vollautomatische Parallelisierung gestattet. Man spricht auch von implizitem Parallelismus, einer Ausdrucksform, die von der Frage sequentiell oder parallel abstrahiert. Ein Beispiel für einen solchen Parallelismus ist die Auswertung arithmetischer Ausdrücke. Die Anwendung von Assoziativ- und Kommutativgesetz erlaubt in vielen Fällen eine Parallelbearbeitung, die vom Übersetzer erkannt und in Maschinenbefehle umgesetzt werden kann. Impliziter Parallelismus bleibt bei imperativen Programmiersprachen auf das Anweisungsniveau beschränkt, da jegliche Parallelität zwischen Anweisungen expliziert werden muß. Sieht der Programmierer selbst bewußt parallele Kontrollflüsse vor, spricht man von explizitem Parallelelismus.

Im vorangegangenen Abschnitt sind die Ausführungsmodelle SIMD und MIMD betrachtet worden. Die SIMD-Rechner sind so etwas wie verallgemeinerte Vektorrechner, wo diese nur parallele Rechenwerke auf den Vektorregistern haben, bestehen jene aus kompletten Prozessoren mit eigenem Speicher und einem Kommunikationsnetzwerk. Alle Prozessoren führen strikt synchron dasselbe Programm auf ihren eigenen privaten Daten aus. Parallelität entsteht in beiden Fällen ausschließlich in Form von Datenparallelität. Das Problem muß so in homogene Datenpartitionen zerlegt werden, daß in jedem Schritt möglichst jeder Prozessor mit Daten versorgt ist. Der Programmierer wird in keiner Weise mit parallelen asynchronen Abläufen konfrontiert. Das Ausführungsmodell ist sequentiell wie bei der herkömmlichen Programmierung. Nachteile ergeben sich aus der synchronen datenparallelen Vorgehensweise für die erreichbare Auslastung der Prozessoren. Am Beispiel der Matrizenmultiplikation betrachtet, werden im ersten Schritt bei n^3 verfügbaren Prozessoren alle zur Durchführung der Multiplikation genutzt. Im zweiten Schritt werden für die erste Stufe der Addition dann nur noch $n^2 \cdot \frac{n}{2}$ Prozessoren genutzt. Die durchschnittliche Prozessorauslastung E_p, s. a. Abschnitt 1.5, über den notwendigen $\log n + 1$ Schritte ergibt sich zu

$$E_p = n^2 \sum_{i=0}^{\log n} 2^i / n^3 (\log n + 1) = \frac{2n^3 - n^2}{n^3(\log n + 1)}. \tag{1.2}$$

Das bedeutet, daß Parallelrechner dieser Art ihre volle Leistung nur erreichen können, wenn die zu bearbeitenden Probleme "größer" sind als die Zahl der vorhandenen Prozessoren. Ist das nicht der Fall, bringen zusätzliche Prozessoren keinen weiteren Nutzen. Bei dem MIMD-Ausführungsmodell führt jeder Prozessor potentiell ein anderes Programm aus. Die Prozessoren arbeiten asynchron zueinander und Abstimmungen finden nur auf Grund des Daten- und Kontrollflusses

statt. Dieses Modell stellt natürlich erheblich höhere Anforderungen an den Programmierer. Im folgenden sollen gängige Ausdrucksformen für Parallelität [41] ein "Gefühl" dafür entwickeln, wie parallele Programme strukturiert sein können und wie sie sich verhalten.

Die nun gleichzeitig auftretende Daten- und Funktionsparallelität muß durch entsprechende Konstrukte in der Programmiersprache kontrollierbar werden. Die einfachste Variante um Parallelität auszudrücken wird durch das **Coroutinen-Konzept** möglich. Die Semantik von Coroutinen läßt jedoch die Ausführung nur einer Coroutine zu einem Zeitpunkt zu und ist daher für Multitaskingsysteme auf Einprozessorsystemen geeignet [100]. Modula-2 [99] zählt zu den Programmiersprachen, die das Coroutinen-Konzept anwenden. Weitere Kontrollstrukturen lehnen sich an gängiger sequentieller Programmiersprachen an. Mit zunehmender Komplexität lassen sich eigenständige Teilaufgaben unter dem Aspekt der Beherrschbarkeit des Gesamtsystems identifizieren. Eine Modularisierung kann aber auch unter dem Aspekt der Parallelisierung erfolgen, was nicht unweigerlich zu einer gänzlich anderen Zerlegung führen muß. Ein "natürlicher" Zerlegungsansatz würde beide Aspekte, den der funktionalen Abgeschlossenheit einer solchen Instanz und den der Nebenläufigkeit gleichzeitig berücksichtigen. Das Ergebnis wäre eine Gruppe von Instanzen, die in einem Beziehungsgeflecht zu einem Ganzen verwoben sind. Die Beziehungen zwischen den Instanzen regeln das zeitliche Verhalten sowie den notwendigen Austausch von Informationen. Diese zu Nebenläufigkeit fähigen Instanzen sind als spezielle syntaktische Einheiten, den **Prozeßdeklarationen** (process, task, thread) eingeführt. Mit der Definition eines Prozesses wird so eine Einheit der Parallelverarbeitung explizit festgelegt. Während bei ConcurrentPascal Prozesse im Deklarationsteil statisch vereinbart werden müssen, erlaubt ADA eine dynamische Prozeßmenge [12]. Eine zweite Variante stellt das **fork/join-Konzept** dar. Durch die Ausführung einer `fork`-Anweisung wird die bezeichnete Anweisungsfolge gestartet. Das rufende Programm und die durch `fork` gestartete Anweisungsfolge laufen parallel zueinander ab. Jedes `fork` korrespondiert eindeutig mit einem `join`, wodurch das rufende Programm mit der Beendigung der durch `fork` gestarteten Anweisungsfolge synchronisiert wird, s. Abb. 1.9. Der entscheidende Nachteil des fork/join-Konzeptes besteht darin, daß eine detaillierte Kenntnis der betrachteten Programme erforderlich ist. Eine undisziplinierte Verwendung kann zu nicht gerade einfach zu verstehenden Programmstrukturen führen.

Mit Hilfe des **parallel blocks-Konzeptes** wird die Menge der Anweisungsfolgen, die parallel ausgeführt werden sollen, in übersichtlicher Weise beschrieben. Durch die Ausführung der Anweisung

```
cobegin s1; s2; ...; sn coend
```

wird die parallele Abarbeitung der Anweisungsfolgen `s1, s2,..., sn` veranlaßt. Die einzelnen `si` können wiederum `cobegin` Anweisungen enthalten, s. Abb. 1.10.

```
    A;
    fork f
    B;
    fork g
    C;
    join f
    D;
    join g
    E;
f:  F;
    end
g:  G;
    end
```

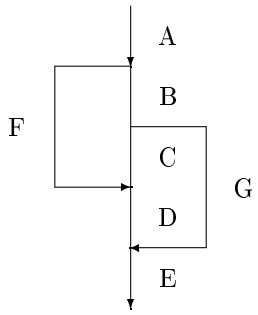


Abbildung 1.9: Beispiel fork/join und Kontrollfluß

Im Unterschied zu fork/join entstehen durch die rekursive Verwendung sauber geschachtelte Parallelitätsstrukturen, die eine gemeinsame Zusammenführung der parallelen Zweige besitzen, `coend` dient als gemeinsamer Synchronisationspunkt. Die Ausführung einer cobegin Anweisung ist nach der Abarbeitung aller `si` Anweisungsfolgen beendet. Auf Grund dieser Struktur ist cobegin weniger mächtig als fork/join, ermöglicht jedoch übersichtlichere Kontrollflußmuster. Ein Kontrollfluß wie in Abb. 1.9 kann beispielsweise nicht erzeugt werden. Varianten dieses Konzeptes werden in den Programmiersprachen ALGOL68 [94], CSP [43] und OCCAM2 [87] verwendet. Es bleibt dem Compiler oder Betriebssystem überlassen, ob nun tatsächlich alles parallel ausgeführt werden kann oder nur partiell parallel.

Die heute am häufigsten eingesetzte Methode ist die **Schleifenparallelisierung**. Statt einer einfachen Repetition des Schleifenkörpers kann eine n-fache Parallelduchführung stattfinden, die im theoretischen Idealfall nur ein n-tel der Zeit in Anspruch nimmt. Die üblichen Schlüsselwörter sind pardo/parend, s. Abb. 1.11.

Im Gegensatz zu cobegin/coend ist bei `pardo` der Parallelitätsgrad nicht statisch festgelegt, sondern datenabhängig und variabel. Der Schleifenkörper kann im allgemeinen recht komplex ausfallen, d. h. Verzweigungen bzw. Unterprogrammaufrufe enthalten. Ebenso wie fork/join und cobegin/coend wirkt das zweite Schlüsselwort, `parend`, als Synchronisationspunkt.

Die Liste der betrachteten Konstrukte ist keineswegs vollständig, es fehlen beispielsweise Konstrukte zur dynamischen Partitionierung von Daten in parallel bearbeitbaren Untermengen, zur Synchronisierung auf gemeinsame Daten im Sinne eines verallgemeinerten Semaphorkonzeptes [100] und zur Kommunikation zwi-

```
A;
cobegin
  B;
  begin
    C;
    cobegin
      D;
      E;
    coend;
    F;
  end;
  G;
coend;
H;
```

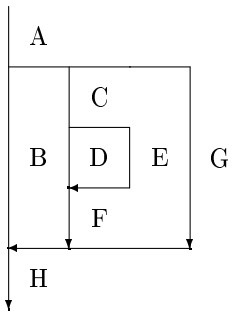


Abbildung 1.10: Beispiel cobegin/coend und Kontrollfluß

schen den parallelen Prozessoren. Selbst wenn alle diese Erweiterungen in geeigneter Weise in eine Programmiersprache eingebettet sind, bleiben noch einige schwierige Fragen bzgl. des Laufzeitsystems offen.

```
A;
pardo i:=1 TO n
  B(i);
  pardo j:=1 TO i
    C(i,j);
  parend;
  D(i);
parend;
E;
```

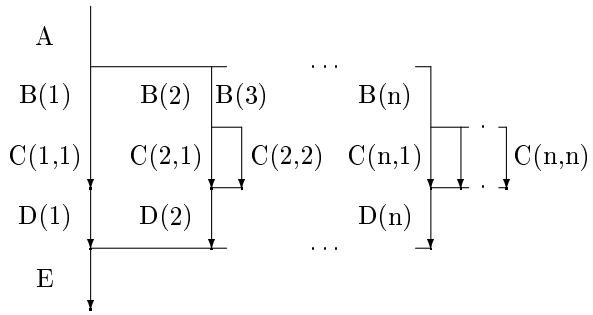


Abbildung 1.11: Beispiel pardo/parend und Kontrollfluß

1.4 Entwurfskonzepte

Innerhalb der Theorie der Parallelisierung von Algorithmen haben sich einige universelle Entwurfstrategien als nützlich herausgestellt. Exemplarisch sollen drei der wichtigsten besprochen werden.

- Teile und Herrsche
- Rekursives Doppeln
- Broadcasting

Die Methoden sind nicht vollständig unabhängig voneinander, vielmehr existieren weitreichende Querverbindungen.

Teile und Herrsche
Ein weithin bekannter Zugang zur Lösung eines Problems ist die Zerlegung dessen in kleinere Teilprobleme. Nach erfolgreicher Lösung dieser Teilprobleme werden die Teillösungen anschließend zur Gesamtlösung zusammengesetzt. Dieser Zugang, zumal in rekursiver Form realisiert, liefert sehr oft effiziente Lösungen zum Problem [1]. Bei vielen Problemstellungen erweist sich das Teile und Herrsche Prinzip als wirkungsvolle Methode der Parallelisierung. Das Teile und Herrsche Prinzip läßt sich beispielsweise dann besonders effizient implementieren, wenn das zur Verfügung stehende Parallelrechnersystem auf eine Kommunikationsstruktur in Form eines Baumes zurückgreifen kann.
Klassisches Beispiel für die Leistungsfähigkeit der Methode Teile und Herrsche ist die Multiplikation von Polynomen und Matrizen.

Beispiel 1.2 *Gegeben seien zwei Polynome $p(x)$ und $q(x)$, zu berechnen ist das Produkt $p(x)q(x)$.*

Der primitive Algorithmus für dieses Problem erfordert n^2 Multiplikationen, wobei jeder der n Summanden von $p(x)$ mit jedem der n Summanden von $q(x)$ multipliziert wird. Eine Verbesserung dieses einfachen Algorithmus wird durch die Verwendung des Teile und Herrsche Prinzips erreicht. Ein Weg, um ein Polynom in zwei zu zerlegen, besteht darin, die Menge der Koeffizienten zu halbieren. Es sei n die Anzahl der Koeffizienten und gerade. Das Polynom

$$p(x) = p_0 + p_1 x + \ldots + p_{n-1} x^{n-1} \tag{1.3}$$

kann in zwei Polynome mit $\frac{n}{2}$ Koeffizienten aufgespaltet werden. Es wird zwischen einem Polynom niedriger (low) und höherer (high) Ordnung unterschieden

$$\begin{aligned} p_l(x) &= p_0 + p_1 x + \ldots + p_{\frac{n}{2}-1} x^{\frac{n}{2}-1}, \\ p_h(x) &= p_{\frac{n}{2}} + p_{\frac{n}{2}+1} x + \ldots + p_{n-1} x^{\frac{n}{2}-1}. \end{aligned} \tag{1.4}$$

Es gilt, wenn $q(x)$ in gleicher Weise aufgespaltet wird

$$\begin{aligned} p(x) &= p_l(x) + x^{\frac{n}{2}} p_h(x), \\ q(x) &= q_l(x) + x^{\frac{n}{2}} q_h(x). \end{aligned} \tag{1.5}$$

Ausgedrückt über die kleineren Polynome ist das Produkt nunmehr gegeben durch

$$p(x)q(x) = p_l(x)q_l(x) + (p_l(x)q_h(x) + q_l(x)p_h(x))x^{\frac{n}{2}} + p_h(x)q_h(x)x^n. \tag{1.6}$$

Der entscheidende Gedanke bei diesem Vorgehen ist nun, daß nur drei Multiplikationen erforderlich sind, um diese Produkte zu berechnen und nicht vier wie es nach obiger Formel den Anschein hat. Zu berechnen sind die Produkte

$$\begin{aligned} r_l(x) &= p_l(x)q_l(x), \\ r_h(x) &= p_h(x)q_h(x), \\ r_m(x) &= (p_l(x) + p_h(x))(q_l(x) + q_h(x)) \end{aligned} \tag{1.7}$$

und das Produkt $p(x)q(x)$ ergibt sich, indem

$$p(x)q(x) = r_l(x) + (r_m(x) - r_l(x) - r_h(x))x^{\frac{n}{2}} + r_h(x)x^n \tag{1.8}$$

berechnet wird. Das Verfahren beruht auf der Tatsache, daß die Addition von Polynomen einen linearen Algorithmus erfordert, die einfache Multiplikation jedoch dagegen quadratisch ist. Es lohnt sich also ein paar "einfache" Additionen auszuführen, um eine "schwierige" Multiplikation einzusparen.

Theorem 1.1 *Zwei Polynome vom Grad n können mit einer Zeitkomplexität von $n^{1.58}$ multipliziert werden.*

Falls M_n die Anzahl der Multiplikationen bezeichnet, die erforderlich sind, um zwei Polynome vom Grad n zu multiplizieren, so gilt:
$M_n = 3M_{\frac{n}{2}}$, für $n \geq 2$, mit $M_1 = 1$
Somit gilt: $M_2 = 3$, $M_4 = 9$ usw. Wenn nun $n = 2^k$ist, gilt:

$$M_{2^k} = 3M_{2^{k-1}} = 3^2 M_{2^{k-2}} = \ldots = 3^k M_1 = 3^k. \tag{1.9}$$

Somit ist $3^k = n^{\log 3}$. Obwohl diese Lösung nur für $n = 2^k$ exakt ist, zeigt sich:

$$M_n = n^{\log 3} = n^{1.58}, \tag{1.10}$$

was im Vergleich zum primitiven Algorithmus eine erhebliche Einsparung ist.

Zu erwähnen ist, daß die schnelle Fourier-Transformation die schnellste Möglichkeit bietet, zwei Polynome zu multiplizieren. Damit erreicht man eine Komplexität von $O(n \log n)$, siehe dazu [21] und Abschnitt 4.4.

Die bekannteste Anwendung des Prinzips Teile und Herrsche auf ein arithmetisches Problem ist die Methode von Strassen für die Multiplikation von Matrizen, s. a. Abschnitt 2.3.

Rekursives Doppeln
Die Idee dieses Konzeptes ist identisch mit dem Teile und Herrsche Prinzip, in dem rekursiv jede Berechnung in zwei unabhängige Teile gleicher Komplexität zerlegt und diese jedoch nun parallel ausgeführt werden.

Beispiel 1.3 *Es sei zu berechnen*

$$y = \prod_{i=0}^{n-1} a_i, \tag{1.11}$$

wobei ohne Einschränkung der Allgemeinheit der Einfachheit halber n eine Potenz von 2, $n = 2^k$ ist.

Um nun y zu berechnen, würde die sequentielle Lösung $n-1$ Schritte verlangen. Das Problem wird jedoch in 2 Teilprobleme zerlegt

$$y = \prod_{i=0}^{n-1} a_i = \prod_{i=0}^{2^{k-1}-1} a_i \cdot \prod_{i=2^{k-1}}^{2^k} a_i. \tag{1.12}$$

Diese beiden Probleme werden so weiter zerlegt, bis sich das Gesamtproblem auf die Elementarprobleme $(a_0 \cdot a_1), (a_2 \cdot a_3), \ldots, (a_{n-2} \cdot a_{n-1})$ reduziert hat. Mit $\frac{n}{2}$ Prozessoren lassen sich alle Elementarprobleme parallel in einem Schritt lösen. Im zweiten Schritt werden die entsprechenden Zwischenergebnisse mit $\frac{n}{4}$ Prozessoren berechnet

$$(a_0 \cdot a_1) \cdot (a_2 \cdot a_3), \ldots, (a_{n-4} \cdot a_{n-3}) \cdot (a_{n-2} \cdot a_{n-1}). \tag{1.13}$$

Nach $k = \log n$ Schritten wird das Ergebnis y erzielt, d. h. die Zeitkomplexität mit rekursivem Doppeln ist gegenüber $T_1(n) = n - 1$

$$T_p(n) \in O(\log n), \quad \text{mit} \quad p = \frac{n}{2}. \tag{1.14}$$

Das rekursive Doppeln ergibt also einen Speedup, s. a. Abschnitt 1.5, von

$$S_p \in O\left(\frac{n}{\log n}\right), \quad \text{für} \quad p = \frac{n}{2}. \tag{1.15}$$

Bei der Anwendung des rekursiven Doppelns brauchen die Teilprobleme der Zerlegung nicht notwendiger Weise kleinere Versionen des ursprünglichen Problems sein, jedoch sollten die Teilprobleme assoziative Eigenschaften [88] aufweisen, die eine weitere Zerlegung erlauben.

Broadcasting
Unter Broadcasting versteht man das explosionsartige Ausbreiten von Information innerhalb eines Gesamtsystems. Die Erläuterung des Grundprinzips gelingt am besten anhand eines konkreten Beispiels.

Beispiel 1.4 *Es sei n eine Potenz von 2, sowie x eine gegebene reelle Zahl. Es sollen alle Potenzen $x, \ldots, x^n$ berechnet werden.*

Mit Hilfe eines gewöhnlichen Einprozessorsystems kann man nicht schneller als n Zeittakte sein, da n verschiedene Werte zu ermitteln sind und je Takt höchstens eine neue Potenz berechnet werden kann. Sind jedoch $\frac{n}{2}$ Prozessoren verfügbar so kann das Broadcasting angewendet werden, wobei nach folgendem Schema gearbeitet wird, s. Tabelle 1.3.

Schritt	Berechnung	beteiligte Prozessoren
1	x^2	1
2	$x^3\ x^4$	2
3	$x^5\ x^6\ x^7\ x^8$	4
...	...	...
$\log n$	$x^{\frac{n}{2}+1}$ ... x^n	$\frac{n}{2}$

Tabelle 1.3: Broadcasting

Deutlich ist die lawinenartige Ausbreitung und das Anwachsen der Ergebnismenge erkennbar. Nachteilig dagegen ist die sehr schlechte Auslastung der Prozessoren. Im dritten Takt arbeiten beispielsweise nur 4 Prozessoren und erst im Schritt $\log n$ sind alle $\frac{n}{2}$ Prozessoren aktiv. Der erreichbare Speedup mit Hilfe des Broadcasting liegt bei

$$S_p(n) \in O\left(\frac{n}{\log n}\right). \tag{1.16}$$

Unter einem Broadcast kann man auch das Kopieren eines Datums von einem Prozessor P_0 auf alle anderen im Netz erreichbaren Prozessoren verstehen. Der Vollständigkeit halber sei in diesem Zusammenhang auf das one-to-one (Unicast), one-to-many (Multicast) und one-to-all (Broadcast) in [47] verwiesen. Wie der

Name bereits verrät, korrespondiert beispielsweise die one-to-many Kommunikation mit dem Senden eines Datums von einer Quelle an eine Gruppe von Zielen. Exemplarisch zeigt Abb. 1.12 das Broadcasting in einem 2D bzw. 3D Gitter. Der markierte Knoten entspricht dabei dem Quellknoten P_0.

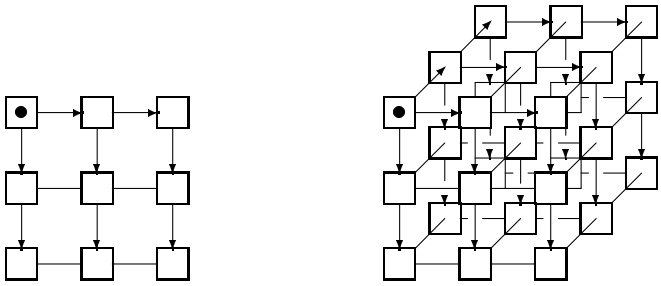


Abbildung 1.12: Broadcasting in einem 2D bzw. 3D Gitter

Die Ausführungszeit beim Broadcasting in einem (m, m) 2D-Gitter beträgt $2T(m-1)$ und in einem (m, m, m) 3D-Gitter $3T(m-1)$ und ist somit proportional dem Durchmesser des Netzwerks. Ähnlich verhält es sich für eine Hypercube Architektur mit dem Durchmesser d. Das theoretische Minimum für die Ausführungszeit ist von der Ordnung $O(\log p)$ und damit ebenfalls proportional dem Durchmesser. Tabelle 1.4 faßt Ausführungszeiten für das Broadcasting für einige Topologien mit p Prozessoren zusammen.

Topologie	Ausführungszeit
Ring	$O(p)$
2D Gitter	$O(p^{\frac{1}{2}})$
Torus	$O(p^{\frac{1}{2}})$
3D Gitter	$O(p^{\frac{1}{3}})$

Tabelle 1.4: Ausführungszeit des Broadcasting für einige Topologien

Analog zum Broadcasting verhält sich das Aggregating, worunter das Einsammeln eines Datums von jedem Prozessor und die Komposition in ein Datum zu verstehen ist.

1.5 Kenngrößen paralleler Algorithmen

Die PRAM[4] ist ein Rechnermodell, in dem sich p Prozessoren einen gemeinsamen Speicher teilen. Der Speicher ist adressierbar und jeder Prozessor kann auf jeden Speicherplatz des gemeinsamen Speichers zugreifen. Darüber hinaus kann jedem Prozessor ein lokaler Speicher zugeordnet werden. Für die Organisation des Speichers stehen rein kombinatorisch vier Zugriffsmodelle zur Verfügung.

EREW	:	exclusiv read	exclusiv write
CREW	:	concurrent read	exclusiv write
ERCW	:	exclusiv read	concurrent write
CRCW	:	concurrent read	concurrent write

Während gleichzeitiges Lesen (CR-concurrent read) einer Adresse des Speichers durch mehrere Prozessoren im allgemeinen keine Probleme hervorruft, kann gleichzeitiges Schreiben problematisch sein. Hier sind Vereinbarungen notwendig, die den Schreibkonflikt vermeiden.
Beispielsweise:

- Prozessor mit niedrigster Adresse hat Schreibpriorität
- Schreibverbot bei unterschiedlichen Eintragungsversuchen
- Summe aller Einträge wird geschrieben

Das folgende Beispiel veranschaulicht nun, wie ein paralleler Algorithmus auf den PRAM-Modellen durchgeführt wird.

Beispiel 1.5 *Es sei $p = 2^k$ die Anzahl der Prozessoren der PRAM-Rechnermodelle. Prüfe, ob in einem unsortierten Datenbereich mit n Plätzen, $n \geq p$, ein bestimmtes Element x enthalten ist.*

Ein natürlich sequentieller Algorithmus fragt den Datenbereich sukzessive ab und benötigt im worst-case n und im average-case $\frac{n}{2}$ Schritte.

EREW-PRAM:
Zur Initialisierung des gesuchten x ist aufgrund des exklusiven Lesens ein Broadcasting von x an alle P_i, für $i = 0 \ldots p-1$ Prozessoren nötig, welches nach $\log p$ Schritten beendet ist. Für den eigentlichen parallelen Algorithmus wird der

[4]PRAM - parallel random access machine

Datenbereich in p Teilbereiche der Größe $\lceil \frac{n}{p} \rceil$ unterteilt. Jeder Prozessor P_i durchsucht seinen aus höchstens $\lceil \frac{n}{p} \rceil$ Plätzen bestehenden Datenbereich auf das Gesuchte x. Der Algorithmus ist im worst-case nach $\lceil \frac{n}{p} \rceil$ Schritten beendet. Die Suche soll nun, nachdem x von einem der Prozessoren gefunden ist, abgebrochen werden. Da gemeinsames Schreiben verboten ist, erhält jeder ein Endsignalregister im Zentralspeicher. Um rechtzeitig abzubrechen, muß nach jedem Suchschritt festgestellt werden, ob x von einem der Prozessoren gefunden wurde. Dies geschieht folgendermaßen in $\log p$ Schritten. Im Schritt j, $0 \leq j \leq \log p - 1$ wenden die Prozessoren P_i, $0 \leq i < 2^{k-1-j}$ logisch "OR" auf ihr Endsignal und das Endsignal von Prozessor $P_{i+2^{k-1-j}}$ an und schreiben das Ergebnis in das Endsignal-Register von P_i, s. Abb. 1.13.

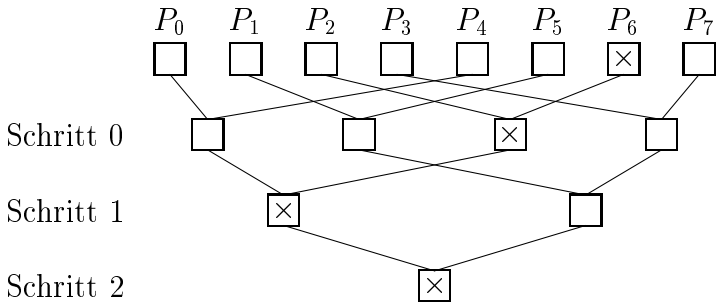


Abbildung 1.13: Informationsfluß

Offensichtlich liegt nach $\log p$ Schritten das Ergebnis in P_0 vor und kann nun in ebenfalls $\log p$ Broadcast-Schritten allen anderen Prozessoren mitgeteilt werden. Insgesamt ergibt sich für das Problem im worst case eine Zeitkomplexität von

$$\log p + \left\lceil \frac{n}{p} \right\rceil (1 + 2 \log p). \tag{1.17}$$

CRCW-PRAM:
Da dieses Rechnermodell gemeinsames Lesen und Schreiben zuläßt, ist zum Einen das anfängliche Broadcasting von x überflüssig, d. h. alle Prozessoren können x in einem Schritt einlesen, und das logische "OR" aller Endsignale kann ebenfalls in einem Schritt berechnet werden. Alle Prozessoren schreiben gleichzeitig mit festgelegter "OR" Schreibkonvention in das Endsignalregister. Mit diesen Überlegungen ergibt sich ein worst case Aufwand von

$$1 + 2 \left\lceil \frac{n}{p} \right\rceil \tag{1.18}$$

Schritten.

Das Beispiel verdeutlicht in anschaulicher Form die Arbeitsweise und vor allem die Leistungsstärke der PRAM sogar in der schwächsten Variante, der EREW-PRAM. Bisher gibt es allerdings keinen Parallelrechner, der dieses Modell realisiert. Der Grund liegt in dem hohen Kodier- und Dekodieraufwand für den Zugriff aller Prozessoren auf den gemeinsamen Speicher [67]. Die PRAM ist aufgrund ihrer uneingeschränkten Kommunikationsmöglichkeiten ein universelles Modell. Die Matrizenmultiplikation auf der CRCW-PRAM beispielsweise beruht auf einer geschickten Auflösung des Schreibkonfliktes. Er wird einfach durch die Vereinbarung aufgelöst, daß die Summe aller Einträge geschrieben wird. Der Algorithmus selbst stellt eine direkte Parallelisierung des sequentiellen "naiven" Algorithmus für die Matrixmultiplikation dar. Er arbeitet mit n^3 Prozessoren P_{ijk}. Zunächst werden die Matrizen $\mathcal{A}$ und $\mathcal{B}$ in den gemeinsamen Speicher gelesen. Alle Prozessoren führen dann parallel in einem Schritt den Befehl

$$c_{ij} := a_{ik} b_{kj} \tag{1.19}$$

aus. Natürlich beträgt die Laufzeit $T(n) \in O(1)$. Da n^3 Prozessoren benötigt werden, erhalten wir die Kosten, s. Formel 1.26, $C(n) \in O(n^3)O(1) = O(n^3)$. Es ist jedoch nicht möglich, eine CRCW-PRAM auf einem Parallelrechner mit festem Verbindungsnetzwerk zu simulieren, in dem jeder Prozessor mit einer begrenzten Anzahl anderer Prozessoren verbunden ist. Dabei verlangsamt sich die Zeit der Ausführung eines parallelen Programms nur um den Faktor der Größenordnung $\log^2 n$ [2]. Ein Vorteil von Verbindungsnetzwerken besteht darin, daß nicht wie bei der PRAM zwischen den Zugriffsmodellen EREW, CREW, ERCW und CRCW unterschieden werden muß. Es gibt also keine Probleme der Exklusivität. Werden p Prozessoren unidirektional oder bidirektional miteinander verbunden so entsteht ein Verbindungsnetzwerk.

Auf der Grundlage des in Abschnitt 1.1 betrachteten hypothetischen Parallelprozessorsystems soll keine abstrakte Komplexitätstheorie betrieben werden sondern pragmatisch Algorithmenanalyse, indem Zeitschritte bestimmt werden (wenigstens der Ordnung nach), die ein Algorithmus, der für einen Parallelrechner mit p Prozessoren entwickelt wurde, tatsächlich bzw. in den Grenzfällen benötigt [89]. Zur Diskussion der Zusammenhänge werden folgende Größen verwendet:

p	sei die Anzahl der Prozessoren im zugrundegelegten Parallelrechner
$T_1(n)$	Zeitkomplexität, benötigte Programmabarbeitungszeit (Anzahl von Zeitschritten) auf einem Einprozessorsystem
$T_p(n)$	Zeitkomplexität, benötigte Programmabarbeitungszeit (Anzahl von Zeitschritten) auf einem p-Prozessorsystem

Dabei ist n das bei Algorithmen allgemein bekannte Maß für die Problemgröße. Das folgende Beispiel veranschaulicht die verschiedenen Möglichkeiten zur Fest-

legung der Problemgröße.

Bei der Multiplikation zweier Matrizen vom Typ (m, m) ergeben sich folgende Möglichkeiten zum "sinnvollen" Festlegen der Klassifikationsinvarianten n:

(i)	$n := m$	
(ii)	$n := 2m$	Spalte + Zeile
(iii)	$n := m^2$	Matrixelemente
(iv)	$n := 2m^2$	
(v)	$n := 2 \cdot 32m^2$	Matrixelemente sind als Bitstring der Länge 32 dargestellt

Die betrachteten Algorithmen beziehen sich meist der Einfachheit halber auf quadratische Matrizen, so daß für die Algorithmenanalyse Variante (i) i. a. verwendet wird.

Zum Vergleich der asymptotischen Komplexitäten definiert man eine (Größen-) Ordnung innerhalb der reellen Funktionen. Die reelle Funktion $f(x)$ heißt von der Ordnung $g(x)$, geschrieben

$$f(x) = O(g(x)), \tag{1.20}$$

wenn für eine Konstante $c \in \mathbb{R}$ und fast alle natürliche Zahlen $n \in \mathbb{N}$ gilt: $f(n) \leq c \cdot g(n)$. Ein Algorithmus ist umso besser, je kleiner das asymptotische Wachstum seiner Komplexitätsfunktion ist.

Der Gewinn an Bearbeitungszeit, der durch Parallelverarbeitung erreicht wird, ist gerade das Verhältnis der Zeitkomplexitäten

$$S_p = \frac{T_1}{T_p}. \tag{1.21}$$

Dieser Quotient wird auch als Speedup [46] bezeichnet. Der Speedup liegt in den Grenzen $1 \leq S_p \leq p$.

Eine weitere Kenngröße ist die Auslastung der p Prozessoren während der Ausführung des Algorithmus, die Effizienz des Systems

$$E_p = \frac{S_p}{p} \leq 1. \tag{1.22}$$

Im theoretischen Idealfall, linearer Speedup, wäre $S_p = p$ und $E_p = 1$. Ein "überlinearer" Speedup verbunden mit einer Effizienz $E_p > 1$ ist irreführend, da von unterschiedlichen Voraussetzungen ausgegangen wird, d. h. es werden

unterschiedliche Algorithmen miteinander verglichen. Für den Fall, daß ein Algorithmus gegeben und der Parallelitätsgrad einstellbar ist, gilt grundsätzlich $S_p \leq p$.
Genauer gesagt spricht man von einem Parallelitätsprofil $p(t)$ eines Programms, also dem Verlauf des Parallelitätsgrades über der Programmabarbeitungszeit. Abb. 1.14 zeigt das Parallelitätsprofil für einen Teile und Herrsche Algorithmus

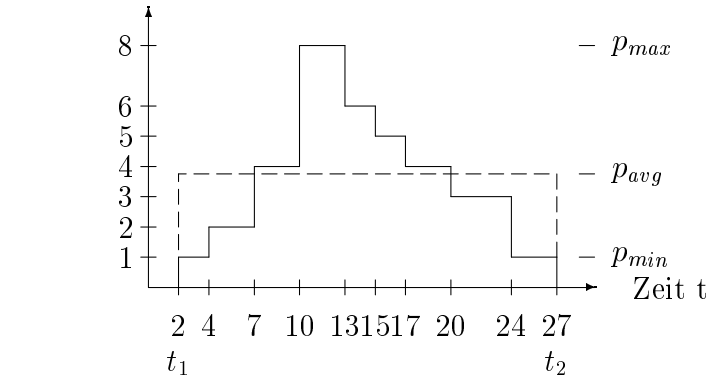


Abbildung 1.14: Parallelitätsprofil für einen Teile und Herrsche Algorithmus

Definition 1.1 *Es sei $(t_{i+1} - t_i)$ ein Zeitintervall und g die von einem Programm während des Zeitintervalls für die Ausführung benutzten Prozessoren eines Parallelrechners, dann ist $p(t) := g$ mit $g > 1$ und geradzahlig der Grad der Parallelität dieses Programms in diesem Zeitintervall unter der Annahme, daß genügend Prozessoren verfügbar sind, $p > g$.*

Es gilt

$$\int_{t_1}^{t_2} p(t)dt = T_1, \tag{1.23}$$

d. h. der Inhalt der Fläche, der von der Kurve $p(t)$, der Zeitachse t, und den Ordinaten $p(t_1)$ und $p(t_2)$ begrenzt wird, gibt die benötigte Rechenzeit an und entspricht daher der Bearbeitungszeit im Einprozessorfall. Der mittlere Parallelitätsgrad p_{avg} läßt sich definieren als

$$p_{avg} = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} p(t)dt \tag{1.24}$$

und entspricht somit dem asymptotischen Speedup

$$p_{avg} = \frac{T_1}{T_\infty} = S_\infty = \lim_{p \to \infty} S_p. \tag{1.25}$$

Auf Grund der vernachlässigten Kommunikationslast ist in der Regel $S_\infty \leq p_{avg}$, d. h. der mittlere Parallelitätsgrad stellt eine obere Grenze des asymptotischen Speedup dar.

Die Verwendung der Begriffe Speedup und Effizienz ist nicht ganz unproblematisch. Parallelrechner werden meist zur Lösung sehr großer Probleme eingesetzt, für die die Kapazitäten eines einzelnen Prozessors nicht ausreichen. In diesen "interessierenden" Fällen können Speedup und Effizienz nicht ohne weitere Überlegungen nach den obigen Definitionen berechnet werden. Eine andere Möglichkeit zum Vergleich von Algorithmen besteht darin, zu messen, wie gut im jeweiligen Algorithmus die potentielle Rechenleistung der Prozessoren ausgenutzt wird. Die Rechenleistung eines Prozessors wird in MFlops gemessen. Für einen Rechner wird vom Hersteller eine Peakperformance angegeben, oft eine rein theoretische Größe. Deshalb sind Vergleiche mit optimierten BLAS[5]-Routinen s. Abschnitt 4.6 meist interessanter.

Mit Formel 1.26 werden die Kosten eines parallelen Algorithmus

$$C(n) = T(n) \cdot p \tag{1.26}$$

angegeben. $C(n)$ ist ein realistisches Kostenmaß. Durch die Produktbildung wird eine übliche Tradeoff Funktion wiedergegeben, d. h. mit anderen Worten führt eine Verminderung der Prozessoranzahl zu einer Verlangsamung des Algorithmus und eine Beschleunigung des Algorithmus zu einem höheren Prozessorbedarf.

1.6 Darstellung der Algorithmen

Alle Algorithmen der nächsten Kapitel werden mit Hilfe eines Pseudocode dargestellt, s. a. [79].

Laufanweisungen haben dabei die Form

```
FORStatement = FOR identifier ":=" expression1 TO expression2
                 {BY expression3} DO
                 StatementSequence END
```

Die Angabe einer Schrittweite `BY expression3` ist dabei optional. Ist keine angegeben, so wird eine Schrittweite von 1 angenommen.

[5]BLAS - basic linear algebra subprograms

```
FORPARStatement = FOR identifier ":="expression1 TO expression2
                    DO IN PARALLEL
                    StatementSequence END
```

Das `DO IN PARALLEL` bewirkt generell das parallele Ausführen der `StatementSequence`, d. h. angewendet im `FOR` Konstrukt wird die Anweisungsfolge nur einmal durchlaufen. Verzweigungen werden mit folgender Syntax

```
IFStatement = IF expression THEN StatementSequence
              {ELSIF expression THEN StatementSequence}
              {ELSE StatementSequence} END
```

angegeben. Hier kann der `ELSIF expression THEN StatementSequence` als auch `ELSE StatementSequence` entfallen. Der `WHILE` Befehl bewirkt, daß die zu wiederholende Anweisung oder Folge von Anweisungen solange ausgeführt wird, wie die kontrollierende Bedingung erfüllt ist. Die Syntax ist durch

```
WHILEStatement = WHILE expression DO
                   StatementSequence END
```

gegeben. Einige Hilfsfunktionen sind durch ein Beispiel selbsterklärend.

```
FUNCTION Bit(m,l);
(*
  liefert den Wert des l-ten Bits f"ur ein Datum m,
  Bit(6,0)=0
*)

FUNCTION BitComplement(m,l);
(*
  liefert den Wert von m, wobei das l-te Bit negiert ist,
  BitComplement(6,0)=7
*)
```

Um parallele Algorithmen darzustellen, werden einige Anweisungen benötigt: Das Verschicken von Daten zwischen zwei Prozessoren wird beim Sender durch die Anweisung

```
send(Daten,Empfaenger);
```

und beim Empfänger durch

```
receive(Daten);
```

bewirkt. Um die Algorithmen möglichst übersichtlich darzustellen, werden bestimmte Operationen mit Hilfe umgangssprachlicher Formulierungen beschrieben.

Kapitel 2

Das Matrizenprodukt

2.1 Eine Einführung

Um dem Leser das Verständnis der Arbeit zu erleichtern, sind im ersten Abschnitt die wichtigsten Begriffe aufgeführt. So ist in der Bedeutung eines rechteckigen Koeffizientenschemas das Wort Matrix zuerst von dem englischen Mathematiker Sylvester [91] benutzt worden.

Definition 2.1 *Eine Matrix $\mathcal{A}$ ist eine Anordnung von $(m \cdot n)$ Ausdrücken in einem rechteckigen Schema von m Zeilen und n Spalten.*

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = \mathcal{A} = (a_{ik}). \tag{2.1}$$

Ein Ausdruck a_{ik} heißt Element der Matrix $\mathcal{A}$ und steht in der i-ten Zeile und der k-ten Spalte. Hat eine Matrix m Zeilen und n Spalten, so ist die Matrix vom Typ (m, n), $(m, n \in N)$.

Definition 2.2 *Sind $\mathcal{A} = (a_{ij})$ und $\mathcal{B} = (b_{ij})$ zwei Matrizen von je m Zeilen und n Spalten, also vom Typ (m, n), so wird die Summe von $\mathcal{A}$ und $\mathcal{B}$ durch eine Matrix vom Typ (m, n)*

$$\mathcal{C} = \mathcal{A} + \mathcal{B} = (c_{ij}) \tag{2.2}$$

mit $c_{ij} = a_{ij} + b_{ij}$ erklärt.

Betrachtet man die Zeitkomplexität der Addition von Matrizen vom Typ (n, n), ergibt sich

$$T(n) = n^2 \in O(n^2) \tag{2.3}$$

Definition 2.3 *Unter dem Produkt $\mathcal{A}x$ einer Matrix $\mathcal{A} = (a_{ik})$ vom Typ (m, n) mit einem n-reihigen Spaltevektor $x = (x_k)$ versteht man den m-reihigen Spaltenvektor $y = (y_i)$, wobei die i-te Komponente y_i als das skalare Produkt*

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n = y_i \tag{2.4}$$

der i-ten Zeile von $\mathcal{A}$ mit der Spalte x entsteht.

Zur Ausführbarkeit eines Produktes $\mathcal{AB}$ [16] ist die Übereinstimmung der Spaltenzahl von $\mathcal{A}$ mit der Zeilenzahl von $\mathcal{B}$ erforderlich. $\mathcal{A}$ ist mit $\mathcal{B}$ in der Reihenfolge $\mathcal{AB}$ verkettbar [13], was gleichbedeutend mit der Multiplizierbarkeit der beiden Matrizen in der angegebenen Reihenfolge ist.

Definition 2.4 *Unter dem Produkt $\mathcal{AB}$ einer Matrix $\mathcal{A}$ vom Typ (m, n) mit einer Matrix $\mathcal{B}$ vom Typ (n, p) in der angegebenen Reihenfolge versteht man die Matrix $\mathcal{C} = \mathcal{AB}$ vom Typ (m, p), deren Elemente c_{ik} als skalares Produkt der i-ten Zeile von $\mathcal{A}$ (des Zeilenvektors a_i) mit der k-ten Spalte von $\mathcal{B}$ (dem Spaltenvektor b_k) gebildet werden.*

$$C = AB = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mp} \end{pmatrix} \tag{2.5}$$

$$c_{ik} = a_{i1}b_{1k} + a_{i2}b_{2k} + \ldots + a_{in}b_{nk} = \sum_{r=1}^{n} a_{ir}b_{rk} = a_i b_k \tag{2.6}$$

wobei $1 \leq i \leq m, 1 \leq k \leq p$.

Dabei stellt der Ausdruck $a_i b_k$ ebenfalls ein Matrizenprodukt dar [77]. Die Multiplikation der Zeile a_i mit der Spalte b_k ergibt eine Matrix c_{ik} vom Typ $(1, 1)$. Aus Formel 2.6 ist ersichtlich, daß die Berechnung eines Elementes c_{ik} des Skalarproduktes der beiden n-stelligen Vektoren a_i und b_k an arithmetischen Rechenoperationen n Multiplikationen und $n-1$ Additionen erfordert. Für die Produktmatrix $\mathcal{C} = \mathcal{AB}$, bestehend aus mp Elementen sind somit mnp Multiplikationen und $mp(n-1)$ Additionen auszuführen. Für die Zeitkomplexität dieses "naiven" Algorithmus ergibt sich

$$T(m,n,p) = mnp + mp(n-1) = mp(2n-1). \tag{2.7}$$

Für den Fall quadratischer Matrizen vom Typ (n,n) ergibt sich eine Zeitkomplexität von

$$T(n) = n^2(n-1) \in O(n^3). \tag{2.8}$$

Die Berechnung der Produktmatrix C ist ein nicht "müheloser" Prozeß, der allerdings recht schematisch abläuft.

```
FOR i=1 TO n DO
  FOR j=1 TO l DO
    t:=0;
    FOR k=1 TO m DO
      t:=t + a(i,k) * b(k,j);
    END (* for *);
    c(i,j)=t;
  END (* for *);
END (* for *);
```

Die Matrizenmultiplikation ist Hauptinhalt des Matrizenkalküls. Viele Probleme der Linearen Algebra, der Kombinatorik, Matrixinversion, Determinantenrechnung etc. können auf die Matrizenmultiplikation zurückgeführt werden. Das heißt, die benötigte Rechenzeit für die Matrizenmultiplikation ist der dominante Teil für solche Probleme. Die Zeitkomplexität kann für eine Vielzahl von Problemen gesenkt werden, wenn die Zeitkomplexität für die Matrizenmultiplikation gesenkt wird. Hieraus stellt sich die Frage: Wie schnell können Matrizen multipliziert werden? Die obere Grenze der Zeitkomplexität wird durch den "naiven" Algorithmus festgelegt. Da n^2 Elemente zu berechnen und diese bestenfalls in einem Schritt gelöst werden könnten, wird so die untere Grenze durch n^2 bestimmt.

$$O(n^2) \leq O(n^\omega) \leq O(n^3). \tag{2.9}$$

Hier ist ω definiert als Exponent der Matrizenmultiplikation.

2.2 Winograd-Algorithmus

Ausgangspunkt der Betrachtungen ist folgende Identität

$$a_1b_1 + a_2b_2 = (a_1 + b_2)(a_2 + b_1) - a_1b_2 - a_2b_1. \tag{2.10}$$

Die sogenannte Winograd-Identität [96] ergibt sich durch Erweiterung der paarweisen Produkte auf die Zahl n, was dem Skalarprodukt zweier Vektoren $a = (a_1, \ldots, a_n)$ und $b = (b_1, \ldots, b_n)$ entspricht. So ergibt sich für das Skalarprodukt nach Winograd

$$\sum_{i=1}^{n} a_i b_i = \begin{cases} \sum_{r=1}^{k} (a_{2r-1} + b_{2r})(a_{2r} + b_{2r-1}) & \\ \\ -\sum_{r=1}^{k} (a_{2r-1} a_{2r}) - \sum_{r=1}^{k} b_{2r-1} b_{2r} & : \quad n = 2k \\ \\ \sum_{r=1}^{k} (a_{2r-1} + b_{2r})(a_{2r} + b_{2r-1}) & \\ \\ -\sum_{r=1}^{k} (a_{2r-1} a_{2r}) - \sum_{r=1}^{k} b_{2r-1} b_{2r} + a_n b_n & : \quad n = 2k+1. \end{cases} \tag{2.11}$$

Zur Bestimmung der Zeitkomplexität sei der Einfachheit halber n gerade. Die linke Seite erfordert bekanntlich $n-1$ Additionen und n Multiplikationen, $T(n) = 2n - 1$. Für die Berechnung des Skalarproduktes nach Winograd ergeben sich demnach aus Formel 2.11 $3(k-1) + 2k + 2 = \frac{5}{2}n - 1$ Additionen und $3k = \frac{3}{2}n$ Multiplikationen, eine Zeitkomplexität von

$$T(n) = 4n - 1. \tag{2.12}$$

Bezogen auf das Skalarprodukt bedeutet die Lösung nach Winograd einen erheblichen Mehraufwand. Für die Matrixmultiplikation erweist sich dieser Ansatz jedoch als vorteilhaft.

Gegeben sind die Matrizen $\mathcal{A}$ vom Typ (n, m) und $\mathcal{B}$ vom Typ (m, l), also $\mathcal{C}$ vom Typ (n, l). Voraussetzung sei wieder $m = 2k$, m ist gerade. Der auf der Winograd-Identität beruhende Algorithmus zur Matrixmultiplikation läßt sich in 3 Prozeduren 2.13, 2.14 und 2.15 aufteilen. Berechne:

$$\forall_{i=1}^{n} \quad e_i = \sum_{r=1}^{k} a_{i,2r-1} a_{i,2r} \tag{2.13}$$

$$\forall_{j=1}^{l} \quad g_j = \sum_{r=1}^{k} b_{2r-1,j} b_{2r,j} \tag{2.14}$$

$$\forall_{i=1}^{n} \forall_{j=1}^{l} \quad c_{ij} = \sum_{r=1}^{k} (a_{i,2r-1} + b_{2r,j})(a_{i,2r} + b_{2r-1,j}) - e_i - g_j. \tag{2.15}$$

Winograd's Matrixmultiplikation erfordert mit $k = \frac{m}{2}$, $\frac{m}{2}(n+l) + \frac{mnl}{2}$ Multiplikationen sowie $\frac{3}{2}mnl + (\frac{m}{2} - 1)(n + l) + nl$ Additionen. Winograd's Matrixmul-

tiplikation hat wie der "naive" Algorithmus für $m = n = l$ eine Zeitkomplexität von

$$T(n) \in O(n^3). \tag{2.16}$$

Vergleicht man jedoch die Werte im einzelnen, so wird die Anzahl der Multiplikationen auf Kosten der Anzahl der Additionen halbiert. Für reale Systeme ist die Multiplikation aufwendiger, so daß für hinreichend große n durchaus ein Gewinn durch den Winograd-Algorithmus erzielt werden kann.

2.3 Strassen-Algorithmus

Strassen [90] entwickelte einen Algorithmus, der das Matrizenprodukt zweier quadratischer Matrizen $\mathcal{A}$ und $\mathcal{B}$ vom Typ (n, n) mit $T(n) \in O(n^{2,81})$ löst. Ausgangspunkt der Betrachtungen sei das Produkt zweier Matrizen $\mathcal{A}$ und $\mathcal{B}$, beide vom Typ $(2, 2)$.

$$\mathcal{C} = \mathcal{AB} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \tag{2.17}$$

Werden die Elemente $c_{ij}, (i, j = 1, 2)$ nach der "naiven" Methode bestimmt, so gilt:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned} \tag{2.18}$$

Wie schon bekannt, erfordert die Berechnung des $(2, 2)$ Matrizenproduktes 8 Multiplikationen und 4 Additionen. Der Algorithmus von Strassen erfordert nun zunächst die folgenden Teilberechnungen:

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_3 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_4 &= (a_{21} + a_{22})b_{11} \\ m_5 &= (a_{11} + a_{12})b_{22} \end{aligned} \tag{2.19}$$

$$m_6 = a_{11}(b_{12} - b_{22})$$
$$m_7 = a_{22}(b_{21} - b_{11})$$

Mit Hilfe dieser Teilberechnungen können die Elemente c_{ij} des Matrizenproduktes wie folgt berechnet werden:

$$\begin{aligned} c_{11} &= m_1 + m_7 - m_5 + m_3 \\ c_{12} &= m_5 + m_6 \\ c_{21} &= m_4 + m_7 \\ c_{22} &= m_1 + m_6 - m_4 + m_2 \end{aligned} \tag{2.20}$$

Das Auszählen der Rechenoperationen ergibt 7 Multiplikationen und 18 Additionen. Es kann also eine Multiplikation gespart werden, dafür werden jedoch erheblich mehr Additionen ausgeführt. Erinnert sei an dieser Stelle, daß die Addition allerdings nur von der Komplexität $O(n^2)$ ist und keinen Einfluß auf den Exponenten der Matrizenmultiplikation hat.
Untersuchungen [37, 101], die im Zusammenhang mit dem Strassen-Algorithmus geführt wurden, haben gezeigt, daß auch andere Teilberechnungen dem Algorithmus von Strassen zugrunde liegen können. Als Beispiel sei hier aufgeführt:

$$\begin{aligned} r_1 &= (a_{12} + a_{21})(b_{11} + b_{22}) \\ r_2 &= (a_{11} + a_{21})(b_{12} - b_{11}) \\ r_3 &= (a_{12} + a_{22})(b_{21} - b_{22}) \\ r_4 &= (a_{11} - a_{12})b_{11} \\ r_5 &= (a_{22} - a_{21})b_{22} \\ r_6 &= -a_{21}(b_{12} + b_{22}) \\ r_7 &= -a_{12}(b_{11} + b_{21}) \end{aligned} \tag{2.21}$$

Die c_{ij} des Matrizenproduktes lassen sich dann wie folgt berechnen:

$$\begin{aligned} c_{11} &= r_4 - r_6 \\ c_{12} &= r_1 + r_2 + r_4 + r_6 \\ c_{21} &= r_1 + r_3 + r_5 + r_7 \\ c_{22} &= r_5 - r_6 \end{aligned} \tag{2.22}$$

Das Auszählen ergibt auch hier 7 Multiplikationen und 18 Additionen.

Es liegt nun nahe, das Vorgehen beim Strassen-Algorithmus zur Berechnung des Produktes von Matrizen vom Typ $(2,2)$ mit den Elementen $a_{ij}, b_{ij}, c_{ij}, (i,j=1,2)$ in der Darstellung 2.17 zu übertragen auf die Multiplikation von Matrizen höherer Ordnung, $n = m \cdot 2^{k+1}$. Für das Produkt kann dann geschrieben werden

$$\mathcal{C} = \mathcal{AB} = \begin{pmatrix} \mathcal{A}_{11} & \mathcal{A}_{12} \\ \mathcal{A}_{21} & \mathcal{A}_{22} \end{pmatrix} \begin{pmatrix} \mathcal{B}_{11} & \mathcal{B}_{12} \\ \mathcal{B}_{21} & \mathcal{B}_{22} \end{pmatrix} = \begin{pmatrix} \mathcal{C}_{11} & \mathcal{C}_{12} \\ \mathcal{C}_{21} & \mathcal{C}_{22} \end{pmatrix} \tag{2.23}$$

wobei $\mathcal{A}_{ij}$, $\mathcal{B}_{ij}$ und $\mathcal{C}_{ij}$ Matrizen der Ordnung $\frac{n}{2} = m2^k$ sind. Dies ist möglich, weil in der Darstellung des Strassen-Algorithmus für Matrizen vom Typ $(2,2)$ bei keiner Multiplikation das Gesetz der Kommutativität der Multiplikation zur Anwendung kam. Es sei erinnert, das Matrizenprodukt ist nicht kommutativ, d.h. im allgemeinen sind die beiden Produktmatrizen $\mathcal{AB}$ und $\mathcal{BA}$ verschieden, von bestimmten Ausnahmen sogenannter vertauschbarer Matrizen $\mathcal{A}, \mathcal{B}$ abgesehen.

Der Strassen-Algorithmus für $(2,2)$ Matrizen kann also auf das Problem 2.23 übertragen werden, indem die Elemente a_{ij}, b_{ij}, c_{ij} durch die Elementematrizen $\mathcal{A}_{ij}, \mathcal{B}_{ij}, \mathcal{C}_{ij}$ zu ersetzen sind. Damit ergeben sich für den so verallgemeinerten Strassen-Algorithmus die folgenden Teilberechnungen:

$$\begin{aligned} M_1 &= (\mathcal{A}_{11} + \mathcal{A}_{22})(\mathcal{B}_{11} + \mathcal{B}_{22}) \\ M_2 &= (\mathcal{A}_{21} - \mathcal{A}_{11})(\mathcal{B}_{11} + \mathcal{B}_{12}) \\ M_3 &= (\mathcal{A}_{12} - \mathcal{A}_{22})(\mathcal{B}_{21} + \mathcal{B}_{22}) \\ M_4 &= (\mathcal{A}_{21} + \mathcal{A}_{22})\mathcal{B}_{11} \\ M_5 &= (\mathcal{A}_{11} + \mathcal{A}_{12})\mathcal{B}_{22} \\ M_6 &= \mathcal{A}_{11}(\mathcal{B}_{12} - \mathcal{B}_{22}) \\ M_7 &= \mathcal{A}_{22}(\mathcal{B}_{21} - \mathcal{B}_{11}). \end{aligned} \tag{2.24}$$

Mit Hilfe dieser Matrizen sind die Untermatrizen $\mathcal{C}_{ik}$ des Matrizenproduktes wie folgt zu berechnen:

$$\begin{aligned} \mathcal{C}_{11} &= M_1 + M_7 - M_5 + M_3 \\ \mathcal{C}_{12} &= M_5 + M_6 \\ \mathcal{C}_{21} &= M_4 + M_7 \\ \mathcal{C}_{22} &= M_1 + M_6 - M_4 + M_2. \end{aligned} \tag{2.25}$$

Der Algorithmus läßt sich nun rekursiv auf die Untermatrizen $\mathcal{A}_{ij}, \mathcal{B}_{ij}, \mathcal{C}_{ij}$ anwenden, bis schließlich nur noch quadratische Matrizen der Ordnung m zu multipli-

zieren sind. Durch Induktion von k läßt sich zeigen, daß für die Multiplikation zweier quadratischer Matrizen $\mathcal{A}$ und $\mathcal{B}$ der Ordnung $n = m2^k$ eine Anzahl von $m^3 7^k$ Multiplikationen und $(5+m)m^2 7^k - 6(m2^k)^2$ Additionen benötigt werden. Hieraus ergibt sich für die Zeitkomplexität

$$T(n) = T(m2^k) = (5+2m)m^2 7^k - 6(m2^k)^2. \tag{2.26}$$

Für den Fall $n = m2^k$ mit $m = 1$ folgt

$$T(n) = 7 \cdot 7^{log_2 n} - 6n^2 \in O(n^{2.81}). \tag{2.27}$$

Im Vergleich zur "naiven" Methode ein erheblicher Zeitgewinn. Wenn nun für kleine Matrizen die "naive" Methode und für große Matrizen Strassen effizienter ist, stellt sich die Frage: Ab welcher Matrixgröße greift der Strassen-Algorithmus. Für die Zeitkomplexitäten gilt zunächst:

$$\begin{aligned} T(n)^{naiv} &= 4^k(2 \cdot 2^k - 1) \\ T(n)^{Strassen} &= 7 \cdot 7^k - 6 \cdot 4^k \end{aligned} \tag{2.28}$$

Durch Lösung der Gleichung $T(n)^{naiv} = T(n)^{Strassen}$ ergibt sich ein Schnittpunkt für beide Funktionen aus Formel 2.28

$$\begin{aligned} \tfrac{7}{8}^k &\leq \tfrac{2}{7} \\ k &\leq \tfrac{ln\frac{2}{7}}{ln\frac{7}{8}} \sim 9.38. \end{aligned} \tag{2.29}$$

Ein theoretisch zu erwartende Gewinn durch den Strassen-Algorithmus ist für quadratische Matrizen vom Typ größer $(1024, 1024)$ zu erwarten. Tabelle 2.1 faßt berechnete Zeitkomplexitäten für beide betrachteten Algorithmen zusammen.

k	$T(n)^{naiv}$	$T(n)^{Strassen}$
8	$3.3488896 \cdot 10^7$	$3.9960391 \cdot 10^7$
9	$2.68173312 \cdot 10^8$	$2.80902385 \cdot 10^8$
10	$2.146435072 \cdot 10^9$	$1.971035287 \cdot 10^9$
11	$1.717567488 \cdot 10^{10}$	$1.3816122377 \cdot 10^{10}$

Tabelle 2.1: Strassen und "naiver" Algorithmus, Vergleich der Zeitkomplexitäten

Abb. 2.1 illustriert anschaulich den Kurvenverlauf beider Funktionen. Deutlich ist der Schnittpunkt zwischen "naivem" Algorithmus und dem approximierten

Verlauf (gestrichelte Linie) des Strassen-Algorithmus für ein $k \sim 9.38$ zu erkennen.

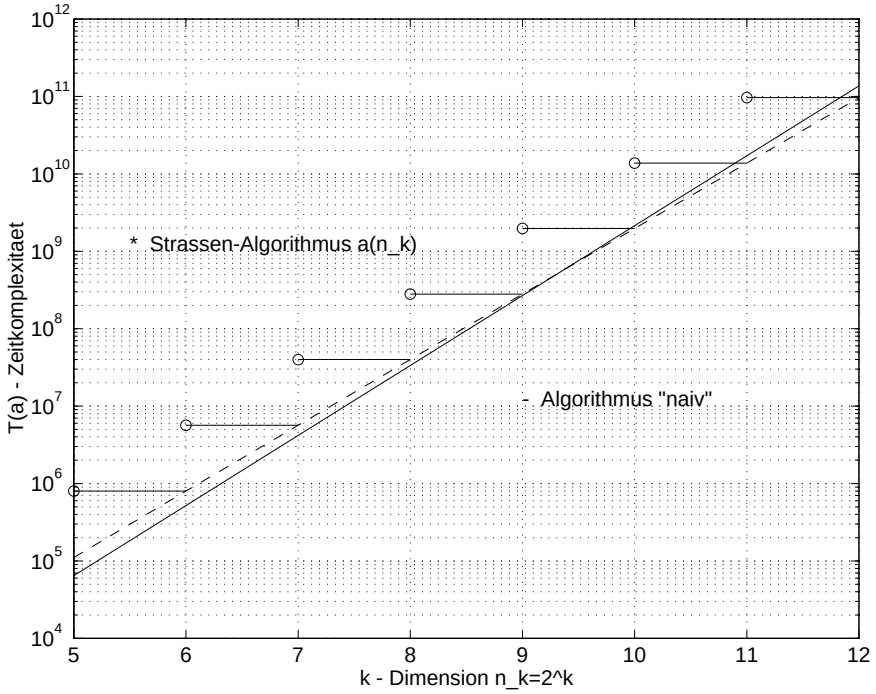


Abbildung 2.1: Zeitkomplexität in Abhängigkeit von der Matrixgröße für den Strassen und "naiven" Algorithmus

Der Strassen Algorithmus läßt sich insbesondere rekursiv effizient implementieren, wenn auch ein Laufzeitgewinn erst für sehr große Matrizen zu erwarten ist. Er stellt aber für derartige Anwendungen eine echte Alternative zum "naiven" Algorithmus dar und fand Einklang in eine Vielzahl von Implementierungen und Bibliotheken [102, 7].

2.4 Bilineare Algorithmen

In [30] werden Methoden zur Ableitung unterer Schranken für arithmetische und algebraische Probleme vorgestellt. In [45] und [97] wird gezeigt, daß die 7 Multiplikationen für das Produkt zweier Matrizen vom Typ $(2, 2)$, wie beim Strassen-Algorithmus, minimal ist. Jede weitere Verbesserung kann also nur durch die

Betrachtung von Matrizen vom Typ $(3,3)$, $(4,4)$ oder größer zustande kommen und/oder durch einen neuen Algorithmus. Bilineare Algorithmen sind auf die Klassen von Problemen nutzbringend anwendbar, wenn eine Menge von Bilinearformen für diese Probleme entwickelt werden kann [1, 70], d. h. eine Anwendung von bilinearen Algorithmen auf die Matrizenmultiplikation [69] erscheint als sinnvoll.

Definition 2.5 *Das Matrizenprodukt zweier Matrizen $\mathcal{A} = (a_{ij})$ vom Typ (m,n) und $\mathcal{B} = (b_{jk})$ vom Typ (n,p) berechnet sich wie folgt. Zunächst werden Linearkombinationen der a- und b-Variablen entwickelt*

$$l_q = \sum_{ij} e'(i,j,q)a_{ij}, \qquad l'_q = \sum_{jk} e''(j,k,q)b_{jk}. \tag{2.30}$$

Hieraus werden die Produkte $P_q = l_q l'_q$ mit $q = 0 \ldots (M-1)$ berechnet. Die Skalarprodukte c_{ik} der Ergebnismatrix ergeben sich

$$c_{ik} = \sum_{j} a_{ij}b_{jk} = \sum_{q=0}^{M-1} e'''(k,i,q)l_q l'_q. \tag{2.31}$$

Die Koeffizienten e', e'', e''' sind Konstanten aus $e \in \{-1,0,1\}$.

M, die Gesamtanzahl der Multiplikationen von l_q und l'_q, wird auch als Rang des bilinearen Algorithmus bezeichnet. An dieser Stelle sei angemerkt, daß der "naive" Algorithmus für Matrizen vom Typ (n,n) bilinear und vom Rang n^3 ist. Strassen's Algorithmus für Matrizen vom Typ $(2,2)$ ist ebenfalls bilinear und vom Rang 7.
In [101] ist ein bilinearer Algorithmus für die Multiplikation von Matrizen vom Typ $(3,3)$ entwickelt worden.

$$\begin{aligned}
m_1 &= l_0 l'_0 = (a_{12} + a_{13} - a_{33})(b_{21} - b_{31} + b_{33}) \\
m_2 &= l_1 l'_1 = (-a_{11} + a_{21} + a_{22})(b_{11} - b_{12} + b_{22}) \\
m_3 &= l_2 l'_2 = (-a_{21} + a_{31} + a_{32})(b_{12} - b_{13} + b_{23}) \\
m_4 &= l_3 l'_3 = (a_{12} + a_{13})(b_{31} - b_{33}) \\
m_5 &= l_4 l'_4 = (a_{21} + a_{22})(b_{12} - b_{11}) \\
m_6 &= l_5 l'_5 = (a_{31} + a_{32})(b_{13} - b_{12}) \\
m_7 &= l_6 l'_6 = (a_{12} - a_{33})(b_{31} - b_{21}) \\
m_8 &= l_7 l'_7 = (a_{11} - a_{21})(b_{22} - b_{12}) \\
m_9 &= l_8 l'_8 = (a_{21} - a_{31})(b_{23} - b_{13})
\end{aligned}$$

$$
\begin{aligned}
m_{10} &= l_9 l'_9 = a_{11} b_{11} \\
m_{11} &= l_{10} l'_{10} = a_{21} b_{21} \\
m_{12} &= l_{11} l'_{11} = a_{31} b_{32} \\
m_{13} &= l_{12} l'_{12} = a_{33} b_{33} \\
m_{14} &= l_{13} l'_{13} = (-a_{12} - a_{13} + a_{32} + a_{33}) b_{21} \\
m_{15} &= l_{14} l'_{14} = (a_{11} + a_{12} - a_{21} - a_{22}) b_{22} \\
m_{16} &= l_{15} l'_{15} = (a_{21} + a_{22} - a_{31} - a_{32}) b_{23} \\
m_{17} &= l_{16} l'_{16} = a_{12} (-b_{21} + b_{31} + b_{23} - b_{33}) \\
m_{18} &= l_{17} l'_{17} = a_{22} (-b_{11} + b_{21} + b_{12} - b_{22}) \\
m_{19} &= l_{18} l'_{18} = a_{32} (-b_{12} + b_{22} + b_{13} - b_{23}) \\
m_{20} &= l_{19} l'_{19} = a_{11} b_{13} \\
m_{21} &= l_{20} l'_{20} = a_{13} b_{32} \\
m_{22} &= l_{21} l'_{21} = a_{23} b_{31} \\
m_{23} &= l_{22} l'_{22} = a_{23} b_{33} \\
m_{24} &= l_{23} l'_{23} = a_{31} b_{11} \\
m_{25} &= l_{24} l'_{24} = a_{33} b_{32}
\end{aligned}
\tag{2.32}
$$

$$
\begin{aligned}
c_{11} &= m_1 + m_4 + m_7 + m_{10} + m_{13} \\
c_{12} &= m_2 + m_5 + m_{10} + m_{15} + m_{21} \\
c_{13} &= m_1 + m_7 + m_{13} + m_{17} + m_{20} \\
c_{21} &= m_2 + m_8 + m_{10} + m_{18} + m_{22} \\
c_{22} &= m_2 + m_5 + m_8 + m_{10} + m_{12} \\
c_{23} &= m_3 + m_6 + m_{11} + m_{16} + m_{23} \\
c_{31} &= m_1 + m_4 + m_{13} + m_{14} + m_{22} \\
c_{32} &= m_3 + m_9 + m_{11} + m_{19} + m_{25} \\
c_{33} &= m_3 + m_6 + m_9 + m_{11} + m_{13}
\end{aligned}
\tag{2.33}
$$

Durch Auszählen ergeben sich 25 Multiplikationen und 87 Additionen. Ähnlich dem Vorgehen im Abschnitt 2.3 liegt es nun nahe, den Algorithmus zur Berechnung des Produktes von Matrizen vom Typ $(3, 3)$ in der Darstellung 2.32, 2.33 zu übertragen auf die Multiplikation von Matrizen höherer Ordnung $n_k = 3^k$.

Der Algorithmus $a(n_k)$ benötigt also 25 Multiplikationen von Matrizen vom Typ (n_{k-1}, n_{k-1}) mit $n_{k-1} = 3^{k-1}$. Also gilt für die Anzahl der Multiplikationen

$$M(a(n_k)) = 25 \cdot M(a(n_{k-1})) = 25^k. \tag{2.34}$$

Demzufolge erfordert der Algorithmus auch 87 Additionen von Matrizen vom Typ (n_{k-1}, n_{k-1}). Darüber hinaus erfordern jedoch die 25 Multiplikationen von Matrizen ihrerseits jeweils $S(a(n_{k-1}))$ Additionen. Es gilt:

$$S(a(n_k)) = 87 \cdot n_{k-1}^2 + 25 \cdot S(a(n_{k-1})). \tag{2.35}$$

Diese rekursive Beziehung besitzt die Lösung

$$S(a(n_k)) = 87 \cdot \sum_{i=0}^{k-1} 9^i \cdot 25^{k-1-i}. \tag{2.36}$$

Aus $M(a(n_k))$ und $S(a(n_k))$ ergibt sich die Zeitkomplexität als Gesamtzahl arithmetischer Operationen

$$T(a(n_k)) = 25^k + 87 \cdot \sum_{i=0}^{k-1} 9^i \cdot 25^{k-1-i}. \tag{2.37}$$

Es gilt nun unter Zuhilfenahme von $\sum_{s=0}^{\infty} r \cdot q^s = \frac{r}{1-q}$

$$\lim_{k\to\infty} \frac{S(a(n_k))}{M(b(n_k))} = \lim_{k\to\infty} \frac{87 \cdot \sum_{i=0}^{k-1} 9^i \cdot 25^{k-1-i}}{25^k} = 6.4375 \tag{2.38}$$

woraus für die Zeitkomplexität folgt

$$T(a(n_k)) = 6.4375 \cdot n^{2.929} \in O(n^{2.929}). \tag{2.39}$$

Bemerkung: Die Bedeutung bilinearer Algorithmen für die Matrizenmultiplikation ist zurückzuführen auf Theorem 2.1 in [70].

Theorem 2.1 *Ein gegebener bilinearer Algorithmus in der Darstellung von 2.30 und 2.31 für m, n, p mit $m, n, p > 1$ und nur M Multiplikationen besitzt einen Koeffizienten der Matrizenmultiplikation von*

$$\omega \leq \frac{3 \log M}{\log(mnp)}. \tag{2.40}$$

Unter der Voraussetzung quadratischer Matrizen mit $n = 3^k$ und bei Strassen ähnlicher rekursiver Konstruktion mit $M = 25^k$ Multiplikationen ergibt sich für den Exponenten Matrizenmultiplikation: $\omega = \frac{3\log(25^k)}{\log(3^{3\cdot k})} = 2.929$.

Eine sorgfältige Betrachtung der Formeln 2.32 und 2.33 verrät, daß 30 additive Operationen für die linken und rechten Seiten der Produkte m_i und 28 additive Operationen zur Berechnung der c_{ij} genügen. Ein Algorithmus $b(n_k)$ benötigt neben 25 Multiplikationen nur noch 58 Additionen. Für die Multplikation von Matrizen höherer Ordnung $n_k = 3^k$ ergibt sich eine Zeitkomplexität von

$$T(b(n_k)) = 25^k + 58 \cdot \sum_{i=0}^{k-1} 9^i \cdot 25^{k-1-i}. \tag{2.41}$$

Eine Grenzwertbetrachtung zeigt den Zeitgewinn der durch Verminderung der Anzahl der Additionen erzielt wird.

$$\lim_{k\to\infty} \frac{T(a(n_k))}{T(b(n_k))} = \frac{25^k + 87 \cdot \sum_{i=0}^{k-1} 9^i \cdot 25^{k-1-i}}{25^k + 58 \cdot \sum_{i=0}^{k-1} 9^i \cdot 25^{k-1-i}} \tag{2.42}$$

und mit $\sum_{s=0}^{\infty} r \cdot q^s = \frac{r}{1-q}$ folgt

$$\lim_{k\to\infty} \frac{T(a(n_k))}{T(b(n_k))} = 1.39\overline{189}. \tag{2.43}$$

Die Gesamtzahl der Zeitschritte konnte um 28.1% gesenkt werden, d. h.

$$T(b(n_k)) = 4.625 \cdot n^{2.929} \in O(n^{2.929}). \tag{2.44}$$

Die Ergebnisse der Berechnungen des Abschnitts sind in der Abb. 2.4 dargestellt. In [36] wird gezeigt, daß der Strassen-Algorithmus eine Repräsentation des Produktes von $(2, 2)$ Matrizen durch das Hadamard-Produkt in einem 7-dimensionalen Raum ist. Weiterhin wird gezeigt, daß es für Matrizen vom Typ (n, n) möglich ist, eine Repräsentation in einem $n^3 - n + 1$ dimensionalen Raum zu erzielen. Gastinel benötigt neben den 25 Multiplikationen jedoch 76 Additionen. Da die Additionen keinen Einfluß auf den Exponenten ω haben, sind beide Algorithmen von der selben Zeitkomplexität

$$T(n) \in O(n^{2,929}). \tag{2.45}$$

Um eine weitere Beschleunigung gegenüber dem Strassen-Algorithmus zu erzielen, ist aus der großen Menge der möglichen bilinearer Algorithmen 2.30, 2.31 für alle $\forall m, n, p, M$ einer zu finden mit

$$\frac{3 \log M}{\log(mnp)} < \log_2 7. \tag{2.46}$$

Das heißt der Rang eines bilinearen Algorithmus für die $(3, 3)$ Matrizenmultiplikation muß $M = 21$ betragen. Die bislang erreichte unterste Grenze für die Multiplikation zweier Matrizen vom Typ $(3, 3)$ beschreibt Laderman in [60] mit

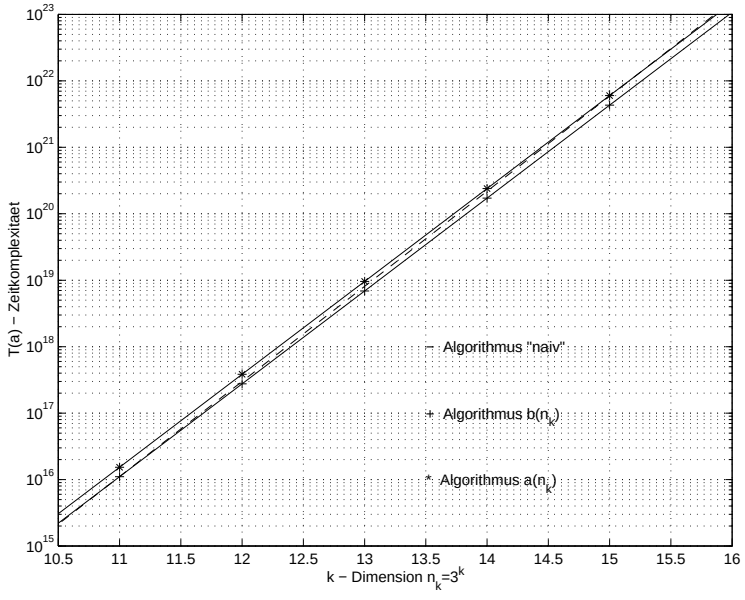


Abbildung 2.2: Vergleich der Zeitkomplexitäten in Abhänigkeit von der Matrixgröße für die Algorithmen $a(n_k)$, $b(n_k)$ und "naiv"

$M = 23$ Multiplikationen. Einen Fortschritt hat Pan [70] 1978 erreicht für die Multiplikation von quadratischen Matrizen vom Typ $(70, 70)$ mit $M = 143640$. Dieser Algorithmus führt zu einer Komplexität mit $\omega \leq \frac{\log 143640}{\log 70} \sim 2.795$. Aus Anwendersicht (nicht implementierbar) stellt er jedoch keine Alternative zu Strassen's Variante dar.

2.5 Kettenmultiplikation von Matrizen

Zum Abschluß dieses Kapitels sei ein Algorithmus entwickelt, der auf dem Prinzip der dynamischen Programmierung beruht [30]. Im Vordergrund steht dabei nicht die Multiplikation an sich, sondern die Minimierung des Rechenaufwandes, der für die Multiplikation einer Kette von Matrizen unterschiedlichen Typs erforderlich ist. Im Gegensatz zum Teile und Herrsche Prinzip, welches eine top down Methode ist, ist das dynamische Programmieren eine bottom up Methode.

Das heißt, um für ein Problem der Größe n eine Lösung zu finden, werden alle für das Problem relevanten Teilprobleme der Größe $1 \ldots (n-1)$ gelöst und die Ergebnisse zum Zweck der späteren Verwendung bei der Lösung größerer Probleme in einer Tabelle abgelegt. Für die Berechnung der nächstgrößeren Teillösung muß/kann nun auf die Tabelle zurückgegriffen werden. Eine optimal berechnete und in der Tabelle fixierte Lösung eines Teilproblems kann für verschiedene Berechnungen größerer Teilprobleme immer wiederverwendet werden, ohne neu berechnet werden zu müssen. Auf Einzelheiten dieser Technik wird hier verzichtet und auf entsprechende Literatur verwiesen [21, 93]. Dieser Ansatz ist im Operation Research zur Lösung von Optimierungsproblemen weit verbreitet. Auf zwei Probleme sei hingewiesen, die bei jeder Anwendung der dynamischen Programmierung auftreten können. Es ist zunächst nicht immer möglich, die Lösungen kleinerer Probleme so zu kombinieren, daß sich die Lösung eines größeren Problems ergibt. Zweitens kann es sich ergeben, daß die Anzahl der zu lösenden kleinen Probleme unvertretbar groß wird.

Für die Kettenmultiplikation von Matrizen ist die dynamische Programmierung sehr effizient einsetzbar. Gegeben sei eine "Kette" $[\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n]$ von n Matrizen die in angegebener Reihenfolge miteinander multipliziert werden sollen. Selbstverständlich muß die Anzahl der Spalten einer Matrix mit der Anzahl der Zeilen der folgenden Matrix übereinstimmen. Matrix $\mathcal{A}_1$ sei vom Typ (p_0, p_1), $\mathcal{A}_2$ vom Typ (p_1, p_2) usw., d. h. $\mathcal{A}_n$ ist vom Typ (p_{n-1}, p_n). In welcher Weise können Matrizen multipliziert werden, so daß die Zeitkomplexität minimal ist?

Beispiel 2.1 *illustriert die Multiplikation einer 3-er Matrixkette* $[\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3]$ *mit* $\mathcal{A}_1$ *sei vom Typ* $(30, 35)$, $\mathcal{A}_2$ *eine* $(35, 15)$ *und* $\mathcal{A}_3$ *eine* $(15, 5)$ *Matrix. Die Multiplikation ist assoziativ, so daß die beiden möglichen Klammerungen das selbe Ergebnis liefern,*

$$(\mathcal{A}_1 \cdot \mathcal{A}_2)\mathcal{A}_3 = \mathcal{A}_1(\mathcal{A}_2 \cdot \mathcal{A}_3). \tag{2.47}$$

Für die Lösung des Produktes $(\mathcal{A}_1 \cdot \mathcal{A}_2)\mathcal{A}_3$ werden 18000 Multiplikationen und die Lösung des Produktes $\mathcal{A}_1(\mathcal{A}_2 \cdot \mathcal{A}_3)$ dagegen nur 7875 benötigt, was einem Speedup von 2.29 entspricht.

Bei der Multiplikation von n Matrizen können durch günstige bzw. ungünstige Klammerungen weit extremere Unterschiede in der Komplexität auftreten. An dieser Stelle sei angemerkt, daß die Anzahl der Klammerungen von exponentieller Komplexität ist und einer gründlich erforschten kombinatorischen Funktion, der Catalanischen Zahl $c(n)$, entspricht. Die Anzahl der möglichen Klammerungen beträgt $p(k) = c(n-1)$ wobei

$$c(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right) \tag{2.48}$$

ist. Da n im Vergleich zur Anzahl der Multiplikationen sehr klein ist, lohnt es sich einigen Aufwand zu treiben, um eine "optimale" Lösung zu finden.

Es sei $m(i,j)$ die minimale Anzahl von Multiplikationen, so daß man bei optimaler Klammerung den Abschnitt $\mathcal{A}_i \ldots \mathcal{A}_j$ mit den Zeilen- und Spaltenanzahlen $(p_{i-1}, p_i, \ldots, p_j)$ berechnen kann. Angenommen $(\mathcal{A}_i \ldots \mathcal{A}_k) \cdot (\mathcal{A}_{k+1} \ldots \mathcal{A}_j)$ für ein $k \in i, \ldots, j-1$ sei eine optimale Klammerung von $\mathcal{A}_i \ldots \mathcal{A}_j$, dann gilt für dieses k auf Grund des Optimalitätsprinzips

$$m(i,j) = m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j. \tag{2.49}$$

Der Anteil von $p_{i-1} \cdot p_k \cdot p_j$ ergibt sich aus der Multiplikation der Matrix $(\mathcal{A}_i \ldots \mathcal{A}_k)$ vom Typ (p_{i-1}, p_k) mit der Matrix $(\mathcal{A}_{k+1} \ldots \mathcal{A}_j)$ vom Typ (p_k, p_j). Da stets eine Gruppe in zwei kleinere Gruppen zerlegt wird, sind die minimalen Kosten der Berechnung für die beiden Gruppen nicht stets neu zu berechnen, sondern können aus einer Tabelle entnommen werden. Zur Berechnung der Werte der Tabelle gilt nun folgende rekursive Definition:

$$m(i,j) = \begin{cases} 0, & i = j \\ \min_{i \le k < j}(m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j), & i < j. \end{cases} \tag{2.50}$$

Die Größe der Tabelle, $[1 \ldots n, 1 \ldots n]$, wird durch Formel 2.50 bestimmt, wobei nur ein Teil (Form einer Dreicksmatrix) verwendet wird, also für die Indizes (i,j) mit $1 \le i \le j \le n$. Der Algorithmus trägt alle entsprechenden Tabelleneinträge (i,j) in der Reihenfolge $j - i = 0, 1, \ldots, n-1$ ein.

```
n:=length[p]-1
FOR i:=1 TO n DO m[i,i]:=0
  FOR l:=2 TO n DO
    FOR i:=1 TO n-l+1 DO
      j:=i+l-1;
      m[i,j]:=maxint;
      FOR k:=i TO j-1 DO
        q:=m[i,k]+m[k+1,j]+p_i-1*p_k*p_j
        IF q<m[i,j] THEN m[i,j]:=q;
                         s[i,j]:=k
        END (* if *);
      END (* for *);
    END (* for *);
  END (* for *);
END (* for *);
```

Auf diese Weise berechnet das Programm die entsprechenden Kosten und erhält im Tabellenelement $m[1,n]$ die gesuchte minimale Anzahl von Multiplikationen für die Berechnung von $\mathcal{A}_1\mathcal{A}_2\ldots\mathcal{A}_n$. Die optimale Klammerung selbst wird durch den Algorithmus mittels $s[i,j]$ bestimmt, indem man sich merkt, durch welches k sich das jeweilige Minimum ergeben hat.

Beispiel 2.2 *sei abschließend zur Kettenmultiplikation von Matrizen betrachtet. Gegeben seien die Matrizen* $\mathcal{A}_1,\ldots,\mathcal{A}_6$. *Im einzelnen gilt:*

Matrix:	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	$\mathcal{A}_5$	$\mathcal{A}_6$
Typ:	$(30,35)$	$(35,15)$	$(15,5)$	$(5,10)$	$(10,20)$	$(20,25)$

Der Algorithmus baut nun folgende Tabellen in Abb. 2.3 auf.

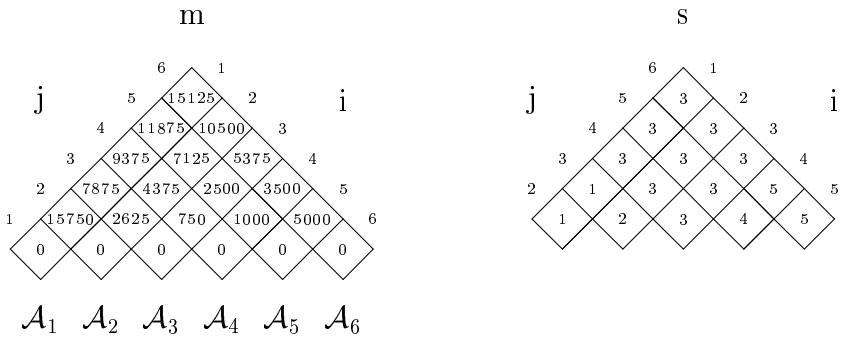


Abbildung 2.3: $m[i,j]$ bzw. $s[i,j]$ Tabellen der Kettenmultiplikation von Matrizen für $n=6$

Die optimale Klammerung ergibt bei diesem Beispiel $m[1,6]$ eine Gesamtzahl von 15125 Multiplikationen. Das folgende Programm stellt die Implementation des Prozesses der Ermittlung der optimalen Klammerung anhand der gespeicherten $m[i,j]$ und $s[i,j]$ Werte dar.

```
PROCEDURE order(i,j:integer);
  IF i=j THEN write(name(i)) ELSE
    write('(');
    order(i,s[i,j]-1);
    order(s[i,j],j);
    write(')');
  END (* if *);
```

Für das Beispiel ist die ermittelte Anordnung der Klammern

$$((\mathcal{A}_1(\mathcal{A}_2 \cdot \mathcal{A}_3))((\mathcal{A}_4 \cdot \mathcal{A}_5)\mathcal{A}_6)). \tag{2.51}$$

Für das weiter oben benannte Beispiel 2.1 läßt sich ebenfalls die Tabelle 2.3 nutzen und die optimale Klammerung $\mathcal{A}_1(\mathcal{A}_2\mathcal{A}_3)$ herauslesen.

Mit Hilfe der dynamischen Programmierung kann das Problem der Kettenmultiplikation von Matrizen in einer zu n^3 proportionalen Zeit und mit einem zu n^2 proportionalen Speicherbedarf gelöst werden. Dynamische Programmieralgorithmen können unter Umständen in ihrer Effizienz auf $O(n^2)$ verbessert werden, s. a. [93].

2.6 Geschichte

Pan [70] diskutiert die Frage: How fast can we multiply matrices? Der Exponent der Matrizenmultiplikation bewegt sich in den Grenzen $2 \leq \omega \leq 3$. Da n^2 unabhängige Elemente zu berechnen sind, ist die untere Grenze für sequentielle Algorithmen fest und von der Ordnung $\Omega(n^2)$. Die obere Grenze stand zunächst bei $O(n^3)$, denn der "naive" Algorithmus löst das Problem mit n^3 Multiplikationen und $n^3 - n^2$ Additionen. Initiert durch die Arbeiten von Strassen 1969 und Pan 1978, gab es um 1980 eine Vielzahl von Veröffentlichungen durch Schönhage, Romani, Bini, Pan, Winograd, Coppersmith etc. zur Frage Pan's nach dem asymptotisch schnellsten Algorithmus. Die obere Grenze hat sich in der Zeit ständig verringert und steht seit 1984 auf $n^{2.376}$ erzielt durch Coppersmith. Die Entwicklung des Exponenten der Matrizenmultiplikation zeigt Abb. 2.4.

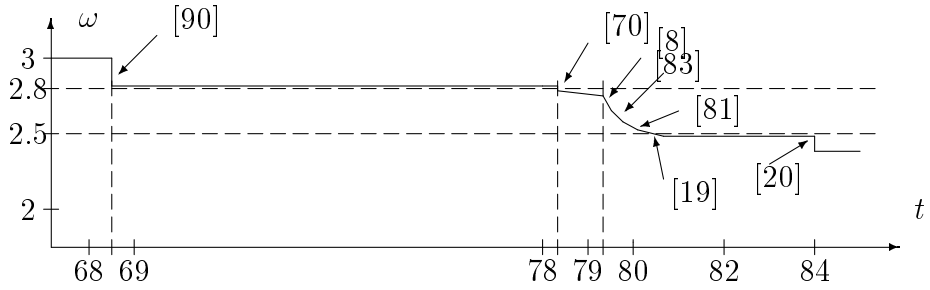


Abbildung 2.4: Entwicklung des Exponenten

An dieser Stelle mag man sich darüber wundern, daß an der Matrizenmultiplikation so großes Interesse besteht. Der Grund dafür besteht u. a. darin, daß andere typische Matrizenoperationen, wie z.B. die Inversion einer Matrix oder das Bestimmen einer Determinante unmittelbar mit der Matrizenmultiplikation verknüpft sind in dem Sinne, daß ein effizienter Algorithmus für eine dieser Operationen einen ähnlich effizienten für die anderen Operationen nach sich zieht.

Dennoch hat bis zuletzt der erzielte Forschritt, wegen der erheblichen Unkosten an asymptotischer Beschleunigung der Matrizenmultiplikation, keine Auswirkungen auf die praktische Berechnung von Matrixprodukten gezeigt. Die Auswirkungen auf die Theorie sind jedoch wesentlich. Das bloße Bestehen von asymptotisch schnellen Algorithmen für die Matrizenmultiplikation und für in Verbindung stehende Berechnungsprobleme ist anregend. Wichtiger ist, durch das Studium von Algorithmen der Matrizenmultiplikation einen tieferen Einblick in den Entwurf von leistungsfähigen arithmetischen Algorithmen zu gewinnen.

Kapitel 3

Parallele Matrizenmultiplikation

Das folgende Kapitel behandelt die parallele Matrizenmultiplikation. Der Schwerpunkt liegt nun nicht mehr allein auf der Multiplikation, sondern auf der gemeinsamen Betrachtung der Kategorien Architektur und Algorithmus.

3.1 Lineares Feld

Wie verhält es sich zunächst mit der Integermultiplikation auf einem linearen Feld? Die "Schulmethode" zerlegt den sequentiellen Algorithmus für die Multiplikation zweier n-bit Integerwerte als Summe von n Integerwerten, s. Abb. 3.1.

		1	1	1
		0	1	1
		1	1	1
	1	1	1	
0	0	0		
1	0	1	0	1

		b_3	b_2	b_1
		a_3	a_2	a_1
		a_1b_3	a_1b_2	a_1b_1
	a_2b_3	a_2b_2	a_2b_1	
a_3b_3	a_3b_2	a_3b_1		
y_5	y_4	y_3	y_2	y_1

Abbildung 3.1: "Schulmethode" zur Lösung der Multiplikation zweier Integerwerte, $7 \cdot 3 = 21$.

Formal können die Integerwerte a und b als zwei n-Vektoren $\vec{a} = [a_n, \ldots, a_1]$ und $\vec{b} = [b_n, \ldots, b_1]$ aufgefaßt werden. Das Produkt ergibt einen $(2n-1)$ Vektor $\vec{y} = [y_{2n-1}, \ldots, y_1]$. Verallgemeinert gilt auch:

$$(a_1+a_2x+\ldots+a_nx^{n-1})(b_1+b_2x+\ldots+b_nx^{n-1}) = y_1+y_2x+\ldots+y_{2n-1}x^{2n-2} \quad (3.1)$$

mit

$$y_k = \sum^{i+j=k+1} a_i b_j \qquad 1 \leq k \leq 2n-1. \quad (3.2)$$

An dieser Stelle muß erwähnt werden, daß nach Formel 3.1 $\vec{y}$ die Faltung [68] von $\vec{a}$ und $\vec{b}$ ist.

Bei der Berechnung der y_k's besteht das Problem in der Erzeugung eines Datenstromes, so daß sich die Elemente a_i, b_j mit dem selben Index $i+j-1$ zu unterschiedlichen Zeitpunkten in den Zellen des linearen Feldes zur Berechnung der y_k's treffen. Glücklicherweise gibt es einige Möglichkeiten einen solchen Datenstrom zu erzeugen. Den vielleicht einfachsten Algorithmus illustriert Abb. 3.2 am Beispiel $n = 3$.

Mit jedem Schritt bewegen sich die a_i's um einen Schritt nach rechts und die b_j's um einen Schritt nach links. Mit dem Schritt (1) wird a_1 in Zelle 1 und mit Schritt (2) das Element b_3 in Zelle $2n$ des linearen Feldes geschoben. Wie man leicht sieht, treffen sich die a_i und b_j in der Zelle 2n-i-j+2 während des Schrittes $2n+i-j$. Das letzte Paar a_nb_1 trifft sich in Zelle $n+1$ während des Schrittes $3n-1$. In jedem Schritt berechnet jede Zelle das Produkt der beiden Inputs und akkumuliert den Wert lokal. Die Berechnung der Faltung ist nach $3n-1$ Schritten beendet. Bei der Integermultiplikation werden die y_k's als individuelle Bits betrachtet und es müssen unter Umständen durch die Summation entstehende Überträge beachtet werden, s. a. Volladderfunktion [56]. Ein Übertrag wird generiert, wenn $a_i = 1$, $b_j = 1$ und der lokal gespeicherte Wert von y_{i+j-1} ebenfalls TRUE ist. Aus diesem Grund werden mit den b_i's noch c_i's, (c steht für carry) durch das lineare Feld geschoben, welche initial auf FALSE gesetzt sind und während der Laufzeit entsprechend geändert werden können. Der Algorithmus wird durch Abb. 3.3 für 3-bit Integerzahlen illustriert.

Die Schnappschüsse in der Abb. 3.3 repräsentieren den Zustand des Feldes jeweils nach dem aktuellen Schritt. Wenn der letzte Übertrag das Feld nach $4n-1$ Schritten verläßt ist die Multiplikation beendet. Die Zeitkomplexität der Integermultiplikation zweier n-bit Zahlen auf einem linearen Feld ist

$$T_p = 4n-1 \in O(n). \quad (3.3)$$

Der aufmerksame Leser wird erkennen, daß der Algorithmus nicht sehr effizient ist, zumal während jeden Schrittes nur die Hälfte der Prozessoren aktiv sind. In [65] sind einige Möglichkeiten aufgezeigt, die Effizienz zu erhöhen. Abschließend ist zu bemerken, daß der Algorithmus dennoch eine angemessene Effizienz im

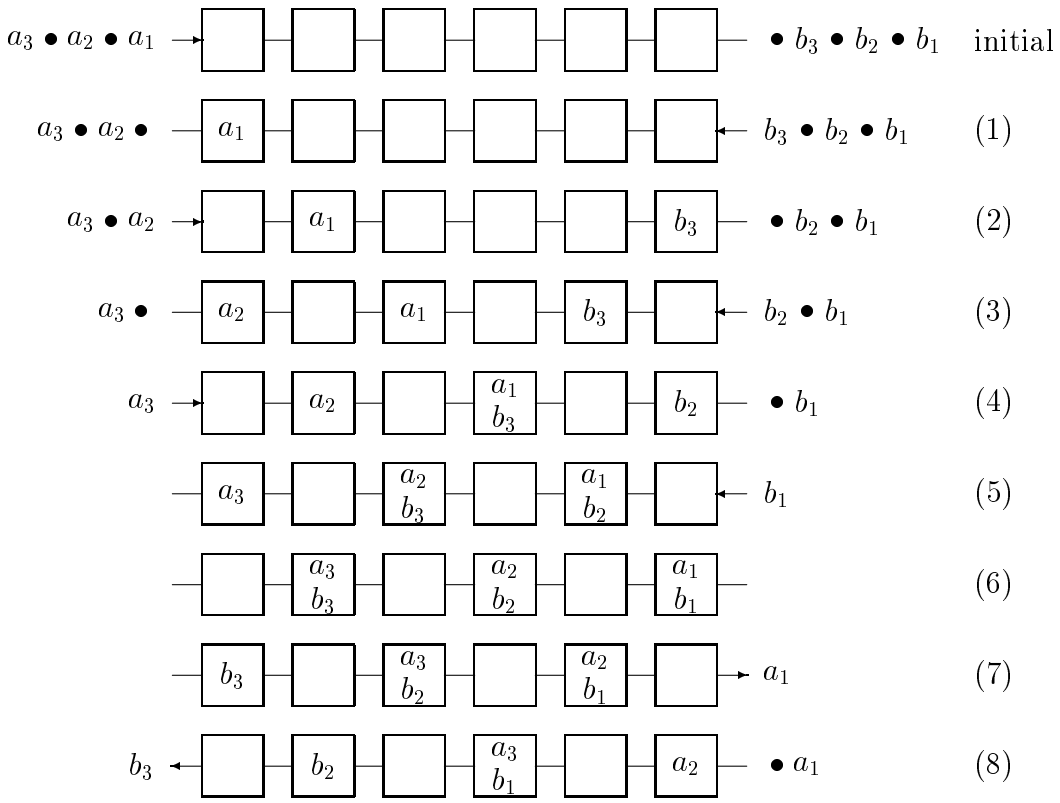


Abbildung 3.2: Datenfluß am Beispiel $n = 3$ in einem linearen Feld doppelter Länge, [65]

Vergleich zum sequentiellen Algorithmus mit $T \in O(n^2)$ Zeitschritten aufweist und somit einen Speedup von $S_p \in O(n)$ für $p = 2n$ Prozessoren erzielt.

3.2 Gitter

In diesem Abschnitt sei untersucht, ob sich die Verwendung einer Gitter-Architektur als lohnenswert erweist, um eine parallele Multiplikation von Matrizen auszuführen. Betrachtet sei zunächst die Matrix-Vektor-Multiplikation auf einem linearen Feld mit einer Matrix $\mathcal{A} = (a_{ij})$ vom Typ (n, n), einem n stelligen Vektor $\vec{x} = (x_j)$ und es sei zu berechnen das Matrix-Vektor-Produkt $\vec{y} = \mathcal{A}\vec{x}$ mit $\vec{y} = (y_i)$ und

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \tag{3.4}$$

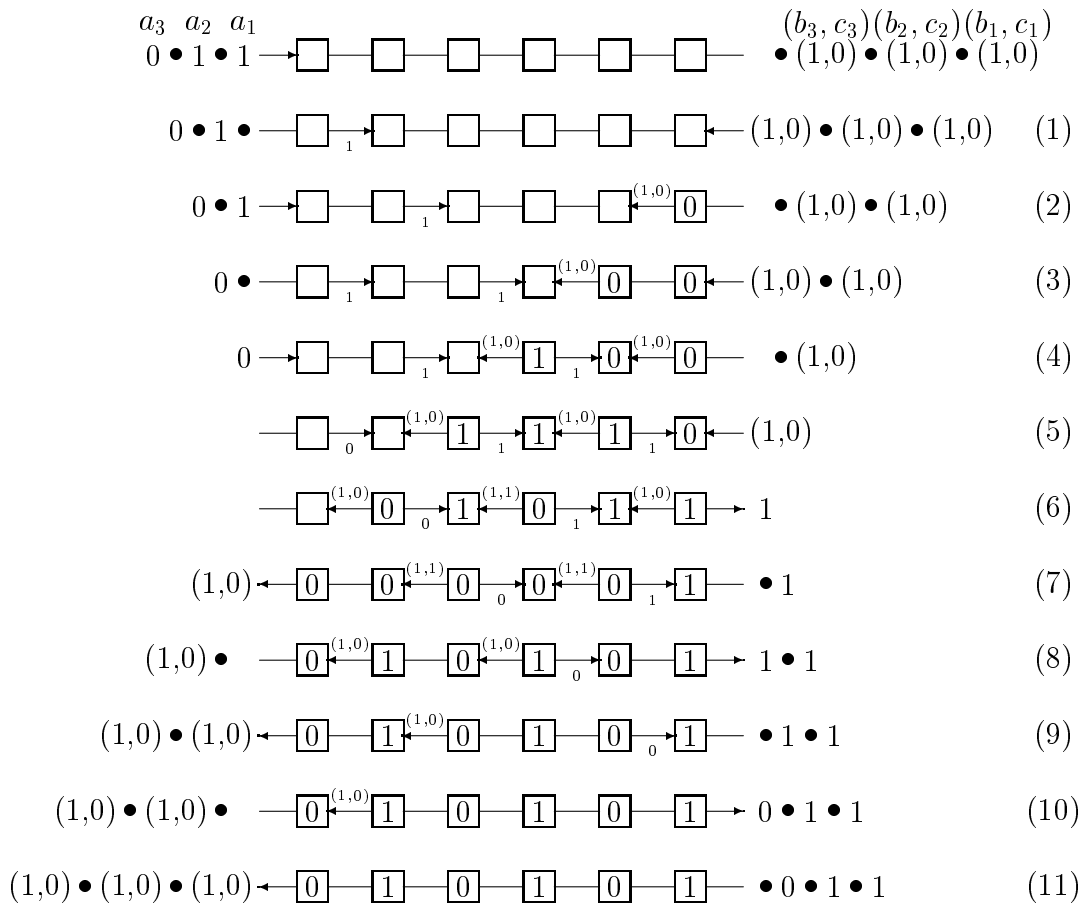


Abbildung 3.3: Multiplikation von $a = 3 = [011]$ und $b = 7 = [111]$, [65]

für alle $1 \geq i \geq n$. Die einfache sequentielle Methode benötigt n Multiplikationen und $n-1$ Additionen für jedes zu berechnende y_i, also insgesamt $2n^2 - n$ Schritte. Der Algorithmus für die Matrix-Vektor-Multiplikation auf einem linearen Feld ist ebenfalls einfach. Die x_i's werden schrittweise von links nach rechts, beginnend mit x_1, x_2, usw., durch das Feld geschoben. Die a_{ij}'s werden entsprechend Abb. 3.4 in das Feld eingelesen.

Die i-te Zelle des Feldes berechnet das Produkt aus linkem und oberen Eingabedatum und akkumuliert den Wert in seinem lokalen Speicher. x_j und a_{ij} erreichen Zelle i zur selben Zeit, im Schritt $i + j - 1$, so daß die exakte Berechnung von y_i erfolgen kann und nach $n + i - 1$ Schritten beendet ist. Also sind alle Variablen nach $2n - 1$ Schritten berechnet.

Der Algorithmus für die Matrix-Vektor-Multiplikation auf einem linearen Feld kann sehr einfach erweitert werden, um zwei Matrizen auf einem zweidimensio-

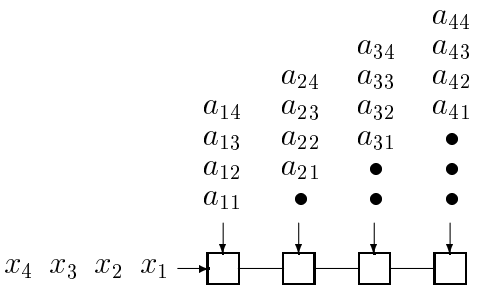


Abbildung 3.4: Berechnung des Matrix-Vektor-Produktes $\vec{y} = \mathcal{A}\vec{x}$ auf einem linearen Feld für $n = 4$

nalen Feld, einem Gitter, zu multiplizieren. Im einzelnen sind gegeben Matrix $\mathcal{A} = (a_{ij})$, Matrix $\mathcal{B} = (b_{ij})$ vom Typ (n, n) und die Produktmatrix $\mathcal{C} = \mathcal{AB}$ mit $\mathcal{C} = (c_{ij})$ und die c_{ij} können in $3n - 2$ Schritten gelöst werden. Der Algorithmus benötigt n^2 Prozessoren P_{ij}. Die Randprozessoren des (n, n) Gitters werden zur Eingabe der Matrizen $\mathcal{A}$, $\mathcal{B}$ benutzt. Die Eingabe erfolgt zeilen- und spaltenweise nach einem Scherungsschema (Skewing). Die Zeile i der Matrix $\mathcal{A}$ wird einen Schritt später als die Zeile $i - 1$, $2 \leq i \leq n$ eingelesen. Analog wird Spalte j der Matrix $\mathcal{B}$ einen Schritt später als Spalte $j-1$, $2 \leq j \leq n$ eingelesen, s.a. Abb. 3.5.

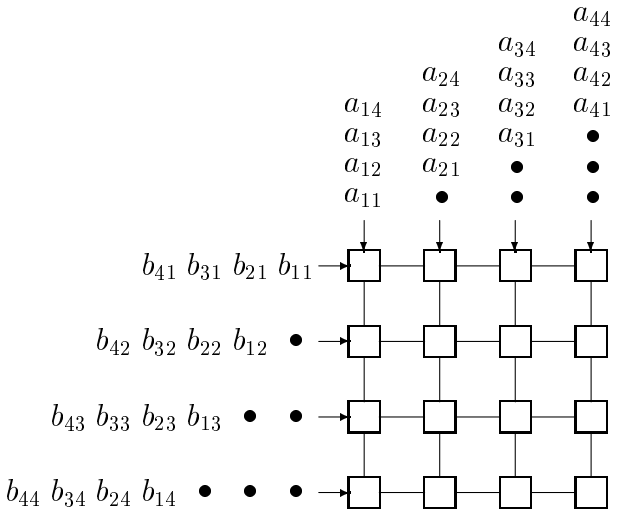


Abbildung 3.5: Berechnung des Matrix-Matrix-Produktes $\mathcal{AB}$ auf einem Gitter für $n = 4$

Diese versetzte Eingabe gewährleistet, daß die Matrixelemente a_{ik} und b_{kj} im

Zeittakt $i+j+k-2$ gleichzeitig den Prozessor P_{ij} erreichen. Abb. 3.6 friert den Datenfluß zum 5. Schritt ein.

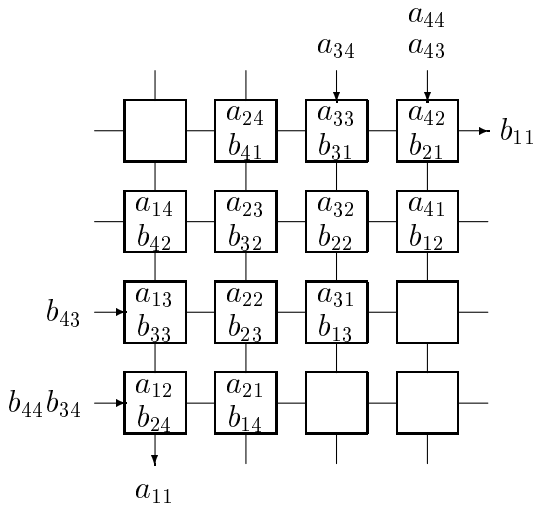


Abbildung 3.6: Schnappschuß des Datenflusses zum 5. Schritt

Wenn ein Prozessor P_{ij} die Eingaben a und b erhält, so bildet er

$$c_{ij} := c_{ij} + ab. \tag{3.5}$$

Gleichzeitig werden die Eingaben gespeichert. Im nächsten Schritt wird a horizontal zu $P_{i,j+1}$ und b wird vertikal zu $P_{i+1,j}$ weitergesendet. Am Ende des Algorithmus ist das Element c_{ij} der Produktmatrix $\mathcal{C}$ in Prozessor P_{ij} gespeichert.

```
FOR i=1 TO n DO IN PARALLEL
  FOR j=1 TO n DO IN PARALLEL
    c(i,j):=0;
    WHILE receive(a,b) DO
      c(i,j):=c(i,j)+(a*b)
      IF j<n THEN
       send(a,P(i,j+1);
      END (* if *);
      IF i<n THEN
       send(b,P(i+1,j);
      END (* if *);
    END (* while *);
  END (* for *);
END (* for *);
```

Die Matrixelemente a_{n1} und b_{1n} benötigen $n+n+n-2$ Schritte, bis sie P_{nn} erreichen. Da P_{nn} der letzte tätige Prozessor ist, beträgt die Laufzeit des Algorithmus

$$T(n) = 3n - 2 \in O(n). \tag{3.6}$$

Da der Algorithmus mit n^2 Prozessoren arbeitet, werden die gleichen Kosten erzielt wie mit der CRCW-PRAM

$$C(n) \in O(n^2) \cdot O(n) = O(n^3). \tag{3.7}$$

Der spezifizierte Algorithmus steht stellvertretend für eine ganze Reihe systolischer Matrixalgorithmen wie beispielsweise die systolische Lösung von Gleichungssystemen [55].

3.3 Torus

Durch sogenanntes Pipelining des Netzwerkes ist es möglich, die Effizienz von Algorithmen weiter zu verbessern. Als Beispiel sei zunächst wieder die Matrix-Vektor-Multiplikation über einem Ring betrachtet, Abb. 3.7.

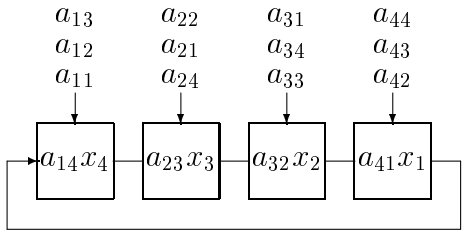


Abbildung 3.7: Matrix-Vektor-Multiplikation auf einem 4-Prozessor-Ring

Initial ist die Variable x_j im Prozessor P_{n-j+1} für $1 \leq j \leq n$ gespeichert und die Variable a_{ij} erreicht Prozessor P_i nach $i+j$ bzw. $i+j-n$ Schritten, jenachdem welche der Formeln einen Wert zwischen 1 und n liefert. y_i läßt sich nun im Prozessor P_i für $(1 \leq i \leq n)$ in n Schritten berechnen, wenn nach jedem Multiplikation/Addition's Schritt der Vektor $\vec{x}$ rechtswärts

geroutet wird. Ein ähnlicher Zugang kann genutzt werden, um 2 Matrizen auf einem (n,n)-Torus zu multiplizieren.

Es sei erinnert, daß die Multiplikation zweier Matrizen $\mathcal{A}$ vom Typ (n,m) und $\mathcal{B}$ vom Typ (m,l) nach der "naiven" Methode die Berechnung von $n \cdot l$ Skalarprodukten zweier m Vektoren erfordert. Sind $p = n \cdot l$ Prozessoren verfügbar, dann läßt sich die Matrix $\mathcal{C}$ in einer einzigen seriellen Schleife multiplizieren.

```
  FOR k=1 TO m DO IN PARALLEL
    c(i,j)=c(i,j) + a(i,k) * b(k,j);
(* parallel in allen n,l Prozessorelementen *)
  END (* for *);
```

Bezogen auf einen festen Index (i, j) schiebt der Schleifenindex k durch Inkrementierung um 1 nacheinander alle Spalten von $\mathcal{A}$ und alle Zeilen von $\mathcal{B}$ über den Index. Hieraus ergibt sich für $p = n \cdot l$ eine Zeitkomplexität

$$T_p(n) = 2m \in O(m). \tag{3.8}$$

Im Folgenden wird die Matrizenmultiplikation auf einer Torus-Architektur betrachtet. Dieser Algorithmus geht auf Cannon [14] zurück, der für einen bestimmten Arrayrechner mit festem Anwendungsspektrum entwickelt wurde. Verfügbar seien p^2 Prozessoren, die in Form eines Torus, s. Abb. 3.8, vernetzt sind.

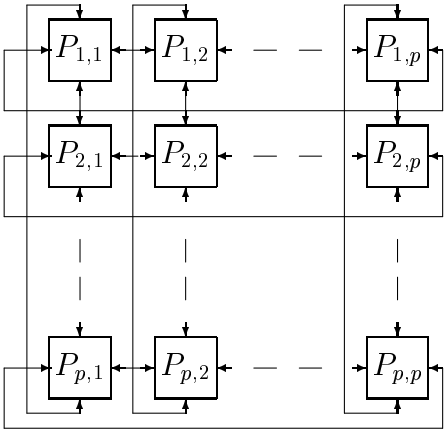


Abbildung 3.8: p^2 Torus-Architektur

Diese Kommunikationsstruktur ermöglicht ein zyklisches Schieben von Daten in horizontaler und vertikaler Richtung. Der Einfachheit halber werden nur quadratische Matrizen vom Typ (n, n), mit n als ganzzahligem Vielfachen von p, multipliziert. Die Matrizen $\mathcal{A}$, $\mathcal{B}$ und $\mathcal{C}$ können nun als p^2 Untermatrizen $\mathcal{A}_{ij}$, $\mathcal{B}_{ij}$, $\mathcal{C}_{ij}$ vom Typ (o, o), wobei gilt $o = \frac{n}{p}$, aufgefaßt werden. Ausgangspunkt des Algorithmus ist die Initialisierung jedes Prozessors P_{ij} mit den Untermatrizen $\mathcal{A}_{ij}$ und $\mathcal{B}_{ij}$. Für die Matrizenmultiplikation ergibt sich mit $i, j = 1 \ldots p$

$$\mathcal{C}_{ij} = \sum_{k=1}^{p} \mathcal{A}_{ik}\mathcal{B}_{kj} + \mathcal{C}_{ij}. \tag{3.9}$$

Richtet man die Aufmerksamkeit zunächst nur auf Prozessor P_{11}, so ist zu berechnen

$$\mathcal{C}_{11} = \mathcal{A}_{11}\mathcal{B}_{11} + \mathcal{A}_{12}\mathcal{B}_{21} + \ldots + \mathcal{A}_{1p}\mathcal{B}_{p1} + \mathcal{C}_{11}. \tag{3.10}$$

Da P_{11} mit den Untermatrizen $\mathcal{A}_{11}$, $\mathcal{B}_{11}$ initialisiert ist, kann sofort das Produkt berechnet und das Ergebnis in $\mathcal{C}_{11}$ akkumuliert werden. Da die $\mathcal{A}$ Untermatrizen nach links und die $\mathcal{B}$ Untermatrizen nach oben geschoben werden, kann nach einer Rotation $\mathcal{A}_{12}\mathcal{B}_{21}$ berechnet und nach $p-1$ Rotationen die Berechnung von $\mathcal{C}_{11}$ abgeschlossen werden. Mit dieser Initialisierung der Untermatrizen zeigen die Prozessoren in der Diagonalen ein ähnlich angenehmes Verhalten, so daß ebenfalls nach $p-1$ Rotationen alle Berechnungen abgeschlossen sind. Nun sei noch Prozessor P_{12} betrachtet, der

$$\mathcal{C}_{12} = \mathcal{A}_{11}\mathcal{B}_{12} + \mathcal{A}_{12}\mathcal{B}_{22} + \ldots + \mathcal{A}_{1p}\mathcal{B}_{p2} + \mathcal{C}_{12} \tag{3.11}$$

zu lösen hat. Es ist leicht zu erkennen, daß erst nach einer Rotation eine Berechnung möglich ist. Dieses Problem tritt bei allen Prozessoren auf, die nicht auf der Diagonalen liegen und kann durch eine Initialrotation korrigiert werden. Für die Matrizenmultiplikation ergibt sich nun folgender Algorithmus:

1. Initialisierung
   Initialisieren der Prozessorelemente P_{ij} mit den zugehörigen Untermatrizen $\mathcal{A}_{ij}$, $\mathcal{B}_{ij}$.

2. Initialrotation
   Verschiebe die Untermatrizen $\mathcal{A}_{ij}$ um i Plätze nach links und die Untermatrizen $\mathcal{B}_{ij}$ um j Plätze nach oben.

3. Schritt 1
   Berechne das Produkt der neuen Untermatrizen $\mathcal{A}$, $\mathcal{B}$ vom Typ (o, o) und akkumuliere in $\mathcal{C}$.

4. Schritt $k = 2 \ldots p$

   Verschiebe Untermatrix $\mathcal{A}$ eine Stelle nach rechts. Verschiebe Untermatrix $\mathcal{B}$ eine Stelle nach unten. Berechne das Produkt der neuen Untermatrizen $\mathcal{A}$, $\mathcal{B}$ vom Typ (o, o) und akkumuliere in $\mathcal{C}$.

Alle p Schritte laufen synchron und parallel auf allen P_{ij} ab. Vernachlässigt man die Operationen Verschieben und zählt wie bisher nur Additionen und Multiplikationen, so ergibt sich eine Zeitkomplexität

$$T_p(n) = (2o^3 - o^2)p + p. \tag{3.12}$$

Im schlechtesten Fall ist nur ein Prozessor verfügbar mit

$$T_1(n) = 2n^3 - n^2 \in O(n^3) \tag{3.13}$$

und im besten Fall $p = n$ mit

$$T_p(n) = 2n \in O(n). \tag{3.14}$$

Algorithmen, deren Funktionsweise auf rythmischen Strömen von Daten durch Prozessorfelder beruht, gewinnen mehr und mehr an Interesse, da sie sich sehr gut für die vollständige Hardwareimplementierung mit der VLSI-Technologie eignen.

3.4 Netz von Bäumen

Diese hybride Netzwerkarchitektur, formal definiert durch Leighton [63, 64], basiert auf einem $(n \times n \times n)$ Würfel. Jedem Knoten ist entsprechend einem kartesischen Koordinatensystem eine Adresse $[i, j, k]$ mit $1 \leq i, j, k \leq n$. zugeordnet. Für jedes j, k mit $1 \leq j, k \leq n$, d. h. in der x-Ausbreitung, werden interne Knoten und Kanten eingeführt, so daß ein kompletter binärer Baum entsteht, dessen Blätter die Knoten $[i, j, k]$ mit $1 \leq i, j, k \leq n$ sind. Diese Erweiterung wird analog für die verbleibenden i, k bzw. i, j Ebenen, d. h. in der y- bzw. z-Ausbreitung durchgeführt. Jeder Knoten des Originalwürfels ist Blatt eines Baumes in den 3 Ausbreitungen. Abb. 3.9 illustriert ein $(2 \times 2 \times 2)$ Netz von Bäumen, s. [65].

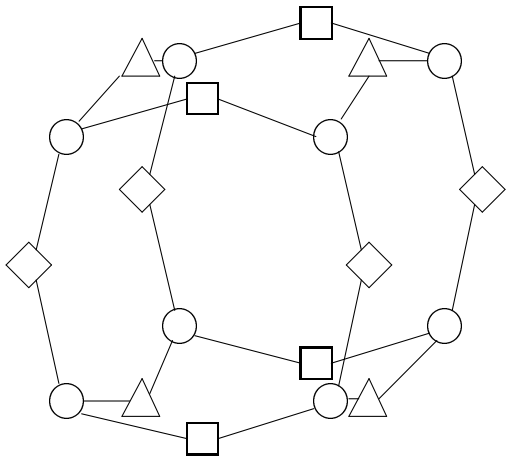


Abbildung 3.9: Ein $(2 \times 2 \times 2)$ Netz von Bäumen. Die Kreise stellen die 8 Original Knoten dar. Die verbleibenden 12 Knoten sind die Wurzelknoten der Bäume in den 3 Ausbreitungen.

Insgesamt gesehen besteht das $(n \times n \times n)$ Netz von Bäumen aus $n^3 + 3n^2(n-1)$ Knoten und $3n^2(2n-2)$ Kanten mit folgenden Eigenschaften. Der Durchmesser des Graphen beträgt $6 \log n$ und der Grad jedes Knoten, außer der $3n^2$ Wurzelknoten (Grad=2), ist 3. An dieser Stelle ist anzumerken, daß diese Graphkonstruktion implizit im Algorithmus von Preparata und Vuillemin [74] für die Matrizenmultiplikation beschrieben ist.

Beispiel 3.1 *Wieder sei $\mathcal{A} = (a_{ij})$ eine Matrix vom Typ (n, n) und $\vec{x} = (x_j)$ ein n-stelliger Vektor. Zu berechnen ist das Matrix-Vektor-Produkt $\vec{y} = \mathcal{A}\vec{x}$ mit $\vec{y} = (y_i)$.*

Die x_j Variable werden über den j-ten Spalten-Wurzelknoten für $1 \leq j \leq n$ durch den Spalten-Baum gesendet, so daß in jedem Blatt des j-ten Spalten-Baumes x_j nach $\log n$ Schritten empfangen wird. Zum Zeitpunkt $\log n$ werden die a_{ij} Variable direkt in die Blätter (ij) für $1 \leq i, j \leq n$ eingegeben und das Produkt $a_{ij}x_j$ gebildet, s. Abb. 3.10.

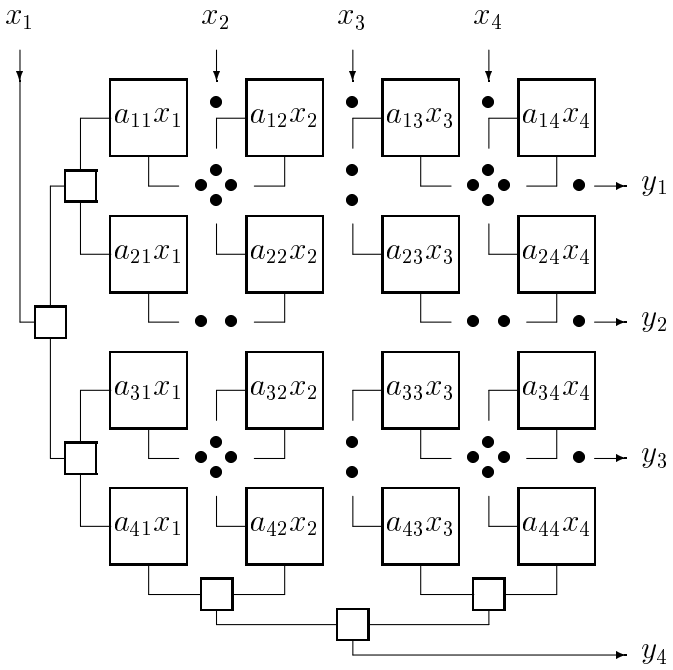
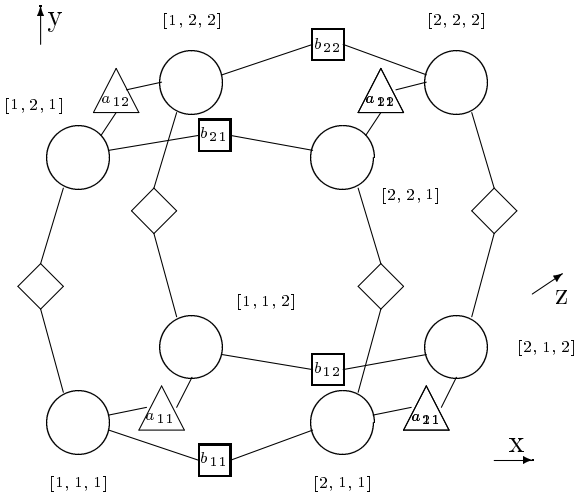


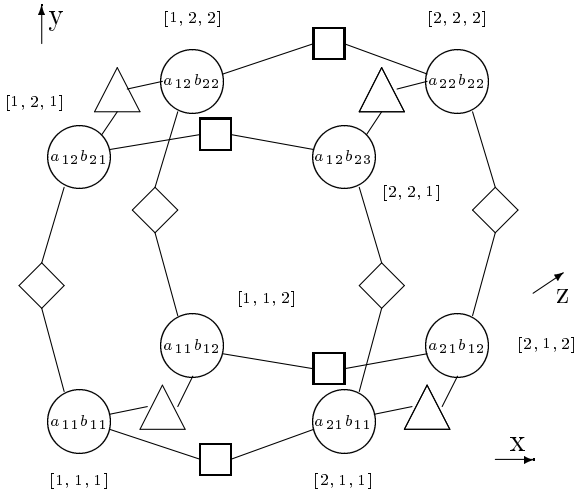
Abbildung 3.10: Matrix-Vektor-Produkt auf einem Netz von Bäumen. Die Baumstruktur ist aus Übersichtlichkeit nur teilweise angedeutet.

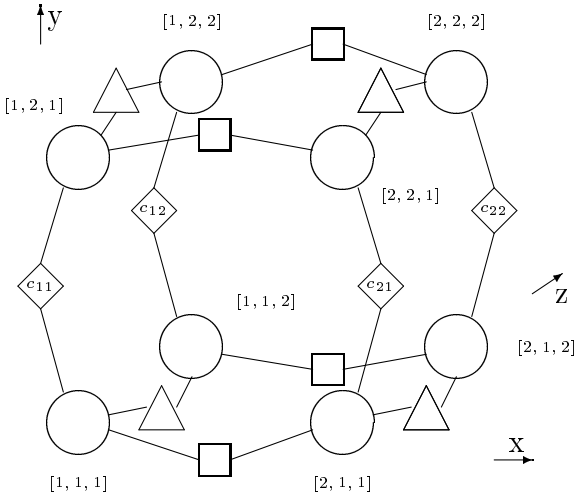
Die Produkte werden nun akkumulierend durch den jeweiligen Zeilen-Baum von den Blättern zur Wurzel gesendet. Nach insgesamt $2 \log n$ Schritten liegt das Ergebnis



Schritt 1

Schritt 2





Schritt 1

$$
\begin{aligned}
c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\
c_{12} &= a_{11}b_{21} + a_{12}b_{22} \\
c_{21} &= a_{21}b_{11} + a_{22}b_{12} \\
c_{22} &= a_{21}b_{12} + a_{22}b_{22}
\end{aligned}
$$

Abbildung 3.11: Multiplikation zweier Matrizen vom Typ $(2, 2)$ auf einem $(2 \times 2 \times 2)$ Netz von Bäumen.

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \tag{3.15}$$

im Wurzelknoten des i-ten Zeilen-Baumes an.

Beispiel 3.2 *Der Algorithmus für die Multiplikation zweier Matrizen $\mathcal{A}$ und $\mathcal{B}$ vom Typ (n, n) auf einem solchen Graphen gestaltet sich nun einfach.*

Die Variablen a_{ij} werden von den Wurzelknoten der i, j Ebene, z-Ausbreitung, für $1 \leq i, j \leq n$ eingelesen. Zur selben Zeit lesen die Wurzelknoten der Ebene j, k, x-Ausbreitung, für $1 \leq j, k \leq n$, die Variablen b_{jk}. Die $\mathcal{A}$ und $\mathcal{B}$ Variablen werden durch den Baum gesendet, so daß nach $\log(n+1)$ in den Blättern $[i, j, k]$ die Variable a_{ij}, b_{jk} für $1 \leq i, j, k \leq n$ empfangen werden. Nach Empfang werden in jedem Blatt die Variablen in einem Schritt multipliziert und das Ergebnis an die Eltern in der y-Ausbreitung des Baumes gesendet. Die verbleibenden $\log n$ Schritte in der y-Ausbreitung bilden eine Summe. Der Algorithmus für die Multiplikation von Matrizen vom Typ (n, n) für $n = 2$ wird illustriert in der Abb. 3.11. Der vorgestellte Algorithmus hat eine Zeitkomplexität von

$$T_p = 2\log(n+1) \in O(\log n). \tag{3.16}$$

3.5 Hypercube

Bei gegebenem Hypercube SIMD-Model mit $n^3 = 2^{3q}$ Prozessoren können Matrizen vom Typ (n, n) in einer Laufzeit von $O(\log n)$ multipliziert werden. Dieser Algorithmus geht auf Dekel [23] zurück.

Bei der Multiplikation von zwei (n, n) Matrizen werden zur parallelen Berechnung eines Elementes c_{ij} n Multiplikationen benötigt, somit für alle n^2 Elemente n^3 Prozessoren. Es ist also ein Verfahren erforderlich, welches alle n^3 Prozessoren mit den "richtigen" Elementen initialisiert.

Es sei $n = 2^q$. Die $n^3 = 2^{3q}$ Prozessoren P_r sind in einem (n, n, n) Gitter mit der Adresse (k, i, j) angeordnet. r ist ein Binärvektor mit $3q$ als Stelligkeit von r.

$$r = [r_{3q-1} r_{3q-2} \ldots r_{2q} r_{2q-1} \ldots r_q r_{q-1} r_{q-2} \ldots r_0] \tag{3.17}$$

mit $r_m \in \{0, 1\}$

$k = [r_{3q-1} r_{3q-2} \ldots r_{2q}]$

$i = [r_{2q-1} r_{2q-2} \ldots r_q]$

$j = [r_{q-1} r_{q-2} \ldots r_0]$.

Es gilt auch:

$$r = k \cdot n^2 + i \cdot n + j \quad mit \quad 0 \leq k, i, j \leq n-1 \tag{3.18}$$

Jeder Prozessor P_r ist mit den Registern A_r, B_r und C_r ausgestattet. Der Algorithmus von Dekel zerfällt nun in zwei Phasen. Während Phase 1 für ein intelligentes Data-Routing steht, ist Phase 2 die eigentliche Compute-Phase. Ausgangspunkt ist die Initialisierung der Prozessoren $P_{(2^q i+j)}$ mit den Matrixelementen $a_{i,j}$, $b_{i,j}$. Geometrisch bedeutet die Initialisierung, daß die Matrixelemente in der Ebene $k = 0$ abgelegt werden, s.a. Abb. 3.12 a).

$$A(0, i, j) = a_{ij} \qquad B(0, i, j) = b_{ij} \tag{3.19}$$

1. Phase

Zunächst werden die Matrizen $\mathcal{A}$ und $\mathcal{B}$ in die übrigen $k-1$ Ebenen kopiert. Dies wird mit einem parallelen Broadcast der Matrixelemente $a_{i,j}$, $b_{i,j}$ erreicht:

```
(* Broadcasting der Matrizen A, B *)
  FOR l=0 TO q-1 DO IN PARALLEL
    IF Bit(k,q-1-l)=0
      A(BitComplement(k,q-1-l),i,j):=A(k,i,j);
      B(BitComplement(k,q-1-l),i,j):=B(k,i,j);
    END (* if *)
  END (* for *);
```

Am Ende dieser Phase gilt:

$$\mathcal{A}(k, i, j) = a_{ij} \qquad \mathcal{B}(k, i, j) = b_{ij} \tag{3.20}$$

d. h. jedes Matrixelement steht in n verschiedenen Registern. Dazu werden insgesamt $2q$ Schritte benötigt, wobei nur jeder zweite Prozessor aktiv ist. Die Prozessoren mit Adressbit 1 an der Stelle $3q-1-l$ empfangen nur. Die Ausbreitung über die k-Ebenen wird durch Abb. 3.12 für $n = 4$ dokumentiert.

Durch eine sogenannte "Umfächerung" wird der Inhalt von $\mathcal{A}(k, i, k)$ in die Register der Prozessoren mit Adresse (k, i, j) mit $0 \leq j \leq n-1$ kopiert.

```
(* Spaltenverteilung *)
  FOR l=0 TO q-1 DO IN PARALLEL
    IF Bit(j,q-1-l)=Bit(k,q-1-l)
      A(k,i,BitComplement(j,q-1-l)):=A(k,i,j);
    END (* if *)
  END (* for *);
```

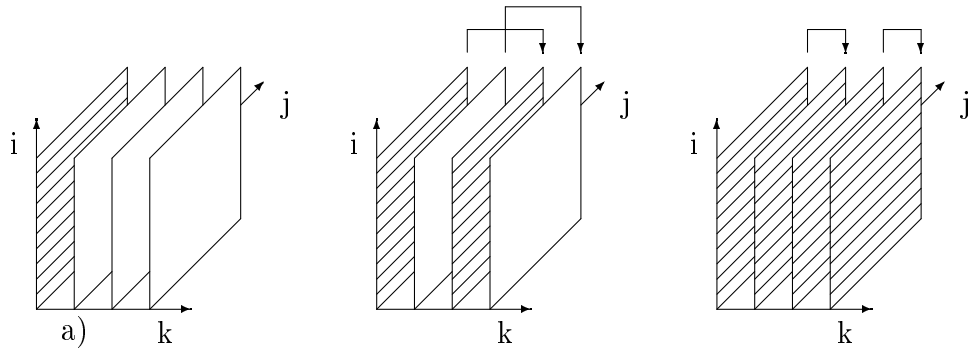


Abbildung 3.12: Broadcasting der Matrizen $\mathcal{A}$, $\mathcal{B}$

Nach Umfächerung gilt:

$$\mathcal{A}(k,i,j) = a_{ik} \tag{3.21}$$

d. h. das Matrixelement a_{ik} steht n mal hintereinander in den vertikalen j-Säulen, oder der Inhalt der Register $\mathcal{A}(k,i,k)$, die gerade die Spalte k in der k-ten Ebene bilden, breitet sich wie gewünscht in j-Richtung aus, s. a. Abb. 3.13.

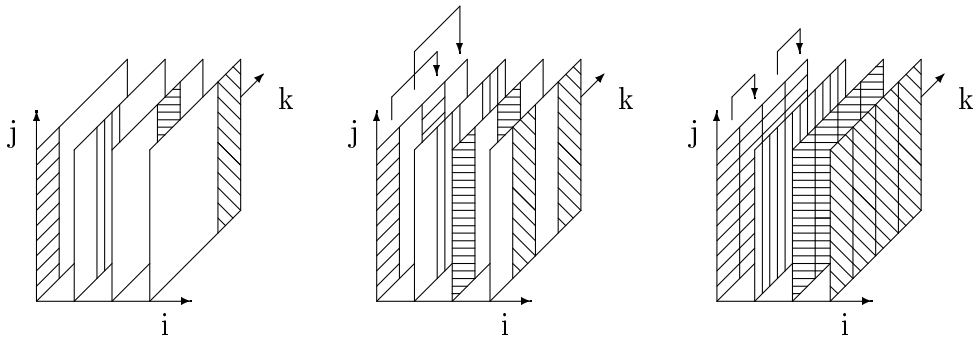


Abbildung 3.13: Spaltenverteilung

Analog dazu wird der Inhalt der Register $\mathcal{B}(k,k,j)$ also der Zeile k in der Ebene k in die Register der Prozessoren mit Adresse (k,i,j) mit $0 \leq i \leq n-1$ kopiert.

```
(* Zeilenverteilung *)
  FOR l=0 TO q-1 DO IN PARALLEL
    IF Bit(i,q-1-l)=Bit(k,q-1-l)
      B(k,BitComplement(i,q-1-l),j):=A(k,i,j);
    END (* if *)
  END (* for *);
```

Am Ende dieser Umfächerung gilt: $\mathcal{B}(k,i,j) = b_{kj}$, d. h. das Matrixelement b_{kj} wird n mal in i Richtung ausgebreitet, s. a. Abb. 3.14.

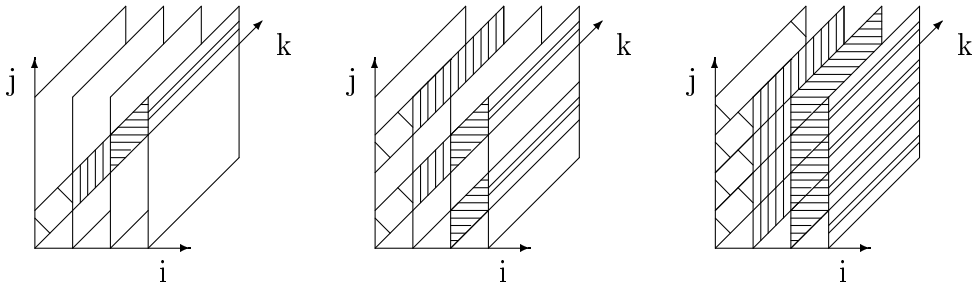


Abbildung 3.14: Zeilenverteilung

Die Umfächerung ist wie auch das Broadcasting nach $2q$ Schritten, mit 50 % Prozessorauslastung, beendet.

2. Phase

Die eigentliche Compute-Phase beginnt. In einem Schritt führen alle Prozessoren eine Multiplikation aus.

```
(* Multiplikation *)
  DO IN PARALLEL
    C(k,i,j):= A(k,i,j)B(k,i,j)
```

Durch ein paralleles "Concentrate" werden nun die Produkte in den k-Ebenen addiert.

```
(* Addition *)
  FOR l=0 TO q-1 DO IN PARALLEL
    IF Bit(k,l)=1
      C(BitComplement(k,l),i,j):=C(BitComplement(k,l),i,j)+C(k,i,j)
    END (* if *)
  END (* for *);
```

In den Registern $\mathcal{C}(0,i,j)$ steht schließlich nach q Schritten das gewünschte c_{ij}. Für die Laufzeit des Algorithmus ergibt sich

$$T_p(n) = 2q + 2q + 1 + q \in O(\log n). \tag{3.22}$$

Auf Grund der Tatsache, daß zur Berechnung der Summe c_{ij} von n Zahlen mindestens $\log n$ Schritte erforderlich sind, ist die logarithmische Laufzeit für eine parallele Matrixmultiplikation auf einem Hypercube optimal. Das wird erzielt auf Kosten eines erhöhten Mehraufwandes. Während der "naive" Algorithmus mit $O(n^3)$ zu Buche schlägt, beträgt der Aufwand hier $C(n) \in O(n^3 \log n)$. Der Grund liegt u. a. darin, daß zum Teil nur 50 % der Prozessoren aktiv sind. Ein Vorteil des Algorithmus ist, daß die Ein- und Ausgabe jeweils an der gleichen Stelle, in der Ebene $k = 0$, erfolgt.

3.6 Systolisches Feld

Das systolische Feld-Konzept ist von Kung und Leiserson [59] eingeführt worden. Die Abb. 3.15 stellt das Ablaufschema für die Multiplikation zweier Bandmatrizen mit einem systolischen Algorithmus auf einer hexagonalen Prozessorfeld-Struktur nach [58] dar.

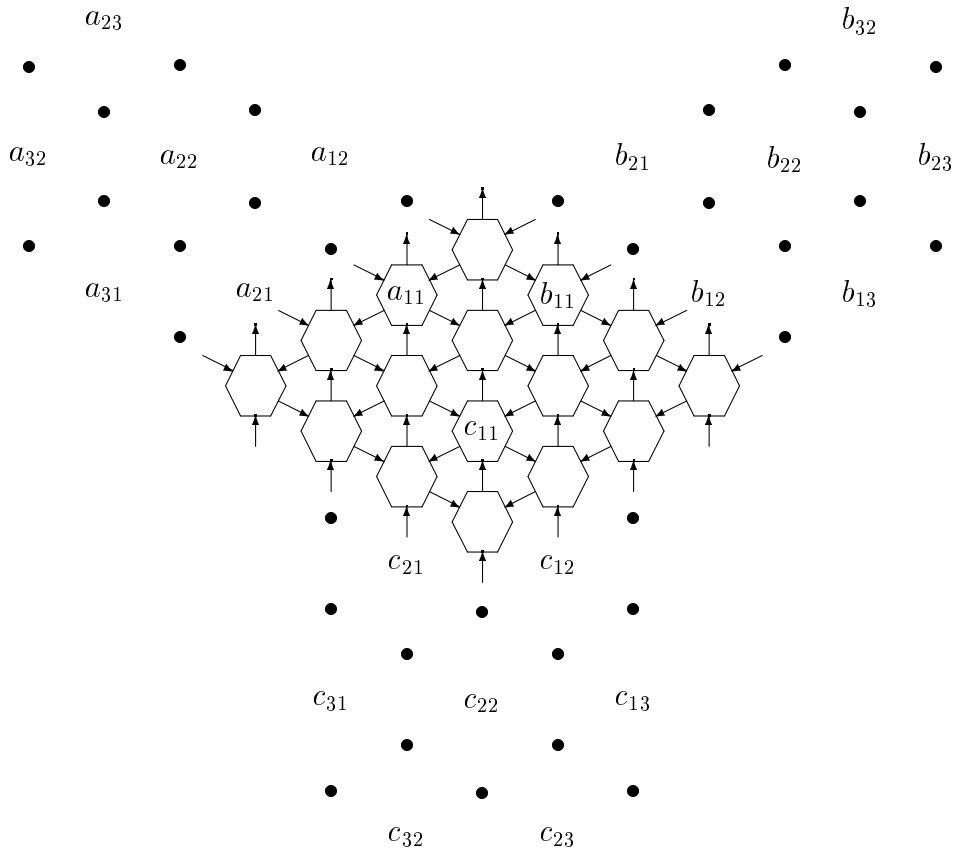


Abbildung 3.15: Systolischer Algorithmus zur Multiplikation von Bandmatrizen

Neben den Datenströmen der Eingangsmatrizen ist auch der virtuelle Strom der Ergebnismatrix angegeben.

Mit den ersten Arbeiten von Kung und Leiserson hat sich das Studium von Algorithmen (Matrixmultiplikation) auf systolischen Feldern intensiviert [17, 50, 51, 73, 75]. Als eine typische Anwendung für die Benutzung systolischer Felder gilt die Multiplikation einer Matrix mit einem Vektor.

Beispiel 3.3 *Gegeben ist eine Matrix* $\mathcal{A} = (a_{ij})$ *vom Typ* (n, n)*, ein* n*-Vektor* $\vec{x} = (x_j)$ *und es sei zu berechnen das Matrix-Vektor-Produkt* $\vec{y} = \mathcal{A}\vec{x}$ *mit* $\vec{y} = (y_i)$ *und*

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \tag{3.23}$$

für alle $1 \geq i \geq n$*.*

Die einfache sequentielle Methode benötigt $2n^2 - n$ Schritte. Diese Berechnung soll nun mit einem systolischen Feld bestehend aus $2n-1$ Prozessoren ausgeführt werden, Abb. 3.16.

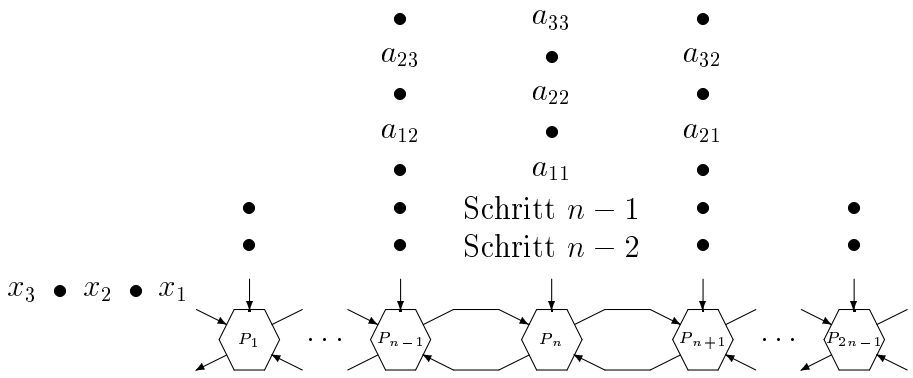


Abbildung 3.16: Ein lineares systolisches Array

Die zeitliche Abstimmung der Datenströme ist ein wesentliches Merkmal der systolischen Maschine. Das allgemeine Schema besteht darin, daß die Matrix über die oberen Eingänge der Prozessoren in einer an den Hauptdiagonalen gespiegelten und um 45° gedrehten Gestalt und der Vektor über den linken Eingang des ersten Prozessors durch das Feld geroutet werden. Zwischenergebnisse werden im Feld von rechts nach links weitergegeben, wobei das Ausgabedatum letztendlich am Ausgang des Prozessors P_1 erscheint. Eine Berechnung setzt sich wie folgt zusammen: Linkes und oberes Eingabedatum werden multipliziert und das rechte

Zwischenergebnis hinzuaddiert. Es erfolgt stets eine dynamische Transformation von Eingabedaten in Ausgabedaten, d. h. es ist nicht notwendig, sich berechnete Werte zu "merken".

Beispiel 3.4 *Exemplarisch sei die zeitliche Abstimmung für folgendes Beispiel dargestellt:*

$$\begin{bmatrix} 1 & 3 & -4 \\ 2 & 1 & -5 \\ 6 & -2 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} = \begin{bmatrix} 8 \\ -3 \\ 0 \end{bmatrix}$$

Der einzugebende Vektor wird über den linken Eingang des Prozessors P_1 in den Schritten 1, 3 und 5 eingegeben. Die einzugebende Matrix wird beginnend mit Schritt 3 über die oberen Eingänge der Prozessoren eingegeben. Der auszugebende Vektor erscheint in den Schritten 6, 8 und 10 am linken Ausgang des Prozessors P_1. Abb. 3.17 illustriert die einzelnen Schritte für das kleine Rechenbeispiel.

Das Verfahren läßt sich offensichtlich in dieser Weise auf die Multiplikation einer Matrix vom Typ (n, n) und einem n-Vektor unter Verwendung von $2n - 1$ Prozessoren in $4n - 2$ Schritten verallgemeinern. Dies kommt der idealen Situation nahe, daß jeder Prozessor in jedem Schritt eine nützliche Operation ausführt; ein quadratischer Algorithmus wird unter Benutzung einer linearen Anzahl von Prozessoren auf einen linearen Algorithmus reduziert. Bereits dieses Beispiel verdeutlicht recht eindrucksvoll, daß systolische Felder einfach und leistungsfähig sind. Das Wesen liegt in der Zusammenschaltung und der zeitlich abgestimmten Eingabe der Eingangsdaten.

Aus der Vielzahl der systolischen Feld-Algorithmen für die Matrixmultiplikation soll ein Algorithmus vorgestellt werden, der auf die Arbeiten von Benaini, Robert [6] zurückgeht. Er basiert auf dem Winograd-Algorithmus (s. Abschnitt 2.2)

$$c_{ij} = d_{ij} - \alpha_i - \beta_j. \tag{3.24}$$

Der Vorteil bei Winograd ist der, daß die Koeffizienten α_i und β_j nur einmal berechnet werden müssen, weil sie für die gesamte Zeile i bzw. Spalte j der Matrix $\mathcal{C}$ benutzt werden. Das systolische Feld setzt sich zusammen aus $\frac{1}{2}n$ Zeilen von n Zellen. Abb. 3.18 zeigt die Struktur und Eingabe der Koeffizienten exemplarisch für $n = 4$.

Wegen der Rückführung wird die Struktur zweimal durchlaufen. Beide Phasen sind identisch. In Phase 1 berechnet das Array $\frac{1}{2}n^2$ der Koeffizienten c_{ij} und in der Phase 2 die verbleibenden $\frac{1}{2}n^2$ Elemente. Die erste Zeile des Array unterscheidet sich von

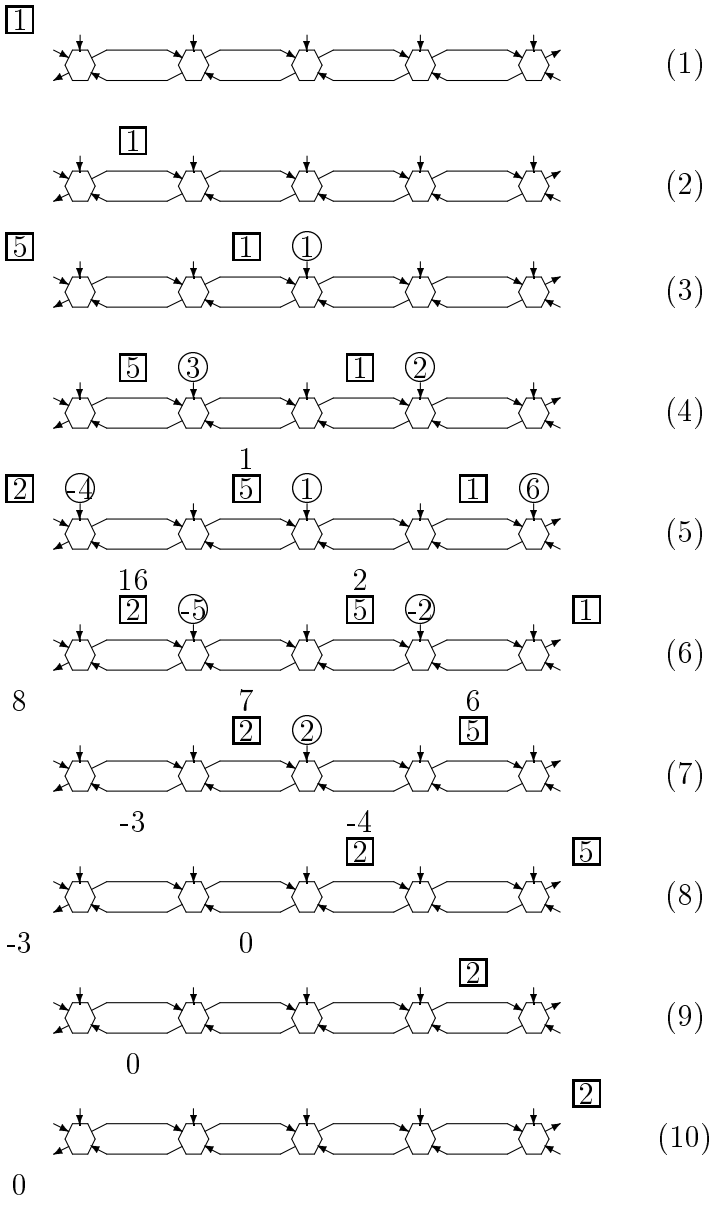


Abbildung 3.17: Multiplikation einer Matrix mit einem Vektor mit Hilfe eines systolischen Feldes

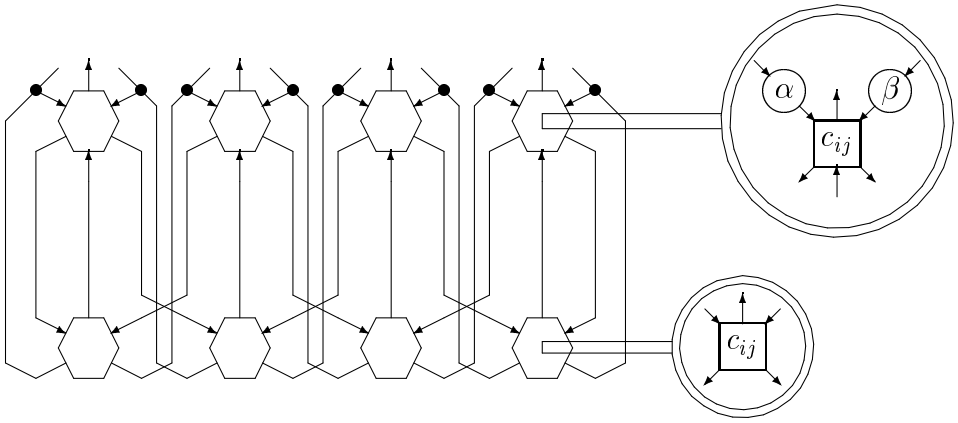


Abbildung 3.18: Systolisches Array ($n = 4$), incl. Basiszelle für die Winograd-Multiplikation

der zweiten insofern, daß sie zusätzlich noch die Koeffizienten α_i, β_j berechnet. Der Programmablauf des Array's wird gesteuert durch die Boolesche Variable t. Die Operationen der Zellen sind detailliert in Tabelle 3.1 und den Programmfragmenten dargestellt.

Kontrollbit t	"Kreis"-Prozess	"Quadrat"-Prozess
0	akkumuliere für α_i oder β_j	akkumuliere für d_{ij}
1	sende α_i oder β_j	berechne c_{ij}

Tabelle 3.1: Operationen der Winogradzellen

```
(* Kreis-Prozess *)
t_out:=t_in;
IF t_in =FALSE
  r:=r+a_in_1*a_in_2;
  a_out_1:=a_in_1;
  a_out_2:=a_in_2;
ELSE
  a_out_1:=r;
  a_out_2:=0;
END (* if *)
```

```
(* Quadrat-Prozess *)
t_out:=t_in;
a_out_1:=a_in_1;
a_out_2:=a_in_2;
b_out_1:=b_in_1;
b_out_2:=b_in_2;
IF t_in=FALSE
(a_in_1+b_in_2)(a_in_2+b_in_1)DO IN PARALLEL
  r:=r+t*t1;
  c_out:=c_in;
ELSE
  c_out:=r-a_in_1_b_in_1;
END (* if *)
```

1. Phase	Takt								
1. Zellreihe									
Kreisprozess		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
	1, 2	α_1	β_1	α_2	β_2	α_3	β_3	α_4	β_4
Quadratprozess		(1,1)		(1,2)		(1,3)		(1,4)	
	2	d_{11}		d_{22}		d_{33}		d_{44}	
	3	d_{11}		d_{22}		d_{33}		d_{44}	
	4	$d_{11} - \alpha_1 - \beta_1$		$d_{22} - \alpha_2 - \beta_2$		$d_{33} - \alpha_3 - \beta_3$		$d_{44} - \alpha_4 - \beta_4$	
2. Zellreihe									
Quadratprozess		(2,1)		(2,2)		(2,3)		(2,4)	
	3	d_{21}		d_{13}		d_{42}		d_{34}	
	4	d_{21}		d_{13}		d_{42}		d_{34}	
	5	$d_{21} - \alpha_1 - \beta_1$		$d_{13} - \alpha_1 - \beta_3$		$d_{42} - \alpha_4 - \beta_2$		$d_{34} - \alpha_3 - \beta_4$	

Tabelle 3.2: Phase 1

2. Phase	Takt								
1. Zellreihe									
Kreisprozess		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
	4, 5	α_2	β_3	β_1	α_4	α_1	β_4	β_2	α_3
Quadratprozess		(1,1)		(1,2)		(1,3)		(1,4)	
	5	d_{23}		d_{41}		d_{14}		d_{32}	
	6	d_{23}		d_{41}		d_{14}		d_{32}	
	7	$d_{23} - \alpha_2 - \beta_3$		$d_{41} - \alpha_4 - \beta_1$		$d_{14} - \alpha_1 - \beta_4$		$d_{32} - \alpha_3 - \beta_2$	
2. Zellreihe									
Quadratprozess		(2,1)		(2,2)		(2,3)		(2,4)	
	6	d_{43}		d_{24}		d_{31}		d_{12}	
	7	d_{43}		d_{24}		d_{31}		d_{12}	
	8	$d_{43} - \alpha_4 - \beta_3$		$d_{24} - \alpha_2 - \beta_4$		$d_{31} - \alpha_3 - \beta_1$		$d_{12} - \alpha_1 - \beta_2$	

Tabelle 3.3: Phase 2

Ein Schritt bei Winograd besteht aus 2 Additionen `DO IN PARALLEL` einer weiteren Addition und Multiplikation, so daß sich eine Zeiteinheit von $\tau = 2 \cdot \tau_{add} + \tau_{mul}$ ergibt. Für ein besseres Verständnis der Operationen im Array sind die Aktivitäten in Tabelle 3.2 für Phase 1 und Tabelle 3.3 für Phase 2 für den Fall $n = 4$ zusammengestellt.

Die Operationen des Feldes wurden durch Simulation validiert [102]. Die Familie von Matrixmultiplikations-Feldern, die auf Winograd basieren, sind in vielerlei Hinsicht anderen existenten Feldern vorzuziehen. Winograd's Algorithmus ist optimal [40]. Alternative bilineare Techniken für die Matrixmultiplikation, vorgeschlagen in [52, 70] und [98], reduzieren zwar asymptotisch die Anzahl der arithmetischen Operationen, sind aber wegen des erheblichen Aufwandes bei der Berechnung der Koeffizienten hier nicht geeignet, ausgenommen für sehr große Matrizen. Mehr noch, diese Algorithmen lassen sich nicht günstig auf reguläre Felder-Strukturen implementieren.

Kapitel 4

Anwendungen paralleler Algorithmen

4.1 Matrizeninversion und biomagnetische Felder

Neben der Matrizenmultiplikation ist die Inversion für viele Bereiche der Mathematik von großer Bedeutung. Zusammen mit einem effizienten parallelen Algorithmus zur Matrizenmultiplikation ist die Inversion grundlegend für einen effizienten Algorithmus zur Lösung linearer Gleichungssysteme, denn es gilt:

$$\mathcal{A}x = b \leftrightarrow x = \mathcal{A}^{-1}b \tag{4.1}$$

falls $\mathcal{A}$ invertierbar ist. Leider ist bis heute kein Algorithmus bekannt, der eine Matrix in $O(\log n)$ Schritten invertiert. Nach heutigem Kenntnisstand liegen die Probleme Matrizenmultiplikation und Matrizeninversion in verschiedenen parallelen Zeitkomplexitätsklassen, obwohl sie zur gleichen seriellen Komplexitätsklasse gehören [11].

Muskelaktivitäten des menschlichen Körpers sind aufgrund ihrer elektrochemischen Wirkungsmechanismen immer mit biomagnetischen Felderscheinungen verbunden. Das Herz ist unter diesem Gesichtspunkt eine besonders aktive Region. Die Nutzung magnetischer Felder zur Gewinnung von Informationen über die Herztätigkeit bedarf sehr empfindlicher Sensortechnik. Biomagnetische Felder sind extrem schwach $B = 10^{-11} - 10^{-14}T$ (im Vergleich [76]: Magnetfeld der Erde $B \sim 0.5 \cdot 10^{-4}T$). Das erste Magnetokardiogramm (MKG) wurde 1963 von Baule und McFee [5, 18] aufgenommen. Mehrkanalanordnungen erlauben die gleichzeitige Messung an mehreren Punkten bei einer Justierung. Dies sichert eine definierte geometrische Lage der Meßpunkte zueinander und erlaubt die Elimi-

nierung von Störeinflüssen mit Hilfe von Referenzmessungen.

Eine bei Herzfehlfunktionen anwendbare Diagnosemethode [92] neben dem Elektrokardiogramm (EKG) besteht darin, dem Patienten ein individuelles Referenzmodell des Herzfeldes eines gesunden Menschen anzupassen. Über die vom Modell abweichenden tatsächlich gemessenen biomagnetischen Feldparameter kann Art und Ort des Defektbereiches im Herz bestimmt werden. Die Struktur des menschlichen Körpers aus feldtheoretischer Sicht kann als dreidimensionale Verteilung des Leitfähigkeitstensors im Raum betrachtet werden. Die Leitfähigkeit ist nicht nur inhomogen, sondern auch stark anisotrop. Diese Verhältnisse werden durch Gebiete konstanter isotroper Leitfähigkeit approximiert, wodurch die Randelemente-Methode angewendet werden kann. Sie beruht auf der Annahme, daß der menschliche Körper näherungsweise in Teilvolumina konstanter Leitfähigkeit unterteilt werden kann. Als Referenzmodelle werden realistische Torsomodelle als Volumenleitermodelle und Stromdipole als Quellenmodelle herangezogen. Mit Hilfe bikubischer Splinefunktionen erfolgt die Approximation der Grenzflächen der einzelnen Volumenleiter. Die Ermittlung der Parameter des zugrundegelegten Stromquellenmodells aus den Meßwerten wird als inverses biomagnetisches Problem bezeichnet. Bei der Lösung dieses Problems hat sich der Marquardt-Levenberg-Algorithmus [66] vielfach bewährt, der zur Lage- und Parameterbestimmung der Quelle im auf den Patienten zugeschnittenen Modell führt. Somit wird eine Diagnose und Lokalisation der Fehlfunktion im menschlichen Organismus möglich.

Die Lösung des eigentlichen Problems geht einher mit der Lösung von Gleichungssystemen aufgrund umfassender Matrizenoperationen vom Typ $(2000, 2000)$ für praxisrelevante Modelle.

Die Matrixinversion ist mit ca. 26% an der Gesamtlaufzeit wesentlicher Bestandteil des Herzfeldsimulationsalgorithmus. Die Inversion läßt sich zurückführen auf die Lösung linearer Gleichungssysteme. Deshalb kann der Gauß'sche Algorithmus [13] auch auf die Matrixinversion angewendet werden. Die Rechenvorschrift lautet im einzelnen:

$$
\begin{array}{lllll}
1. & a_{kl} := \frac{1}{a_{kl}} & & & \text{Pivotelement,} \\
2. & a_{kj} := \frac{a_{kj}}{a_{kl}} & j \neq l, & & \text{Elemente der Pivotspalte,} \\
3. & a_{il} := -\frac{a_{il}}{a_{kl}} & i \neq k, & & \text{Elemente der Pivotzeile,} \\
4. & a_{ij} := a_{ij} - \frac{a_{il}}{a_{kl}} a_{kj} & i \neq k,\ j \neq l, & & \text{alle übrigen Matrixelemente.}
\end{array}
\tag{4.2}
$$

Ein spezielles Problem stellt die Wahl des Pivotelementes, hier a_{kl} dar. Wegen der internen Zahlendarstellung wird die Mantisse einer Gleitkommazahl an der letzten Stelle u. U. abgerundet, so daß der entstehende Fehler zu scheinbarer

Singularität (Division durch ein Pivotelement nahe oder gleich Null) und damit zum Abbruch der Berechnung führt. Auf Möglichkeiten der Pivotisierung und Stabilitätsbetrachtungen [31] bei der Lösung von linearen Gleichungssystemen, im Speziellen der Matrizeninversion, wird hier nicht weiter eingegangen.

Nach Formel 4.2 wird in den Schritten 1-4 für jede der n Iterationen folgende Anzahl von Operationen

$$T(n) = n(3(n-1)^2 + 2(n-1) + 1) \in O(n^3) \tag{4.3}$$

ausgeführt. Wie die getroffene Abschätzung, obwohl Vergleiche etc. nicht berücksichtigt sind, zeigt, gehört die Matrizeninversion zur Klasse BLAS-3, s. a. Abschnitt 4.6. Eine genaue Betrachtung zeigt, daß die Schritte 1-4 in jeder Iteration parallel ausgeführt werden können. Für derartige Probleme ist in der Regel die Anzahl der Prozessoren kleiner als die Problemgröße, $p < n$, so daß ein Mapping der Matrizen auf die Prozessoren erfolgen muß. Diese Verteilung der Daten auf eine MIMD-Architektur mit verteiltem Speicher kann das Laufzeitverhalten eines Algorithmus entscheidend beeinflussen. Es muß eine Abbildung von Daten auf die Prozessoren gefunden werden, die bei verfügbarer Kommunikationsbandbreite eine optimale parallele Lösung gewährleistet. Die Speicherung von Matrizen erfolgt meist zeilenweise. Somit bietet sich natürlich die Verwendung der Zeilenpivotisierung und ein sogenanntes Zeilenmapping an. Die Organisation im Speicher erleichtert so z. B. die notwendige Übertragung von Pivotzeile und Pivotelement. Abb. 4.1 zeigt die prinzipielle Aufteilung der Matrix auf die Prozessoren.

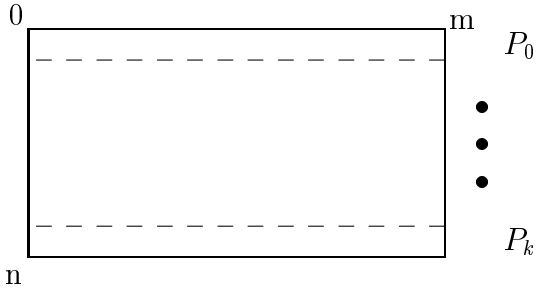


Abbildung 4.1: Mapping

Die Abbildungsvorschrift der Ausgangsmatrix $\mathcal{A}$ auf alle P_k Prozessoren lautet:

$$(a_{ij} \to P_r) \leftrightarrow (r = \left\lfloor \frac{i}{d} \right\rfloor) \tag{4.4}$$

mit $d = \lceil \frac{n}{k} \rceil$, wobei d der Anzahl der abgebildeten Zeilen je Prozessor entspricht. Diese Abbildung erzeugt eine möglichst gleichmäßige Verteilung (Lastbalance)

der Daten. Auftretende Unregelmäßigkeiten wirken sich hier immer auf die Prozessoren mit den höchsten Indizes aus. Durch das Mapping der Daten und die Nummerierung der Prozessoren ist die parallele Steuerung der Inversion bereits festgelegt. Diese muß schrittweise nach Zeilen erfolgen und vernünftigerweise beginnend mit der ersten Zeile. Da nur ein Prozessor über die gegenwärtige Pivotzeile verfügt, diese und der Pivotelementindex aber von allen P_k Prozessoren zur Berechnung benötigt werden, muß in jedem Schritt das ganze Netz über die Pivotzeile und den Spaltenindex des Pivotelementes informiert werden. Dem Prozessor mit der aktuellen Pivotzeile kommt somit die Aufgabe zu, die Steuerung der anderen Prozessoren zu übernehmen (Master-Slave-Prinzip). Die folgenden Programmfragmente skizieren die Aktionen für den Master-Prozessor und die Slave-Prozessoren:

```
(* Master *)
  find (Pivotelement);
  broadcast (Pivotzeile, Pivotindex);
  calculate (alle a(i,j), fuer r*d<= i<(r+1)*d);
  synchronize (mit allen Prozessoren);

(* Slave *)
  receive (Pivotzeile, Pivotindex);
  calculate (alle a(i,j), fuer r*d<= i<(r+1)*d);
  synchronize (mit allen Prozessoren);
```

Als Implementierungsplattform wurde ein Transputernetzwerk (MC2/16-1 mit 12 T800) verwendet. Es sind sowohl Untersuchungen bzgl. der Parallelisierung der Matrizeninversion als auch des gesamten Algorithmus zur biomagnetischen Herzfeldberechnung angestellt worden. Diese Applikationen wurden mit verschiedenen anwendungsrelevanten Netzwerktopologien erprobt. Die verwendeten Routing- und Message-Passing- Algorithmen, s. [103], gestatten eine Anpassung des Gesamtsystems an jede mittels Cross-Bar-Switches frei konfigurierbare Topologie. Der Einfluß der Topologie auf den Speedup zeigt Abb. 4.2.

Die Messungen ergaben, daß Gewinne mit Topologien hoher Konnektivität nur dann Vorteile bringen, wenn die Kommunikationslast überaus hoch und die Effizienz des Netzes bezogen auf die Problemgröße noch gering ist. Ist das Verhältnis von Kommunikation und Berechnung ausgewogen, dann erweist sich der Ring als günstigste Topologie. Das rührt daher, daß bei der Komplexität $O(n^3)$ des Algorithmus bei wachsender Problemgröße die Rechenlast stark steigt, während die Kommunikationslast nur linear mit der Dimension wächst. Zudem trägt die Struktur des Message-Passing-Systems zusammen mit der Verwaltung der Links durch den Transputer dazu bei, daß die Kommunikation überlappend durchgeführt werden kann. Somit schlägt sich erhöhte Kommunikationslast in Netzen

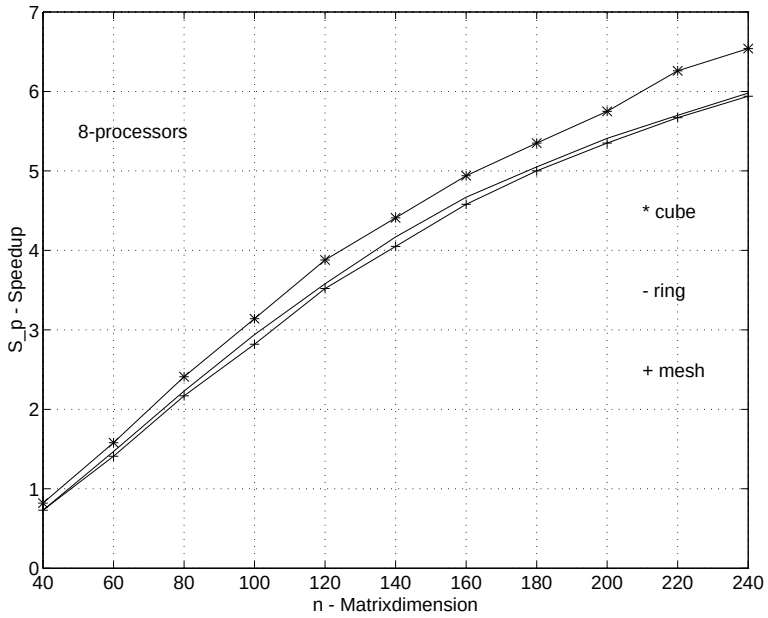


Abbildung 4.2: Speedup in Abhängigkeit von der Problemgröße mit $p = 8$ Prozessoren vernetzt als Ring, Gitter und Hypercube

der hier betrachteten Größe nicht negativ auf die Gesamtleistung des Systems nieder, einfache Strukturen sind letztlich günstiger. Abb. 4.3 illustriert den Einfluß der Anzahl der Prozessoren auf den Speedup.

Erkennbar wird, daß sich die Optima des Speedup mit wachsendem n zu immer längeren Laufzeiten t verlagern, oder, der Einsatz von 11 Prozessoren bringt keine signifikant kürzere Laufzeit als 5 Prozessoren für eine Problemgröße $n = 100$. Es ist offenbar schwieriger und kostenaufwendiger ein Problem konstanter Dimension in deutlich kürzerer Zeit zu lösen, als ein größeres Problem in der gleichen Zeit. Das entspricht der Erkenntnis, daß sich in MIMD-Architekturen der Scaleup meist wesentlich einfacher gestaltet, als der Speedup, s. a. Kapitel 5. Abschließend noch einige Bemerkungen zu den Ergebnissen bei der Parallelisierung des gesamten Algorithmus zur biomagnetischen Herzfeldberechnung. Tabelle 4.1 faßt Laufzeiten und prozentuale Anteile zusammen. Die Problemgröße sei $n = 240$ mit $p = 10$ Prozessoren vernetzt in einem Ring.

Obwohl ein beachtlicher Speedup $S_p = 7.74$ für die Inversion erzielt wurde, eig-

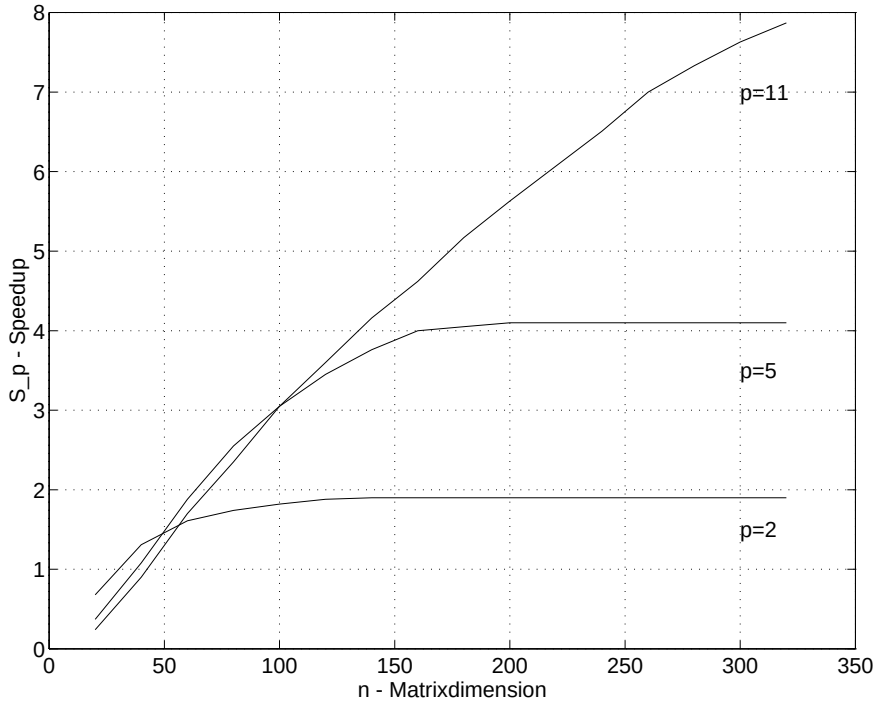


Abbildung 4.3: Speedup in Abhängigkeit von der Problemgröße für eine unterschiedliche Anzahl von Prozessoren vernetzt in einem Ring

net sich die Lösung mehr für die Bearbeitung deutlich größerer Probleme. Die veraltete Technologie des T800 wirk insgesamt limitierend, d. h. sowohl Kommunikationsgeschwindigkeit als auch Rechenleistung sind zu gering, um mit modernen Architekturen konkurrieren zu können. Der Gesamtalgorithmus konnte um 21.9% beschleunigt werden, was einem Speedup von $S_p = 1.28$ entspricht. Interessanterweise entspricht der Speedup dem zu erwartetenden, berechenbaren theoretischen Speedup mit Hilfe des Amdahl'schen Gesetzes, s. a. Abschnitt 5.

Im folgenden sei ein Algorithmus zur exakten Inversion in $O(\log^2 n)$ Schritten präsentiert. Das Vorgehen dieses Algorithmus unterscheidet sich ganz fundamental von dem Vorgehen sequentieller Inversionsalgorithmen. Es sei zunächst an das charakteristische Polynom einer Matrix und an den Satz von Cayley-Hamilton aus der linearen Algebra erinnert.

	Laufzeit t in $[s]$	Anteil in %
Gesamtalgorithmus	398.4	
Matrizeninversion	101.4	25.5
paralleler Gesamtalgorithmus	311.1	
parallele Matrizeninversion	14.1	4.5

Tabelle 4.1: Ergebnisse durch parallele Berechnung des Herzfeldes

Definition 4.1 *Das Polynom n-ten Grades [77]*

$$p(\lambda) = \det(\mathcal{A} - \lambda\mathcal{I}) = \lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \ldots + c_{n-1}\lambda + c_n \tag{4.5}$$

heißt das charakteristische Polynom der Matrix $\mathcal{A}$. Die Nullstellen von p heißen die Eigenwerte von $\mathcal{A}$.

Für die Existenz einer Inversen mit der Eigenschaft $\mathcal{A}^{-1}\cdot\mathcal{A} = \mathcal{I}$ ist die Regularität der Matrix $\mathcal{A}$, $\det\mathcal{A} \neq 0$, notwendige und hinreichende Bedingung. Aus der Polynomdarstellung von $p(\lambda)$ folgt für $\lambda = 0$, $\det\mathcal{A} = c_n$, also muß $c_n = \det\mathcal{A} \neq 0$ vorausgesetzt werden.

Theorem 4.1 *Cayley-Hamilton: Eine beliebige quadratische Matrix $\mathcal{A}$ genügt ihrer eigenen charakteristischen Gleichung, d. h. hat $\mathcal{A}$ das charakteristische Polynom*

$$p(\lambda) = \lambda^n + \sum_{i=1}^{n} c_i\lambda^{n-i}, \tag{4.6}$$

so erfüllt $\mathcal{A}$ die Cayley-Hamiltonsche Gleichung

$$p(\mathcal{A}) = \mathcal{A}^n + c_1\mathcal{A}^{n-1} + c_2\mathcal{A}^{n-2} + \ldots + c_{n-1}\mathcal{A} + c_n\mathcal{I} = \mathcal{O}. \tag{4.7}$$

Das Matrizenpolynom $p(\mathcal{A})$ ist gleich der Nullmatrix.

Wir benutzen nun den Satz von Cayley-Hamilton unter der Voraussetzung, daß $\mathcal{A}^{-1}$ existiert, um eine Darstellung für $\mathcal{A}^{-1}$ zu bestimmen. Dazu multiplizieren wir die Matrizengleichung mit $\mathcal{A}^{-1}$ und erhalten

$$\mathcal{O} = \mathcal{A}^{n-1} + c_1\mathcal{A}^{n-2} + c_2\mathcal{A}^{n-3} + \ldots + c_{n-1}\mathcal{I} + c_n\mathcal{A}^{-1}, \tag{4.8}$$

$$\mathcal{A}^{-1} = -\frac{1}{c_n}(\mathcal{A}^{n-1} + c_1\mathcal{A}^{n-2} + \ldots + c_{n-1}\mathcal{I}). \tag{4.9}$$

Das Problem der Matrix-Inversion ist darauf zurückgeführt, die Potenzen der Matrix $\mathcal{A}$ und die Koeffizienten des charakteristischen Polynoms zu berechnen. Die Koeffizienten des charakteristischen Polynom einer Matrix $\mathcal{A}$ erfüllen das folgende Gleichungssystem, Lemma-Le Verrier [1],

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ s_1 & 2 & 0 & \cdots & 0 \\ s_2 & s_1 & 3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{n-1} & s_{n-2} & s_{n-3} & \cdots & n \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix} = - \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_n \end{pmatrix} \tag{4.10}$$

wobei s_k die Spur von $\mathcal{A}^k$, $1 \leq k \leq n$, bezeichnet. Mit Hilfe des Lemma von Le Verrier ist es leichter, die Koeffizienten des charakteristischen Polynoms zu bestimmen, denn das Problem wird darauf reduziert, eine untere Dreiecksmatrix zu invertieren.
Csanky [22] zeigte, daß eine Matrix vom Typ (n, n) mit $p = \frac{n^4}{4}$ Prozessoren in

$$T_p(n) = \frac{3}{2}\log^2 n + O(\log n) \in O(\log^2 n) \tag{4.11}$$

invertiert werden kann. Der Csanky-Algorithmus umfaßt die folgenden vier Berechnungsstufen:

1. Berechnung aller Potenzen $\mathcal{A}^i$ von $\mathcal{A}, 1 \leq i \leq n$
2. Berechnung der Spur s_i von $\mathcal{A}^i, 1 \leq i \leq n$
3. Lösung des Dreiecksystems nach den Koeffizienten c_j des charakteristischen Polynoms
4. Berechnung der Inversen $\mathcal{A}^{-1}$ unter Verwendung der Potenzen von $\mathcal{A}$ und der Koeffizienten c_j

Zur Bestimmung der Zeitkomplexität werden die einzelnen Berechnungsstufen analysiert.

[1] Le Verrier berechnete 1846 allein aus Störungen der Bahnen von Uranus und Merkur die Bahn des Planeten Neptun.

1. Eine Potenz $y^i, 1 \leq i \leq n$ berechnet sich mit $p = \lfloor \frac{n}{2} \rfloor$ Prozessoren in der Zeit $T_p = \lceil \log n \rceil$. Ausgedehnt auf die Berechnung des n-fachen Produktes quadratischer Matrizen $\mathcal{A}$ vom Typ (m, m) setzt sich der Aufwand an Zeitschritten so zusammen: $T_p = \lceil \log n \rceil \cdot T_{p'}(\mathcal{A}^2)$ wobei $T_{p'}(\mathcal{A}^2)$ die Zeit für die Durchführung der Matrizenmultiplikation zweier (m, m) Matrizen ist. Dabei bedeuten p die Gesamtzahl der notwendigen Prozessoren und p' die Zahl der für die individuelle Matrizenmultiplikation nötigen Prozessoren. Für die anfallende Matrizenmultiplikation sei der Hypercube-Algorithmus aus Abschnitt 3.5 benutzt, der in $O(\log n)$ Zeit mit n^3 Prozessoren läuft. Werden Auf- bzw. Abrundungen vernachlässigt, so ergibt sich für die Potenzen von $\mathcal{A}$

$$T_p = \log^2 n, \text{mit } p = \frac{n}{2} n^3. \tag{4.12}$$

2. Da die Spur einer Matrix M eine Summation impliziert, lassen sich mit Rekursivem Doppeln die $s_i, 1 \leq i \leq n$ in

$$T_p = \lceil \log n \rceil \tag{4.13}$$

Schritten gleichzeitig für $i = 1, 2 \ldots n$ mit insgesamt $p = n \lfloor \frac{n}{2} \rfloor$Prozessoren berechnen.

3. Die Lösung des Dreiecksystems nach den Koeffizienten $c_j, 1 \leq j \leq n$, läßt sich nach Sameh/Brent [82] in den Schritten

$$T_p = \frac{1}{2} \log^2 n + \frac{3}{2} \log n + 3 \in O(\log^2 n), \text{mit } p = \frac{n^3}{68} + O(n^2) \tag{4.14}$$

lösen.

4. Die Berechnung der Inversen $\mathcal{A}^{-1}$ aus Formel 4.9. kann mittels Rekursivem Doppeln in $\lceil \log n \rceil$ Schritten für die Summation, einen vorhergehenden Schritt für die Multiplikation mit den c_j und schließlich einer Division durch $-c_n$ in

$$T_p \lceil \log n \rceil + 2, \text{Schritten mit } p = n^3 \tag{4.15}$$

Prozessoren durchgeführt werden.

Unter Berücksichtigung der Schrittfolgen 1 bis 4 ergibt sich für den Csanky-Algorithmus eine Zeitkomplexität

$$T_p = \frac{3}{2} \log^2 n + O(\log n) \in O(\log^2 n) \tag{4.16}$$

mit $p = \frac{n^4}{2}$ Prozessoren.
Abschließend muß betont werden, daß die Berechnung der Koeffizienten $c_j, 1 \leq j \leq n$, extrem empfindlich gegenüber Rundungsfehlern ist, die bei der Spurberechnung entstehen. Wegen dieser inhärenten Instabilität genügt dieser Algorithmus kaum den Anforderungen der Praxis. Die Anwendung iterativer Methoden der Numerik sind weitaus praxisrelevanter. So berechnet der Algorithmus von Pan und Reif [71] mit nur n^3 Prozessoren in $O(\log^2 n)$ Zeitschritten sehr genaue Approximationen der Inversen. Dennoch hat Csanky ein exellentes theoretisches Ergebnis vorgelegt, das die Komplexität für die Lösung von Gleichungssystemen und der Inversion auf $O(\log^2 n)$ reduziert.

4.2 Neuronale Netze auf einer Prozessorfarm

Die Extrahierung wesentlicher Informationen aus Grauwertbildern mit Hilfe neuronaler Netze [72] ist ein Beispiel für die Anwendung paralleler Matrizenmultiplikation. Der Algorithmus sei auf einer Prozessorfarm zu implementieren. Zur Verfügung stand das Parsytec GCel1024 System des Paderborner Center for Parallel Computing mit 1024 Prozessoren, 4.4 GFlop/s peak performance und 4 GByte Speicher.

Beispiel 4.1 *Ein gegebenes Bild mit 256 Graustufen wird zunächst in quadratische Teilbilder der Dimension* 16×16 *zerlegt, welche als 256 Byte große Bildvektoren behandelt werden. Dieser Bildvektor von Graupunkten wird durch ein Neuronales Netz mit 45 Neuronen bewertet. Der 45-reihige Outputvektor entsteht durch die Multiplikation des Bildvektors mit einer Wichtungsmatrix vom Typ* $(256, 45)$.

Für die Berechnung $y = \mathcal{A}x$ sind nach Formel 2.3 22995 Multiplikationen und Additionen auszuführen. Bei einer floatingpoint performance eines Tranputers von 4.35 MFlop/s [49] ergibt sich eine Berechnungsdauer von 52 ms. Der nachfolgende Auszug zeigt den Berechnungsschritt, welcher über jedem Teilbild angewendet wird.

```
for (i=0; i<45; i++) {
  Output [i]=0.0;
  for (j=0; j<256; j++)
    Output [i]+=Picture [j]*WeightMatrics [i][j];
  for (j=0; j<256; j++)
    Output [i]-=Picture [j]*WeightMatrics [i][j]; }
```

Zur Kontrolle des Ergebnisses wird mit Hilfe des Output-Vektors und der Wichtungsmatrix wieder ein Bild generiert, welches mit dem Eingangsbild optisch identisch ist. Als Programmiermodell zur Parallelisierung wurde das Farmer-Prinzip

[48] angewendet. Dieses Prinzip ist dann sinnvoll, wenn ein Algorithmus mehrmals über einer Menge von Daten angewendet wird und die Berechnung einzelner Teilmengen unabhängig vom Ergebnis der anderen Teilmengen ist. In diesem Fall könnte der Algorithmus auch genauso oft gleichzeitig ausgeführt werden, wie sich die Grundmenge der Daten in Teilmengen zerlegen läßt.

```
PAR
  farmer()
  PAR i=0 FOR 4
    harvester(i)
```

Die logische Struktur kann folgendermaßen beschrieben werden: Ein farmer-Prozeß produziert kontinuierlich Nachrichten, welche über Kanäle zu den harvester-Prozessen geschickt werden, s. Abb. 4.4. Die Ergebnisse wiederum werden an den farmer-Prozeß zurückgesendet und dort zusammengesetzt.

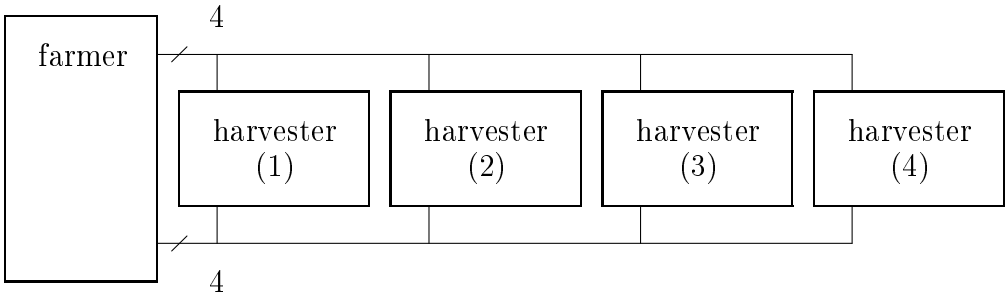


Abbildung 4.4: Farmer-Prinzip

Für ein bestehendes System aus p harvester-Prozessen muß die logische Struktur von p Kanälen des farmer-Prozesses auf die physische Struktur der Transputerarchitektur mit maximal 4 Links abgebildet werden. Das Farmer-Prinzip wurde auf einer Pipe-Topologie realisiert. Das erfordert folgende veränderte Prozeßstruktur: Der farmer-Prozeß teilt sich in einen Producer und Consumer. Die Aufgabenverteilung ist bereits durch den Namen der Prozesse spezifiziert. Desweiteren werden ein distributor- und collector-Prozeß eingeführt, s. auch Abb. 4.5.

Erhält ein distributor-Prozeß eine Nachricht, testet er zunächst, ob der harvester-Prozeß frei ist, wenn ja bekommt er die Daten, wenn nicht erfolgt die Weiterleitung.

Während die Implementierung der Matrizenmultiplikation trivial ist, sei eine Abschätzung der Parallelisierbarkeit des Problems im folgenden näher betrachtet. Bei den bisherigen Betrachtungen sind eine Reihe von Faktoren nicht beachtet worden, die den Speedup S_p vermindern: Kommunikationsverzögerungszeiten, Overheadzeiten des Betriebssystems, Interprozessorkommunikationszeiten

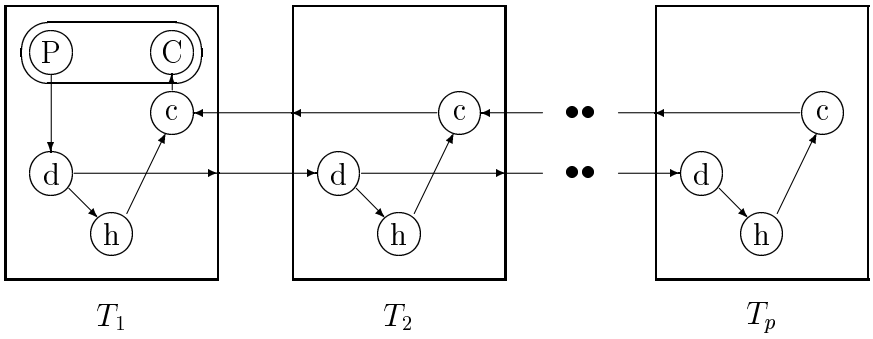


Abbildung 4.5: Prozeßstruktur

und auch Einflüsse durch den Programmierer selbst. $H(p)$ vereinigt nun die Summe aller Einflüsse in einem p-Prozessorsystem und ist sowohl maschinen- als anwendungsabhängig. Der Speedup berechnet sich dann zu

$$S_p = \frac{T_1}{T_p + H(p)}. \tag{4.17}$$

Eine geschlossene analytische Form ist für $H(p)$ i. a. nicht zugänglich. In Vereinfachung setzen Amdahl [3] und Gustafson [39] in ihren Speedup-Gesetzen $H(p) = 0$. Die Laufzeit eines parallelen Programms hängt im wesentlichen von der Abarbeitungszeit und von der Kommunikationszeit für den Datenaustausch ab. Die Kommunikation in einem MIMD-System mit verteiltem Speicher, erfolgt über ein Message-Passing-System. Für den Speedup gilt nun Formel 4.18

$$S_p = \frac{T_1}{T_p + T_c}. \tag{4.18}$$

In Bezug auf eine gegebene Architektur beschreibt T_c das Kommunikationsverhalten, [38].

$$\begin{aligned} T_c &= f(r, d, k_{nt}) \\ k_{nt} &= f(p, m, b) \end{aligned} \tag{4.19}$$

mit r - Datenübertragungsrate

d - Länge eines Datenpaketes

k_{nt} - Kommunikationsaufwand der Netzwerktopologie

p - Anzahl der Prozessoren

m - max. Anzahl atomarer Rechenschritte des Gesamtproblems

b - Bandbreite des Netzwerkes

Es sei T_c für eine Pipe-Topologie untersucht. Für die Pipe-Topologie beträgt der Durchmesser $n-1$ und die Bandbreite $b = 2(n-1)$. Der Kommunikationsaufwand innerhalb einer Pipe zum Verteilen und Einsammeln der m Datenpakete erfordert

$$\frac{m}{n} 2 \sum_{i-1}^{n} = m(n-1) \tag{4.20}$$

Schritte. Bei einer Bandbreite von $b = 2(n-1)$ reduziert sich der Kommunikationsaufwand um den Faktor $\frac{1}{b}$. Unberücksichtigt blieb bislang das initiale Auffüllen und Ausleeren der Pipe. Für den Gesamtkommunikationsaufwand ergibt sich

$$k_c = \frac{m}{2} + 2(n-1). \tag{4.21}$$

Für die Kommunikationszeit gilt nun

$$T_c = k_c t_h \tag{4.22}$$

mit $t_h = \frac{d}{r}$ als Übertragungszeit eines Datenpaketes zwischen zwei Knoten, unter der Voraussetzung, daß immer Datenpakete zur Übertragung vorhanden sind. Für die Berechnung des Speedup darf jedoch T_c nicht einfach mit der Rechenzeit T_p addiert werden, da die Kommunikation und die Berechnung auf dem Transputer weitgehend unabhängig voneinander ausgeführt werden. Für die Gesamtausführungszeit ist daher nur die längere der beiden Zeiten ausschlaggebend.

Beachte: Da die Rechenzeit mit der Anzahl der zur Verfügung stehenden Prozessoren proportional abnimmt, würden theoretisch unendlich viele Prozessoren keine Zeit zur Lösung des Problems benötigen. Dem wirkt entgegen, daß bei unendlich vielen Prozessoren die Kommunikation unendlich lange dauert. Daraus folgt, daß die Parallelisierung nur bis zu einer bestimmten Anzahl der einzusetzenden Prozessoren sinnvoll ist, siehe Abschnitt 1.1 einführendes Kartensortierbeispiel.

Zur Ermittlung der optimalen Anzahl von Prozessoren seien zunächst zwei Ansätze verfolgt:

- Die Kommunikationszeit T_c darf nicht größer werden als die Rechenzeit T_p,

$$T_p > T_c. \tag{4.23}$$

  Durch Ersetzen beider Terme und Umstellung läßt sich eine erste obere Grenze für die Anzahl n der sinnvoll einsetzbaren Prozessoren angeben.

- Innerhalb einer Pipe-Topologie gibt es Bereiche mit hohem Kommunikationsaufkommen, welche "hot spots" genannt werden. Dieser Effekt ist auf dem Kanal zwischen dem ersten und zweiten Prozessor zu erwarten, d. h. die Datenübertragungsrate ist in diesem Netzwerkabschnitt für den notwendigen Datendurchsatz zu gering. Der "hot spot" Effekt tritt nicht auf, so lange folgende Bedingung erfüllt ist

$$t_p > (n-1)t_h \tag{4.24}$$

$$t_h = \begin{cases} t_{hd} & : \quad t_{hd} > t_{hc} \quad \text{Übertragungszeit der Eingangsdaten zwischen zwei Knoten} \\ t_{hc} & : \quad sonst \quad \text{Übertragungszeit der Ergebnisdaten zwischen zwei Knoten} \end{cases} \tag{4.25}$$

  wobei t_p die Dauer der Berechnung eines Paketes der Länge d beschreibt und eine Funktion der Form

$$t_p = f(d, M) \tag{4.26}$$

  mit M ist floatingpoint performance.

Löst man die Ungleichungen, dann folgt aus Formel 4.23

$$n < -\frac{m}{8} + \frac{1}{2} \pm \sqrt{\frac{(m-4)^2}{64} + \frac{T_1}{2t_h}} \tag{4.27}$$

und Formel 4.24

$$n < \frac{t_p}{t_h} + 1. \tag{4.28}$$

Da der Algorithmus sich nicht beliebig zerlegen läßt, sondern nur bestimmte Paketgrößen sinnvoll sind, wurde für d zunächst eine Größe von 256 Byte Daten + 12 Byte Zusatzinformation gewählt. Mit Hilfe einer Implementierung, die nur zwei Prozessoren benutzt, wurden die Werte für t_h bzw. t_p experimentell ermittelt.

Für t_h wurde eine Zeit von 5.2 ms und für t_p eine Zeit von 200 ms gemessen. Mit $d = 256$ ergibt sich für m der Wert 1024 und für T_1 gemessene 200 s.
Mit diesen Werten wird Ungleichung 4.23 für $n < 61$ und Ungleichung für $n < 40$ erfüllt. Als Speedup ist unter diesen Bedingungen nur maximal 40 Prozessoren zu erwarten. Messungen wurden zunächst mit 14 Prozessoren durchgeführt. Tabelle 4.2 gibt ermittelte Laufzeiten auf einer Pipe-Topologie mit unterschiedlicher Anzahl von Prozessoren an.

Prozessor-anzahl	1	2	4	6	8	10	12	14
Laufzeit [s]	200.64	100.59	50.55	33.97	25.65	20.81	17.50	15.56
Speedup	1.00	1.99	3.97	5.91	7.82	9.64	11.46	12.89

Tabelle 4.2: Meßwerte der Laufzeit

Bei Untersuchungen auf einem System mit 80 verfügbaren Prozessoren haben sich folgende Resultate ergeben, s. Abb. 4.6

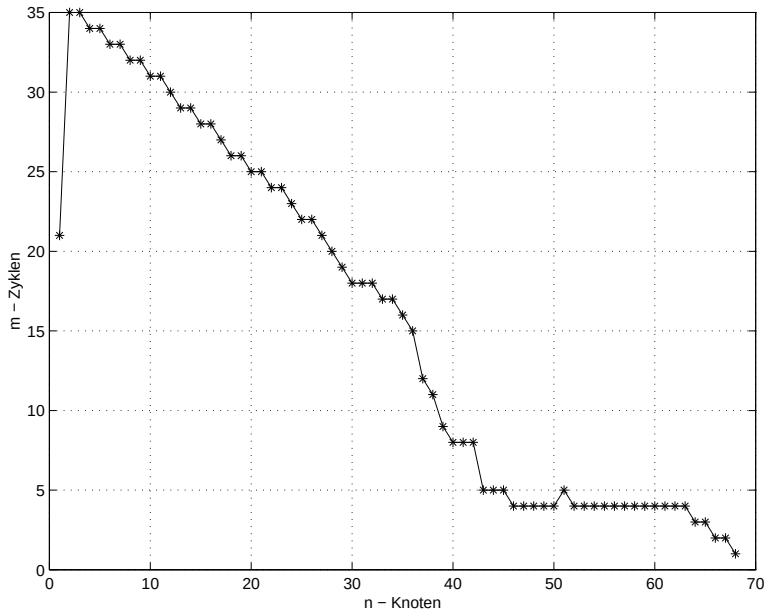


Abbildung 4.6: Lastverteilung bei der Berechnung mit 80 Prozessoren

Es ist zu erkennen, daß mehr als 60 Prozessoren bei dieser Topologie nicht beschäftigt werden können. Außerdem ist bei etwa 35 Prozessoren ein deutlicher Einbruch ("hot spot" Effekt) der Lastverteilung zu verzeichnen. Bei dieser Anzahl von Prozessoren ist die Kommunikationsleistung der Pipe-Topologie an ihrer Leistungsgrenze angelangt und eine Gleichverteilung der Last nicht mehr gewährleistet. Theoretisch kann die Prozessorzahl verdoppelt werden, wenn als Netzwerktopologie eine Baumstruktur verwendet wird. Auf Grund des erhöhten Rechenaufwandes der distributor- und collector-Prozesse hat sich diese Überlegung jedoch nicht bestätigt.

4.3 Matrizenmultiplikation auf einem Transputernetz

Der nachfolgend beschriebene Algorithmus läßt sich nach [15] ebenfalls als Farm klassifizieren, s. a. Abschnitt 4.2.
Für die Matrizenmultiplikation $\mathcal{AB}$ mit $\mathcal{A}$ vom Typ (m, n) und $\mathcal{B}$ vom Typ (n, q) verteilen sich die Aufgaben für den Master (farmer-Prozeß) und Slave (harvester-Prozeß) wie folgt, [42]:

- Der Master erzeugt m Slaves.
- Jeder der m Slaves ist mit einem Zeilenvektor der ersten Matrix initialisiert, wobei dem i-ten Slave der i-te Zeilenvektor der ersten Matrix zugeordnet wird.
- Jedem der m Slaves werden nacheinander alle q Spaltenvektoren der zweiten Matrix übergeben, wodurch schrittweise alle Skalare und somit der gesamte Zeilenvektor der Produktmatrix berechnet werden kann.
- Der Master übernimmt von den m Slaves die berechneten Zeilenvektoren und setzt diese zur gewünschten Produktmatrix zusammen, wobei wiederum der vom i-ten Slave gelieferte Vektor dem i-ten Zeilenvektor der Produktmatrix entspricht.

Eine auf einem realen Prozessorsystem abgebildete Netzwerktopologie ist durch die Anzahl von verfügbaren Prozessoren beschränkt. Dies impliziert Quasiparallelität. Ein Prozeß, welchem exklusiv ein Prozessor zugeteilt wurde, arbeitet nicht wie bisher über einem einzigen Zeilenvektor der Matrix, sondern über einem Matrixfragment, bestehend aus einem oder mehreren Zeilenvektoren. Die Anzahl der Vektoren je Matrixfragment berechnet sich als ganzzahliges Vielfache $\lfloor \frac{m}{p} \rfloor$, s. a. Abschnitt 3.3.

Die sequentielle Implementierung dient der Generierung von Vergleichswerten zur späteren Bewertung des Leistungszuwachses durch die einzelnen Stufen der Parallelisierung. Zielplattform ist auch hier ein Transputercluster (Multicluster 2/16) der Firma Parsytec. Als Entwicklungsplattform und Hostsystem diente eine SUN. Die single Prozeßstruktur zur Generierung der Vergleichswerte zeigt Abb. 4.7.

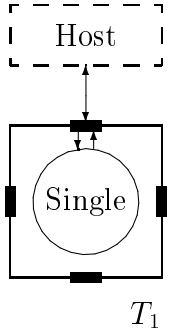


Abbildung 4.7: single Prozeßstruktur

Für die parallele Implementierung wurde eine vollständig ausgeglichene Baumstruktur gewählt, da diese dem Farming sehr ähnlich ist. Um bei der weiteren Erhöhung der Prozessorzahl unter Beibehaltung der Baumstruktur als Prozeßstruktur zu entsprechen, s. Abb. 4.8, ist die Einführung von Agents notwendig [10].

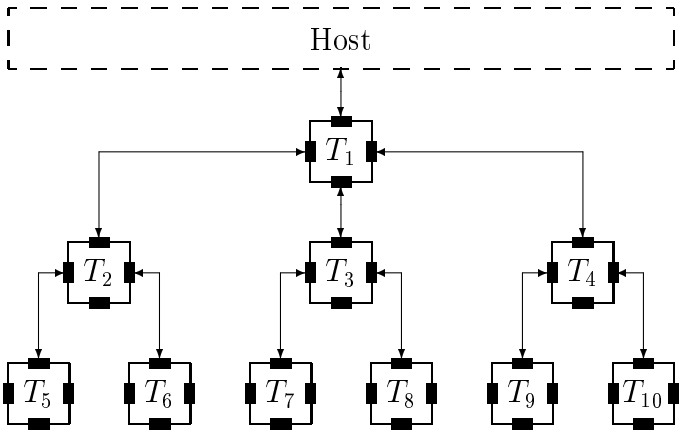


Abbildung 4.8: multi Prozeßstruktur

Ein Agent, s. Abb. 4.9, stellt hierbei die Vereinigung der Funktionalität von sowohl Master als auch Slave dar. Das ihm vom Master oder einem übergeordneten Agent zugeteilte Matrixfragment wird zur Bearbeitung auf diesem Agent untergeordneten Agents oder Slaves einschließlich sich selbst aufgeteilt. Daran schließen sich die Berechnung des eigenen Teilergebnisses, das Einsammeln der Ergebnisse der untergeordneten Komponenten und letztendlich das Weiterreichen des zusammengesetzten Ergebnisses an die übergeordnete Komponente an.

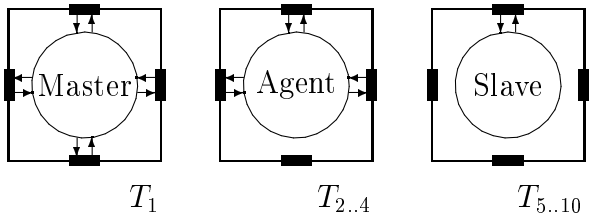


Abbildung 4.9: Master, Agent und Slave

Bei der Meßwertbestimmung des realen Laufzeitverhaltens der Applikation in Abhängigkeit von der Matrixdimension wurde von folgenden Voraussetzungen ausgegangen:

- keine Messung von Peripheriezugriffen durch Unterbindung von Ein- und Ausgabeoperationen über den Host,
- Messung der Zeit in Clock-Ticks ($\frac{1}{15625}s$), um im Bereich kleinerer Matrixgrößen eine entsprechende Genauigkeit zu erzielen,
- für die Messungen wurden nur quadratische Matrizen miteinander multipliziert,
- schrittweise Erhöhung der Matrixgröße n bis zur vollständigen Auslastung des limitierten Primärspeichers.

Tabelle 4.3 faßt wesentliche Ergebnisse der Implementierungen auf den skizierten Prozeßstrukturen zusammen.

Die Abbildung 4.10 zeigt die Abhängkeit des Speedup von der Matrixgröße für die in Abb. 4.8 angegebene Struktur, bestehend aus 10 Prozessoren und vernetzt in einer Baumstruktur.

Matrixgröße	10	50	100	200	1000	2000
t(Prozeßstruktur Abb. 4.7)	77	8742	68926	548644	68740740	549949934
t(Prozeßstruktur Abb. 4.8)	53	1552	10069	69132	7615521	59853034
Speedup	1.34	5.1	6.18	7.1	8.03	8.17

Tabelle 4.3: Meßwerte der Laufzeit in Clock-Ticks, wobei ein Clock-Tick $\frac{1}{15625}$s entspricht.

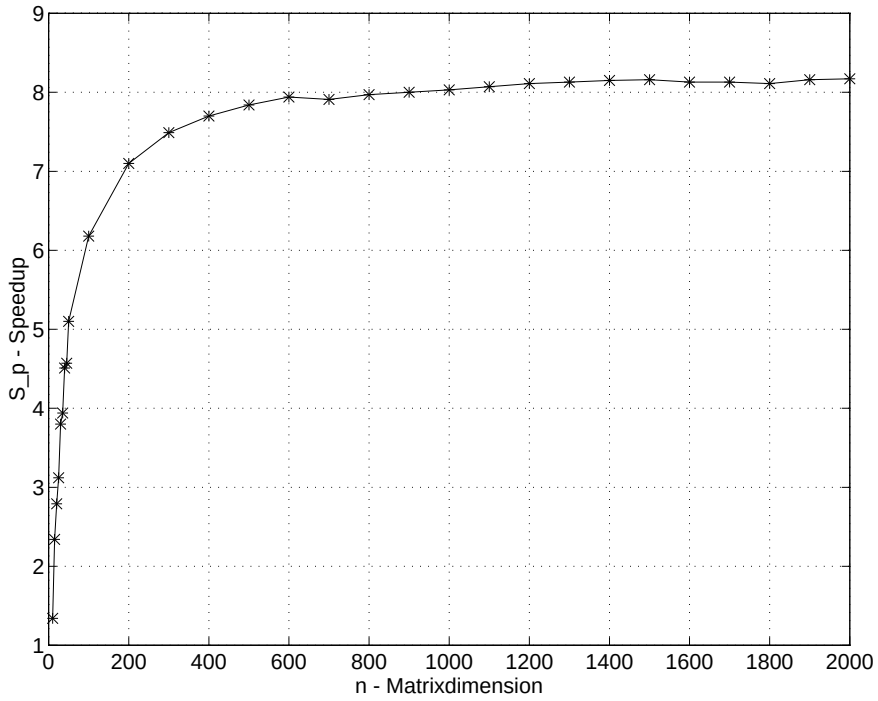


Abbildung 4.10: Abhängigkeit des Speedup von der Matrixgröße

4.4 Schnelle Fourier-Transformation und Multiplikation von Polynomen

Ausgangspunkt der Betrachtungen sei die Multiplikation zweier Polynome. Im Abschnitt 1.4 ist dieses Thema mit der Bemerkung abgeschlossen worden, daß die schnelle Fourier-Transfomation hierfür eine Komplexität von $O(n \log n)$ erreicht und somit die schnellste Möglichkeit bietet, zwei Polynome zu multiplizieren. Das folgende Schema 4.11 skizziert zum einen den primitiven Algorithmus zur Bildung des Produktpolynoms $r(x) = p(x) \cdot q(x)$ von der Komplexität $O(n^2)$ in Koeffizientendarstellung und ferner einen Algorithmus in Punktwertdarstellung, dessen Lösung eine Komplexität von $O(n \log n)$ erreicht.

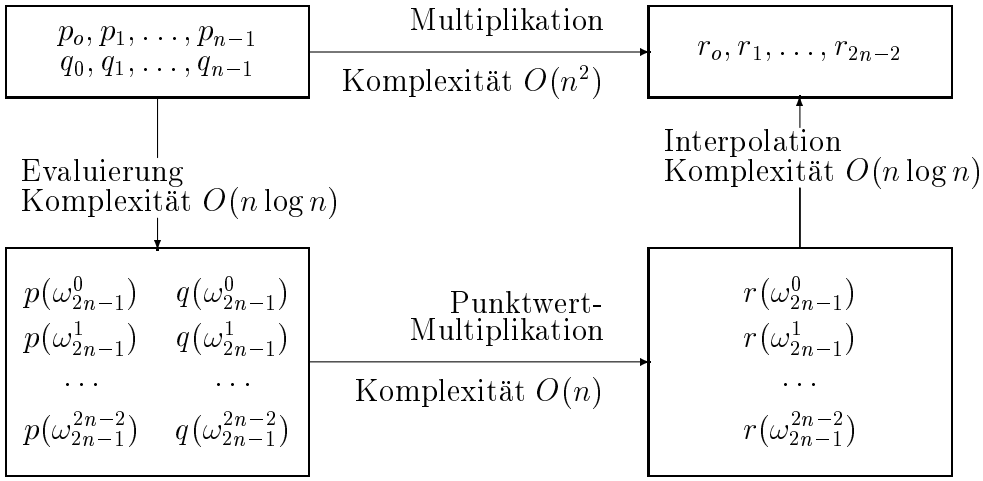


Abbildung 4.11: Graphische Darstellung zur effizienten Berechnung von Polynomen. ω_{2n-1} ist $(2n-1)$-te Einheitswurzel.

Die Polynome $p(x), q(x)$ werden an beliebigen $2n-1$ Punkten (Stützstellen) evaluiert, danach punktweise multipliziert und schließlich werden aus den $2n-1$ Werten die Koeffizienten des Produktpolynoms durch Interpolation rekonstruiert. Verwendet man das Horner-Schema zur Evaluierung und Lagrange zur Interpolation, so ergibt jede dieser Operationen eine Komplexität von $O(n^2)$. Der Trick nun besteht darin, als Stützpunkte die komplexen n-ten Einheitswurzeln zu verwenden. Eine komplexe Zahl ω ist n-te Einheitswurzel, falls $\omega^n = 1$. Es gibt im Komplexen genau n Lösungen der Gleichung $\omega^n = 1$. Dies sind die Zahlen $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. In der Evaluierungsphase sind Polynome vom Grad $n-1$ in $2n-1$ Punkten zu berechnen, d. h. $n-1$ Koeffizienten (Koeffizienten der Glieder mit dem höchsten Grad) haben den Wert null.

Der Algorithmus zur Berechnung eines Polynoms vom Grad $n-1$ in n Punkten beruht auf der Teile und Herrsche Strategie. Zunächst wird $p(x)$ in zwei Polynome vom Grad $\frac{n}{2}$ aufgespaltet.

$$\begin{aligned} p^{[0]}(x) &= p_0 + p_2 x + p_4 x^2 + \ldots p_{n-2} x^{\frac{n}{2}-1} \\ p^{[1]}(x) &= p_1 + p_3 x + p_5 x^2 + \ldots p_{n-1} x^{\frac{n}{2}-1} \end{aligned} \tag{4.29}$$

$p^{[0]}$ vereinigt die geraden und $p^{[1]}$ die ungeraden Koeffizienten und es gilt

$$p(x) = p^{[0]}(x^2) + x \cdot p^{[1]}(x^2) \tag{4.30}$$

Die n-ten Einheitswurzeln sind für diese Zerlegung geeignet, da man beim Quadrieren einer Einheitswurzel eine andere Einheitswurzel erhält. Tatsächlich gilt noch mehr: Beim Quadrieren einer n-ten Einheitswurzel erhält man für ein gerades n eine $\frac{1}{2}n$-te Einheitswurzel, eine Zahl, deren $\frac{1}{2}n$-te Potenz den Wert 1 hat. Eine Eigenschaft, die sich nutzbringend für den Teile und Herrsche Algorithmus verwenden läßt. Im Allgemeinen werden die Polynome $p^{[0]}(x), p^{[1]}(x)$ für die $\frac{n}{2}$ Einheitswurzeln rekursiv berechnet, was nur möglich ist, wenn n gerade. Ist n nun eine Zweierpotenz, dann bleibt n während der gesamten Rekursion gerade und die Aufspaltung kann solange durchgeführt werden, bis ein triviales Problem, nämlich die Auswertung eines konstanten Polynoms vorliegt, die Rekursion bricht für $n = 2$ ab. Die Anzahl der auszuführenden Multiplikationen genügt der fundamentalen rekursiven Beziehung für das Teile und Herrsche Prinzip und besitzt eine Zeitkomplexität

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n), \tag{4.31}$$

deren Lösung bekanntermaßen

$$T(n) = O(n \log n) \tag{4.32}$$

ist. Die Umwandlung des Polynoms aus seiner üblichen Darstellung mit n Koeffizienten in seine Darstellung mit Hilfe seiner Werte für die Einheitswurzeln ist die Fourier-Transformation und die rekursive Berechnung die schnelle Fourier-Transformation.

```
(* schnelle Fourier-Transformation *)
fft(p);
n:=length[p]
IF n=1 THEN RETURN p
  w_n:=e^2pi*i/n;
  w:=1;
  p[0]:=(po,p2,...,pn-2);
```

```
p[1]:=(p1,p3,...,pn-1);
y[0]:=fft(p[0]);
y[1]:=fft(p[1]);
FOR k:=1 TO n/2-1 DO
  yk:=yk[0]+w*yk[1];
  yk+(n/2):=yk[0]-w*yk[1];
  w:=w*wn;
RETURN y
```

Das gleiche Verfahren läßt sich auf allgemeinere Funktionen als Polynome anwenden. Man spricht dann von der diskreten Fourier-Transformation.

Definition 4.2 *Zu einem gegebenen Koeffizientenvektor* $p = (p_0, \ldots, p_{n-1})$ *bezeichnet der Vektor* $y = (y_0, \ldots, y_{n-1})$ *mit*

$$y_k = \sum_{j=0}^{n-1} p_j \cdot \omega_n^{kj} \tag{4.33}$$

die diskrete Fourier-Transformation von p*.*

Die Berechnung von y bedeutet das Evaluieren von p an den n-ten Einheitswurzeln, $y = DFT_n(p)$. Für das Interpolieren ist die inverse diskrete Fourier-Transformation auszuführen. Diese unterscheidet sich geringfügig (ein zusätzlicher Faktor) von der diskreten Fourier- Transformation.

$$p_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \cdot \omega_n^{-kj} \tag{4.34}$$

Die inverse diskrete Fourier-Transformation kann so ebenfalls in $O(n \log n)$ gelöst werden. Symbolisch kann nun die Polynommultiplikation, die sich mit

$$T(n) = 2n \log n + O(n) \in O(n \log n) \tag{4.35}$$

komplexen Multiplikationen lösen läßt, wie folgt ausgedrückt werden: DFT sei ein Operator, der den Koeffizientenvektor p in den Wertevektor y und DFT^{-1} umgekehrt y nach p überführt, dann gilt für den Koeffizientenvektor des Produktpolynoms:

$$r = DFT^{-1}(DFT(p) * DFT(q)). \tag{4.36}$$

Hierbei bedeutet "*" die komponentenweise Multiplikation der beiden Vektoren also $(y_0, \ldots, y_{n-1}) * (y_0', \ldots, y_{n-1}') = (y_0 y_0', \ldots, y_{n-1} y_{n-1}')$.

Die Teile und Herrsche Strategie liefert zusammen mit einigen Eigenschaften komplexer n-ter Einheitswurzeln einen eleganten Ansatz für die schnelle Fourier-Transformation und die effiziente Implementierung der Polynommultiplikation.

Die diskrete Fourier-Transformation läßt sich äquivalent durch eine Vektor-Matrix-Multiplikation beschreiben.

$$\vec{y} = V_n \cdot \vec{a} \tag{4.37}$$

mit V_n - ist Vandermonde'sche Matrix, mit den i, j Einträgen ω_n^{kj}, für $(j, k = 0 \ldots n-1)$.

Bei Anwendung des Vektor-Matrix-Multipikationsalgorithmus auf einem Netz von Bäumen, s. Abschnitt 3.4, ist es möglich, die Fourier-Transformation in $O(\log n)$ Schritten mit $O(n^2)$ Prozessoren zu lösen. Im folgenden sei eine Struktur vorgestellt, die eine parallele Implementierung auf nur $O(n)$ Prozessoren mit gleicher Zeitkomplexität zuläßt. Aus Gründen der Effizienz wird man bei Anwendungen meist an einer iterativen Implementierung interessiert sein. Hierzu sei der oben angegebene Algorithmus näher betrachtet. Die `FOR` Schleife läßt sich idealerweise in einer Butterfly-Operation ausführen, s. Abb. 4.12, was von entsprechenden graphischen Darstellungen suggeriert ist.

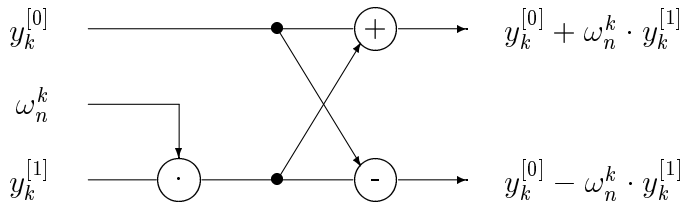


Abbildung 4.12: Eine Butterfly Operation.

Eine solche Butterfly-Operation kann auch als kombinatorische Schaltung interpretiert werden kann.

Beispiel 4.2 *Zur Illustration sei $n = 8$ gewählt. Formel 4.37 gestaltet sich in der Form*

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_7 \end{pmatrix} = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n & \omega_n^2 & \cdots & \omega_n^7 \\ \omega_n^0 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2\cdot 7} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^7 & \omega_n^{2\cdot 7} & \cdots & \omega_n^{7\cdot 7} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_7 \end{pmatrix} \tag{4.38}$$

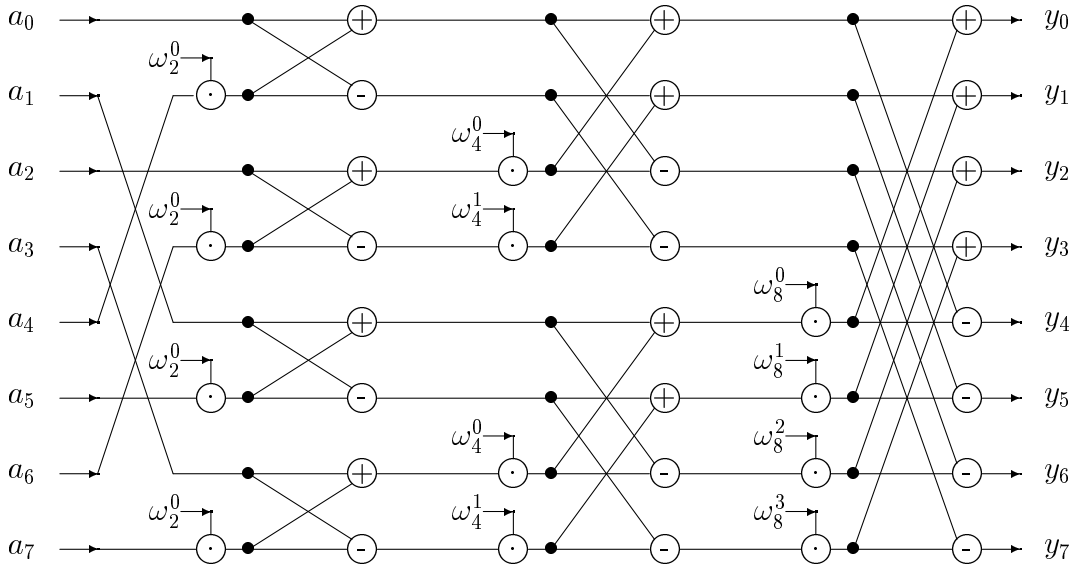


Abbildung 4.13: Kombinatorische FFT für $n = 8$, s. a. [21]

Man beachte die Periodizität der Einheitswurzel und so ergibt sich für $n = 8$ die kombinatorische Schaltung , s. Abb. 4.13.

Die schnelle Fourier-Transformation der Länge n kann mit $\frac{n}{2}(\log n)$ Butterfly-Knoten in $O(\log n)$ Takten berechnet werden, wenn man einen Takt pro Butterfly-Operation ansetzt. Es sollte erwähnt werden, daß die schnelle Fourier-Transformation für Spezialanwendungen vielfach bereits hardwaremäßig implementiert worden ist. Sie ist Grundlage für wesentliche Operationen bei der Signalverarbeitung, s. a. [57].

4.5 Numerische Resultate

In diesem Abschnitt werden die mit den im Kapitel 2 und 3 beschriebenen Verfahren erzielten numerischen Ergebnisse (insbesondere Zeitmessungen) auszugsweise vorgestellt und diskutiert. Zur Vollständigkeit wird auf [102] verwiesen. Die Messungen wurden auf verschiedenen Plattformen wie Challenge SC800 - TU Ilmenau, GCel1024 - Paderborn und Multicluster 2/16 - TU Ilmenau durchgeführt. Da Hardware und auch Compilerversionen nicht identisch sind, sind die Ergebnisse unterschiedlich zu bewerten.

Die Challenge SC800 ist ein Vertreter der MIMD-Rechner mit gemeinsamen Speicher. Die Konfiguration bestand aus 4 Prozessoren mit nur 500 kByte Cache. Die Parallelisierung der Algorithmen erfolgte mit PowerC. Untersucht worden ist die Matrizenmultiplikation nach dem “naiven” Algorithmus und dem Algorithmus

nach Strassen, sowohl seriell als auch parallel mit einfacher und doppelter Genauigkeit. In Tabelle 4.4 soll gezeigt werden, wie sich die gemessene Laufzeit für den "naiven" und den Strassen-Algorithmus verhält, wenn gleichzeitig Matrixgröße und Prozessorzahl variiert werden.

n	"naiv"		Strassen		"naiv" $p = 3$
	single	double	single	double	double
64	0.04	0.04	0.22	0.24	0.01
128	0.35	0.32	1.6	1.6	0.11
256	2.9	3.8	11.5	12	1.2
512	37	198	79	85	70
1024	1667	1803	566	625	630

Tabelle 4.4: Laufzeitmessungen für den "naiven" und Strassen Algorithmus

Der parallele Algorithmus mit $p = 3$ Prozessoren erzielt zum sequentiellen äquivalent einen Speedup von $S_p \sim 3$. Sehr gut ist auch das Verhalten des Strassen Algorithmus gegenüber dem "naiven" Algorithmus erkennbar. Mit steigender Matrixgröße nimmt die Berechnungszeit gegenüber dem "naiven" Algorithmus ab. Bereits für eine Matrixgröße von $n = 512$ erzielt Strassen ein besseres Laufzeitverhalten. Abb. 4.14 veranschaulicht die durchschnittliche Rechenzeit für die Berechnung in doppelter Genauigkeit.

Im Vergleich dazu ist eine Implementierung von Strassen auf einem Pentium III-450 mit 128 MB RAM unter Windows NT erfolgt. Es wurden Berechnungen nur bis zu einer Matrixgröße $n = 2000$ ausgwertet, da hier noch keine Verzögerungen durch "Swappen" auftraten. Abb. 4.15 faßt die erzielten Ergebnisse zusammen. Strassen ist durchführbar, wenn die Matrixgröße einer Zweierpotenz entspricht, so daß untersuchte Matrizen auf die notwendige Größe aufgefüllt wurden. Deutlich ist das "Treppen"-Verhalten sichtbar, s. a. Abb. 2.1.

Der "mixed"-Algorithmus ist dahingehend optimiert, daß er bis zu einer Schwelle (hier $n = 128$) nach Strassen und bei Unterschreitung nach dem "naiven" Algorithmus die Matrizenmultiplikation ausführt. Für diese Implementierung sind die besten Ergebnisse erzielt worden.

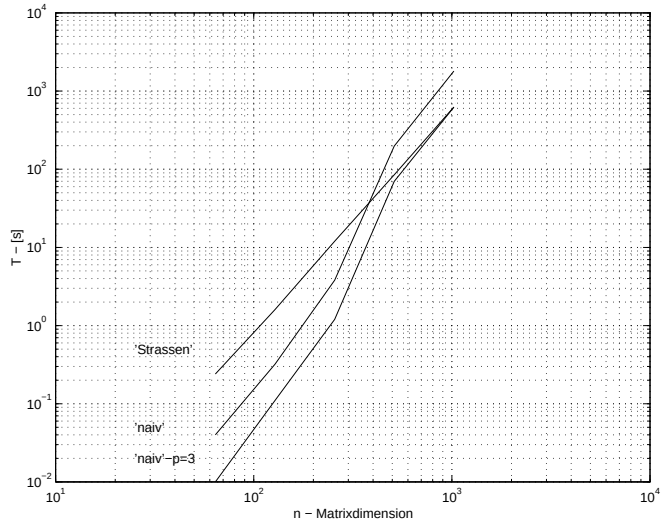


Abbildung 4.14: Rechenzeit in Abhängigkeit von der Matrixgröße, SC 800

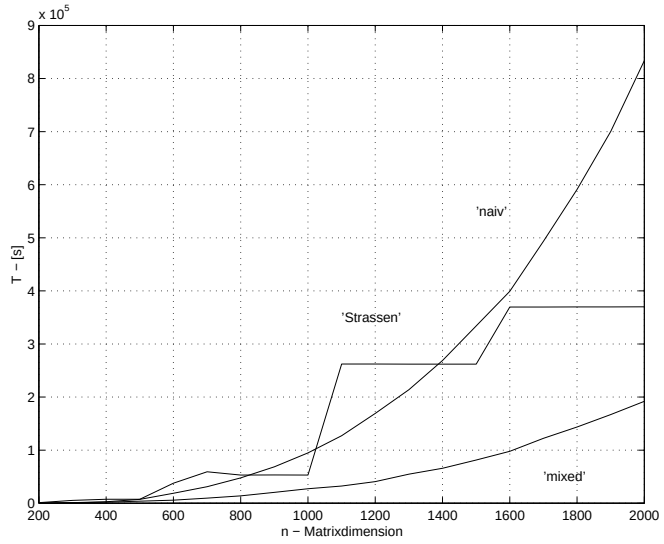


Abbildung 4.15: Rechenzeit in Abhängigkeit von der Matrixgröße, Pentium III

Ausgangspunkt einer parallelen Implementierung von Strassen ist die Idee, die 7 Multiplikationen, s. a. Formeln 2.19 und 2.20, gleichzeitig ausführen zu lassen. Abb. 4.16 zeigt die Verteilung der Teiloperationen auf eine Prozessorarchitektur. Hier wiederum ist eine Butterfly-Architektur, s. Abschnitt 4.4 möglich.

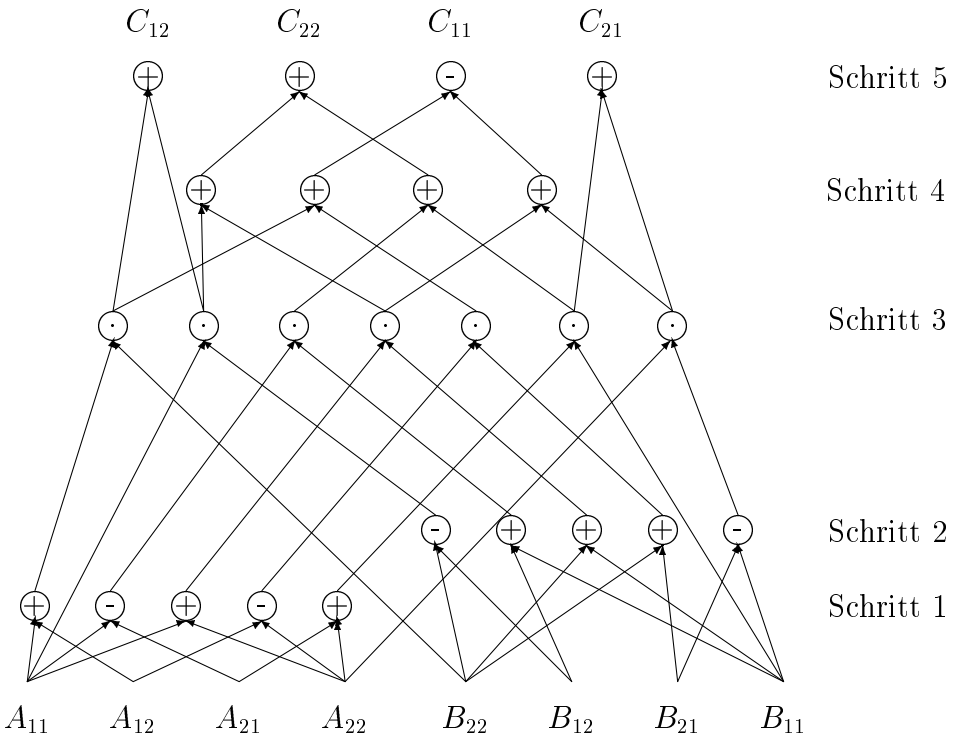


Abbildung 4.16: Verteilung der Teiloperationen

Die Abb. 4.16 ist von unten nach oben zu lesen. Zunächst werden auf 5 Prozessoren jeweils zwei Matrizen $\mathcal{A}_{ik}$ und danach zwei Matrizen $\mathcal{B}_{ik}$ addiert. In Schritt 3 werden diese Summen durch rekursives Aufrufen des Strassen Algorithmus multipliziert. In Schritt 4 und 5 werden die Zwischenergebnisse addiert. Eine Grenzwertbetrachtung ermittelt für $n = m \cdot 2^k$ mit $m = 1$, s.a. Abschnitt 2.3 einen Speedup von

$$S_p = \frac{T_1}{T_p} = \lim_{k \to \infty} \frac{7(7^k - 6 \cdot 2^{2k})}{(7^k + 2^{2k}} = 7. \tag{4.39}$$

Die erzielten Ergebnisse auf dem Multicluster sind gemittelt worden und in Abb. 4.17 dargestellt.

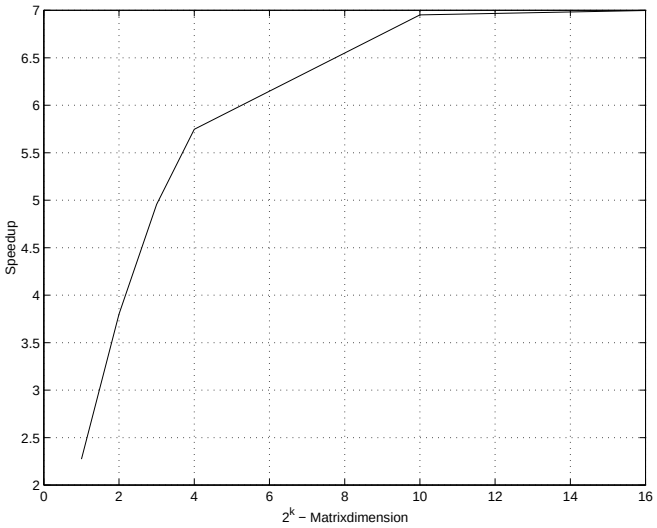


Abbildung 4.17: Rechenzeit in Abhängigkeit von der Matrixgröße, Multicluster

4.6 Standard-Bibliotheken

Eine Vielzahl der Anwendungen, die für die Parallelverarbeitung von Interesse sind, liegen im "High Performance Computing" Bereich, erfordern also die Rechenleistung von Hochleistungsrechnern. Durch die Bereitstellung numerischer Bibliotheken, einheitlich konzipierter Subroutinen, ist die Anwendersoftware effizient auf einem breiten Spektrum von Hochleistungsrechnern lauffähig. In den vergangenen 25 Jahren sind unter Leitung von J. J. Dongarra eine Vielzahl von bekannten Software Bibliotheken entwickelt worden: EISPACK, LINPACK, LAPACK und BLAS. Als Ergänzung ist ScaLAPACK zu nennen, einer skalierten Version von LAPACK für MIMD-Architekturen mit verteiltem Speicher.

- **EISPACK** ist ein Paket von 70 FORTRAN Subroutinen zur Lösung von Eigenwert/- vektor Problemen bei Matrizen [86, 35]. Darüber hinaus enthält EISPACK auch spezielle Subroutinen zur singulären Zerlegung von Matrizen. EISPACK besteht hauptsächlich aus ALGOL Prozeduren, entwickelt bereits im Jahre 1960.
- **LINPACK** ist ein Paket von ca. 50 FORTRAN Subroutinen zur Lösung von linearen Gleichungssystemen [24]. Mit LINPACK können Matrizen ana-

lysiert werden (Berechnung der Determinanten, Abschätzung der Konditionszahl). Das Paket enthält auch Routinen für nicht quadratische Matrizen (Singuläre Zerlegung von Matrizen, least squares Lösungen von linearen Gleichungssystemen).

- **LAPACK** steht für Linear Algebra Package [4] und beinhaltet unter anderem Routinen zur Lösung von linearen Gleichungssystemen, über- und unterbestimmten Eigenwert-Problemen, einfach und verallgemeinert und von Singulärwert-Problemen. Die neue Bibliothek erweitert die erfolgreichen EISPACK und LINPACK Bibliotheken, indem sie die Funktionalität der beiden Bibliotheken integriert. Es wurde großer Wert darauf gelegt, die Eignung der LAPACK Algorithmen für heutige Hochleistungsarchitekturen zu erhöhen. Einen speziellen Schwerpunkt bilden Mehrprozessorsysteme mit gemeinsamen Speicher oder Vektorrechner mit einer großen Anzahl von Pipes. Schließlich liefert die Genauigkeit und Robustheit der Programme einen Maßstab zur Bewertung konkurrierender Implementierungen und Algorithmen und dient als Benchmark zum Vergleich der Rechengeschwindigkeiten verschiedener Rechner.

- **ScaLAPACK** ist seit 5 Jahren verfügbar und stellt eine skalierte Version der LAPACK Bibliothek für MIMD-Rechner mit verteiltem Speicher und Workstationclustern dar. Um eine einfache Portierung auf verschiedenste Rechnerarchitekturen und Kommunikationsbibliotheken zu ermöglichen, wurden interne Schnittstellen definiert, PBLAS und BLACS.

  PBLAS steht für Parallel Basic Linear Algebra Subprograms und stellt eine Erweiterung der bekannten BLAS (Basic Linear Algebra Subprograms) für Parallelrechner mit verteiltem Speicher dar. In den PBLAS sind also z. B. Funktionen für Matrix-Matrix- und Matrix-Vektor-Operationen enthalten, wobei die Matrizen und Vektoren auf die lokalen Speicher der einzelnen Prozessoren verteilt sind. Um eine möglichst effiziente Nutzung der Einzelprozessoren zu gewährleisten, werden für die lokalen Rechenoperationen soweit wie möglich BLAS-Routinen eingesetzt, die in der Regel in einer optimierten Version vorliegen.

  Für die Kommunikation werden BLACS (Basic Linear Algebra Communication Subprograms) verwendet, die eine einfache Schnittstelle zum Versenden und Empfangen von rechteckigen oder trapezförmigen Matrizen und Teilmatrizen enthalten. Die BLACS setzen wiederum auf Message Passing Primitiven wie MPI (message passing interface), MPL (message passing library) oder PVM (parallel virtuel machine) auf. Das Schichtenmodell von ScaLAPACK wird durch Abb. 4.18 illustriert.

- Die **BLAS** bilden eine Schnittstelle für elementare Matrix- und Vektoroperation. Ihre Verwendung trägt zur Übersichtlichkeit und Portabilität

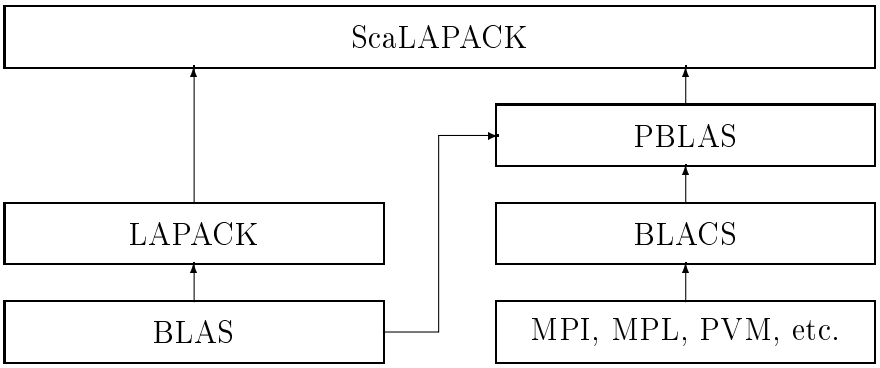


Abbildung 4.18: ScaLAPACK Softwarehierarchie

von Programmen bei. Oft liegen die BLAS auf den jeweiligen Rechnern in auf die Prozessorarchitektur hin optimierter Form vor, so daß bei ihrer Benutzung die Ressourcen des zugrundeliegenden Rechners gut ausgenutzt werden. Es werden 3 BLAS-Level unterschieden:

- Level 1 BLAS [62, 61]
  (sind Vektor-Vektor-Operationen, Operationen von $O(n)$ über $O(n)$ Daten)
- Level 2 BLAS [28, 27]
  (sind Matrix-Vektor-Operationen, Operationen von $O(n^2)$ über $O(n^2)$ Daten)
- Level 3 BLAS [25, 26]
  (sind Matrix-Matrix-Operationen, Operationen von $O(n^3)$ über $O(n^2)$ Daten)

Level 1 BLAS implementieren allgemein Vektor-Vektor-Operationen wie das Skalarprodukt (_AXPY-Operationen)

$$y := y + \alpha x, \tag{4.40}$$

wobei x und y Vektoren sind und α ein Skalar. Soll beispielsweise ein Matrix-Vektor-Produkt $y = \mathcal{A}x$ berechnet werden, so kann dies u. a. mit Hilfe der _AXPY-Operation 4.40 durchgeführt werden. Es sei $\mathcal{A} = (a_1, \ldots, a_n) \in \mathbb{R}^{n \times n}$ mit a_j bezeichnet die j-te Spalte von $\mathcal{A}$ und $x \in \mathbb{R}^n$., dann gilt

$$y = \sum_{j=1}^{n} a_j x(j). \tag{4.41}$$

Es ergibt sich y, indem für $j = 1, \ldots, n$ die Zuweisung $y := x(j)a_j + y$ ausgeführt wird. Diese stellt jeweils eine _AXPY dar. Die hohe Rechengeschwindigkeit der Vektorprozessoren kann jedoch nur erreicht werden, wenn der Prozessor schnell genug mit den erforderlichen Daten versorgt wird. Da der Zugriff auf den Hauptspeicher i. a. zu lange dauert, werden die Prozessoren über schnelle Zwischenspeicher (Vektorregister, Cache) versorgt. Es ist Sorge zu tragen, daß die Rechenleistung des Prozessors nicht durch eine zu langsame Versorgung der schnellen Zwischenspeicher aus dem Hauptspeicher reduziert wird. Elemente der aus dem Hauptspeicher zu ladenden Vektoren sollten aufeinanderfolgend abgelegt sein. Das bedeutet, daß möglichst mit Spalten statt mit Zeilen von Matrizen gearbeitet werden sollte. Daten in den Vektorregistern bzw. Cache sollten so oft wie möglich wiederverwendet werden, um unnötige Speicherzugriffe zu vermeiden.

Bei der Berechnung des Matrix-Vektor-Produktes könnte während der Ausführung der Schleife der Vektor y die ganze Zeit in einem Vektorregister gehalten werden. In jedem Schritt müßte dann nur noch die Spalte a_j von $\mathcal{A}$ und eventuell die j-te Komponente von x geladen werden. Jeder Aufruf der BLAS 1 Routine _AXPY bewirkt jedoch ein erneutes Laden und Zurückschreiben von y. $3n^2$ Zugriffe auf den Hauptspeicher stehen $2n^2$ Operationen gegenüber. Es bietet sich der Übergang zu BLAS 2 Routinen für Matrix-Vektor-Operationen an.

$$y := \beta y + \alpha \mathcal{A} x \tag{4.42}$$

Die Entwicklung der Level 2 BLAS wurde durch vektorverarbeitende Maschinen motiviert. In einer einzigen Matrix-Vektor-Produkt Routine kann der Zugriff auf die Vektoren optimiert werden. Für die Matrix $\mathcal{A}$ werden n^2 Speicherzugriffe gegenüber $2n^2$ arithmetischen Operationen benötigt. Will man nun das Produkt $\mathcal{C} = \mathcal{A}\mathcal{B}$ zweier quadratischer Matrizen mit Hilfe von BLAS 2 Aufrufen berechnen, so muß für jede Spalte von $\mathcal{C}$ eine Matrix-Vektor-Multiplikation von $\mathcal{A}$ mit der entsprechenden Spalte von $\mathcal{B}$ durchgeführt werden. $\mathcal{A}$ müßte also insgesamt n mal geladen werden, so daß insgesamt n^3 Elemente aus dem Hauptspeicher geholt werden müßten. Nehme man an, $\mathcal{A}$ passe in einen sehr schnellen (Cache) Speicher, so würde sich der Datenzugriff auf $3n^2$ Elemente beschränken. Die Anzahl der durchgeführten Operationen wäre dagegen $2n^3$, so daß sich Speicherzugriffe und Anzahl der Operationen wie $O(\frac{1}{n})$ verhalten (Surface-to-Volumne-Effekt, s. a. [9, 29]). Dies würde bedeuten, daß der Zugriff auf den Hauptspeicher keinen großen Einfluß mehr auf die erzielte Rechenleistung hat. Tabelle 4.6 faßt das Verhältnis von Speicherzugriffen zu arithmetischen Operationen (Rechendichte) für ein "SAXPY", eine Matrix-Vektor- und eine Matrix-Matrix-Multiplikation für n Vektoren und (n, n) Matrizen zusammen.

	Level 1 BLAS	Level 2 BLAS	Level 3 BLAS
Rechendichte	3:2	1:2	1.5:n

Tabelle 4.5: Rechendichte für BLAS

In der Regel werden allerdings Matrizen nicht ganz in einen solchen Cache passen. Beim Übergang zu BLAS 3-Routinen

$$\mathcal{C} := \alpha\mathcal{A}\mathcal{B} + \beta\mathcal{C} \tag{4.43}$$

können diese aber intern so implementiert werden, daß der Cache möglichst gut ausgenutzt wird. Die Matrix wird dabei in kleinere, zusammenhängende Teilmatrizen (Blöcke) zerlegt, die in den Cache passen. Die Berechnungen einer Routine setzen sich dann aus Verknüpfungen dieser Teilmatrizen miteinander zusammen.

Sei beispielsweise $n = o \cdot p$, und $\mathcal{A}, \mathcal{B}, \mathcal{C}$ seien jeweils in p^2 Teilmatrizen vom Typ (o, o) zerlegt. o ist also die Größe eines Teilblocks der Matrix, o ist Blockgröße. Die Matrizenmultiplikation kann dann in der BLAS 3-Routine beispielsweise nach folgendem Algorithmus realisiert werden, s. a. [34].

```
FOR i=1 TO p
  FOR j=1 TO p
    FOR k=1 TO p
      C(i,j):=C(i,j)+A(i,k)B(k,j)
    END (* for *);
  END (* for *);
END (* for *);
```

Ist nun im Cache Platz für 3 Blöcke der Größe (o, o), so müssen in der innersten Schleife zwei Matrizen der Größe (o, o) in den Cache geladen werden, während $C(i, j)$ während des Durchlaufs dieser Schleife immer im Cache gehalten werden kann. Insgesamt hat man also einen Zugriff auf $\frac{2n^3}{o}$ Elemente im Hauptspeicher gegenüber $2n^3$ Operationen.

An dieser Stelle sei noch einmal betont, daß die Anpassung der BLAS an die Prozessorarchitektur innerhalb der Routinen stattfindet. Der Nutzer der Bibliothek hat wegen der Anpassung keine Sorge zu tragen. Sollen schnelle Speicher gut ausgenutzt werden, so sollte man Algorithmen so formulieren, daß der Hauptanteil der Rechenoperationen innerhalb solcher Matrix-Matrix-Operationen ausgeführt werden kann. Man arbeitet dann

nicht mehr mit einzelnen Elementen oder Vektoren, sondern mit Matrizen, indem man die Ausgangsmatrix in mehrere Teilmatrizen (Blöcke) zerlegt. Solche Algorithmen werden auch deshalb Blockalgorithmen [84] genannt. In der LAPACK-Bibliothek sind die Verfahren so realisiert, daß ein möglichst großer Anteil der Rechenarbeit innerhalb von BLAS 3-Routinen stattfinden kann. Bei Verwendung dieser Routinen kann daher i. a. eine relativ hohe Rechenleistung erzielt werden.

Kapitel 5

Zusammenfassung

Die Arbeit entstand angeregt durch die Forschungstätigkeit auf dem Gebiet System Level Design mit der Anwendung in Simulation komplexer Systeme und Implementation von Echtzeitsystemen. In beiden Fällen führen eine Vielzahl der Algorithmen auf die Matrizenmultiplikation zurück. Sei es nun beispielsweise die Beschreibung von großen Modellen mit Hilfe von Matrizen oder die effiziente Implementierung von Echtzeitalgorithmen (Bildkompression [78]). In Anlehnung an Pan [70] stellt die Arbeit eine sinnvolle Erweiterung dar. Im Vordergrund stand eine umfassende Darstellung von Algorithmen für die Matrizenmultiplikation, die vor allem von praktischem Interesse sind. Insbesondere erfolgte eine Gegenüberstellung von repräsentativen sequentiellen und originären parallelen Algorithmen und ihr Bezug zur Rechnerarchitektur. Erst die ganzheitliche Betrachtung von Algorithmus, Architektur, Programmiersprache und Anwendung erlaubt eine effiziente Lösung. Generell muß hier festgestellt werden, daß nur sehr wenige Sprachentwürfe programmiersprachliche Instrumente zur Parallelisierung oder gar Softwaretechnologien enthalten, die dem Potential von Parallelarchitekturen angemessen sind.

Algorithmen und numerische Verfahren für Vektorprozessoren und Supercomputer wurden hier nicht explizit behandelt. Diese Systeme weisen Parallelismus meist in der Form von arithmetischen Pipeline-Strukturen auf. Die Programmierung richtet sich eng an den architektonischen Charakteristiken wie Vektorregister und speziellen Funktionseinheiten aus. Die sprachlichen Möglickeiten sind dabei in unterschiedlicher Weise begrenzt und werden meist von autoparallelisierenden Compilern übernommen.

Abschließend seien zwei Sachverhalte kurz diskutiert. Parallelrechner können dazu dienen, gegebene Probleme schneller zu lösen oder aber in gegebener Zeit größere Probleme zu lösen. Die in Verbindung damit stehenden Begriffe Speedup und Scaleup werden im folgendem Abschnitt diskutiert. Das Kapitel schließt ab mit einer Betrachtung zu den Kategorien Algorithmus und Architektur.

5.1 Speedup und Scaleup

Parallelrechner sind in der Vergangenheit wiederholt hinter den in sie gesetzten Erwartungen zurückgeblieben. Der Grund war jedesmal, daß ihre Leistung von der nächsten Generation der Singleprozessormaschinen erreicht oder übertroffen wurde. Zum einen sind diese einfacher zu bauen und zum anderen einfacher zu bedienen. Aus physikalischen Gründen läßt sich dieses Spiel

"Parallelrechner sind die Rechner der Zukunft und werden es immer sein."

nicht endlos wiederholen und deshalb ist die Entwicklung paralleler Rechner und geeigneter paralleler Algorithmen sinnvoll und dringend erforderlich. Insbesondere haben Parallelrechner ihre Berechtigung zur Erreichung eines guten Scaleup, d. h. Leistungssteigerung durch Parallelität, indem in gegebener Zeit größere Probleme gelöst werden. Dem Erreichen eines Speedup (gegebene Probleme durch Parallelität schneller zu lösen) sind dagegen ebenfalls natürliche Grenzen gesetzt. Daraus folgt auch, daß das Potential von Rechnern der Teraflop-Klasse vor allem der Erschließung sehr großer Probleme dient und nicht der noch schnelleren Erledigung "kleiner" Probleme. Folgendes Problem bleibt bestehen: Es gibt Algorithmen, die inhärent sequentiell sind und die durch mehr Hardwareeinsatz also höchstens verlangsamt werden können. Zudem existieren Probleme aus dem Bereich der Suche und Optimierung, wo durch Beweis klar ist, daß die Kosten der zugehörigen Algorithmen exponentiell mit der Problemgröße wachsen, während ein Parallelrechner bestenfalls einen linearen Speedup erbringen kann, die Lücke bleibt also prinzipiell bestehen.

Das am häufigsten verwendete Maß für die Leistungsbewertung paralleler Algorithmen ist der Speedup, Formel 1.21. 1967 hat Amdahl [3] das Gesetz des fixed load Speedup für den Spezialfall hergeleitet, daß der Rechner entweder im sequentiellen oder im perfekt parallelen Modus am Problem arbeitet. In einer normalisierten Version mit α als sequentiellen, nicht parallelisierbarem und entsprechend $1-\alpha$ als parallelisierbarem Anteil, kann Amdahl's Gesetz auch in der Form

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} = \frac{p}{1+(p-1)\alpha} \tag{5.1}$$

geschrieben werden. Der Verlauf dieser Funktion ist wenig ermutigend, denn es gilt der Amdahl'sche (sequentielle) Flaschenhals, d. h. der Speedup ist nach oben durch $\frac{1}{\alpha}$ beschränkt, unabhängig von der Anzahl der Prozessorknoten. Es gibt eine Vielzahl von Messungen, die Amdahl's Gesetz scheinbar widerlegen. Es wird ein Speedup nahe p gemessen, ohne daß der sequentielle Anteil α verschwindend klein ist. Gustafson [39] schlägt hier den sogenannten scaled Speedup vor und leitet folgende Formel ab.

$$S_p = \alpha + p(1-\alpha) = p - \alpha(p-1) \tag{5.2}$$

Er betrachtet das Parallelprogramm mit seinem sequentiellen und parallelen Zeitanteil und skaliert letzteren in linearer Weise, p mal. Dieses nun lineare Gesetz gilt unter der recht kritischen Annahme, daß der relative sequentielle Anteil im Verhältnis zur Parallelität sinkt. Mit anderen Worten, Gustafson's Voraussetzung ist, daß mit steigender Parallelität auch proportional größere Probleme behandelt werden und daß in Relation zu den größeren Problemen der sequentielle Anteil des Algorithmus kleiner wird. Auf Grund der problematischen Verschmelzung der beiden Parameter Parallelität und Problemgröße [80] sollte man für Situationen, wo Parallelität zur Lösung größerer Probleme statt zur schnelleren Lösung vorhandener Probleme, das Maß Scaleup verwenden. Wenn $T_1(m) = T_p(n)$ für $n > m$, dann ist $\frac{n}{m}$ der Scaleup bei Parallelität p. Der Scaleup gibt also an, ein um welchen Faktor größeres Problem man mit Parallelität p in der selben Zeit lösen kann wie im sequentiellen Fall.

5.2 Algorithmus und Architektur

Die Schwierigkeit bei Entwurf und Analyse paralleler Algorithmen ist die im Vergleich zu sequentiellen Algorithmen stärkere Abhängigkeit von der zugrundeliegenden parallelen Architektur. Die Auswahl möglicher effizienter Algorithmen wird durch die Architektur beschränkt. Die Mehrzahl der in der Literatur dargestellten Algorithmen gehen von hypothetischen Rechnermodellen aus, Voraussetzungen, die natürlich stark idealisierend sind. Damit bleiben zwei wesentliche Problembereiche unberücksichtigt, die eine entscheidende Auswirkung auf die Ausführung des Algorithmus haben. Das ist zum Einen die Frage der Organisation des Speichers und der Zugriffskonflikte und zum Anderen die Frage nach der Kommunikation zwischen den Prozessoren und zwischen Prozessor und Speicher. Viele Algorithmen sind bedingt durch die Entwicklungen der VLSI-Technik nun auch hardwarerealisierbar. Damit können die beiden Problembereiche nicht mehr vernachlässigt werden. Die parallel verfügbaren Prozessoren können den inhärenten Parallelismus nur dann ausnutzen, wenn die Operationen zur richtigen Zeit und ohne Verzug ausgeführt werden. Es hat sich gezeigt, daß die Kommunikationskomplexität gleichrangig neben der Zeitkomplexität betrachtet werden muß. Ziel ist es, eine optimale Anpassung zwischen Algorithmus und Datenkommunikation zu erzielen. Für die optimale Matizenmultiplikation mit der Zeitkomplexität $O(\log n)$ bietet sich so idealerweise eine Baumstruktur an. Es muß angestrebt werden, daß ein Kommunikationsnetzwerk für eine möglichst große Klasse optimal ist und sich auch in die VLSI- Technologie einbetten läßt. Als Beispiel sei hier schnelle Fourier-Transformation genannt, die sich als Butterfly-Netz in einem integrierten Prozessor hardwaremäßig realisieren läßt. Der erzielbare Speedup und die Effi-

zienz solcher Lösungen ist bei gleichzeitiger Spezialisierung sehr hoch. In [102] sind Untersuchungen gemacht worden, Schaltungsmodule (spezialisierte parallele Funktionseinheiten) für eine Klasse von Anwendungen einzusetzen (schnelle Fourier-Transformation, Matrizenmultiplikation, Bildkompressionsalgorithmen). Eine Vielzahl der betrachteten Algorithmen sind auf einem Transputersystem implementiert worden. Wenn auch die veraltete Technologie des Transputers verwendet wurde und somit ein Vergleich mit heutigen Rechnerarchitekturen nicht sinnvoll ist, so konnten doch wichtige Erkenntnisse zu parallelen Algorithmen gewonnen werden.

Literaturverzeichnis

[1] Aho, A. V. ; Hopcroft, J. E.: *The Design and Analysis of Computer Algorithms.* Addision Wesley, 1976

[2] Alt, H. ; Hagerup, T. [u. a.]: Deterministic simulation of idealized parallel computers on more realistic ones. In: *SIAM J. Comput.* (1987), S. 808–835

[3] Amdahl, G. M.: Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. In: *Proc. of the AFIPS Conference* (1967)

[4] Anderson, E. ; Hammerling, S. [u. a.]: *LAPACK Users's Guide.* SIAM Press Philadelphia, 1992

[5] Baule, G. ; McFee, R.: Detection of the magnetic field of the heart. In: *Americ. Heart Journal* (1963), S. 66–95

[6] Benaini, A. ; Robert, Y.: An even faster systolic array for matrix multiplication. In: *Parallel Computing* (1989), S. 249–254

[7] Berkhin, P. ; Brown, J.: A Transform Approach to Fast Matrix Multiplication. In: *in J. Dongarra : Parallel Scientific Computing* (1994), S. 67–79

[8] Bini, D. ; Capovani, M. [u. a.]: $O(n^{2.7799})$ complexity for matrix multiplication. In: *Inform. Proc. Lett.* (1979), S. 234–235

[9] Bischof, C.: Fundamental Linear Algebra Computations on High Performance Computers. In: *in H. W. Meuer : SUPERCOMPUTER '90* (1990), S. 167–182

[10] Booch, G.: *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, 1991

[11] Borodin, A. ; Munro, I.: *The computational complexity of algebraic and numerical problems.* American Elsevier, 1975

[12] Braeunl, T.: *Parallel Programming.* Prentice Hall, 1993

[13] Bronstein, I. N. ; Semendjajew, K. A.: *Taschenbuch der Mathematik.* B. G. Teubner Verlag, 25. Auflage, 1991

[14] Cannon, L. E.: *A cellular computer to implement the Kalman filter algorithm.*, Montana State Univ., Bozman, MT., PhD thesis, 1969

[15] Carlini, U. ; Villano, U.: *Transputers And Parallel Architectures.* Ellis Horwood Limited, 1991

[16] Cayley, A.: *Trans. London philos. Soc. Bd. 148 S. 17-37.* 1858

[17] Cheng, K. H. ; Sahni, S.: VLSI systems for matrix multiplication. in: S. N. Maheshwari, ed. Foundation of Software technology and Theoretical Computer Science. In: *Lecture Notes in Computer Science* (1985), S. 428–456

[18] Cohen, D. ; Edelsack, E. A. ; Zimmermann, J. E.: Magnetocardiograms taken inside a shielded room with superconduting point-contact magnetometer. In: *Appl. Phys. Lett.* (1970), S. 278 ff.

[19] Coppersmith, D. ; Winograd, S.: On the asymptotic complexity of matrix multiplications. In: *SIAM J. Comput.* (1982), S. 472–492

[20] Coppersmith, D. ; Winograd, S.: Matrix multiplication via arithmetical progressions. In: *Proceedings 19th ACM STOC* (1987), S. 1–6

[21] Cormen, T. H. ; Leiserson, C. E. ; Rivest, R. L.: *Introduction to Algorithms.* MIT Press, 1990

[22] Csanky, L.: Fast parallel matrix inversion algorithms. In: *SIAM Journal Computing* 5 (1976), S. 618–623

[23] Dekel, E. ; Nassimi, D. ; Sahni, S.: Parallel matrix and graph algorithms. In: *SIAM Journal Computing* 10 (1981), S. 657–675

[24] Dongarra, J. J. ; Bunch, J. R. [u. a.]: *LINPACK Users's Guide.* SIAM Press Philadelphia, 1979

[25] Dongarra, J. J. ; Croz, J. D. ; Hammerling, S. ; Duff, I.: A set of Level 3 basic linear algebra subprograms. In: *ACM Trans. Math. Software* 16 (1990), S. 1–17

[26] Dongarra, J. J. ; Croz, J. D. ; Hammerling, S. ; Duff, I.: Algorithm 679: A set of Level 3 basic linear algebra subprograms. In: *ACM Trans. Math. Software* 16 (1990), S. 18–28

[27] Dongarra, J. J. ; Croz, J. D. ; Hammerling, S. ; Hanson, R.: Algorithm 656: An extended set of Fortran basic linear Algebra subprograms. In: *ACM Trans. Math. Software* 14 (1988), S. 18–32

[28] Dongarra, J. J. ; Croz, J. D. ; Hammerling, S. ; Hanson, R.: An extended set of Fortran basic linear Algebra subprograms. In: *ACM Trans. Math. Software* 14 (1988), S. 1–17

[29] Dongarra, J. J. ; Duff, I. ; Sorensen, D. ; Vorst, H. van d.: *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM Press Philadelphia, 1991

[30] E.Horowitz ; Sahni, S.: *Algorithmen - Entwurf und Analyse.* Springer Verlag, 1978

[31] F. Röser: *Untersuchungen möglicher Laufzeitverbesserungen des sequentiellen Herzfeldsimulationsalgorithmus durch Parallelisierung und Implementation auf einem Transputersystem.*, TH Ilmenau, Diplomarbeit, 1993

[32] Flynn, M. J.: Very high speed computing systems. In: *Proceedings of the IEEE* 54 (1966), S. 1901–1909

[33] Flynn, M. J.: Some computer organisations and their effectiveness. In: *IEEE Trans. Comput.* 21 (1972), S. 948–960

[34] Gallivan, K. ; Plemmons, R. ; Sameh, A.: Parallel Algorithms for Dense Linear Algebra Computations. In: *SIAM Rev. 32* 1 (1990), S. 54–135

[35] Garbow, B. ; Boyle, J. [u. a.]: Matrix Eigensystems Routines - EISPACK Guide Extension. In: *Lecture Notes in Computer Science* 51 (1977)

[36] Gastinel, N.: Sur le calcul des produits de matrices. In: *Numerische Mathematik 17* (1971), S. 222–229

[37] Glover, K.: *A note on Strassen's matrix multiplication method.*. – unpublished Paper

[38] Grzemba, C.: Leistungsbetrachtung einer Farmimplementierung. In: *Euro Transputer Forum - Bielefeld* (1994), S. 87–94

[39] Gustafson, J. L.: Reevaluating Amdahl's Law. In: *Communications of the ACM* (1988)

[40] Harter, R.: The Optimality of Winograd's formula. In: *Communications of the ACM* (1971), S. 352

[41] Heiss, H. U.: *Prozessorzuteilung in lose gekoppelten Systemen.*, Technische Universität Karlsruhe, Germany, Habilitation, 1993

[42] Hirschfeld, R. ; Zerbe, V.: Untersuchung und Implementierung von Algorithmen der linearen Algebra auf Systemen mit verteiltem Speicher. In: *Euro-Transputer-Forum, Bielefeld* (1994), S. 105–114

[43] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall International, 1978

[44] Hockney, R. W. ; Jesshope, C. R.: *Parallel Computers 2.* Adam Hilger, Bristol and Philadelphia, 1988

[45] Hopcroft, J. E. ; Keer, L. R.: Some techniques for proving certain simple programs optimal. In: *Proceedings Tenth Ann. Symposium on Switching and Automata Theory* (1969), S. 36–45

[46] Hwang, K.: *Advanced Computer Architecture.* Mc Graw-Hill, 1993

[47] Hwang, K. ; Briggs, F. A.: *Computer Architecture and Parallel Processing.* Mc Graw-Hill, 1987

[48] inmos: *OCCAM 2, Reference Mannual.* Prentice Hall, 1988

[49] inmos: The Transputer Applications Notebook. / INMOS. 1989. – Forschungsbericht

[50] Jagadish, H. V. ; Kailath, T.: A family of new efficient arrays for matrix multiplication. In: *IEEE Trans. Comput.* (1989), S. 149–155

[51] Kak, S.: A two-layered mesh array for matrix multiplication. In: *Parallel Computing* (1988), S. 383–385

[52] Karatsuba, A. ; Ofman, Y.: Multiplication of multiple numbers by means of automata. In: *Doklady Akad. Nauk USSR* (1962), S. 293–294

[53] Kluge, J. ; Bruening, U. [u. a.]: The ATOLL approach for a fast and reliable System Area Network. In: *Proceedings 3rd Workshop on Advanced Parallel Processing Technologies* (1999), S. 99–104

[54] Kober, R.: *Parallelrechner-Architekturen.* Springer Verlag, Berlin, 1988

[55] Kramer, M. ; Leeuwen, J. van: Systolische Berechnungen und VLSI. In: *Informatik-Spektrum 7* (1984), S. 154–165

[56] Krapp, M.: *Digitale Automaten.* Verlag Technik, Berlin, 1988

[57] Kroll, P. ; Radtke, T. ; Zerbe, V.: *Compression of still Images. submitted Chapter for: Design and Managment of Multimedia Information Systems: Opportunities and Challenges.* Idea Group Publishing, 2000

[58] KUNG, H. T.: *The structure of Parallel Algorithms in Advances in computers.* Academic Press, Vol 19, 1980

[59] KUNG, H. T. ; LEISERSON, C. E.: *Systolic arrays for VLSI. in: Introduction to VLSI Systems.* Addison Wesley, 1980

[60] LADERMANN, J. D.: A noncommutative algorithm for multiplying 3x3 matrices using 23 multiplications. In: *Bulletin of the american mathematical society* 82 (1976), S. 126–128

[61] LAWSON, C. ; HANSON, R. ; KINCAID, D. ; KROGH, F.: Algorithm 539: Basic linear algebra subprograms for Fortran usage. In: *ACM Trans. Math. Software* 5 (1979), S. 324–325

[62] LAWSON, C. ; HANSON, R. ; KINCAID, D. ; KROGH, F.: Basic linear algebra subprograms for Fortran usage. In: *ACM Trans. Math. Software* 5 (1979), S. 308–323

[63] LEIGHTON, T.: *Layouts for the Shuffle-Exchange Graph and Lower Bound Techniques for VLSI.*, Massachusetts Institute of Technology, PhD thesis, 1981

[64] LEIGHTON, T.: New lower bound techniqes for VLSI. In: *IEEE 22nd Annual Symposium on Foundations of Computer Science* (1981), S. 1–12

[65] LEIGHTON, T.: *Introduction to parallel algorithms and architectures.* Morgan Kaufmann Publisher, Inc., 1992

[66] MARQUARDT, P. W.: An algorithm for least square estimation of nonlinear parameters. In: *J. Soc. Indust. Appl. Math.* (1963), S. 431–441

[67] OBERSCHELP, W. ; SCHIKARSKI, A. ; ERHARDT, B.: *Parallele Algorithmen.* RWTH Aachen, Bericht 154, 1993

[68] OPPENHEIM, A. V. ; WILLSKY, A. S.: *Signale und Systeme.* VCH Verlagsgesellschaft, 1992

[69] PAN, V.: How can we speed up matrix multiplications? In: *SIAM Review, Society for industrial and Applied Mathematics* 26 (1984), S. 393–415

[70] PAN, V.: *How to Multiply Matrix Faster.* Springer Verlag, 1984

[71] PAN, V. ; REIF, J.: *Efficient parallel solution of linear systems.* Proc. of the 17th Annual ACM Symp. on Theors of Computing, 1985

[72] POMIERSKI, T. ; GROSS, H. M. ; WENDT, A.: Multicolumnar system for Primary Cortical Analysis of Real World scenes. / TU Ilmenau. 1993. – Forschungsbericht

[73] Porter, W. A. ; Aravena, J. L.: Cylindrial arrays for matrix multiplication. In: *Proc. 24th Annual Conference on Comm., Control and Computing.* (1986)

[74] Preparata, F. ; Vuillemin, J.: Area-time optimal VLSI networks for matrix multiplication. In: *Proceedings of th 14th Princeton Conference on Information Science and Systems.* (1980), S. 300–309

[75] Quinton, P. ; Joinnault, B. ; Gachet, P.: A new matrix multiplication systolic array. In: *North Holland Amsterdam* (1986), S. 259–268

[76] R. Göbel and P. Schulze and others: *Wissensspeicher der Physik.* Volk und Wissen Verlag, 1975

[77] R. Zurmühl: *Matrizen und ihre technischen Anwendungen.* Springer Verlag, 1964

[78] Radtke, T. ; Zerbe, V.: Image Compression using Modified SPIHT Algorithm and linear Prediction. In: *Proceedings of Advances in Intelligent Computing and Multimedia Systems, Baden-Baden* (1999), S. 49–52

[79] Ranka, S. ; Sahni, S.: *Hypercube Algorithms.* Springer Verlag, 1990

[80] Reuter, A.: Grenzen der Parallelität. In: *informationstechnik* 34 (1992), S. 62–74

[81] Romani, F.: Some properties of disjoint sums of tensor related to matrix multiplication. In: *SIAM Journal Computing* (1982), S. 263–267

[82] Sameh, A. H. ; Brent, R. P.: Solving triangular systems on a parallel computer. In: *SIAM J. Numer. Anal. 14* (1977), S. 793–796

[83] Schoenhage, A.: Partial and total matrix multiplication. In: *SIAM Journal Computing* (1981), S. 434–456

[84] Schreiber, R.: Block Algorithmen for Parallel Machines. In: *IMA Volumes in Mathematics and its Applications* (1988), S. 197–207

[85] Schwartz, J. T.: Ultracomputers. In: *ACM TOPLAS 2* (1980), S. 484–521

[86] Smith, B. T.: *Matrix Eigensystem Routines - EISPACK Guide.* Springer Verlag, 1976

[87] Steinmetz, R.: *OCCAM2.* Dr. Alfred Huethig Verlag, Heidelberg, 1987

[88] Stone, H. S.: An efficient parallel algorithm for the solution of a triangular linear system of equations. In: *Journal of ACM* (1973), S. 27–38

[89] Stone, H. S.: *Problems of parallel computation.* Academic Press New York, 1973

[90] Strassen, V.: Gaussian elimination is not optimal. In: *Numer. Math. 32* (1969), S. 354–356

[91] Sylvester, J. J.: *Philos. Mag. Bd. 37.* 1850

[92] T. Knösche: *Magnetokardiographische Feldmodellierung und Quellenlokalisation mit Hilfe der Randelementemethode.*, TU Ilmenau, Diplomarbeit, 1992

[93] U. Schöning: *Algorithmen - kurz gefasst.* Spektrum Akademischer Verlag, 1997

[94] Vijngaarden, V. [u. a.]: *Algol68.* Springer Verlag, Berlin, 1969

[95] Wenzel, L.: *Parallele Programmierkonzepte.* Franzis Verlag, 1991

[96] Winograd, S.: A new Algorithm for inner product. In: *IEEE Transaction on computers* (1968), S. 693–694

[97] Winograd, S.: On multiplication of 2x2 matrices. In: *Lineare Algebra and its applications* 4 (1971), S. 381–388

[98] Winograd, S.: *Some remarks on fast multiplication of polynomials. in: Complexity of Sequential and Parallel Numerical Algorithms.* New York Academic Press, 1973

[99] Wirth, N.: *Programming in Modula-2.* Springer Verlag, 1991

[100] Zerbe, V.: *Spezifikation und HLL-Implementierung von Multitaskbetriebssystemen.*, TH Ilmenau, Dissertation, 1990

[101] Zerbe, V.: *Strassen 'like' Algorithmus.* 1995. – unpublished Paper

[102] Zerbe, V. [u. a.]: Matrizenmultiplikation: sequentiell versus parallel. / TU Ilmenau. 1998. – Forschungsbericht

[103] Zerbe, V. ; Keller, H. ; Schorcht, G.: Parallel Matrix Computations and their Application for Biomagnetic Fields. In: *Proc. of Advances in Parallel and Distributed Computing Shanghai (China)* 1 (1997), S. 139–143