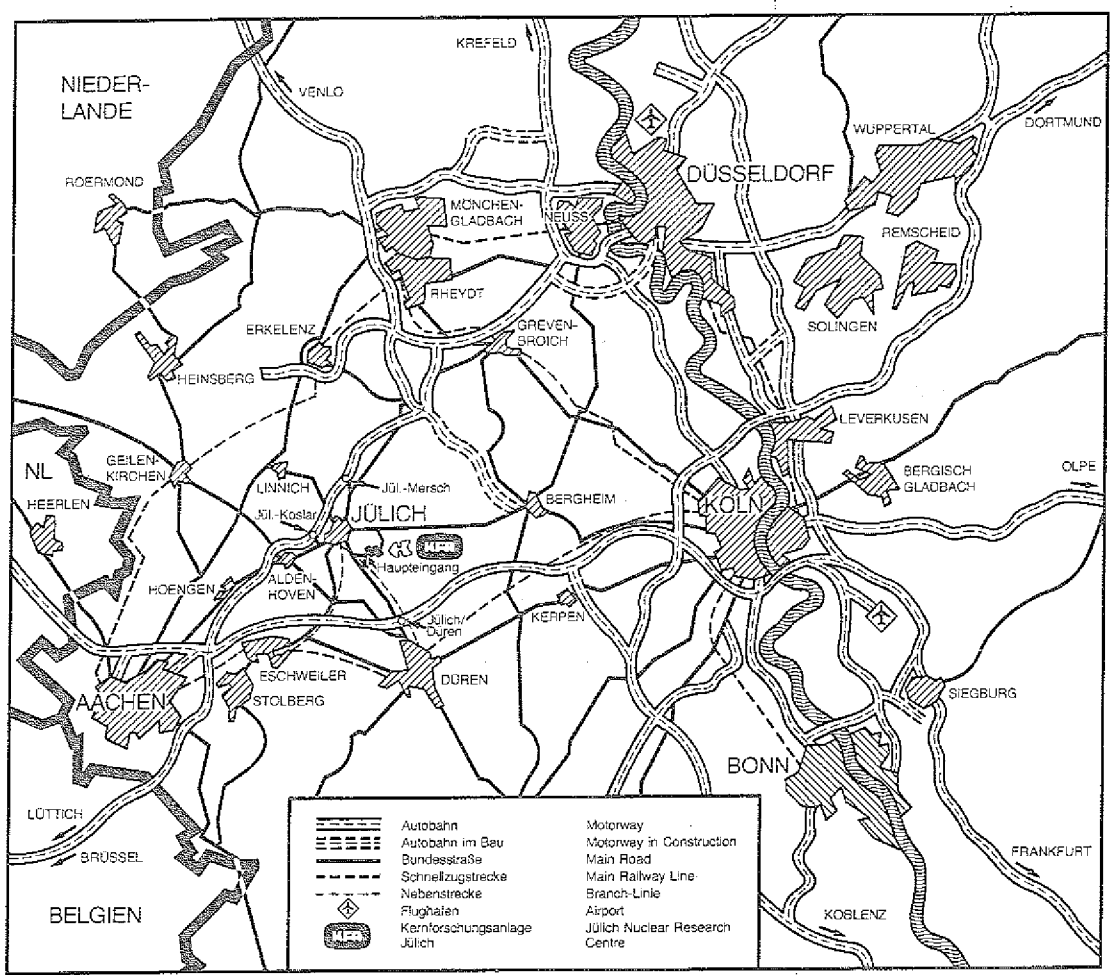
KERNFORSCHUNGSANLAGE JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik

Entwicklung der Grundlagen einer Theorie paralleler partitionierter Algorithmen unter besonderer Berücksichtigung partitionierter Netzwerkalgorithmen

von

Thula Vogell

Jül-2108
Dezember 1986
ISSN 0366-0885



Als Manuskript gedruckt

Berichte der Kernforschungsanlage Jülich – Nr. 2108
Zentralinstitut für Angewandte Mathematik Jül-2108

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH
Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)
Telefon: 02461/610 · Telex: 833556-0 kf d

Entwicklung der Grundlagen einer Theorie paralleler partitionierter Algorithmen unter besonderer Berücksichtigung partitionierter Netzwerkalgorithmen

von

Thula Vogell

D 82 (Diss. T.H. Aachen)

Diese Arbeit ist am Zentralinstitut für Angewandte Mathematik der Kernforschungsanlage Jülich entstanden. Für die damit verbundene Unterstützung möchte ich mich bei Prof. Dr. F. Hoßfeld bedanken. Darüberhinaus gilt mein Dank den Herren Prof. Dr. O. Lange und Prof. Dr. O. Spaniol der RWTH Aachen für die interessanten inhaltlichen Diskussionen und die Betreuung der Arbeit.

INHALTSVERZEICHNIS

0. Einleitung 3

1. Die Struktur paralleler partitionierter Algorithmen 11
1.1. Entwicklung allgemeiner Methoden und Bezeichnungen zur systematischen Untersuchung partitionierter Algorithmen 12
1.2. Beispiele 19
1.3. Partitionierbarkeit und Rekursion 32

2. Effizienzen und Berechnungszeiten 37
2.1. Bestimmung der Berechnungszeiten im allgemeinen Fall und für die Beispielalgorithmen 38
2.2. Zeitkomplexität sequentieller partitionierter Algorithmen . . . 43
2.3. Vergleich der Berechnungszeiten partitionierter und nicht-partitionierter paralleler Algorithmen 45
2.4. Komplexitätsvergleich verschiedener Parallelisierungen eines partitionierten Algorithmus 49

3. Partitionierte Prozessornetzwerke 55
3.1. Netzwerkpartitionierung durch Hypernetze 56
3.2. Duale Hypernetze 67
3.3. Mehrfach partitionierte Netze und d-dimensionale Graphen . . . 70
3.4. Zusammenhang von Hypernetzen und Netzwerkemulationen 78

4. Partitionierte Netzwerkalgorithmen 81
4.1. Partitionierte Netzwerke als Kommunikationsgraphen partitionierter Algorithmen 82
4.2. Bestimmung der Kommunikationszeit partitionierter Algorithmen und ihrer dualen Versionen 90

5. Kommunikationszeiten in Netzen mit beschränkter Prozessorzahl 97

6. Die partitionierte LU-Zerlegung - ein Beispiel 103
6.1. Berechnungszeit 105

6.2. Partitionierungen der Tiefe 2 108
6.3. Die partitionierte LU-Zerlegung als Netzwerkalgorithmus . . . 112
6.4. Kommunikationszeiten 131

7. Zusammenfassung und Ausblick 135

Literatur 141

0. EINLEITUNG

Im Rahmen der Theorie paralleler Algorithmen sind eine Vielzahl von Algorithmen zur Lösung von Problemen der Größe n für theoretische Prozessoranzahlfunktionen p(n) entwickelt worden. Zusammenstellungen einiger wesentlicher paralleler Algorithmen aus den Bereichen der Linearen Algebra, der Graphen- und Sortieralgorithmen finden sich beispielsweise in [10], [21], [2] und [11]. Die Annahme eines unbeschränkten Parallelismus ermöglicht es, sehr schnelle Algorithmen zu entwickeln, jedoch um den Preis einer großen Prozessoranzahl, die häufig keine praktische Relevanz besitzt.
Dieser theoretischen Behandlung paralleler Algorithmen stehen die Restriktionen gegenüber, die sich aus der vorhandenen Technologie ergeben. Hierzu gehören vor allem die Beschränkung der Prozessoranzahl sowie die Organisation des Datenaustausches zwischen den Prozessoren.
Ein Ziel ist die Abbildung der theoretisch entwickelten Algorithmen auf die real verfügbaren Parallelrechner, so daß ein möglichst geringer Zeitverlust erfolgt. Um dies zu ermöglichen, ist es sinnvoll, die wesentlichen technologischen Restriktionen in die Theorie paralleler Algorithmen zu integrieren.

Unser Ausgangspunkt ist die Beschränkung der Prozessoranzahl. Parallele Algorithmen für eine fest vorgegebene Prozessorzahl p können aus den theoretischen Algorithmen für p(n) Prozessoren gewonnen werden, indem die Algorithmen "partitioniert" werden. Der Begriff der Partitionierung wird in der Literatur nicht einheitlich verwendet. In Bezug auf Algorithmen werden zwei verschiedene Methoden damit bezeichnet. In [18] werden parallele Algorithmen partitioniert, um sie optimal auf systolische Felder beschränkter Größe abzubilden. Konkret wird in diesem Fall der Indexraum des Algorithmus in Teilräume zerlegt, die dann den einzelnen Prozessoren zugeordnet werden. Dies entspricht einer Partitionierung der Menge der vom Algorithmus auszuführenden Operationen.
Die in [14] beschriebenen partitionierten Algorithmen dienen dem Zweck, Chips, die zur Berechnung elementarer Matrix-Operationen für kleine Blockmatrizen entwickelt worden sind, einzusetzen. Obwohl die Arbeit [18]

diese Algorithmen mit als Beispiele für partitionierte Algorithmen heranzieht, liegt in mathematischer Hinsicht ein völlig anderes Prinzip vor: Während im ersten Fall unter Beibehaltung des Algorithmus die Menge der Operationen zerlegt wird, wird im zweiten Fall die zugrundeliegende Datenmenge partitioniert und ein Algorithmus entwickelt, der auf diesen partitionierten Datenmengen operiert.
Datenpartitionierungen können ebenfalls verwendet werden, um einen parallelen Algorithmus mit beschränkter Prozessoranzahl auszuführen. Als Beispiel mögen die Blocksortieralgorithmen (z.B. in [2]) dienen.

Partitionierungen (paralleler) Algorithmen treten in der Literatur lediglich als Hilfsmittel zur Lösung bestimmter Probleme (Chip-Einsatz, Prozessorenbeschränkung) auf; uns ist keine Arbeit bekannt, in der eine detailliertere Untersuchung des eigenständigen Charakters partitionierter Algorithmen erfolgt.

Ein weiterer Zugang zur Problematik der Beschränkung der Prozessoranzahl findet sich in [4]. Dort werden für Prozessorverbindungsnetzwerke $G(n)$ beliebiger Größe Quotientennetzwerke G mit beschränkter Knotenanzahl angegeben, die $G(n)$ vermöge einer Abbildung $\phi: V(G(n)) \to V(G)$ emulieren. Diese Herangehensweise beinhaltet darüberhinaus eine möglichst exakte Wiedergabe der Verbindungsstruktur von $G(n)$ durch G, indem verlangt wird, daß Kanten von $G(n)$ durch ϕ nur auf Kanten oder Knoten von G abgebildet werden. In diesem Fall induziert ϕ eine Partitionierung der Knotenmenge von $G(n)$, die durch die Urbilder $\phi^{-1}(v)$, $v \in V(G)$, gegeben ist. Interpretiert man $G(n)$ als Kommunikationsgraph eines Netzwerkalgorithmus, so impliziert die Partitionierung von $V(G(n))$ eine Partitionierung des Algorithmus im Sinne einer Partitionierung der Operationsmenge.

Wir verstehen im folgenden unter partitionierten Algorithmen ausschließlich Algorithmen, die auf partitionierten Datenmengen operieren. Algorithmen, deren Operationsmenge im Hinblick auf eine beschränkte Prozessorzahl partitioniert wird, bezeichnen wir als nicht-partitionierte Algorithmen ausgeführt mit p Prozessoren.

Den Gegenstand dieser Arbeit bilden parallele partitionierte Algorithmen. Für die durch einen Algorithmus zu berechnende Funktion f setzen wir vor-

aus: $f: (K^n \times K^n)^k \to K^n \times K^n$, wobei K einen Körper bezeichnet. Für einen Teiler m von n ("m|n") sind die Operanden eines partitionierten Algorithmus dann als $(m \times m)$-Matrizen leicht zu beschreiben.
Im Mittelpunkt steht die Untersuchung der Berechnungs- und Kommunikationskomplexität partitionierter Algorithmen. Dabei legen wir für die Bestimmung der Kommunikationskomplexität als Parallelrechnermodell eine Multiprozessoranordnung mit Prozessorverbindungsnetzwerk und lokalen Speichern zugrunde. Wir wollen partitionierte Algorithmen verwenden, um parallele Algorithmen für eine beschränkte Prozessoranzahl p anzugeben. Aus diesem Grund liegt es nahe, ihre Berechnungs- und Kommunikationskosten zu vergleichen mit den entsprechenden Zeiten nicht-partitionierter Algorithmen ausgeführt mit p Prozessoren. Um diesen Vergleich durchführen zu können, ist eine präzise Bestimmung des Begriffs der "Partitionierbarkeit" bzw. der "Partitionierung" eines (Netzwerk-)Algorithmus M Voraussetzung. Es können dann die Komplexitäten von M ausgeführt mit p Prozessoren und einer Partitionierung von M für p Prozessoren verglichen werden.

Im einzelnen bestehen die wesentlichen Resultate der Arbeit in folgenden Punkten:
Durch eine verallgemeinerte Fassung des Konzeptes einer PRAM (parallel random access machine) (vgl. Kap. 1 und [5]) wird eine Definition des Begriffs "partitionierter Algorithmus" ermöglicht, die eine einheitliche Beschreibung einer großen Klasse teilweise sehr verschiedenartiger partitionierter Algorithmen erlaubt. Die grundlegende Idee dabei ist, PRAMs über beliebigen Ringen R zu betrachten. Damit enthält die Klasse der PRAMs sowohl Algorithmen über $K^m \times K^m$ als auch über K. Die Simulation einer über dem Matrix-Ring $K^m \times K^m$ arbeitenden PRAM M durch eine PRAM über K liefert dann einen partitionierten Algorithmus. Diese simulierende Maschine kann als Zusammensetzung aus einem "externen" Algorithmus M und "internen" Algorithmen zur effektiven Berechnung der Operationen von M über K aufgefaßt werden. Partitionierte Netzwerkalgorithmen lassen sich weitgehend analog definieren, wobei zu berücksichtigen ist, daß neben der Datenpartitionierung eine Partitionierung des Kommunikationsgraphen erfolgen muß.

Externer Algorithmus und interne Algorithmen einer partitionierten PRAM können unabhängig voneinander Parallelisierungen für mehrere Prozessoranzahlfunktionen besitzen. Die Betrachtung dieser verschiedenen partitionierten Algorithmen zur Berechnung von f weist auf eine Dualitätseigenschaft derjenigen Algorithmen hin, deren externer Algorithmus und internen Algorithmen Parallelisierungen für dieselben Prozessoranzahlfunktionen besitzen. In diesem Fall können die Parallelisierungen von externem Algorithmus und internen Algorithmen getauscht werden. Man erhält einen neuen partitionierten Algorithmus zur Berechnung von f für dieselbe Prozessoranzahl, der im Falle konstanter Effizienzen dieselbe Laufzeit besitzt, dessen Kommunikationszeit bezogen auf denselben Kommunikationsgraphen sich jedoch signifikant von der Kommunikationszeit des Ausgangsalgorithmus unterscheiden kann.

Die naheliegendste Möglichkeit, einen partitionierten Algorithmus mit einer beschränkten Prozessoranzahl auszuführen, besteht darin, für jede Blockoperation von M jeweils einen Prozessor einzusetzen. Verwendete der ursprüngliche Algorithmus p(n) Prozessoren, so werden nun nur noch p(n/m) Prozessoren benötigt. Dieses Vorgehen entspricht dem in der Literatur üblichen. So werden die Blocksortieralgorithmen in [2] durch Verwendung jeweils eines Prozessors für die merge-split-Operationen realisiert; die Algorithmen in [15] zur Produkt- bzw. Prefix-Berechnung mit p Prozessoren beruhen auf einer Partitionierung, die ebenfalls jeweils einen Prozessor für die internen Operationen verwendet.
In unserem Kontext stellt sich diese Vorgehensweise als Spezialfall einer allgemeineren Verwendung partitionierter Algorithmen zur parallelen Berechnung bei beschränktem Parallelismus dar. Ausgehend von unbeschränktem Parallelismus sind externer Algorithmus und interne Algorithmen als parallele Algorithmen für die Prozessoranzahl*funktionen* p_0 bzw. p_1 gegeben: Der partitionierte Algorithmus verwendet insgesamt $p_0(n/m)p_1(m)$ Prozessoren. Wenn es möglich ist, die Blockgröße m in Abhängigkeit von n und p so zu bestimmen, daß gilt

$$p_0(n/m)p_1(m) = p,$$

so erhält man einen parallelen Algorithmus für p Prozessoren, welcher die durch die Funktionen p_0, p_1 beschriebenen Parallelisierungen von externem Algorithmus und internen Algorithmen aufrecht erhält. Die oben genannten Beispiele entsprechen dann dem Spezialfall $p_1 \equiv 1$.

Es stellt sich die Frage, welche der möglichen Parallelisierungen p_0 und p_1 eine optimale Abbildung des unbeschränkten Parallelismus auf den beschränkten Parallelismus liefern. Wir untersuchen diese Frage im Hinblick auf die Berechnungszeit und weisen nach (2.4.5.): Unter allen Parallelisierungen, die für festes n und p eine Wahl der Blockgröße m mit $p_0(n/m)p_1(m) = p$ zulassen, besitzt entweder eine Parallelisierung mit $p_0(n/m) = 1$, $p_1(m) = p$ oder mit $p_0(n/m) = p$, $p_1(m) = 1$ eine maximale Effizienz. Damit wird begründet, daß das intuitive Vorgehen in vielen Fällen eine minimale Laufzeit besitzt. Umgekehrt ermöglicht dies Ergebnis beispielsweise, einen Effizienzverlust des Matrix-Multiplikationsalgorithmus für einen cube-connected-computer mit n^2m Prozessoren aus [3] nachzuweisen und zu bestimmen, ohne hierfür die globale Struktur des Algorithmus zu benötigen. Dieser Algorithmus verdeutlicht gleichzeitig aber auch, daß es sinnvoll sein kann, gemischte Parallelisierungen zu betrachten, da es keine p(n)-Parallelisierung der Matrix-Multiplikation mit $n^2 < p(n) < n^3$ gibt.

Es entspricht der Anschauung, daß partitionierte Algorithmen eine größere Berechnungs- und Kommunikationskomplexität besitzen als nicht-partitionierte Algorithmen. Die Quantifizierung dieser Intuition erfolgt in dieser Arbeit. Mit Hilfe der Effizienzen der auftretenden Algorithmen geben wir an, um welchen Faktor sich die Berechnungszeit einer p-Partitionierung eines Algorithmus M mindestens und höchstens von der Berechnungszeit einer nicht-partitionierten Ausführung von M mit p Prozessoren unterscheidet (2.3.8.): Im Gegensatz zur Kommunikationszeit ist der Unterschied in den Berechnungszeiten nicht signifikant.
Für den Vergleich der Kommunikationszeiten nehmen wir an, daß der nicht-partitionierte Netzwerkalgorithmus D vermöge einer uniformen Netzwerkemulation im Sinne von [4] auf einem Netz mit p Prozessoren realisiert wird. Die Kommunikationszeiten werden als Funktionen der Laufzeiten beschrieben; damit werden insbesondere Datenflußalgorithmen ([9], [16]) erfaßt. Wir erhalten im ungünstigsten Fall (5.0.7.) einen Zeitverlustfaktor des partitionierten Algorithmus von $(p(n)/p)^{1-\beta}$, $0 \leq \beta \leq 1$, wobei β von der Topologie des Netzwerks abhängt und z.B. für ebene Gitterstrukturen durch 1/2 und für lineare Netze durch 0 gegeben ist.

Grundsätzlich führen wir die Kommunikationszeit partitionierter Netzwerkalgorithmen zurück auf die Berechnung der Kommunikationszeiten von internen und externem Algorithmus. Damit lassen sich bekannte Verfahren zur Angabe unterer Schranken, wie z.B. die in [6] beschriebene Bisektionsmethode oder die Bestimmung des Durchmessers des Teilnetzes der Prozessoren, die durch einen Datenweg miteinander verbunden sein müssen ([7]), auch auf partitionierte Netzwerkalgorithmen anwenden.
Grundlage für unsere allgemeine Darstellung der Kommunikationszeit ist die Angabe partitionierter Kommunikationsgraphen für partitionierte Algorithmen. Vereinzelt finden sich in der Literatur zusammengesetzte Netzwerke. Beispiele sind die mesh-of-tree-Netze (z.B. in [12]) und die cube-connected-cycles ([20]). Unser Zugang zur Netzwerkpartitionierung beruht auf einer Verallgemeinerung solcher Netzwerkkonstruktionen.

Durch die Bildung von Äquivalenzklassen in Graphen konstruieren wir Hypernetze, die als Kommunikationsgraphen paralleler Algorithmen über $K^m \times K^m$ dienen. Unser Hypernetzkonzept erweist sich darüberhinaus als geeignet, um beliebige mehrdimensionale Graphen zu definieren. Obwohl solche Graphen aufgrund ihrer nachweisbar großen Kantenzahl nicht als reale Prozessorverbindungsnetzwerke in Betracht zu ziehen sind, ist ihre Verwendung als Kommunikationsgraphen sinnvoll, da sie einfache Quotientennetze besitzen (3.4.3.).

Die in der Arbeit angeführten Beispielalgorithmen sind vorzugsweise grundlegende Algorithmen der Linearen Algebra. Sie sind, insbesondere hinsichtlich der gewählten Parallelisierungen, elementar. Eine Ausnahme bildet der partitionierte Algorithmus zur Berechnung der Fast-Fourier-Transformation. Neben ihren vielfältigen Anwendungsmöglichkeiten sind diese Beispiele aufgrund ihrer einfachen Struktur gut geeignet, unsere wesentlichen Ergebnisse zu verdeutlichen.

Ausgehend von den Untersuchungen in dieser Arbeit sehen wir die Vorteile partitionierter Algorithmen gegenüber nicht-partitionierten Algorithmen vor allem in folgenden Punkten:

- Partitionierte Algorithmen können als parallele Algorithmen direkt für $p(n,m) = p$ Prozessoren entworfen werden. Hierdurch kann z.B. die

Organisation des Datenflusses systolischer Algorithmen unmittelbar auf ein systolisches Feld beschränkter Größe übertragen werden.

- Da die Kommunikationsstruktur partitionierter Algorithmen sich elementar aus dem Kommunikationsmuster von internen Algorithmen und externem Algorithmus zusammensetzen läßt, ist zu erwarten, daß ihre Kommunikation - trotz des größeren Informationsflusses - im Einzelfall schneller realisierbar ist als die Kommunikation nicht-partitionierter Algorithmen ausgeführt mit p Prozessoren. (Z.B. wenn die Übertragung eines aus m Einzeldaten zusammengesetzten Datenblocks lediglich Zeit o(m) erfordert.)

Die Inhalte der einzelnen Kapitel im Überblick:
In Kapitel 1 entwickeln wir den theoretischen Rahmen für die Behandlung partitionierter paralleler Algorithmen und erläutern ihn durch Beispielalgorithmen. Es wird die Partitionierbarkeit eines Algorithmus definiert, sowie eine enge Verwandtschaft zwischen partitionierten und rekursiven Algorithmen nachgewiesen (1.3.1.). Kapitel 2 untersucht die Berechnungskomplexität partitionierter Algorithmen für p Prozessoren. Wir geben eine obere und untere Schranke für die Laufzeit einer Algorithmenpartitionierung im Verhältnis zur nicht-partitionierten Ausführung des Algorithmus mit p Prozessoren an. Für eine große Klasse partitionierter Algorithmen wird nachgewiesen, daß die sequentielle Ausführung des externen Algorithmus oder der internen Algorithmen hinsichtlich der Gesamteffizienz optimal ist. Im 3. Kapitel werden die graphentheoretischen Grundlagen für die Behandlung partitionierter Netzwerkalgorithmen gelegt. Wir entwickeln das Hypernetzkonzept zur Partitionierung von Netzwerken bzw. zur Konstruktion partitionierter Netzwerke (3.1.). Wir untersuchen duale Kommunikationsgraphen (3.2.) und verwenden Hypernetze zur Definition allgemeiner d-dimensionaler Netzwerke (3.3.). Der Zusammenhang von Quotientennetzwerken ([4]) und dem Hypernetzkonzept wird präzisiert (3.4.). In Kapitel 4 wird gezeigt, wie partitionierte parallele Algorithmen auf den zuvor entworfenen partitionierten Netzen realisiert werden, und wie die Kommunikationszeit der so gewonnenen Netzwerkalgorithmen zu berechnen ist. Die Kommunikationszeit des dualen partitionierten Algorithmus wird untersucht. Den Abschluß der allgemeinen Untersuchungen bildet in Kapitel 5 der Vergleich

der Kommunikationszeit eines nicht-partitionierten Netzwerkalgorithmus D ausgeführt auf einem Quotientennetzwerk mit p Prozessoren mit der Kommunikationszeit einer Partitionierung von D. Wir weisen einen Zeitverlustfaktor $(p(n)/p)^{1/\beta}$, $0 \leq \beta \leq 1$, des partitionierten Netzwerkalgorithmus gegenüber seiner nicht-partitionierten Version nach. Das letzte Kapitel illustriert Ergebnisse der Arbeit anhand des Beispiels der partitionierten LU-Zerlegung einer Matrix.

1. DIE STRUKTUR PARALLELER PARTITIONIERTER ALGORITHMEN

In diesem Kapitel wird eine Formalisierung des Begriffs "partitionierter Algorithmus" entwickelt, mit der wir in der Lage sind, verschiedenartige Beispiele partitionierter Algorithmen einheitlich zu behandeln. Kernbestandteil unseres Konzeptes ist die Definition paralleler Algorithmen durch PRAMs (parallel random access machines). Dieser Zugang zu parallelen Algorithmen hat sich in der Literatur weitgehend durchgesetzt. Explizite Beschreibungen von PRAMs finden sich beispielsweise in [5] und [17]. Wir verallgemeinern die üblichen Definitionen, indem wir PRAMs über beliebigen Ringen R betrachten. In der hiermit gewonnenen größeren Klasse paralleler Algorithmen sind partitionierte Algorithmen leicht als Algorithmen zu charakterisieren, deren Argumentbereich aus Datenblöcken besteht.

Die grundlegenden Definitionen, die den ersten Teil des Kapitels bilden, werden ergänzt durch die Untersuchung des Begriffs "Partitionierung eines (bekannten) Algorithmus". Dieser Begriff ist von zentraler Bedeutung für einen sinnvollen Vergleich der Komplexität partitionierter und nicht-partitionierter Algorithmen (Kap.2 und Kap.5).

Der anschließende Beispielteil beinhaltet neben bekannten Algorithmen zur Berechnung der elementaren Matrix-Operationen, partitionierten Algorithmen der Linearen Algebra ([10]) und Blocksortieralgorithmen ([2]) eine von uns entwickelte partitionierte Version der Fast-Fourier-Transformation. Die theoretischen Überlegungen des ersten Teils werden genutzt, um eine Partitionierung des Rücksubstitutionsalgorithmus anzugeben, die gegenüber der in [14] dargestellten Version nur leicht modifiziert ist, jedoch ein besseres Laufzeitverhalten aufweist (vgl. Kap.2).

Die auf der intuitiven Ebene nahegelegte Verwandtschaft zwischen rekursiven und partitionierten Algorithmen wird durch Satz 1.3.2. im letzten Teil des Kapitels präzisiert.

1.1. ENTWICKLUNG ALLGEMEINER METHODEN UND BEZEICHNUNGEN ZUR SYSTEMATISCHEN UNTERSUCHUNG PARTITIONIERTER ALGORITHMEN

Unter partitionierten Algorithmen verstehen wir Algorithmen, die auf partitionierten Datenmengen operieren. Um die Datenpartitionierung einfach beschreiben zu können, betrachten wir nur solche Algorithmen, deren Ein- und Ausgabe in Form von Matrizen (oder Vektoren) gegeben ist. Nachfolgend sei K ein beliebiger Körper, also in der Regel $K = \mathbb{R}$ oder $K=\{0,1\}$. Im allgemeinen legen wir damit Algorithmen zugrunde, die Funktionen

$$f: K^{n_1} \times K^{n_2} \to K^{n'_1} \times K^{n'_2}$$

mit $n_1, n'_1, n_2, n'_2 \in \mathbb{N}$ berechnen.

Wir beschränken uns auf den Fall von Algorithmen mit quadratischen Eingabematrizen $A(1),\ldots,A(k) \in K^n \times K^n$ und einer quadratischen Ausgabematrix $X \in K^n \times K^n$ bzw. auf den Fall von k Eingabevektoren $A(1),\ldots,A(k) \in K^n$ und einem Ausgabevektor $X \in K^n$, $k,n \in \mathbb{N}$, d.h. es werden Funktionen

$$f: (K^n \times K^n)^k \to K^n \times K^n \quad \text{bzw.} \quad f: (K^n)^k \to K^n$$

berechnet. Wir interpretieren Vektoren als Diagonalmatrizen, wodurch eine geschlossene Darstellung der theoretischen Grundlagen erreicht wird.
Im folgenden wird vorausgesetzt, daß $m \in \mathbb{N}$ ein Teiler von n ist, $m|n$.

1.1.1. Definition

Sei $A = (a_{ij})_{1\le i,j\le n}$ eine $(n \times n)$-Matrix über K. *Die Partitionierung $\bar{A}$ von A mit Blockgröße m* ist die Matrix $\bar{A} = (\bar{a}_{ij})_{1\le i,j\le n/m}$ mit

$$\bar{a}_{ij} := \begin{bmatrix} a_{(i-1)m+1,(j-1)m+1} & \cdots & a_{(i-1)m+1,jm} \\ \vdots & & \vdots \\ a_{im,(j-1)m+1} & \cdots & a_{im,jm} \end{bmatrix}$$

□

1.1.2. Definition

Sei $f: (K^n \times K^n)^k \to K^n \times K^n$ eine Funktion und $\bar{K} := K^m \times K^m$. Eine *m-Partitionierung von f* ist eine Funktion

$$\bar{f}_m: (\bar{K}^{n/m} \times \bar{K}^{n/m})^k \to \bar{K}^{n/m} \times \bar{K}^{n/m}$$

mit $\bar{f}_m(\overline{A(1)},\ldots,\overline{A(k)}) := \overline{f(A(1),\ldots,A(k))}$ für alle $A(1),\ldots,A(k) \in K^n \times K^n$.

□

Bei der Betrachtung partitionierter Algorithmen legen wir von vorneherein den parallelen Fall zugrunde. Sequentielle partitionierte Algorithmen ergeben sich dann als Spezialfall. Eine präzise Begriffsbestimmung erfordert es, zunächst festzulegen, was unter einem parallelen Algorithmus verstanden wird. Wir verwenden hierfür das Modell einer PRAM ([5], [17]).

1.1.3. Definition

Sei R ein Ring. Eine *(polynomielle) PRAM M(n) über R* ist gegeben durch

- p(n) Prozessoren $P_1,\ldots,P_{p(n)}$, $p \in \mathbb{R}[X]$, $p(1) = 1$,
- eine unbeschränkte Anzahl Register 1,2,... ,
- p(n) ausgezeichnete Register 01,...,0p(n), die Akkumulatoren der P_i ,
- eine endliche Menge S von Grundoperationen ,
- p(n) den P_i zugeordneten Programmen prog P_i, $1 \le i \le p(n)$.

Jedes Register kann ein Element aus R oder $\mathbb{N}$ speichern, <r> bezeichne den Inhalt von Register r für $r \in \mathbb{N}$. S besteht aus Funktionen $g: R' \to R$ mit $R' = R$ oder $R' = R \times R$. prog P_i besteht aus einer endlichen Folge $(\text{prog } P_i)_1,\ldots,(\text{prog } P_i)_{t_i}$ von Befehlen aus folgender Liste:

	Bezeichnung:	*Wirkung:*
$X \in R$:	c-load (X)	<0i> := X
$r \in \mathbb{N}$:	load (r)	<0i> := <r>
$r \in \mathbb{N}$:	store (r)	<r> := <0i>
<r> $\in \mathbb{N}$:	i-load (r)	<0i> := <<r>>
<r> $\in \mathbb{N}$:	i-store (r)	<<r>> := <0i>
<0i> $\in R$:	g	<0i> := g(<0i>)
$r \in \mathbb{N}$, (<0i>,<r>) $\in R \times R$:	g(r)	<0i> := g(<0i>,<r>)
	stop	Maschine hält
	wait	P_i setzt einen Schritt aus

Falls die in der linken Spalte angegebene Bedingung nicht erfüllt ist, ist der Befehl nicht definiert.

Bezeichnet l_i die Anzahl der arithmetischen Operationen (Befehle vom Typ (f) und (g)) in prog P_i, so heißt

$$t(n) := \max \{l_i \mid 1 \le i \le p(n)\}$$

die *Laufzeit* oder auch *Berechnungszeit* von M(n).

□

In der Regel ist S = {+,-,*,/}. Im Falle von Sortieralgorithmen enthält S eine Vergleichsoperation.

1.1.4. Definition

M(n) berechnet die Funktion $f: (K^n \times K^n)^k \to K^n \times K^n$:<=>
Wird M(n) gestartet mit $x \in (K^n \times K^n)^k$, so enthalten nach Beendigung aller Programme der Prozessoren von M(n) die Register $1,\ldots,n^2$ die Werte $(f(x))_{ij}$, $1 \leq i,j \leq n$.

□

1.1.5. Definition

Ein *paralleler Algorithmus* zur Berechnung von f ist eine PRAM M(n), die f berechnet.

□

Auf dieser Grundlage geben wir nun ein Konzept zur Beschreibung paralleler partitionierter Algorithmen an.
$\overline{K} = K^m \times K^m$ ist ein Ring, so daß PRAMs über $\overline{K}$ betrachtet werden können. Die Überlegung besteht darin, durch die Verknüpfung einer PRAM über $\overline{K}$ mit PRAMs über K zwischen einer globalen und einer lokalen Struktur der untersuchten Algorithmen zu unterscheiden. Sei $\overline{M}(n/m)$ eine PRAM über $\overline{K}$ mit $p_0(n/m)$ Prozessoren, die eine m-Partitionierung $\overline{f}_m$ von f berechnet. (Wir bezeichnen eine PRAM $\overline{M}(n/m)$ über $\overline{K}$ gelegentlich auch als *Funktional-PRAM*.) Seien ferner $M_g(m)$, $g \in S_m$, PRAMs über K mit $p_1(m)$ Prozessoren, welche die Funktionen g aus der Operationsmenge S_m von $\overline{M}(n/m)$ berechnen. Wir beschreiben eine PRAM M(n,m) über K, die f berechnet, indem sie $\overline{M}(n/m)$ schrittweise simuliert.

M(n,m) besitzt $p_0(n/m)p_1(m)$ Prozessoren. Das grundsätzliche Verhalten von M(n,m) ist dadurch charakterisiert, daß jedem Prozessor $\overline{P}_i$ von $\overline{M}(n/m)$ $p_1(m)$ Prozessoren von M(n,m) zugeordnet werden, die sich wie $M_g(m)$ verhalten, wenn im Programm von $\overline{P}_i$ die Operation g ausgeführt werden soll. Eine mögliche Präzisierung der Simulation wird nachfolgend angegeben:

Der Prozessor $\overline{P}_i$ von $\overline{M}(n/m)$ wird durch die $p_1(m)$ Prozessoren

$$P_{(i-1)p_1(m)+1},\ldots,P_{ip_1(m)}$$

von $M(n,m)$ und jedes Register $\bar{r}$ von $\bar{M}(n/m)$ durch die m^2 aufeinanderfolgenden Register

$$Z + (r-1)(m^2+1) +1,\ldots,Z + r(m^2+1) -1$$

von $M(n,m)$ simuliert, $Z = p_0(n/m)m^2$. Die Funktion des Akkumulators $\overline{0i}$ von $\bar{M}(n/m)$ wird von den Registern

$$(i-1)m^2 + 1,\ldots,im^2$$

von $M(n,m)$ übernommen.

Enthält Register $\bar{r}$ von $\bar{M}(n/m)$ die Matrix $A \in \bar{K}$, $\langle\bar{r}\rangle = A$, so ist für $s = (i-1)m + j$, $1 \le i,j \le m$, in $M(n,m)$ $\langle Z + (r-1)(m^2+1) + s\rangle = a_{ij}$. Eine im Akkumulator $\overline{0i}$ von $\bar{M}(n/m)$ enthaltene Matrix wird völlig analog zeilenweise in den Registern $(i-1)m^2 + 1,\ldots,im^2$ von $M(n,m)$ gespeichert. Natürliche Zahlen in Registern von $\bar{M}(n/m)$ werden in allen zugehörigen Registern von $M(n,m)$ gespeichert. Die Simulation eines load- bzw. store-Befehls von $\bar{M}(n/m)$ wird dann durch $2\lceil m^2/p_1(m)\rceil$ (parallele) load- und store-Befehle von $M(n,m)$ realisiert.

Ein arithmetischer Befehl g aus dem Programm des $\bar{M}(n/m)$-Prozessors $\bar{P}_i$ wird simuliert, indem die $p_1(m)$ zu $\bar{P}_i$ gehörigen Prozessoren sowie die Register

$(i-1)m^2 + j$, $1 \le j \le m^2$, $Z + r(m^2+1)$, $r = 1,2,\ldots$ von $M(n,m)$ $M_g(m)$ simulieren. Die Simulation eines (parallelen) Schrittes von $\bar{M}(n/m)$ wird begonnen, sobald die Simulation des vorhergehenden Schrittes beendet worden ist.

Da $\bar{M}(n/m)$ $\bar{f}_m$ berechnet, enthält jedes Register $\bar{r}$, $1 \le \bar{r} \le (n/m)^2$, von $\bar{M}(n/m)$ nach Beendigung der Rechnung von $\bar{M}(n/m)$ einen Wert $(\bar{f}_m(\overline{A(1)},\ldots,\overline{A(k)}))_{ij}$, $1 \le i,j \le n/m$. Damit sind nach Beendigung der Simulation die Elemente von $(\overline{f(A(1),\ldots,A(k))})_{ij}$ in den Registern

$(r-1)(m^2+1) + 1,\ldots,r(m^2+1) - 1$ von $M(n,m)$ gegeben. $M(n,m)$ überträgt deshalb nach dem stop-Befehl von $\overline{M}(n/m)$ die Inhalte dieser Register in die Register $1,\ldots,n^2$. Damit haben wir insgesamt:

1.1.6. Satz

Sei $\overline{M}(n/m)$ eine PRAM über $K^m \times K^m$, die eine m-Partitionierung $\overline{f}_m$ von f berechnet. Ist $M(n,m)$ eine PRAM, die $\overline{M}(n/m)$ in der beschriebenen Weise simuliert, so berechnet $M(n,m)$ die Funktion f.

□

Wir nennen $M(n,m)$ *partitionierte PRAM* zur Berechnung von f.
Wir verwenden synonym auch den Begriff *partitionierter Algorithmus*. In dieser Terminologie wird $\overline{M}(n/m)$ als *externer* Algorithmus und die $M_g(m)$ als *interne* Algorithmen bezeichnet.

Kapitel 2 beschäftigt sich im Hinblick auf die Berechnungszeit mit den Zeitkosten der oben beschriebenen Simulation. Die Tatsache, daß die Simulationskosten der Speicherbefehle für $p_1(m) < m^2$ groß werden können, berücksichtigen wir im allgemeinen PRAM-Modell nicht. Die Untersuchung partitionierter Netzwerkalgorithmen in Kapitel 4 trägt jedoch bei der Berechnung der Kommunikationszeit der Engpaßwirkung der Akkumulatoren im Fall $p_1(m) < m^2$ Rechnung.

Ausgehend von der Operationsmenge S_m einer Funktional-PRAM $\overline{M}(n/m)$, z.B.
S_m = {LU-Zerlegung, Inversion einer $(m \times m)$-Matrix; Multiplikation, koeffizientenweise Multiplikation, Addition zweier $(m \times m)$-Matrizen},
lassen sich partitionierte Algorithmen im Sinne obiger Definition konstruieren. In der Regel ist der Ausgangspunkt jedoch ein bekannter nichtpartitionierter Algorithmus $M(n)$ zur Berechnung von f. Ziel ist es, aus $M(n)$ einen partitionierten Algorithmus zu gewinnen, der eine ähnliche Struktur wie $M(n)$ besitzt. Eine sinnvolle Definition des Begriffs der Partitionierung eines Algorithmus $M(n)$ erfordert es, anstelle der Grundoperationen S_m eine gemeinsame Operationsmenge S für alle betrachteten PRAMs festzulegen. Dabei wird unter einer Operation g aus S eine Klasse von Funktionen $(g_m)_{m \in \mathbb{N}}$, $g_m \in S_m$, verstanden.

1.1.7. Definition

Seien $M'(n)$ und $M''(n)$ PRAMs über $K^{m'} \times K^{m'}$ bzw. $K^{m''} \times K^{m''}$ mit $p'(n)$ bzw. $p''(n)$ Prozessoren und gleicher Operationsmenge S.

$M'(n) = M''(n)$:<=>

$p'(n) = p''(n)$, $n \in \mathbb{N}$, und zu jedem Prozessor P' von $M'(n)$ gibt es genau einen Prozessor P'' von $M''(n)$ mit prog P' = prog P''.

□

Damit kann die Partitionierbarkeit von $M(n)$ formal erfaßt werden:

1.1.8. Definition

Eine PRAM $M(n)$ über K mit $p(n)$ Prozessoren, die f berechnet, heißt *partitionierbar* :<=>

Für alle $m \in \mathbb{N}$ mit $m|n$ existiert eine PRAM $\overline{M}(n/m)$ über $K^m \times K^m$ mit $p(n/m)$ Prozessoren, die eine m-Partitionierung $\overline{f}_m$ von f berechnet, und es gilt: $\overline{M}(n) = M(n,1) = M(n)$.

Die partitionierten PRAMs $M(n,m)$, welche die $\overline{M}(n/m)$ in der oben dargestellten Weise simulieren, heißen *Partitionierungen* oder *partitionierte Versionen* von $M(n)$.

□

Wir erhalten für alle betrachteten Funktionen f: Ein partitionierter Algorithmus $M(n,m)$ ist Partitionierung von $M(n)$, wenn der externe Algorithmus $\overline{M}(n/m)$ und der ursprüngliche Algorithmus $M(n)$ gleich sind. Sei nämlich $f = (f_n)_{n \in N}$ eine Klasse von Funktionen $f_n: (K^n \times K^n)^k \to K^n \times K^n$, und für jedes n existiere eine partitionierbare PRAM $M(n)$ zur Berechnung von f. Unter Zugrundelegung einer gemeinsamen Operationsmenge S für alle PRAMs gilt dann :

$$M(n/m) = \overline{M}(n/m) \ , \ m \in \mathbb{N},\ m|n.$$

(Man ersetze in 1.1.8. n durch n/m.)

Ein Zusammenhang zwischen $M(n)$ und den internen Algorithmen einer partitionierten Version $M(n,m)$ ist wie folgt gegeben:

1.1.9. Fakt

Sei $M(n)$ eine partitionierbare PRAM, die f berechnet, und bezeichne S die gemeinsame Operationsmenge von $M(n)$ und den zugehörigen Funktional-PRAMs $\overline{M}(n/m)$. Für $g \in S$ sei $M_g(m)$ eine PRAM mit $p_1(m)$ Prozessoren, die g berechnet. Dann gibt es eine Folge $(g_1,\ldots,g_r)$, $g_i \in S$, mit: Die Nacheinander-

ausführung der internen Algorithmen $M_{g_1}(n) \ldots M_{g_r}(n)$ ist ein paralleler Algorithmus zur Berechnung von f.

Beweis:

Für $m = n$ ist $p_0(n/m) = p_0(1) = 1$. $(g_1, \ldots, g_r)$ ist dann gerade das Programm von $\overline{M}(1)$.

□

Häufig ist $r = 1$, also $f \in S$, und für $p_1 \equiv p$ kann der interne Algorithmus so gewählt werden, daß $M_f(n) = M(n)$.

Dann erhält man als typisches Beispiel für die Grundoperationen S_m von $\overline{M}(n/m)$:

S_m = {Addition, Subtraktion, Multiplikation, Inversion von $(m \times m)$-Matrizen, f angewendet auf $(m \times m)$-Matrizen}.

Die Aufnahme der Matrix-Inversion in S_m bedeutet in der Praxis, daß es Eingaben gibt, für die der partitionierte Algorithmus keine Lösungen liefert, obwohl es Lösungen gibt.

Wir beschließen diesen Abschnitt mit der Definition des dualen partitionierten Algorithmus:

1.1.10. Definition

Seien $\overline{M}(n/m)$ und $\overline{M}^*(m)$ PRAMs über $K^m \times K^m$ bzw. über $K^{n/m} \times K^{n/m}$, die eine m- bzw. n/m-Partitionierung von f mit $p_0(n/m)$ bzw. mit $p_1(m)$ Prozessoren berechnen.

Seien ferner $M_g(m)$ und $M_g^*(n/m)$ PRAMs über K mit $p_1(m)$ bzw. $p_0(n/m)$ Prozessoren, die für $g \in S$ g_m bzw. $g_{n/m}$ berechnen. Bezeichnet man die zugehörigen partitionierten PRAMs mit $M(n,m)$ und $M^*(n,n/m)$, so heißt $M^*(n,n/m)$ *dualer* partitionierter Algorithmus von $M(n,m)$.

□

Offensichtlich berechnet $M^*(n,n/m)$ ebenso wie $M(n,m)$ f mit $p_0(n/m)p_1(m)$ Prozessoren.

1.2. BEISPIELE

Als Beispiele führen wir zunächst partitionierte Versionen mehrerer paralleler Algorithmen der Linearen Algebra an. Die Beispiele werden beschrieben durch die Angabe der Menge S der Grundoperationen (interne Algorithmen) und die Darstellung der externen Algorithmen als Prozeduren, welche die Operationen aus S verwenden. Man überzeugt sich im Einzelfall leicht davon, daß die angegebenen externen Algorithmen m-Partitionierungen der gewünschten Funktionen berechnen, so daß wir auf den formalen Nachweis verzichten können. Da es sich um Partitionierungen bekannter paralleler Algorithmen handelt, sind die Programme der externen Algorithmen entweder identisch zu Programmen geläufiger Algorithmen über K oder unterscheiden sich lediglich um Operationen, die im Falle der Blockgröße 1 redundant sind, wie z.B. die LU-Zerlegung oder Transposition einer (1×1)-Matrix. Da nur die Struktur partitionierter Algorithmen veranschaulicht werden soll, wird die Nicht-Singularität aller Blockmatrizen, die im Verlauf eines externen Algorithmus invertiert werden, vorausgesetzt. Im parallelen Fall verwenden wir ausschließlich Prozessoranzahlfunktionen der Form $p_0(x) = x^r$, $r \in \mathbb{N}$. Der Zeitbedarf der Algorithmen wird in Kapitel 2 untersucht.

Im folgenden seien $A, B, C, X \in K^n \times K^n$, falls nicht ausdrücklich etwas anderes verlangt wird.

1.2.1. Matrix-Addition $A + B = C$

$S = \{\text{Matrix-Addition}\}$.

Für $p_0(n/m) = n/m$ ist $\overline{M}_+(n/m)$ gegeben durch:

```
begin
  for i = 1,...,n/m do
    for all j = 1,...,n/m pardo
      c̄_ij := ā_ij + b̄_ij
end
```

$\overline{M}_+(n/m)$ mit $p_0(n/m) = 1$ bzw. $p_0(n/m) = (n/m)^2$ sind ähnlich trivial gegeben.

1.2.2. Matrix-Multiplikation A*B = C

S = {Matrix-Addition, Matrix-Multiplikation}

Für $p_0(n/m) = 1$ ist $\overline{M}_*(n/m)$ gegeben durch:

```
begin
  for i = 1,...,n/m do
   for j = 1,...,n/m do
    begin
     c̄_ij := 0̄;
     for k = 1,...,n/m do
      c̄_ij := c̄_ij + ā_ik b̄_kj
    end
end
```

Im Falle $p_0(n/m) = n/m$ wird entweder die i- oder die j-Schleife, für $p_0(n/m) = (n/m)^2$ werden beide äußeren Schleifen parallel ausgeführt. Für $n/m = 2^s$, $s \in \mathbb{N}$, ist ein $\overline{M}_*(n/m)$ für $(n/m)^3$ Prozessoren beispielsweise gegeben durch:

```
begin
  for all i,j,k = 1,...,n/m pardo
   c̄_ij(k) := ā_ik b̄_kj;
  for r = 1,...,log n/m do
   for all k = 1,...,n/(2^r m) pardo
    c̄_ij(k) := c̄_ij(2k-1) + c̄_ij(2k);
  c̄_ij := c̄_ij(1)
end
```

1.2.3. Matrix-Transposition

S = {Matrix-Transposition, Zuweisung a := b von Blockmatrizen}.

Für $p_0(n/m) = 1$ ist $\overline{M}_{trans}(n/m)$ gegeben durch:

```
begin
  for i = 1,...,n/m do
   for j = 1,...,n/m do
    begin
     ā_ij := ā_ij^T;
     ā_ij := ā_ji
    end
end
```

Durch Parallelisierung der Schleifen erhält man $\overline{M}_{trans}(n/m)$ für n/m und

$(n/m)^2$ Prozessoren.

1.2.4. Gauß-Jordan-Algorithmus

Es soll das lineare Gleichungssystem AX = B gelöst werden.

S = {Matrix-Addition, Matrix-Multiplikation, Matrix-Inversion}.

Für $p_0(n/m) = 1$ ist $\overline{M}_{GJ}(n/m)$ gegeben durch:

```
begin
  for j = 1,...,n/m do
    begin
     ā'_jj := ā_jj^-1;
     for i = 1,...,n/m do                    (**)
      begin
       t̄_ij := ā_ij ā'_jj;
       for k = 1,...,n/m do                  (*)
        ā_ik := ā_ik - t̄_ij ā_jk;
       for k = 1,...,n/m do                  (*)
        b̄_ik := b̄_ik - t̄_ij b̄_jk;
       x̄_ij := b̄_ij ā'_ii
      end
    end
end
```

Durch Parallelisierung der mit (*) gekennzeichneten Schleifen erhält man eine parallele Version für n/m Prozessoren. Parallelisiert man zusätzlich noch die i-Schleife (**), erhält man $\overline{M}_{GJ}(n/m)$ für $p_0(n/m) = (n/m)^2$.

Die folgenden drei Beispiele fügen die partitionierten Algorithmen aus [14] in unser Modell ein.

1.2.5. LU-Zerlegung A = LU

Es wird vorausgesetzt, daß es eine untere Dreiecksmatrix L mit $l_{ii} = 1$, $1 \le i \le n$, und eine obere Dreiecksmatrix U gibt, so daß A = LU. L und U sollen berechnet werden. Der angegebene Algorithmus ist eine partitionierte Version des Crout-Algorithmus.

S = {Matrix-Multiplikation, Matrix-Addition, Inversion von Dreiecksmatrizen, LU-Zerlegung}.

Für $p_0(n/m) = 1$ ist $\overline{M}_{LU}(n/m)$ gegeben durch:

```
begin
 ā_11 = l̄_11 ū_11;

 l̄'_11 := l̄_11^-1;

 ū'_11 := ū_11^-1;
 for i = 1,...,n/m do                    (*)
  begin
   ū_1i := l̄'_11 ā_1i;
   l̄_i1 := ā_i1 ū'_11
  end
 for k = 2,...,n/m do
  begin
   for j = 1,...,k-1 do
   ā_kk := ā_kk - l̄_kj ū_jk;
   ā_kk = l̄_kk ū_kk;

   l̄'_kk := l̄_kk^-1;

   ū'_kk := ū_kk^-1;
   for i = k+1,...,n/m do                (*)
```

```
      begin
       for j = 1,...,k-1 do
        begin
         ā_ki := ā_ki - l̄_kj ū_ji;
         ā_ik := ā_ik - l̄_ij ū_jk
        end
       ū_ki := l̄'_kk ā_ki;
       l̄_ik := ā_ik ū'_kk
      end
    end
  end
```

Ein $\overline{M}(n/m)$ für $p_0(n/m) = n/m$ erhält man durch Parallelisierung der mit (*) gekennzeichneten Schleifen.

Für $p_0(n/m) = (n/m)^2$ führen wir $\overline{M}(n/m)$ aus:

```
 begin
  for k = 1,...,n/m - 1 do
   begin
    ā_kk = l̄_kk ū_kk;

    l̄'_kk := l̄_kk^-1;

    ū'_kk := ū_kk^-1;
    for all i = k+1,...,n/m pardo
     begin
      ū_ki := l̄'_kk ā_ki;
      l̄_ik := ā_ik ū'_kk
     end
    for all i,j = k+1,...,n/m pardo
     ā_ij := ā_ij - l̄_ik ū_kj
   end
  ā_n/m,n/m = l̄_n/m,n/m ū_n/m,n/m
 end
```

1.2.6. Inversion von Dreiecksmatrizen

A sei eine obere Dreiecksmatrix. Es soll die Matrix $A' = A^{-1}$ berechnet werden.

S = {Matrix-Addition, Matrix-Multiplikation, Inversion von Dreiecksmatrizen}.

Für $p_0(n/m) = 1$ ist $\overline{M}_{/}(n/m)$ gegeben durch:

```
 begin
  for i = 1,...,n/m do                          (*)
   ā'ii := āii^-1;
  for i = 1,...,n/m - 1 do
   for j = 1,...,n/m - i do                     (*)
    begin
     for k = 1,...,j do
      āi,i+j := āi,i+k ā'i+k,i+j + āi,i+j;
     ā'i,i+j := -ā'ii āi,i+j
    end
 end
```

Ein $\overline{M}_{/}(n/m)$ für n/m Prozessoren erhält man wiederum durch Parallelisierung der mit (*) gekennzeichneten Schleifen. Die Parallelisierung für $p_0(n/m) = (n/m)^2$ ist ähnlich wie in Beispiel 1.2.5.:

```
 begin
  for all i = 1,...,n/m pardo
   ā'ii := āii^-1;
  for k = 1,...,n/m - 1 do
   begin
    for all i = 1,...,n/m - k , j = k,...,n/m - i pardo
     āi,i+j := āi,i+j + āi,i+j-k ā'i+j-k,i+j;
    for all i = 1,...,n/m - k pardo
     ā'i,i+k := -ā'ii āi,i+k
   end
 end
```

1.2.7. Rücksubstitution

Sei A eine obere Dreiecksmatrix, B, X ε K^n. Es ist AX = B zu lösen.

S = {Vektor-Addition, Matrix-Vektor-Multiplikation, Inversion von Dreiecksmatrizen}.

Gemäß [14] ist eine Partitionierung des Rücksubstitutionsalgorithmus für $p_0(n/m) = 1$ gegeben durch:

```
begin
  for i = 1,...,n/m do                    (*)
   a'_ii := a_ii^-1;
  x_n/m := a'_n/m,n/m b_n/m;
  for j = n/m,...,2 do
   begin
    for i = 1,...,j-1 do                  (*)
     b_i := b_i - a_ij x_j;
    x_j-1 := a'_j-1,j-1 b_j-1
   end
end
```

Parallelisierung der markierten Schleifen liefert den partitionierten Algorithmus für n/m Prozessoren.

1.2.8. Rücksubstitution ohne Inversion

In 1.2.7. reduziert sich der Algorithmus für die Blockgröße $m = n$ auf die beiden internen Algorithmen Inversion einer Dreiecksmatrix und Matrix-Vektor-Multiplikation. Fakt 1.1.9. und die anschließende Bemerkung legen es jedoch nahe, eine Partitionierung des Rücksubstitutionsalgorithmus anzugeben, welche für die Blockgröße $m = n$ selbst wiederum die Rücksubstitution liefert. In diesem Fall haben wir

$S = \{$Vektor-Addition, Matrix-Vektor-Multiplikation, Rücksubstitution$\}$

$\overline{M}_R(n/m)$ erhält man aus dem in 1.2.7. angegebenen externen Algorithmus, indem man auf die Matrix-Inversion zu Beginn verzichtet und die Blöcke $\overline{x}_{n/m}$ bzw. $\overline{x}_{j-1}$ des Lösungsvektors in Zeile 4 bzw. 9 mit Hilfe der internen Rücksubstitution berechnet.

Die bis hierher angeführten Beispiele erhielten wir auf elementarem Wege, indem wir Operationen bekannter Algorithmen über K durch Blockoperationen ersetzt haben. Auf diese Weise ist gleichzeitig sichergestellt worden, daß die partitionierten Algorithmen Partitionierungen des Ausgangsalgorithmus sind. Im nächsten Beispiel verwenden wir die Rekursivität des Ausgangsalgorithmus, um eine partitionierte Version zu entwickeln.

1.2.9. Fast-Fourier-Transformation, FFT

Ziel ist die Berechnung der diskreten Fourier-Transformation (DFT) $(q(0),\ldots,q(N-1))^T \in \mathbb{R}^N$ von $(c(0),\ldots,c(N-1))^T \in \mathbb{R}^N$, $N = 2^n$, $n \in \mathbb{N}$.
Ist ω eine primitive N-te Einheitswurzel, so sind die Koeffizienten $q(j)$ der DFT definiert durch

$$\text{(i)} \quad q(j) := \sum_{k=0}^{N-1} c_0(k)\omega^{jk}$$

Für $b \in \mathbb{N}$, $0 \le b < 2^r$ bezeichne $\mathrm{bin}_r b = b_{r-1}\ldots b_0$ die Binärdarstellung der Länge r von b und $\mathrm{rev}_r b = b_0\ldots b_{r-1}$ die Binärdarstellung von $2^{r-1}b_0 + \ldots + 2b_{r-2} + b_{r-1}$. Dem Rekursionsschema des FFT-Algorithmus liegt folgende Darstellung von (i) zugrunde (vgl. [11]):

$$\text{(ii)} \quad q(j) = \sum_{k_{n-1}=0,1} \cdots \sum_{k_0=0,1} c_0(k_{n-1}\ldots k_0)\omega^{j(2^{n-1}k_{n-1}+\ldots+k_0)}$$

Bei der Ausführung des FFT-Algorithmus werden rekursiv folgende Werte berechnet:

$$c_1(j_0 k_{n-2}\ldots k_0) := \sum_{k_{n-1}=0,1} c_0(k_{n-1}\ldots k_0)\ \omega^{j_0 k_{n-1} 2^{n-1}}.$$

Für $2 \le d \le n$:

$$c_d(j_0\ldots j_{d-1}k_{n-d-1}\ldots k_0) :=$$

$$\sum_{k_{n-d}=0,1} c_{d-1}(j_0\ldots j_{d-2}k_{n-d}\ldots k_0)\ \omega^{(j_0+2j_1+\ldots+2^{d-1}j_{d-1})k_1)k_{n-d}2^{n-d}},$$

und es ist

$$(q(0),\ldots,q(N-1))^T = (c_n(\mathrm{rev}_n 0),\ldots,c_n(\mathrm{rev}_n(N-1))^T.$$

Ziel ist die Partitionierung der FFT mit Blockgröße M, $M = 2^m$, $0 \le m \le n$. Wir fassen deshalb in (ii) die Summation über $k_0,\ldots,k_{m-1}$ zusammen:

$$\text{(iii)} \quad q(j) = \sum_{k_{n-1},\ldots,k_m} \sum_{k_{m-1},\ldots,k_0} c_0(K^m + K_m)\omega^{jK_m}\omega^{jK^m}$$

Dabei ist $K^m = k_{n-1}2^{n-1} + \ldots + k_m 2^m$ und $K_m = k_{m-1}2^{m-1} + \ldots + 2^0$.

Diese Darstellung mit einer inneren und äußeren Summe führt zur Angabe von internen Algorithmen und externem Algorithmus der partitionierten FFT. Das Vorgehen ist verwandt mit dem Prim-Faktor-Algorithmus zur Berechnung einer 2-dimensionalen FFT (s. [19]). Der Unterschied besteht darin, daß wir im Falle $N = 2^n$ keine teilerfremden Faktoren von N angeben können, so daß für interne Algorithmen und externen Algorithmus keine *primitiven* Einheitswurzeln zur Verfügung stehen.
Die nachfolgende partitionierte Version der FFT berechnet aus dem Eingabevektor

$$(c_0(0), c_0(N/M), \ldots, c_0((M-1)N/M), c_0(1), \ldots, c_0((M-1)N/M+1), \ldots,$$
$$c_0(N/M-1), \ldots, c_0(N-1))^T$$

die Ausgabe $(q(0), \ldots, q(N-1))^T$. D.h. um den partitionierten Algorithmus beginnen zu können, muß eine Permutation der Koeffizienten von $(c_0(0), \ldots, c_0(N-1))^T$ erfolgen, z.B. durch Transponieren der Matrix

$$\begin{bmatrix} c_0(0) & c_0(1) & \ldots & c_0(N/M-1) \\ c_0(N/M) & c_0(N/M+1) & \ldots & c_0(2N/M-1) \\ & & & \\ c_0((M-1)N/M) & & \ldots & c_0(N-1) \end{bmatrix}$$

Die Operationsmenge S_M besteht aus der koeffizientenweisen Multiplikation zweier Vektoren, sowie aus den Abbildungen f_M und ρ_M, die definiert sind durch

$$f_M: \mathbb{R}^M \to \mathbb{R}^M$$
$$\begin{bmatrix} c_0(k) \\ \cdot \\ \cdot \\ \cdot \\ c_0((M-1)N/M+k) \end{bmatrix} \to \begin{bmatrix} c_m(\mathrm{bin}_m 0, \mathrm{bin}_{n-m} k) \\ \cdot \\ \cdot \\ \cdot \\ c_m(\mathrm{bin}_m M-1, \mathrm{bin}_{n-m} k) \end{bmatrix} =: \bar{c}_m(k).$$

f_M kann berechnet werden, indem die ersten m Iterationen der oben angegebenen Rekursion ausgeführt werden.

$$\rho_M: \mathbb{R}^M \to \mathbb{R}^M$$
$$\begin{bmatrix} v(0) \\ \\ v(M-1) \end{bmatrix} \to \begin{bmatrix} v(rev_M 0) \\ \\ v(rev_M(M-1)) \end{bmatrix} .$$

Für $M = N$ liefert $\rho_N \circ f_N$ gerade die DFT von $(c_0(0),\ldots,c_0(N-1))^T$.
Wir bezeichnen die koeffizientenweise Multiplikation zweier Vektoren v und w durch $v * w$ und setzen für $d \in \mathbb{N}$ $v^d := v * \ldots * v$.
Sei $\Omega := (1,\omega,\omega^2,\ldots,\omega^{M-1})^T$. Wir beschreiben den externen Algorithmus $\overline{M}_{FFT}(N/M)$ zur Berechnung der DFT von $(c_0(0),\ldots,c_0(N-1))^T$:

Eingabe:

$(c_0(0),\ldots,c_0((M-1)N/M),\ldots,c_0(N/M-1),\ldots,c_0(N-1))^T =$

$(\overline{c_0}(0),\ldots,\overline{c_0}(N/M-1))^T$, $\overline{c_0}(k) \in \mathbb{R}^M$.

Schritt 1:

Berechnung von $f_M(\overline{c_0}(k)) = \overline{c_m}(k)$ für $k = 0,\ldots,N/M-1$.

Schritt 2:

Berechnung von $\rho_M(\overline{c_m}(k))$ für $k = 0,\ldots,N/M-1$ und von $\rho_M(\Omega)$.

Schritt 3:

Rekursive Berechnung von

$\overline{c}_{m+i}(k) = (c_{m+i}(rev_m 0, bin_{n-m} k),\ldots,c_{m+i}(rev_m M-1, bin_{n-m} k))^T$

für $1 \le i \le n-m$:

$i = 1$:

$$\overline{c}_{m+1}(j_0 k_{n-m-2}\ldots k_0) := \overline{c}_m(0k_{n-m-2}\ldots k_0) + \overline{c}_m(1k_{n-m-2}\ldots k_0) * \Omega^{2^{n-m-1}}.$$

Für $i = 2,\ldots,n-m$:

$$\overline{c}_{m+i}(j_0\ldots j_{i-1}k_{n-m-i-1}\ldots k_0) :=$$
$$\overline{c}_{m+i-1}(j_0\ldots j_{i-2}0k_{n-m-i-1}\ldots k_0) + \overline{c}_{m+i-1}(j_0\ldots j_{i-2}1k_{n-m-i-1}\ldots k_0)$$
$$*\Omega^{(j_0+\ldots+j_{i-1}2^{i-1})2^{n-m-i}}.$$

Resultat: $\overline{c}_n(j_0\ldots j_{n-m-1}) = \overline{q}(j_{n-m-1}\ldots j_0)$ mit

$\overline{q}(j) = (q(jM),q(jM+1),\ldots,q((j+1)M-1))^T$, $0 \le j \le N/M-1$.

Schritt 4:

Berechnung von $rev_{n-m} j$ für alle $j = 0,\ldots,N/M-1$ und $\overline{q}(j) = \overline{c}_n(rev_{n-m} j)$.

1.2.10. Block-Sortieralgorithmen

In [2] werden Möglichkeiten zur Partitionierung von Sortieralgorithmen dargestellt. Wir beschreiben das allgemeine Vorgehen in unserem Modell. S_m enthält zwei Grundoperationen σ und μ. σ sortiert eine Folge der Länge m. Die merge-split-Operation μ ist eine verallgemeinerte Vergleichsoperation: μ berechnet aus zwei Folgen $(v_1,\ldots,v_m)$, $(w_1,\ldots,w_m)$ mit $v_i \le v_{i+1}$, $w_i \le w_{i+1}$, $1 \le i \le m-1$, die beiden Folgen $(v_1',\ldots,v_m')$, $(v_{m+1}',\ldots,v_{2m}')$ mit $v_i' \in \{v_1,\ldots,v_m,w_1,\ldots,w_m\}$, $v_i' \le v_{i+1}'$, $1 \le i \le 2m$.
Um eine Folge $(v_1,\ldots,v_n)$ zu sortieren, verfährt der externe Algorithmus $\overline{M}_{sort}(n/m)$ eines partitionierten Sortieralgorithmus wie folgt: In einem ersten Schritt werden die Blöcke $\overline{v}_j = (v_{(j-1)m+1},\ldots,v_{jm})$, $1 \le j \le n/m$, sortiert (Berechnung von σ). Danach kann ein beliebiger bekannter Sortieralgorithmus für Folgen der Länge n/m, der nur mit Vergleichsoperationen arbeitet verwendet werden.

1.2.11. Assoziative Verknüpfung

Für eine assoziative Operation o soll $f(x_1,\ldots,x_n) = x_1 \circ \ldots \circ x_n$ berechnet werden. Analog zur Matrix-Transposition (1.2.3.) wendet M(n/m) zunächst f auf die einzelnen Blöcke $\overline{x}_1,\ldots,\overline{x}_{n/m}$ an und berechnet dann mit o das Produkt

$$f(\overline{x}_1) \circ \ldots \circ f(\overline{x}_{n/m}) = (x_1 \circ \ldots \circ x_m) \circ \ldots \circ (x_{(n/m-1)m+1} \circ \ldots \circ x_n).$$

Wir beschließen diesen Abschnitt durch ein sehr elementares Beispiel für einen partitionierten Algorithmus, das jedoch praktische Relevanz besitzt.

1.2.12. Unabhängige Operationen

Für $r \in \mathbb{N}$ bestehe die Aufgabe darin, unabhängig voneinander r Probleme der Größe m zu lösen. Beispielsweise sollen r Gleichungssysteme der Größe m gelöst werden oder r verschiedene Matrixprodukte berechnet werden. Mit $n = rm$ läßt sich die Berechnung in unserer Terminologie als partitionierter Algorithmus beschreiben. Der einzige interne Algorithmus besteht dabei in der Lösung eines der r Teilprobleme, $\overline{M}(n/m)$ ist die r-

fache Anwendung dieser inneren Operation auf die verschiedenen Blöcke der Eingabedaten.

1.3. PARTITIONIERBARKEIT UND REKURSION

Das Beispiel der FFT legt nahe, daß ein enger Zusammenhang zwischen partitionierten und rekursiven Algorithmen existiert. Dort wurde nämlich eine partitionierte Version des Algorithmus gewonnen, indem das bekannte Rekursionsverfahren an der Stelle der gewünschten Blockgröße M aufgetrennt wurde. Die ersten m Iterationen haben dann die Funktion eines internen, die verbleibenden n-m Iterationen die Funktion des externen Algorithmus erfüllt. Der zugrundeliegende allgemeine Sachverhalt soll in diesem Abschnitt präzisiert werden.
Wir setzen voraus $n = 2^r$, $r \in \mathbb{N}$, und die gemeinsamen Grundoperationen aller betrachteten PRAMs seien gegeben durch

S = {Matrix-Addition/Subtraktion, Matrix-Multiplikation,
Matrix-Inversion, f}

Indem wir f in die Menge der Grundoperationen aufnehmen, setzen wir wieder voraus, daß f eine Klasse $(f_n)_{n \in \mathbb{N}}$ von Funktionen $f_n: (K^n \times K^n)^k \to K^n \times K^n$ ist. Um zu gewährleisten, daß allen Algorithmen immer dieselbe Prozessoranzahlfunktion zugrundeliegt, beschränken wir uns auf den sequentiellen Fall, also $p(n) = p_0(n/m) = p_1(m) = 1$. Zunächst gilt dann für die Operationen aus S:

1.3.1. Fakt

Es gibt partitionierbare Algorithmen zur Matrix-Addition, -Multiplikation und -Inversion.

□

Solche Algorithmen sind im vorangegangenen Beispielteil (1.2.1., 1.2.2., 1.2.4.) angegeben worden. D.h. für die Grundoperationen +,-,*,/ können die internen Algorithmen immer so gewählt werden, daß sie selbst wiederum partitionierbar sind. Falls es darüberhinaus partitionierbare Algorithmen M(n), $n \in \mathbb{N}$, zur Berechnung von f gibt, so läßt sich die Partitionierung

von M(n) rekursiv in den internen Algorithmen fortsetzen. Genauer zeigen wir:

1.3.2. Satz

Für alle $n \in \mathbb{N}$ gibt es eine partitionierbare PRAM M(n) über K, die f berechnet <=>

Für alle $n \in \mathbb{N}$ gibt es eine PRAM $\overline{M}_f(2)$ über $\overline{K}^{n/2} \times \overline{K}^{n/2}$, die $\overline{f}_{n/2}$ berechnet.

Beweis:

"=>" ist durch die Definition der Partitionierbarkeit gegeben. Für den Beweis der Rückrichtung muß gezeigt werden: für alle $m \in \mathbb{N}$ mit $m|n$ existiert eine Funktional-PRAM $\overline{M}_f(n/m)$, die $\overline{f}_m$ berechnet. Man erhält M(n) dann, indem man $M(n) := \overline{M}_f(n/1)$ setzt.

Sei $n/m = 2^i$. Wir führen eine Induktion nach i aus. $\overline{M}_f(2)$ existiert nach Voraussetzung. Sei $\overline{M}_f(2^i) = \overline{M}_f(n/2^{r-i})$ eine Funktional-PRAM, die $\overline{f}_{2^{r-i}}$ berechnet.

Wegen 1.3.1. gibt es PRAMs $\overline{M}_{+,-}(2)$, $\overline{M}_{*}(2)$, $\overline{M}_{/}(2)$ über $K^{m/2} \times K^{m/2}$, die (m/2)-Partitionierungen der Addition, Multiplikation und Inversion von $(m \times m)$-Matrizen berechnen. Eine (m/2)-Partitionierung von f angewendet auf $(m \times m)$-Matrizen wird durch $\overline{M}_f(2)$ über $K^{m/2} \times K^{m/2}$, die nach Voraussetzung existiert, berechnet.

Das Programm von $\overline{M}_f(2^{i+1})$ über $K^{m/2} \times K^{m/2}$ wird dann durch die Programme von $\overline{M}_f(2^i)$ und der $\overline{M}_o(2)$ festgelegt, $o \in \{+,-,*,/,f\}$.

Hierzu wird jede Operation o im Programm von $\overline{M}_f(2^i)$ ersetzt durch das Programm von $\overline{M}_o(2)$.

□

Für den Beweis von 1.3.2. ist es nicht erforderlich, die Operationsmenge S in der angegebenen Weise festzulegen. Verwendet wird lediglich, daß es zur Berechnung jeder Grundoperation einen partitionierbaren Algorithmus gibt. 1.3.2. kann für die Konstruktion partitionierter Algorithmen zur Berechnung von f genutzt werden. Es genügt, einen externen Algorithmus anzugeben, der $\overline{f}_{n/2}$ berechnet und dessen interne Algorithmen partitionierbar oder gleich f sind. Insbesondere wird durch den Beweis von 1.3.2. beschrieben, wie aus einem rekursiv dargestellten Algorithmus zur Berechnung von f ein partitionierter Algorithmus konstruiert werden kann.

Als Beispiel betrachten wir das Rekursionsverfahren von Borodin und Munro (vgl. [10]) zur Inversion einer Dreiecksmatrix. Für $n = 2^r$ und eine obere Dreieckmatrix $A \in K^n \times K^n$, die sich darstellen läßt in der Form

$$A = \begin{bmatrix} B & C \\ 0 & D \end{bmatrix}$$

mit $B, C, D \in K^{n/2} \times K^{n/2}$, B und D obere Dreiecksmatrizen, wird das Rekursionsschema beschrieben durch

$$A^{-1} = \begin{bmatrix} B^{-1} & -B^{-1}CD^{-1} \\ 0 & D^{-1} \end{bmatrix}$$

Gemäß Satz 1.3.2. muß es dann einen partitionierbaren Algorithmus M(n) über K geben, der A^{-1} berechnet. Wir beschreiben den externen Algorithmus $\overline{M}(n/m)$ für $m \leq n/2$: Die m-Partitionierung $\overline{A}$ von A ist gegeben durch

$$\begin{bmatrix} \overline{b}_{11} & \cdots & \overline{b}_{1,n/2m} & \overline{c}_{11} & \cdots & \overline{c}_{1,n/2m} \\ \overline{0} & \cdots & \overline{b}_{n/2m,n/2m} & \overline{c}_{1,n/2m} & \cdots & \overline{c}_{n/2m,n/2m} \\ & & & \overline{d}_{11} & \cdots & \overline{d}_{1,n/2m} \\ & 0 & & & & \\ & & & \overline{0} & \cdots & \overline{d}_{n/2m,n/2m} \end{bmatrix}$$

$\overline{M}(n/m)$ verwendet die internen Operationen Multiplikation von $(m \times m)$-Matrizen, Inversion von $(m \times m)$-Dreiecksmatrizen und führt damit $\log(n/2m)$ Iterationen der oben angegebenen Rekursion auf $\overline{B}$ und $\overline{D}$ aus. Abschließend verwendet $\overline{M}(n/m)$ die partitionierte Matrix-Multiplikation zur Berechnung von $-\overline{B^{-1}CD^{-1}}$.

Durch 1.3.2. wird die enge Verwandtschaft von rekursiven und partitionierten Algorithmen ausgedrückt. Da wir darauf verzichtet haben, den Rekursionsbegriff formal zu definieren, können wir jedoch keine prä-

zise Aussage darüber machen, welche rekursiven Algorithmen partitionierbar sind. Auf der intuitiven Ebene ist umgekehrt allerdings klar, daß jeder partitionierte Algorithmus zur Berechnung von f, dessen Grundoperationsmengen die Funktion f selbst enthalten, ein rekursives Verfahren zur Berechnung von f darstellt.

Abschließend verallgemeinern wir deshalb den Begriff der Partitionierung und führen Partitionierungen der Tiefe d eines Algorithmus ein.

1.3.3. Definition

Für $m_i | m_{i-1}$ sei $M_i = \overline{M}_i(m_{i-1}/m_i)$ eine PRAM über $K^{m_i} \times K^{m_i}$, und es gelte: M_i berechnet eine m_i-Partitionierung einer internen Operation von M_{i-1}, $1 \leq i \leq d-1$.
Ist $m_0 = n$ und $M(n)$ eine partitionierbare PRAM mit externem Algorithmus $\overline{M}_0(n/m)$, so heißt $(M_0,\ldots,M_{d-1})$ *Partitionierung der Tiefe d* von $M(n)$.

□

Durch geschachtelte Partitionierungen im Sinne von 1.3.3. werden nicht nur rekursive, sondern auch zusammengesetzte Algorithmen beschrieben. Als Beispiel geben wir eine Partitionierung der Tiefe 5 des Algorithmus $M(n)$ an, der r Produkte von jeweils s $(k \times k)$-Matrizen berechnet.

Für r', s', k' mit $r'|r$, $s'|s$, $k'|k$ setzen wir:
$n = rsk$, $m_1 = r'sk$, $m_2 = s'k$, $m_3 = k$, $m_4 = k'$.

$\overline{M}_1(n/m_1) = \overline{M}_1(r/r')$ berechnet r/r' Produkte von jeweils s $(k \times k)$-Matrizen (vgl. 1.2.13.).
$\overline{M}_2(m_1/m_2) = \overline{M}_2(r's/s')$ entspricht dem r'-mal ausgeführten Algorithmus 1.2.12.
$\overline{M}_3(m_2/m_3) = \overline{M}_3(s')$ ist die Multiplikation von s' $(k \times k)$-Matrizen.
$\overline{M}_4(k/k')$ ist die partitionierte Multiplikation von zwei $(k \times k)$-Matrizen mit Blockgröße k'.

2. EFFIZIENZEN UND BERECHNUNGSZEITEN

In diesem Kapitel soll das Laufzeitverhalten paralleler partitionierter Algorithmen untersucht werden. Abschnitt 2.2. behandelt die Laufzeit im sequentiellen Fall. In vielen Beispielen ist die sequentielle Laufzeit partitionierter Algorithmen von der gleichen Größenordnung wie die sequentielle Laufzeit nicht-partitionierter Algorithmen. Allgemein wird für eine große Klasse von Algorithmen nachgewiesen, daß die Berechnungskomplexität partitionierter Algorithmen im sequentiellen Fall durch die Berechnungszeit nicht-partitionierter Algorithmen nach unten beschränkt ist.

Die Betrachtung des parallelen Falls in 2.3. und 2.4. wird motiviert durch die Notwendigkeit, parallele Algorithmen, die in der Regel für eine allgemeine Prozessoranzahlfunktion $p(n)$ konzipiert sind, mit einer konstanten Anzahl p real verfügbarer Prozessoren auszuführen. Wegen des zusätzlichen freien Parameters der Blockgröße m erlauben es parallele partitionierte Algorithmen, die für allgemeine Prozessoranzahlfunktionen entworfenen Algorithmen unmittelbar (ohne Änderung der Programme der Prozessoren) auf eine Maschine mit p Prozessoren zu übertragen. Der Prozessoranzahlbeschränkung wird dabei durch Wahl einer geeigneten Blockgröße m Rechnung getragen. Die mit dieser Vorgehensweise verbundene Laufzeit wird in 2.3. verglichen mit der Laufzeit einer schrittweisen Simulation der nicht-partitionierten Version desselben parallelen Algorithmus durch eine Maschine mit p Prozessoren.

Berücksichtigt man, daß es verschiedene Möglichkeiten geben kann, ein Verfahren zur Berechnung von f zu parallelisieren (vgl. die Beispiele in 1.2.), d.h. daß es PRAMs mit unterschiedlichen Prozessoranzahlfunktionen für die Ausführung der internen Algorithmen und des externen Algorithmus gibt, so erhält man verschiedene partitionierte Algorithmen zur Berechnung derselben Funktion. In 2.4. vergleichen wir die Effizienzen und Laufzeiten dieser Partitionierungen untereinander. Als Resultat erhalten wir eine optimale Effizienz im Falle der sequentiellen Ausführung des externen oder internen Algorithmus.

2.1. BESTIMMUNG DER BERECHNUNGSZEITEN IM ALLGEMEINEN FALL UND FÜR DIE BEISPIELALGORITHMEN

Für eine partitionierte PRAM $M(n,m)$ mit externem Algorithmus $\overline{M}(n/m)$ und internen Algorithmen $M_{g_1}(m),\ldots,M_{g_r}(m)$ bezeichnen wir die zugehörigen Laufzeiten mit $t(n,m)$, $t_0(n/m)$, $t_i(m)$, $1 \le i \le r$. Im Falle sequentieller Maschinen verwenden wir die Bezeichnungen $t(seq)(n,m)$, $t_0(seq)(n/m)$, $t_i(seq)(m)$. O.E. sei $t_i(m) \ge t_{i+1}(m)$, $1 \le i \le r-1$. Zunächst gilt offensichtlich

2.1.1. Fakt

$$t_0(n/m)\min_i t_i(m) \le t(n,m) \le t_0(n/m)\max_i t_i(m).$$

□

Da wir in der Definition der $\overline{M}(n/m)$ simulierenden Maschine $M(n,m)$ synchronisiertes Verhalten verlangt haben, läßt sich $t(n,m)$ aber auch präziser bestimmen. Ist nämlich $t_{0,i}(n/m)$ die Anzahl der Berechnungsschritte von $\overline{M}(n/m)$, in denen die Operation g_i ausgeführt wird, aber keine der Operationen $g_1,\ldots,g_{i-1}$, so erhält man

2.1.2. Fakt

$$t(n,m) = \sum_i t_{0,i}(n/m)t_i(m)$$

□

Wir berechnen die Laufzeiten der Algorithmen 1.2.2.-1.2.9. gemäß 2.1.2.. Den in Tabelle 2.1.3. zusammengestellten Laufzeiten der internen Algorithmen liegen dabei dieselben Algorithmen und Grundoperationen wie in 1.2. zugrunde, welche für die nicht-partitionierte Ausführung gegebenenfalls um redundante Operationen bereinigt worden sind. Tabelle 2.1.4. enthält die Werte $t_{0,i}(n/m)$ der externen Algorithmen für die verschiedenen Operationen g_i. Die resultierenden Gesamtlaufzeiten sind in 2.1.5. zusammengestellt. Wir fassen bei allen Laufzeitberechnungen Matrix-Multiplikation und -Addition zu einer Operation AB+C zusammen.

2.1.3. Laufzeiten der internen Algorithmen

g_1 : Matrix-Multiplikation/Addition

g_2 : Matrix-Vektor-Multiplikation/Addition

g_3 : Matrix-Transposition

g_4 : Inversion Dreiecksmatrix

g_5 : Matrix-Inversion

g_6 : LU-Zerlegung einer Matrix

g_7 : Rücksubstitution

g_8 : Koeffizientenweise Vektormultiplikation

g_9 : Blockzuweisung Matrix (Vektor)

g_{10}: Berechnung von f_M (1.2.9.)

g_{11}: Berechnung von ρ (1.2.9.)

$p_1(m)$	1	m	m^2	m^3
g_1	m^3	m^2	m	1+log m
g_2	m^2	m	1+log m	
g_3	m^2	m	1	
g_4	$(1/3)m^3-(1/3)m$	m^2-m	2m-2	
g_5	$2m^3+2m^2$	$4m^2$	4m	
g_6	$(1/3)m^3+m^2+(7/6)m$	$(3/2)m^2+(5/2)m-1$	5m-3	
g_7	$(1/2)m^2+(3/2)m$	2m		
g_8	m	1		
g_9	m^2 (m)	m (1)	1	
g_{10}	O(M log M)	O(log M)		
g_{11}	O(M log M)	O(log M)		

2.1.4. Laufzeiten $t_{0,i}(n/m)$ der externen Algorithmen

($y := n/m$, $Y := N/M$)

$\overline{M}(n/m)$	$p_0(y)$	1	y	y^2	y^3
Matrix-Multiplikation	$t_{0,1}(y)$	y^3	y^2	y	$1+\log y$
	$t_0(y)$	y^3	y^2	y	$1+\log y$
Matrix-Transposition	$t_{0,3}(y)$	y^2	y	1	
	$t_{0,9}(y)$	y^2	y	1	
	$t_0(y)$	$2y^2$	$2y$	2	
Gauß-Jordan-Algorithmus	$t_{0,1}(y)$	$2y^3+2y^2$	$4y^2$	$4y$	
	$t_{0,5}(y)$	y	y	y	
	$t_0(y)$	$2y^3+O(y^2)$	$4y^2+y$	$5y$	
LU-Zerlegung	$t_{0,1}(y)$	$y^3/3+O(y^2)$	$3y^2/2+O(y)$	$3y-1$	
	$t_{0,4}(y)$	$2y-2$	$2y-2$	$2y-2$	
	$t_{0,6}(y)$	y	y	y	
	$t_0(y)$	$y^3/3+O(y^2)$	$3y^2/2+O(y)$	$6y-3$	
Inversion Dreiecksmatrix	$t_{0,1}(y)$	$(1/3)(y^3-y)$	y^2-y	$2y-2$	
	$t_{0,4}(y)$	y	1	1	
	$t_0(y)$	$(1/3)y^3+O(y)$	y^2-y+1	$2y-1$	
Rücksubstitution (1.2.7.)	$t_{0,1}(y)$	$(1/2)(y^2+y)$	$2y-1$		
	$t_{0,4}(y)$	y	1		
	$t_0(y)$	$y^2/2+3y/2$	$2y$		
Rücksubstitution (1.2.8.)	$t_{0,1}(y)$	$(1/2)(y^2-y)$	$y-1$		
	$t_{0,7}(y)$	y	y		
	$t_0(y)$	$y^2/2+3y/2$	$2y-1$		
Fast-Fourier-Transformation	$t_{0,8}(y)$	$Y \log Y$	$\log Y$		
	$t_{0,9}(y)$	$Y \log Y$	$\log Y$		
	$t_{0,10}(y)$	Y	1		
	$t_{0,11}(y)$	$Y + 1$	2		
	$t_0(y)$	$2Y(\log Y)+2Y+1$	$2\log Y+3$		

2.1.5. Laufzeiten t(n,m) der partitionierten Algorithmen

Matrix-Multiplikation

$p_0 \backslash p_1$	1	m	m^2	m^3
1	n^3	n^3/m	n^3/m^2	$n^3/m^3(\log m+1)$
n/m	n^2m	n^2	n^2/m	$n^2/m^2(\log m+1)$
$(n/m)^2$	nm^2	nm	n	$n/m\ (\log m+1)$
$(n/m)^3$	$m^2\log(n/m)+m^3$	$m\log(n/m)+m^2$	$\log(n/m)+m$	$\log(n/m)+1$

(Im Falle der Parallelisierung mit $(n/m)^3$ ist die Marix-Addition der dominierende interne Algorithmus!)

Matrix-Transposition

$p_0 \backslash p_1$	1	m	m^2
1	$2n^2$	$2n^2/m$	$2n^2/m^2$
n/m	2nm	2n	2n/m
$(n/m)^2$	$2m^2$	2m	2

Gauß-Jordan-Algorithmus

$p_0 \backslash p_1$	1	m	m^2	m^3
1	$2n^3+O(nm^2)$	$2n^3/m+O(n^2)$	$2n^3/m^2+O(n^2/m)$	$2(n/m)^3\log m+a$
n/m	$4n^2m+O(nm^2$	$4n^2+O(nm)$	$4(n^2/m+n)$	$4(n/m)^2\log m+O(n)$
$(n/m)^2$	$6nm^2+O(nm)$	8nm	8n	$4(n/m)\log m+O(n)$

$a = O((n^2/m^2)\log m)$

LU-Zerlegung

$p_0 \backslash p_1$	1	m	m^2
1	$n^3/3+O(n^2m)$	$n^3/(3m)+O(n^2)$	$n^3/(3m^2)+O(n^2/m)$
n/m	$3n^2m/2+O(nm^2)$	$3n^2/2$	$3n^2/(2m)+O(n)$
$(n/m)^2$	$20nm^2/3+O(nm)$	$11nm-9m^2$	11n-9m

Inversion Dreiecksmatrix

$p_0 \backslash p_1$	1	m	m^2	m^3
1	$n^3/3-n/3$	$n^3/(3m)+O(nm)$	$n^3/(3m^2)+O(n)$	$(1/3)(n^3/m^3)\log m+a$
n/m	$n^2m+O(m^3-nm^2)$	$n^2+O(m^2-nm)$	$n^2/m+O(m-n)$	$(n^2/m^2)\log m+b$
$(n/m)^2$	$2nm^2-O(m^3)$	$2nm-m^2-m$	$2n-2$	$(2n/m-2)(\log m+1)+O(m)$

$a = O(n-(n/m)\log m)$

$b = O(m-(n/m)\log m)$

Rücksubstitution (1.2.8.)

$p_0 \backslash p_1$	1	m	m^2	m^3
1	$n^2m/2-O(nm^2)$	$n^2/2-nm/2+2n$	$n^2/2m+3n/2$	$(1/2)(n^2/m^2-n/m)(\log m+1)+O(n)$
n/m	$nm^2+O(nm-m^3)$	$nm-m^2+2n$	$3n-m$	$(n/m-1)(\log m+1)+O(n)$

Fast-Fourier-Transformation

$p_0 \backslash p_1$	1	M
1	$2N\log N+M\log M$	$2(N/M)\log N+\log M$
N/M	$2M\log N+M\log M$	$2\log N+\log M$

Im folgenden machen wir die vereinfachende Annahme $t_i(m) = t_1(m)$, $1 \le i \le r$. Diese Annahme ist insbesondere dann gerechtfertigt, wenn alle internen Algorithmen dieselbe Berechnungskomplexität besitzen. Größenordnungsmäßig ist dies für 1.2.1.-1.2.7. gegeben. In den anderen Fällen sind die nachfolgenden Aussagen auf die Subalgorithmen zu beziehen, für welche die Annahme gleicher Laufzeiten der internen Algorithmen Gültigkeit hat. Wir haben nun also

$$t(n,m) = t_0(n/m)t_1(m).$$

Wir setzen in 2.2. und 2.3. voraus, daß $M(n,m)$ Partitionierung von $M(n)$ ist. Ist $M(n)$ partitionierbar, also $M(n/m) = \overline{M}(n/m)$ für alle m, so sind die zugehörigen Laufzeiten $t_0(n/m)$ und $t(n/m)$ gleich. Wir interpretieren t_0 und t als Funktionen $\mathbb{R} \to \mathbb{R}$ und können o.E. setzen $t_0(x) = t(x)$, $x \in \mathbb{R}$.

2.2. ZEITKOMPLEXITÄT SEQUENTIELLER PARTITIONIERTER ALGORITHMEN

Sei $p_0 = p_1 = p \equiv 1$. Setzen wir

$$T(seq)(n) := \min_m t(seq)(n,m),$$

so erhalten wir

2.2.1. Fakt

$T(seq)(n) \leq t(seq)(n,1) = t_1(seq)(1)t_0(seq)(n) = \text{const. } t(seq)(n),$

$T(seq)(n) \leq t(seq)(n,m) = t_1(seq)(n)t_0(seq)(1) = \text{const.} t_1(seq)(n).$

□

Im allgemeinen ist davon auszugehen, daß $t_1(seq)(1) = t_0(seq)(1) = 1$. Ist das Produkt $t_0(seq)(n/m)t_1(m) = t(seq)(n,m)$ für alle Blockgrößen m gleich, so ist dies gleichbedeutend damit, daß $t(seq)(n,m) = \theta(t(seq)(n))$, d.h. die sequentielle Berechnungskomplexität des partitionierten Algorithmus ist für alle Blockgrößen m von der gleichen Größenordnung wie die Berechnungszeit des nicht-partitionierten Algorithmus M(n). Im allgemeinen Fall folgt aus 2.2.1.:

Falls die minimale sequentielle Laufzeit des partitionierten Algorithmus für die triviale Partitionierung mit Blockgröße m = 1 oder m = n angenommen wird, ist die Laufzeit des partitionierten Algorithmus für alle Blockgrößen m durch die Laufzeit des nicht-partitionierten Algorithmus M(n) bzw. $M_1(m)$ nach unten beschränkt. Prinzipiell läßt 2.2.1. die Möglichkeit offen, daß es Blockgrößen m gibt, für die der partitionierte Algorithmus schneller ist. Für einfache Laufzeitfunktionen ist jedoch leicht zu sehen, daß die minimale sequentielle Laufzeit des partitionierten Algorithmus für eine der trivialen Partitionierungen (m=1 oder m = n) angenommen werden muß:

2.2.2. Satz

Für $t_0(seq)(x) = x^s \log^r cx$, $t_1(seq)(x) = x^v \log^t cx$, $r,s,t,v \in \mathbb{R}$, c = const. und r = 0 oder t = 0 ist

$T(seq)(n) = t(seq)(n,1)$ oder $T(seq)(n) = t(seq)(n,n)$.

Beweis:

Es sind folgende Laufzeiten zu betrachten:

$t(seq)(n,m) = n^s m^{v-s}$

$n^i \leq p \leq n^j$ für $i \leq j$ bzw. $n^j \leq p \leq n^i$ für $i \geq j$. Wir interessieren uns für einen Vergleich der Laufzeiten $t(n,m_p)$ von $M(n,m_p)$ und t_p von M.

Intutitiv ist klar, daß die Qualität der oben beschriebenen einfachen Simulation von M(n) durch M von der Effizienz des Ausgangsalgorithmus M(n) abhängt, welche die Auslastung der p Prozessoren von M direkt beeinflußt. Ebenso liegt aber auch ein Einfluß der Effizienz von $M_1(m)$ für die Kosten der in 1.1. beschriebenen Simulation von $\overline{M}(n/m)$ durch M(n,m) nahe. Wir präzisieren deshalb den Effizienzbegriff, der in allen weiteren Überlegungen zur Laufzeit eine wichtige Rolle spielt.

Um von der Effizienz eines parallelen Algorithmus bzw. einer PRAM M(n) zur Berechnung von f sprechen zu können, muß ein sequentieller Algorithmus zur Berechnung von f festgelegt werden, dessen Laufzeit t(seq)(n) als Bezugsgröße dient. Üblicherweise wählt man den schnellsten bekannten sequentiellen Algorithmus zur Berechnung von f. Die Effizienz $E_{p(n)}(M(n))$ von M(n) wird dann definiert durch

2.3.2. Definition

$$E_{p(n)}(M(n)) := t(seq)(n)/(p(n)t(n))$$

□

Zur Vereinfachung der Darstellung setzen wir $E_{p(n)}(M(n)) =: E_{p(n)}$, wenn aus dem Zusammenhang klar ist, welche PRAM gemeint ist. Offensichtlich ist die Effizienz eines parallelen Algorithmus ≤ 1.
Um die Effizienz eines parallelen partitionierten Algorithmus sinnvoll zu definieren, muß die Laufzeit des sequentiellen Bezugsalgorithmus unabhängig von m sein. Hierzu gehen wir aus von dem sequentiellen Algorithmus mit Laufzeit $t_0(seq)(n/m)$ zur Berechnung von $\overline{f}_m$, der als Bezugsgröße für $E_{p_0(n/m)}(\overline{M}(n/m))$ dient, sowie von dem sequentiellen internen Algorithmus mit Laufzeit $t_1(seq)(m)$, in Bezug auf den die Effizienz von $M_1(m)$ bestimmt wird. Unter den hiermit festgelegten sequentiellen partitionierten Algorithmen wählen wir denjenigen mit der kürzesten Laufzeit T(seq)(n) aus (vgl. 2.2.) und setzen

2.3.3. Definition

$$E_{p_0(n/m)p_1(m)}(M(n,m)) := T(seq)(n)/[p_0(n/m)p_1(m)t(n,m)].$$

□

Da $M(n,m)$ Partitionierung von $M(n)$ ist, ist $M(n) = \overline{M}(n/1)$. Die Festlegung eines einheitlichen sequentiellen Bezugsalgorithmus zur Bestimmung der Effizienz bedeutet in diesem Fall $t_0(seq)(n) = t(seq)(n)$. (Die Identität $M(n/m)=\overline{M}(n/m)$ beinhaltet hingegen nur dieselben Laufzeiten $t_0(n/m)=t(n/m)$ für die Prozessoranzahl $p_0(n/m) = p(n/m)$, vgl. Definition 1.1.7..)
Wir verwenden den sequentiellen Algorithmus zur Berechnung von f, in Bezug auf den die Effizienz des partitionierten Algorithmus $M(n,m)$ gemessen wird, gleichzeitig als Bezugsgröße für die Bestimmung von $E_{p(n)}(M(n))$, so daß wir insgesamt haben: $t(seq)(n) = T(seq)(n) = t_0(seq)(n)$..

Wir definieren eine Funktion $q(m) \leq 1$, welche für festes n die Abweichung der sequentiellen Laufzeiten $t(seq)(n,m)$ vom minimalen Wert $T(seq)(n)$ mißt:

2.3.4. Definition

$$q(m) := T(seq)(n)/t(seq)(n,m).$$

□

Berücksichtigt man nur den dominierenden Term der Laufzeitpolynome in den Beispielen, so ist mit Ausnahme von 1.2.7. und 1.2.8. (Rücksubstitution) immer $q(m) = 1$. Für die allgemeinen Überlegungen kann jedoch auf den Korrekturfaktor $q(m)$ nicht verzichtet werden.

Wir können nun die Faktorisierung der Prozessoranzahl und der Laufzeit eines partitionierten Algorithmus auf dessen Effizienz $E_{p_0(n/m)p_1(m)}(M(n,m))$ übertragen:

2.3.5. Lemma

$$E_{p_0(n/m)p_1(m)}(M(n,m)) = q(m)E_{p_0(n/m)}(\overline{M}(n/m))E_{p_1(m)}(M_1(m))$$

Beweis:

$$E_{p_0(n/m)p_1(m)} = T(seq)(n,m)/[p_0(n/m)p_1(m)t(n,m)]$$

$$= q(m)\ t(seq)(n,m)/[p_0(n/m)p_1(m)t(n,m)]$$

$$= q(m)\ t_0(seq)(n/m)t_1(seq)(m)/[p_0(n/m)t_0(n/m)p_1(m)t_1(m)]$$

$$= q(m)E_{p_0(n/m)}(\bar{M}(n/m))E_{p_1(m)}(M_1(m)).$$

□

Mit diesen Grundlagen wenden wir uns dem Vergleich der Laufzeiten t_p und $t(n,m_p)$ zu. Wir zeigen:

2.3.6. Lemma

$t(n,m_p) = O(E_p(M(n,m_p))^{-1}\ t_p) = O([\ q(m_p)E_{p_0(n/m_p)}E_{p_1(m_p)}]^{-1}\ t_p)$

Beweis:

Es ist nach 2.3.3. und wegen 2.3.1. $t(n,m_p) = T(seq)(n)/(pE_p(M(n,m_p)))$. Da $M(n,m_p)$ Partitionierung von $M(n)$ ist, ergibt sich analog zu 2.2.1.:

$t(n,m_p) \le const.t_0(seq)(n)/(pE_p(M(n,m_p)))$

$\quad = const.t(seq)(n)/(pE_p(M(n,m_p)))$.

Für die zu Beginn dieses Abschnitts beschriebene Simulation gilt sicher $t_p \ge t(seq)(n)/p$, so daß wir insgesamt erhalten

$t(n,m_p) \le const.t_p/E_p(M(n,m_p))$.

□

Als untere Schranke für $t(n,m_p)$ erhält man:

2.3.7. Lemma

$t(n,m_p) = \Omega(E_{p(n)}(M(n))E_p(M(n,m_p))^{-1}\ t_p)$

$\quad = \Omega(E_{p(n)}[q(m_p)E_{p_1(m_p)}\ E_{p_0(n/m_p)}]^{-1}\ t_p)$.

Beweis:

Die Simulation von $M(n)$ durch M erfordert höchstens Zeit $t(n)p(n)/p$, also

$t_p \le t(n)p(n)/p = (p(n)/p)(t(seq)(n)/[p(n)E_{p(n)})]$ und somit

$t_p \le T(seq)(n)/(pE_{p(n)}) = [E_p(M(n,m_p))/E_{p(n)}(M(n))]\ t(n,m_p)$

□

Die zweite Gleichung in 2.3.6. und 2.3.7. folgt jeweils aus 2.3.5..

Wir können 2.3.6. und 2.3.7. zusammenfassen zu

2.3.8. Satz

Sei $M(n,m_p)$ Partitionierung von $M(n)$ mit $p_0(n/m_p)p_1(m_p) = p$. Sei ferner M eine PRAM mit p Prozessoren, die $M(n)$ schrittweise simuliert. Bezeichnet $t(n,m_p)$ die Laufzeit von $M(n,m_p)$ und t_p die Laufzeit von M, so gilt:

$$\text{const}.E_{p(n)}(M(n))/E_p(M(n,m_p)) \le t(n,m_p)/t_p \le \text{const}.E_p(M(n,m_p))^{-1}.$$

□

Satz 2.3.8. präzisiert die unmittelbare Anschauung: Die Laufzeit des parallelen partitionierten Algorithmus für p Prozessoren ist im günstigsten Fall von der gleichen Größenordnung wie die Laufzeit des nicht-partitionierten Algorithmus ausgeführt auf einer Maschine mit p Prozessoren. Diese Situation tritt ein, wenn der partitionierte Algorithmus eine große Effizienz besitzt, d.h. wenn $E_p(M(n,m_p)) = O(1)$. Im allgemeinen kann jedoch die Partitionierung des Algorithmus einen Zeitverlustfaktor $\ge E_{p(n)}/E_p$ liefern. Als Spezialfall erhalten wir:

2.3.9. Korollar

Ist $E_{p(n)}(M(n)) = O(1)$, so gilt:

$$t(n,m_p) \ge [q(m_p)E_{p_1(m_p)}(M_1(m_p))]^{-1}\, t_p\ .$$

Beweis:

Da $M(n,m_p)$ Partitionierung von $M(n)$ ist, also $p_0 \equiv p$ und $t_0 \equiv t$, haben wir im Falle konstanter Effizienz von $M(n)$ auch

$$E_{p_0(n/m_p)}(\overline{M}(n/m_p)) = E_{p(n)}(M(n)) = O(1).$$

□

2.4. KOMPLEXITÄTSVERGLEICH VERSCHIEDENER PARALLELISIERUNGEN EINES PARTITIONIERTEN ALGORITHMUS

Ist $\overline{M}(n/m)$ eine Funktional-PRAM mit $p_0(n/m)$ Prozessoren zur Berechnung von $\overline{f}_m$, welche die Operation g verwendet, so kann es verschiedene PRAMs $M_1(1)(m),\ldots,M_1(r)(m)$ zur Berechnung von g geben. $M_1(j)(m)$ besitze $p_1(j)(m)$ Prozessoren. O.E. setzen wir $p_1(1)(m) = 1$. Damit erhält man r

verschiedene partitionierte PRAMs $M(j)(n,m)$ mit $p_0(n/m)p_1(j)(m)$ Prozessoren, $1 \le j \le r$, die $\overline{M}(n/m)$ simulieren.
Sind andererseits $\overline{M(1)}(n/m),\ldots,\overline{M(s)}(n/m)$ Funktional-PRAMs mit $p_0(1)(n/m) = 1,\ldots,p_0(s)(n/m)$ Prozessoren zur Berechnung von $\overline{f}_m$, die alle in ihren Programmen nur die Operation g verwenden, so erhält man insgesamt rs partitionierte PRAMs $M(i,j)(n,m)$ mit $p_0(i)(n/m)p_1(j)(m)$ Prozessoren, $1 \le i \le s$, $1 \le j \le r$, die alle die Funktion f berechnen.
Typischerweise sind die verschiedenen PRAMs gegeben durch verschiedene Parallelisierungen desselben mathematischen Verfahrens, also z.B. die klassische Matrix-Multiplikation ausgeführt mit 1, n, n^2 oder n^3 Prozessoren.
Ausgehend von einer festen Prozessoranzahl p untersuchen wir die Frage, ob sich allgemeine Aussagen darüber machen lassen, welcher der rs partitionierten Algorithmen die größte Effizienz bzw. die kleinste Berechnungszeit besitzt. Wir bezeichnen die Laufzeiten von $\overline{M(i)}(n/m)$, $M_1(j)(m)$ und $M(i,j)(n,m)$ mit $t_0(i)(n/m)$, $t_1(j)(m)$ bzw. $t(i,j)(n,m)$ und setzen wie in 2.2.:

$$T(1)(n) := \min_m t_0(1)(n/m)t_1(m).$$

Damit kann q(m) definiert werden wie in 2.3.4., und die in 2.3.5. angegebene Faktorisierung ist gültig für alle i,j.

Zunächst zeigen wir unabhängig von der Beschränkung der Prozessoranzahl, daß ein partitionierter Algorithmus mit maximaler Effizienz entweder den internen oder den externen Algorithmus sequentiell ausführen muß:
(Wir verzichten in 2.4.1. auf die Bezeichnung von n und m.)

2.4.1. Satz

Für festes j ist $E_{p_1(j)}(M(1,j)) \ge E_{p_0(i)p_1(j)}(M(i,j))$, $1 \le i \le s$.

Für festes i ist $E_{p_0(i)}(M(i,1)) \ge E_{p_0(i)p_1(j)}(M(i,j))$, $1 \le j \le r$.

Beweis:

Aus 2.3.5. und wegen $E_{p_0(i)}(\overline{M(i)}) \le 1$, $E_{p_1(j)}(M_1(j)) \le 1$ erhalten wir:

$$E_{p_1(j)}(M(1,j)) = q(m)E_{p_1(j)}(M_1(j)) \ge q(m)E_{p_1(j)}(M_1(j))E_{p_0(i)}(\overline{M(i)})$$

$$= E_{p_0(i)p_1(j)}(M(i,j)) \text{ und}$$

$$E_{p_0(i)}(M(i,1)) = q(m)E_{p_0(i)}(\overline{M(i)}) \geq q(m)E_{p_0(i)}(\overline{M(i)})E_{p_1(j)}(M_1(j))$$

$$= E_{p_0(i)p_1(j)}(M(i,j)).$$

□

Es soll nun untersucht werden, inwieweit das Resultat 2.4.1. Gültigkeit hat für den Fall, daß für alle Algorithmen die gleiche Prozessoranzahl p verwendet wird. D.h. die Blockgrößen m sind in Abhängigkeit von n und p so zu wählen, daß 2.3.1. erfüllt ist. Speziell seien für festes i,j die Werte m, m_i, m_j gegeben durch

2.4.2.

$$p_0(i)(n/m)p_1(j)(m) = p_0(i)(n/m_i) = p_1(j)(m_j) = p.$$

Die Prozessoranzahlfunktionen $p_0(i)$, $p_1(j)$ sind naturgemäß wachsende Funktionen, und somit folgt aus 2.4.2. $m_i \leq m \leq m_j$.
Wir legen für den Rest diese Kapitels die zulässigen Effizienzfunktionen zu $\overline{M(i)}(n/m)$ und $M_1(j)(m)$ fest durch

2.4.3. $E_{p_0(i)(x)}, E_{p_1(j)(x)} \in \{1/c, 1/\log^r x, 1/x^s \mid r, s, c \in \mathbb{R}\}$

Für diese Funktionen weist man ähnlich wie in 2.2.2. nach:

2.4.4. Lemma
Sei $h(x) := E_{p_0(i)(x)}E_{p_1(j)(x)}$, $x \in [1,n]$. Dann gibt es $M \in [1,n]$ mit:

h ist monoton fallend in $[1,M]$, und h ist monoton wachsend in $[M,n]$.

□

Da die Effizienzen von den Blockgrößen abhängen, die durch 2.4.2. gegeben sind, können wir eine zu 2.4.1. vergleichbare Aussage allgemein nur für einen Teil der möglichen Werte von m zeigen. Wir unterscheiden hierzu die beiden Fälle
- q(m) wächst mit m ("$t_1(1)$ wächst stärker als $t_0(1)$");
- q(m) fällt mit m ("$t_0(1)$ wächst stärker als $t_1(1)$").

2.4.5. Satz

Ist q wachsend, so gilt für $m \in [M,n]$

$$E_p(M(1,j)(n,m_j)) \geq E_p(M(i,j)(n,m)).$$

Ist q fallend, so gilt für $m \in [1,M]$

$$E_p(M(i,1)(n,m_i)) \geq E_p(M(i,j)(n,m)).$$

Beweis:

Im ersten Fall ist wegen $m_j \geq m$ auch $h(m_j) \geq h(m)$, also $E_{p_0(i)(n/m_j)}E_{p_1(j)(m_j)} \geq E_{p_0(i)(n/m)}E_{p_1(j)(m)}$.

Dann liefern 2.3.5. und 2.4.1.:

$$\begin{aligned} E_p(M(i,j)(n,m)) &= E_{p_0(i)(n/m)p_1(j)(m)}(M(i,j)) \\ &= q(m)E_{p_0(i)(n/m)}E_{p_1(j)(m)} \\ &\leq q(m_j)\, E_{p_0(i)(n/m_j)}E_{p_1(j)(m_j)} \\ &= E_{p_0(i)(n/m_j)p_1(j)(m_j)}(M(i,j)(n,m_j)) \\ &\leq E_{p_1(j)(m_j)}(M(1,j)(n,m_j)) = E_p(M(1,j)(n,m_j)). \end{aligned}$$

Teil 2 des Beweises verläuft völlig analog, weshalb wir auf die Ausführung verzichten.

□

2.4.5. läßt sich unmittelbar auf das Laufzeitverhalten partitionierter Algorithmen übertragen:

2.4.6. Korollar

Ist q wachsend, so gilt für $m \in [M,n]$

$$t(i,j)(n,m) \geq t(1,j)(n,m_j),$$

Ist q fallend, so gilt für $m \in [1,M]$

$$t(i,j)(n,m) \geq t(i,1)(n,m_i).$$

□

Im Falle konstanter Effizienzen hat man also:

2.4.7. Korollar

Ist q wachsend und $E_{p_1(j)} = O(1)$, so ist

$$t(i,j) \geq t(1,j).$$

Ist q fallend und $E_{p_0(i)} = O(1)$, so ist

$$t(i,j) \geq t(i,1).$$

□

Es entspricht der Anschauung, daß ein sehr effizienter interner/externer Algorithmus auf möglichst große Matrizen angewendet werden sollte, um die Laufzeit zu minimieren. Das in der Literatur zumeist angegebene Vorgehen besteht darin, die internen Algorithmen sequentiell auszuführen. So wird in [2] beispielsweise davon ausgegangen, daß die merge-split-Operationen eines Blocksortieralgorithmus von jeweils einem Prozessor ausgeführt werden. Im Falle des Beispiels 1.2.11., der unabhängigen Berechnung einzelner Teilprobleme, ist dieses Vorgehen naturgemäß optimal: der externe Algorithmus besitzt Effizienz 1. Für weniger elementare partitionierte Algorithmen muß jedoch überprüft werden, ob nicht die sequentielle Ausführung des externen Algorithmus bessere Laufzeitwerte liefert. Letzteres ist insbesondere zu erwarten, da in vielen Beispielen der dominierende interne Algorithmus die Matrix-Multiplikation ist, die für eine lineare oder quadratische Anzahl Prozessoren mit Effizienz 1 zu parallelisieren ist.
Als Beispiel betrachten wir die partitionierte LU-Zerlegung. Wir beschränken uns auf den internen Algorithmus der Matrix-Multiplikation, sowie auf die dominierenden Laufzeitterme.
Für $p_1(m) = m$ und $p_0(n/m) = (n/m)^2$ ist in diesem Fall $t(n,m) = 3nm$.

$p_0(n/m_p)p_1(m_p) = p$ liefert $m_p = n^2/p$, so daß $t(n,m_p) = 3n^2/p$.

Analog erhält man für $p_1(m) = 1$ und $p_0(n/m) = (n/m)^2$ die Laufzeit $3n^2/p$, jedoch für $p_1(m) = m$ und $p_0(n/m) = 1$ die Laufzeit $(1/3)n^2/p$: offensichtlich ist es in diesem Fall zweckmäßiger, nur den internen Algorithmus zu parallelisieren.

Die folgende Tabelle enthält alle Werte $t(n,m_p)$ für die partitionierte Inversion von Dreiecksmatrizen. Der einzige berücksichtigte interne Algorithmus ist wiederum die Matrix-Multiplikation.

$p_0 \backslash p_1$	1	m	m^2	m^3
1		$n^3/3p-np/3$	$n^3/3p-n/3$	$(n^3/9p)\log p$ -a
n/m	$n^3/p-n^2/p^2$		$n^3/p-n$	$(n^3/4p)\log p/n$ -b
$(n/m)^2$	$2n^3/p-2n^3/p^{3/2}$	$2n^3/p-2n^4/p^2$		$(2n^3/p)\log p/n$ -c

$a = O((n/p^{1/3})\log p)$

$b = O((n^{3/2}/p^{1/2})\log(p/n))$

$c = O(\log(p/n^2))$

3. PARTITIONIERTE PROZESSORNETZWERKE

Das vorangegangene Kapitel befaßte sich ausschließlich mit den Zeitkosten der arithmetischen Operationen (partitionierter) paralleler Algorithmen. Der Zeitbedarf für die Realisierung paralleler Algorithmen auf Parallelrechnern wird jedoch auch durch die Kommunikationszeit beeinflußt, d.i. die Zeit, die für den Datenaustausch zwischen verschiedenen parallel arbeitenden Prozessoren benötigt wird. Aus diesem Grund bildet die Untersuchung partitionierter Netzwerkalgorithmen den Gegenstand des zweiten Teils der Arbeit.

Die Kommunikationszeit eines Algorithmus hängt von zwei verschiedenen Komponenten ab, nämlich von den Datenabhängigkeiten, die aufgrund des gewählten mathematischen Verfahrens auftreten, sowie von der konkreten Architektur des Parallelrechners. Im Hinblick auf die Architektur beschränken wir uns auf das Modell einer Multiprozessoranordnung mit statischem Prozessorverbindungsnetzwerk. Damit hängt die Kommunikationszeit eines Algorithmus von der Topologie des jeweiligen Netzwerkes ab. Wir beschreiben diese Netzwerke durch ungerichtete (Kommunikations-)Graphen $G = (V(G),E(G))$. Dabei symbolisieren die Knoten $V(G)$ die Prozessoren, die Kanten $E(G)$ die bestehenden Verbindungen zwischen den Prozessoren. Die Begriffe "Netzwerk" und "Kommunikationsgraph" werden synonym verwendet.

Typische Beispiele für Verbindungsnetzwerke, die durch Graphen beschrieben werden können sind:

- Lineare Netze (Ringe)
- Gitterstrukturen (u.U. mehrdimensional)
- Baumstrukturen
- Perfect-Shuffle-Netze
- Würfelstrukturen
- Plus-Minus-2I-Verbindungen.

Beschreibungen solcher Netzwerke finden sich beispielsweise in [4], [20] und [22].

Netzwerkalgorithmen verwenden Kommunikationsgraphen, deren Verbindungsstruktur intuitiv gesprochen die Abhängigkeitsstruktur des Algorithmus abbildet, um auf diese Weise die Kommunikationszeit zu minimieren. Eine genaue Definition (partitionierter) Netzwerkalgorithmen erfolgt in Kapitel 4. Ziel dieses Kapitels ist es, Netzwerke zu beschreiben, welche die Abhängigkeitsstruktur partitionierter Algorithmen im intuitiven Sinne abbilden.

Beginnend mit der Einführung einiger allgemeiner graphentheoretischer Definitionen stellen wir in 3.1. unser Konzept des *Hypernetzes* vor, mit dem die Möglichkeit gegeben wird, in gegebenen Netzwerken eine partitionierte Struktur festzulegen oder partitionierte Netzwerke mit sehr regelmäßiger Struktur zu konstruieren. Grundlage für die Konstruktion von Hypernetzen ist eine einfache Äquivalenzklassenbildung in einem gegebenen Graphen.
Die Abschnitte 3.2. und 3.3. zeigen Anwendungsmöglichkeiten auf, die durch das Hypernetzkonzept erschlossen werden. In 3.2. wird die Dualität regelmäßiger partitionierter Netzwerke analysiert. Der Zusammenhang von Kommunikationsgraphen mit dualer Struktur und den in Kapitel 1 definierten dualen partitionierten Algorithmen wird im nächsten Kapitel hergestellt.
In 3.3. werden Hypernetze verwendet, um eine allgemeine Definition d-dimensionaler Netzwerke zu liefern. Die Kantenkomplexität d-dimensionaler Graphen wird untersucht.
Der letzte Abschnitt führt auf das Problem der Beschränkung der Prozessoranzahl zurück. In [4] werden sogenannte Quotientennetzwerke verwendet, um Netzwerke beliebiger Größe durch Netzwerke derselben Topologie aber fest vorgegebener Größe zu emulieren. In 3.4. wird der Zusammenhang zwischen diesen Netzwerkemulationen und unserem Hypernetzmodell präzisiert.

3.1. NETZWERKPARTITIONIERUNG DURCH HYPERNETZE

Sei $G = (V(G),E(G))$ ein zusammenhängender ungerichteter Graph mit Knotenmenge $V(G)$ und Kantenmenge $E(G)$. Für $v,w \in V(G)$ bezeichne $(v,w) \in E(G)$ eine ungerichtete Kante zwischen den Knoten v und w. Insbesondere ist also $(v,w) = (w,v)$.

3.1.1. Definition

Für $v_1,\ldots,v_r \in V(G)$ wird der *von $v_1,\ldots,v_r$ erzeugte Subgraph* $G' = [v_1,\ldots,v_r]$ von g definiert durch:

$V(G') := \{v_1,\ldots,v_r\}$,

$E(G') := \{(v,w) \in E(G) \mid v,w \in V(G')\}$.

Für $E' \subseteq E(G)$ wird der *von E' erzeugte Subgraph* $G' = [E']$ von G definiert durch:

$E(G') := E'$,

$V(G') := \{v \in V(G) \mid$ es existiert $w \in V(G)$ mit $(v,w) \in E'\}$.

$G'' = (V(G),E')$ heißt *von E' erzeugter Teilgraph* von G.

□

3.1.2. Fakt

Sei $V' \subseteq V(G)$ und $E' = E([V'])$. Dann gilt:

$$(V',E') = (V([E']),E').$$

□

3.1.3. Definition

Zwei Graphen G, H heißen *isomorph*, $G \simeq H$, $:\Leftrightarrow$

es existiert eine Bijektion $\iota: V(G) \to V(H)$ mit

$(v,w) \in E(G) \Rightarrow (\iota(v),\iota(w)) \in E(H)$,

$(v',w') \in E(H) \Rightarrow (\iota^{-1}(v'),\iota^{-1}(w')) \in E(G)$.

(Insbesondere ist also $|V(G)| = |V(H)|$ und $|E(G)| = |E(H)|$.)

□

In Kapitel 1 sahen wir, daß die Prozessoranzahl paralleler partitionierter Algorithmen als Produkt der Prozessoranzahl des externen Algorithmus und der internen Algorithmen gegeben ist. Wir setzen eine analoge Faktorisierung für die Knotenzahl des Kommunikationsgraphen G voraus und geben eine allgemeine Beschreibung einer partitionierten Netzwerkstruktur in G.

Sei im folgenden also $|V(G)| = p$ und $p = p_0p_1$, $p_0,p_1 \in \mathbb{N}$.
In G seien zusammenhängende Subgraphen $G_1,\ldots,G_{p_0}$ spezifiziert mit

$$V(G) = \dot{\bigcup_i} V(G_i)\ ,\ |V(G_i)| = p_1\ ,\ 1 \le i \le p_0.$$

Durch Äquivalenzklassenbildung erhält man dann aus G einen Graphen $\overline{G}$ mit $|V(\overline{G})| = p_0$:

3.1.4. Definition

$\overline{G} := (V(G)/\sim, E(G)/\sim)$ heißt *p_1-Hypernetz von G*. Dabei ist $\sim$ definiert durch:

Für $v,w \in V(G)$ ist $v \sim w$, genau dann wenn es einen Subgraphen G_i gibt mit $v,w \in V(G_i)$. Für $(v_1,w_1), (v_2,w_2) \in E(G)$ ist $(v_1,w_1) \sim (v_2,w_2)$, genau dann wenn $v_1 \sim v_2$ und $w_1 \sim w_2$.

Die Knoten von $\overline{G}$ werden bezeichnet durch $g_1,\ldots,g_{p_0}$.

□

Durch $\overline{G}$ werden in G zwei disjunkte Kantenklassen $E_0(G)$ und $E_1(G)$ definiert mit $E(G) = E_0(G) \dot\cup E_1(G)$. $E_1(G)$ enthält dabei die Kanten der Subgraphen G_i, $E_0(G)$ die Kanten von G, die verschiedene Subgraphen miteinander verbinden.

Wegen 3.1.2. ist $[E_1(G)] = [\dot\bigcup_i V(G_i)]$.

D.h. die Kantenmenge $E_1(G)$ erzeugt gerade den Teilgraphen von G, der aus den p_0 Subgraphen $G_1,\ldots,G_{p_0}$ besteht.

Insbesondere besitzt $[E_1(G)]$ also genau p_0 Zusammenhangskomponenten.
Im allgemeinen gilt für den Zusammenhang von G offensichtlich:

3.1.5. Fakt

G zusammenhängend $\Leftrightarrow$ $\overline{G}$ zusammenhängend.

□

Seien nun für $p_0p_1 = p_0'p_1' = p$ in G ein p_1-Hypernetz $\overline{G} = (V(G)/\sim_1, E(G)/\sim_1)$ und ein p_1'-Hypernetz $\overline{G}' = (V(G)/\sim_2, E(G)/\sim_2)$ definiert.

3.1.6. Definition

$\overline{G}$ heißt *Verfeinerung* von $\overline{G}'$:$\Leftrightarrow$

Für $v,w \in V(G)$ folgt: $v \sim_1 w \Rightarrow v \sim_2 w$.

□

3.1.7. Lemma

Ist $\overline{G}$ Verfeinerung von $\overline{G}'$, so gilt: $p_1 | p_1'$, $p_0' | p_0$ und $E_1(G) \subseteq E_1'(G)$, $E_0(G) \supseteq E_0'(G)$.

Beweis:

Nach Definition folgt für alle $(v,w) \in E_1(G)$ auch $(v,w) \in E_1'(G)$. Für einen p_1-Subgraphen G_i und einen p_1'-Subgraphen G_j' mit $V(G_i) \cap V(G_j') \neq \emptyset$ folgt insbesondere $V(G_i) \subseteq V(G_j')$. Somit muß auch gelten $p_1 | p_1'$.

□

Eine unmittelbare Konsequenz von 3.1.7. ist, daß für eine Primzahl p_1 ein p_1-Hypernetz keine Verfeinerung besitzen kann.

Durch ein Hypernetz wird in vorgegebenem G eine Partitionierung charakterisiert. Umgekehrt kann das Hypernetzkonzept aber auch genutzt werden, um neue Graphen mit partitionierter Strutur zu konstruieren. Hierzu geht man aus von einem Graphen H mit p_0 Knoten und ersetzt jeden Knoten i durch einen Graphen G_i mit p_1 Knoten. Die Kanten von H werden durch (beliebige) Kantenmengen zwischen den entsprechenden G_i ersetzt. In jedem der so gewonnenen neuen Graphen G ist H p_1-Hypernetz. Hinsichtlich der Ausführung partitionierter Algorithmen bietet es sich an, solche partitionierten Netzwerke zu konstruieren, die eine möglichst regelmäßige Struktur besitzen. Wir führen deshalb zwei weitere Definitionen ein:

3.1.8. Definition

Sei G ein Graph mit p_1-Hypernetz $\overline{G}$ und zugehörigen Subgraphen $G_1,\ldots,G_{p_0}$.

G heißt *symmetrisch* bez. $\overline{G}$ $:\Leftrightarrow$ $G_i \approx G_j$, $1 \leq i,j \leq p_0$.

□

3.1.9. Definition

Sei G symmetrisch bez. $\overline{G}$ und sei $G_1 \approx G_i$ durch die Graph-Isomorphismen ι_i: $V(G_1) \to V(G_i)$ gegeben.

G heißt *vollständig* bez. $\overline{G}$ $:\Leftrightarrow$

$(g_i,g_j) \in E(\overline{G}) \Leftrightarrow (\iota_i(v),\iota_j(v)) \in E(G)$ für alle $v \in V(G_1)$.

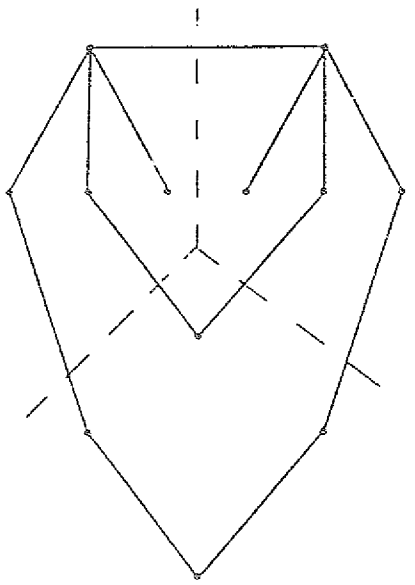
□

Wir beschließen diesen Abschnitt mit einigen Beispielen. Die in den Beispielen angegebene Kapazität κ wird in 3.1.16. definiert.

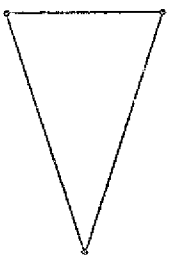
3.1.10. Beliebiger Graph G

G ist 4-Hypernetz von G:

G:



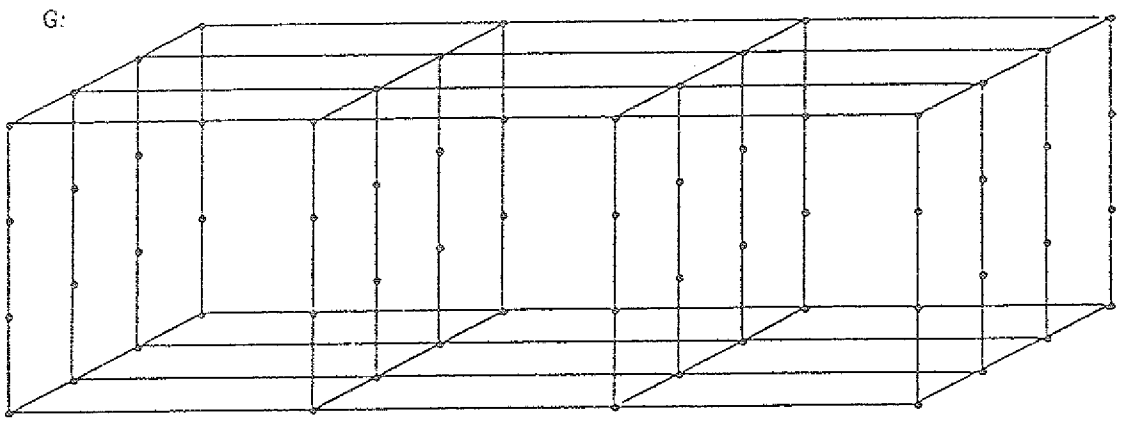
$\overline{G}$:



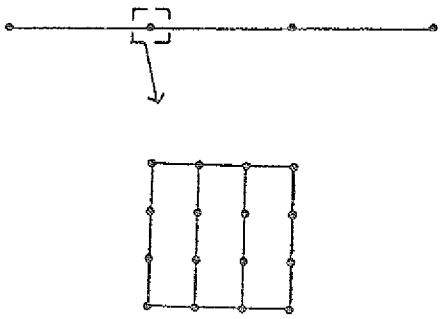
3.1.11. Unvollständiges 3-dimensionales Gitter

G ist symmetrisch bez. allen nachfolgend angegebenen Hypernetzen, aber nicht vollständig. G(3) ist Verfeinerung von G(4), G(2) Verfeinerung von G(2).

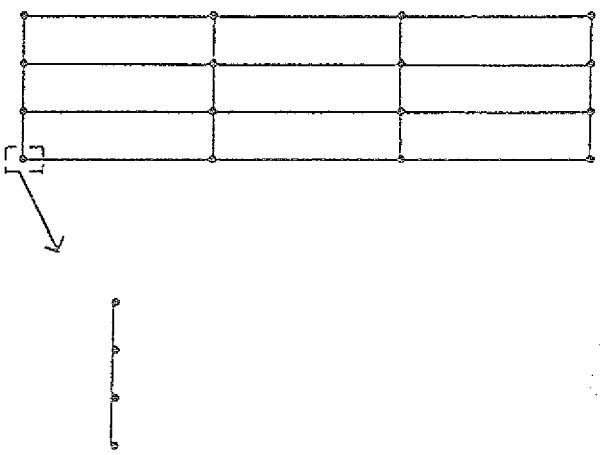
G:



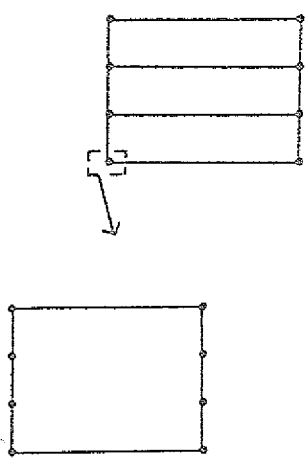
$\overline{G}(1)$ ist 16-Hypernetz; $\kappa(\overline{G}(1)) = 8$:



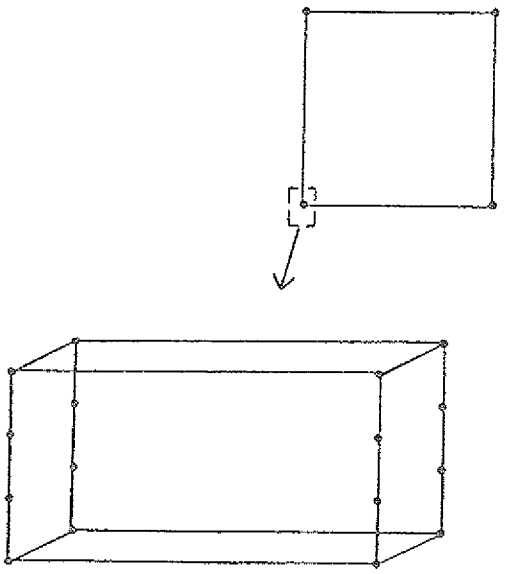
$\overline{G}(2)$ ist 4-Hypernetz; $\kappa(\overline{G}(2)) = 2$:



$\overline{G}(3)$ ist 8-Hypernetz; $\kappa(e_1) = 4$, $\kappa(e_2) = 2$:

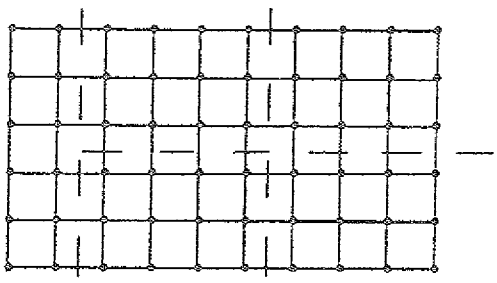


$\overline{G}(4)$ ist 16-Hypernetz; $\kappa(\overline{G}(4)) = 4$:



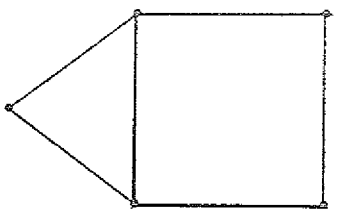
3.1.12. 2-dimensionales Gitter

G:



Das 12-Hypernetz $\overline{G}(1)$ erhält man durch Identifizieren der markierten Knotenmengen:

$\overline{G}(1)$:



Das 6-Hypernetz $\overline{G}(2)$ und das 10-Hypernetz $\overline{G}(3)$ erhält man durch Identifizieren der Knoten der Vertikalen bzw. der Horizontalen; G ist bez. $\overline{G}(1)$ und $\overline{G}(2)$ ein vollständiges symmetrisches Netzwerk.

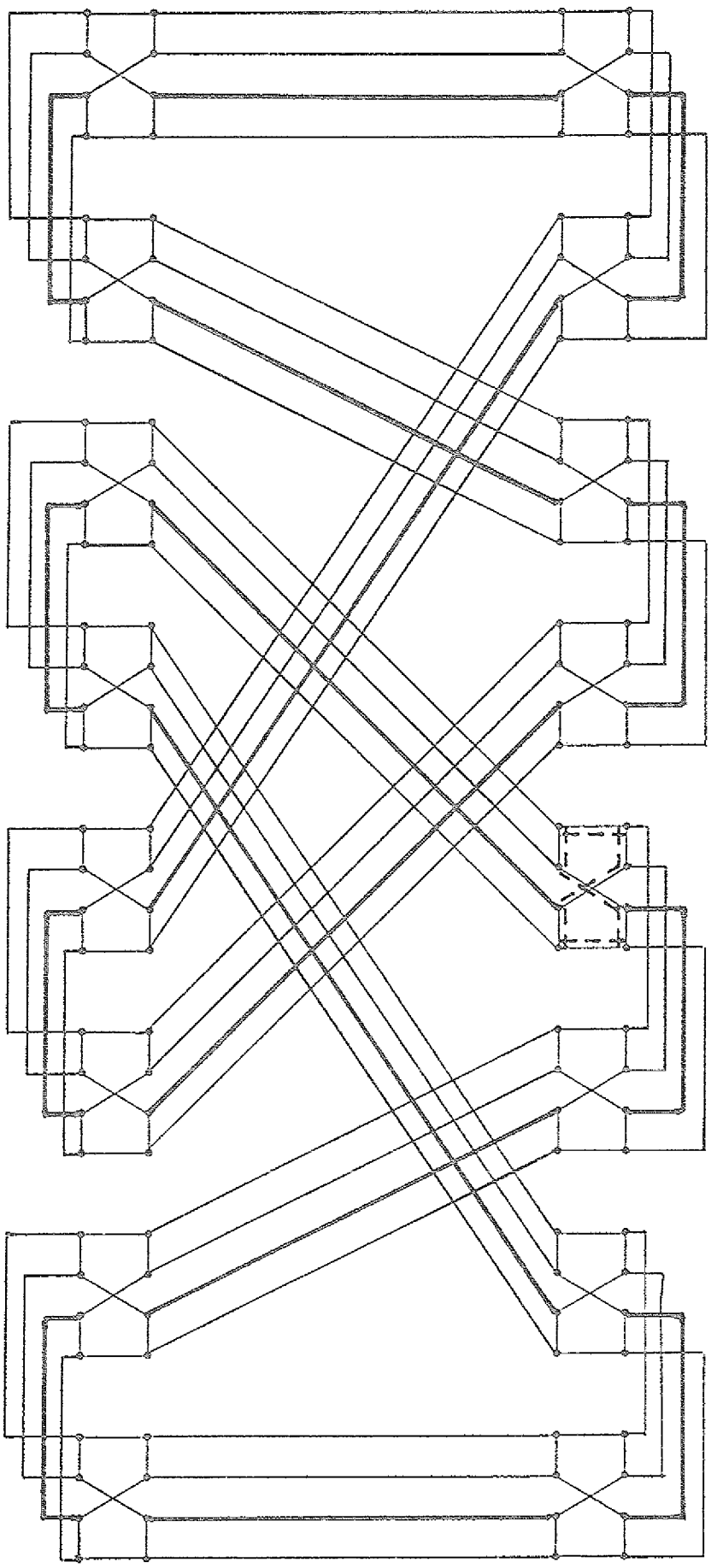
$\overline{G}(2)$:

$\overline{G}(3)$:

3.1.13. Geschachteltes Shuffle-Exchange-Netz

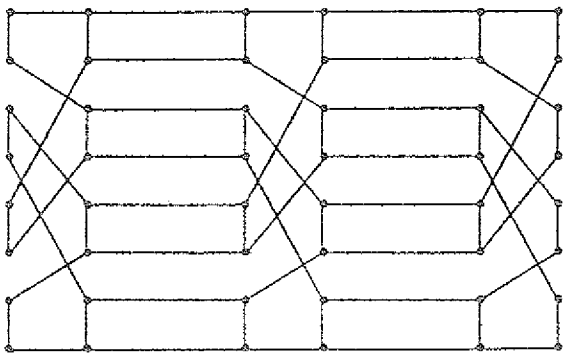
Für die Shuffle-Netze in 3.1.13. und 3.1.14 wählen wir die Darstellung als bipartite Graphen.

$\overline{G}(1)$ ist ein Shuffle-Netz mit 8, $\overline{G}(2)$ ein Shuffle-Netz mit 4 Knoten. D.h. $\overline{G}(1)$ ist 4-Hypernetz, $\overline{G}(2)$ ist 8-Hypernetz. G ist vollständig und symmetrisch bez. $\overline{G}(1)$ und bez. $\overline{G}(2)$.
$\overline{G}(1)$ ist dual zu $\overline{G}(2)$ (s. 3.2.). $\kappa(\overline{G}(1)) = 4 = p_1$, $\kappa(\overline{G}(2)) = 8 = p_1^*$. Für jedes Hypernetz ist jeweils ein Subgraph im Bild markiert.

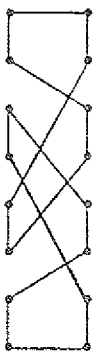


3.1.14. Linear verbundene Shuffle-Netze

G:



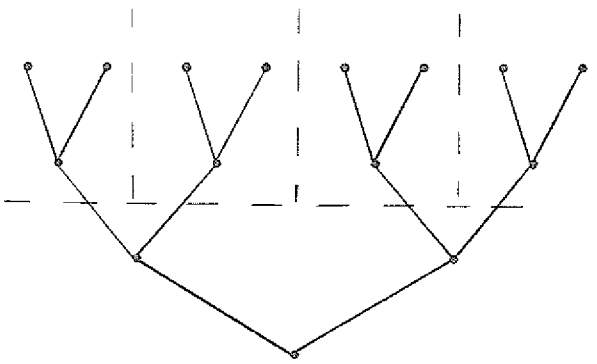
$\overleftarrow{G}(1)$:



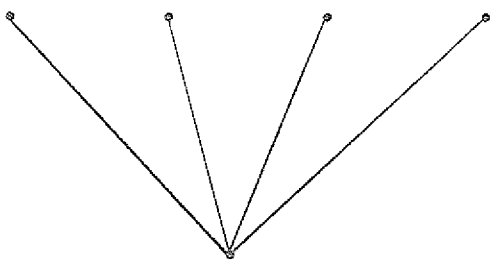
$\overleftarrow{G}(2)$:

3.1.15. Bäume

G:



$\overline{G}$:



$\overline{G}$ ist das einzige nichttriviale Hypernetz von G. Wegen $|V(G)| = 15$, kann G nur 3- oder 5-Hypernetze besitzen. Die Größe p_1 der Subgraphen eines binären Baumes muß von der Form
$p_1 = 1 + 2 + 4 + \ldots + 2^r$, $0 \leq r \leq \log p$, sein. Also kann G kein 5-Hypernetz besitzen.

Die Beispiele zeigen, daß die Anzahl der Kanten von G, die durch eine Hypernetzkante zusammengefaßt werden, variieren kann. Um diesen Parameter, der für die Zeitkosten einer Datenübertragung bedeutsam ist, zu erfassen, definieren wir:

3.1.16. Definition

Sei $\overline{G}$ ein p_1-Hypernetz von G und $(g_i,g_j) \in E(\overline{G})$. Bezeichnet $\kappa((g_i,g_j))$ die Anzahl der Kanten aus $E_0(G)$, welche die Subgraphen G_i und G_j in G miteinander verbinden, so heißt $\kappa((g_i,g_j))$ die *Kapazität* der Kante (g_i,g_j).

Ist $\kappa(e) = \kappa$ für alle $e \in E(\overline{G})$, so heißt $\kappa = \kappa(\overline{G})$ die Kapazität des Hypernetzes $\overline{G}$.

□

3.2. DUALE HYPERNETZE

Angenommen G besitzt ein p_1-Hypernetz $\overline{G}$ und ein p_0-Hypernetz $\overline{G}^*$ mit zugehörigen Subgraphen

$$G_1,\ldots,G_{p_0} \text{ bzw. } G^*_1,\ldots,G^*_{p_1}$$

und Kantenklassen

$$E_0(G),\ E_1(G),\ E^*_0(G),\ E^*_1(G),$$

$$E_0(G) \dot\cup E_1(G) = E(G) = E^*_0(G) \dot\cup E^*_1(G).$$

3.2.1. Definition

$\overline{G}^*$ heißt zu $\overline{G}$ duales Hypernetz :<=>

(i) $E_0(G) = E^*_1(G)$,

(ii) $|V(G_i) \cap V(G^*_j)| = 1$, $1 \le i \le p_0$, $1 \le j \le p_1$.

□

Da mit (i) auch $E_0^*(G) = E_1(G)$ gegeben ist, ist $\overline{G}$ auch dual zu $\overline{G}^*$:

$(\overline{G}^*)^* = \overline{G}$.

3.2.2. Lemma

Sei G ein Graph mit p_1-Hypernetz $\overline{G}$.

Dann besitzt G höchstens ein zu $\overline{G}$ duales p_0-Hypernetz $\overline{G}^*$.

Beweis:

Angenommen $\overline{G}^*$, $\overline{G}'$ sind zu $\overline{G}$ duale p_0-Hypernetze von G. Nach Definition ist

$$E^*_1(G) = E_0(G) = E'_1(G).$$

$E^*_1(G)$ und $E'_1(G)$ erzeugen also den gleichen Teilgraphen von G, bestehend aus den Subgraphen

$$G^*_1,\ldots,G^*_{p_0} \text{ bzw. } G'_1,\ldots,G'_{p_0}.$$

Da dieser Teilgraph genau die p_0 Zusammenhangskomponenten

$$G^*_1,\ldots,G^*_{p_0} \text{ bzw. } G'_1,\ldots,G'_{p_0}$$

besitzt, kann o.E. angenommen werden $G^*_i = G'_i$. Damit gilt für alle Hypernetzkanten

$$(g_i,g_j) \in E(\overline{G}^*) \iff (g_i,g_j) \in E(\overline{G}'),$$

also insgesamt: $\overline{G}' = \overline{G}^*$.

□

In Beispiel 3.1.12. (2-dimensionales Gitter) ist $\overline{G}(3)$ dual zu $\overline{G}(2)$. Ebenso überzeugt man sich leicht davon, daß die beiden Hypernetze $\overline{G}(1)$ und $\overline{G}(2)$ aus Beispiel 3.1.13. und 3.1.14. (geschachteltes Shuffle-Netz, linear verbundene Shuffle-Netze) dual zueinander sind. Die Beispiele zeigen, daß im Falle vollständiger symmetrischer Netzwerke die Subgraphen des dualen Hypernetzes isomorph zu G sind.

3.2.3. Satz

Sei G ein vollständiges symmetrisches Netzwerk bez. dem p_1-Hypernetz $\overline{G}$. Dann besitzt G ein p_0-Hypernetz $\overline{G}^*$ mit $G^*_i \simeq \overline{G}$, und G ist bez. $\overline{G}^*$ vollständig symmetrisch.

3.2.4. Korollar

Ist darüberhinaus die Kapazität der Kanten von $\overline{G}$ minimal in dem Sinne, daß für alle $e \in E(\overline{G})$ $\kappa(e) = p_1$, so gilt:

$\overline{G}^*$ ist dual zu $\overline{G}$ und $\overline{G}^* \simeq G_1$.

Beweis 3.2.3.:

Man erhält $\overline{G}^*$ wie folgt: Jeder Knoten $v_j \in V(G_1)$ repräsentiere eine

Äquivalenzklasse v_j, die einen Knoten g^*_j von $\overline{G}^*$ bildet, $1 \le j \le p_1$.
v_j bestehe aus $\{\iota_k(v_j) \mid 1 \le k \le p_0\}$.
Die von diesen Knotenmengen erzeugten Subgraphen sind gerade die zu $\overline{G}^*$ gehörigen Subgraphen $G^*_1,\ldots,G^*_{p_1}$.

Es ist zu zeigen:

(a) $|V(G^*_i)| = |V(G^*_j)| = p_0$, $1 \le i,j \le p_1$.

(b) $V(G^*_i) \quad V(G^*_j) = \emptyset$

(c) $G^*_i \simeq G^*_j \simeq \overline{G}$, $1 \le i,j \le p_1$.

(d) $(g^*_i, g^*_j) \in E(\overline{G}^*) \Rightarrow (\iota^*_i(v), \iota^*_j(v)) \in E(G)$ für alle $v \in V(G^*_1)$.

(a) und (b) sind nach Konstruktion unmittelbar gegeben.
Die Graph-Isomorphismen ι^*_i werden formal definiert durch

$\iota^*_i : V(G^*_1) \to V(G^*_i)$,
$\iota_k(v_1) \to \iota_k(v_i)$, $1 \le k \le p_0$.

Mit diesen Abbildungen erhält man (c) unter Verwendung der Vollständigkeit von G bez. $\overline{G}$. Mit (c) ist zugleich gewährleistet, daß die G_i^* zusammenhängend sind, da $\overline{G}$ Hypernetz eines zusammenhängenden Graphen ist.

Sei schließlich $(g^*_i, g^*_j) \in E(\overline{G}^*)$. Dann gibt es ein G_k, in dem die Knoten $\iota_k(v_i)$, $\iota_k(v_j)$ durch eine Kante miteinander verbunden sind. Wegen der Symmetrie bez. $\overline{G}$ sind dann in jedem G_l die Knoten $\iota_l(v_i)$ und $\iota_l(v_j)$ miteinander verbunden. Mit der oben eingeführten Darstellung der Knoten $\iota_l(v_i) = \iota^*_i(\iota_l(v_l))$, $\iota_l(v_j) = \iota^*_j(\iota_l(v_l))$ folgt die in (d) verlangte Vollständigkeit von G bez. $\overline{G}^*$.

□

Beweis 3.2.4.:

Allgemein gilt wegen der Vollständigkeit bez. $\overline{G}$ für alle $e \in E(\overline{G})$

$\kappa(e) \geq p_1$ und

$E_0(G) \supseteq \{(\iota_i(v),\iota_j(v)) \in E(G) \mid (g_i,g_j) \in E(\overline{G}),\ v \in V(G_1)\} =: I_0(G)$.

$$|E_0(G)| = \sum_{e \in E(\overline{G})} \kappa(e) \quad \text{und} \quad |I_0(G)| = p_1 |E(\overline{G})|.$$

Falls nun $\kappa(e) = p_1$ für alle $e \in E(\overline{G})$, ist also $E_0(G) = I_0(G)$.

Da nach Konstruktion von $\overline{G}^*$ $E^*_1(G) = I_0(G)$, folgt Bedingung (i) aus Definition 3.2.1..

Bedingung (ii) ist durch die Definition der Subgraphen G^*_j unmittelbar gegeben.

Die Isomorphie $\overline{G}^* \approx G_1$ ist durch $g^*_j \to v_j$ gegeben. Dabei folgt:

$(g^*_i,g^*_j) \in E(\overline{G}^*) \Rightarrow (v_i,v_j) \in E(G_1)$ wegen der Vollständigkeit bez. $\overline{G}$ und

$(v_i,v_j) \in E(G_1) \Rightarrow (g^*_i,g^*_j) \in E(\overline{G}^*)$ nach Definition eines Hypernetzes.

□

3.3. MEHRFACH PARTITIONIERTE NETZE UND D-DIMENSIONALE GRAPHEN

Eine naheliegende Verallgemeinerung der bis hierher beschriebenen partitionierten Kommunikationsgraphen besteht in der Definition mehrdimensionaler Netzwerke. Üblicherweise werden mehrdimensionale Strukturen nur im Falle von Würfeln oder Gittern betrachtet. Wir verwenden den in 3.1.6. eingeführten Begriff der Verfeinerung, um Dimensionen für allgemeine Netzwerkklassen definieren zu können.

Bekannt ist, daß d-dimensionale Würfel sich besonders für die Realisierung rekursiver (Netzwerk-)Algorithmen eignen (vgl. [19]). Intuitiv gesprochen bilden unsere d-dimensionalen Netzwerke die Abhängigkeitsstruktur partitionierter Algorithmen ab, deren Partitionierung mehrfach geschachtelt ist (vgl. 1.3., 6.2.).

3.3.1. Definition

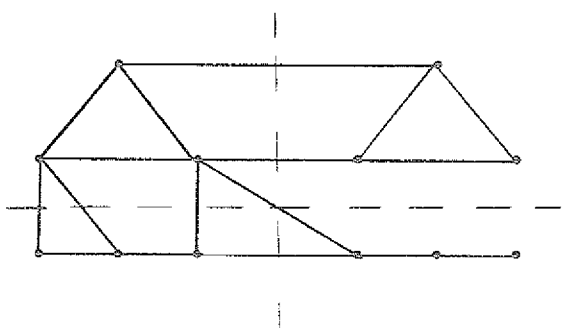
In G seien d Hypernetze $\overline{G}_0,\ldots,\overline{G}_{d-1}$ definiert mit: $\overline{G}_i$ ist Verfeinerung von $\overline{G}_{i-1}$. Dann heißt $(\overline{G}_0,\ldots,\overline{G}_{d-1})$ *Partitionierung der Tiefe d* von G.

□

Im folgenden Beispiel ist für einen Graphen G mit 12 Knoten eine Folge von Hypernetzen $\overline{G}_0$, $\overline{G}_1$, $\overline{G}_2$ angegeben, die eine Partitionierung der Tiefe 3 von G bildet.

3.3.2. Beispiel

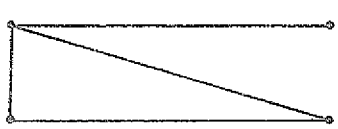
G:



$\overline{G}_0$:

$\overline{G}_1$:

$\overline{G}_2$:



Es entspricht der Anschauung, daß die Kantenanzahl von Netzwerken, die eine Partitionierung der Tiefe d besitzen, abhängig von d ist. Insbesondere ist im Falle vollständiger Netzwerke eine große Kantenanzahl zu erwarten.

Wir verwenden folgende Bezeichnungen:

$p_0(i) := |V(\overline{G}_i)|$, $p_1(i) := p/p_0(i)$, $0 \le i \le d$.

$\alpha := \min_i(1 - p_1(i)/p_1(i-1))$ und zeigen:

3.3.3. Satz

Sei $(G_0,\ldots,G_d)$ eine Partitionierung der Tiefe d+1 von G, und sei G bez. G_i vollständig und symmetrisch, $0 \le i \le d$.
Dann gilt für die Kantenanzahl von G

$$|E(G)| \ge \alpha dp.$$

Beweis:

Seien $E_0(i)$, $E_1(i)$ die durch $\overline{G}_i$ in G induzierten disjunkten Kantenklassen mit $E_0(i) \dot\cup E_1(i) = E(G)$, $0 \le i \le d$. $|E(G)|$ wird nun abgeschätzt durch $|E_0(d)|$.
Es ist $E_0(i-1) \subseteq E_0(i)$, $E_1(i-1) \supseteq E_1(i)$ (s. 3.1.7.).
Um eine untere Schranke für $|E_0(d)|$ anzugeben überlegt man sich, daß

$$|E_0(i)| = |E_0(i-1)| + |E_1(i-1) \cap E_0(i)|$$

und zählt die Kanten, die mindestens in $E_1(i-1) \cap E_0(i)$ enthalten sein müssen. Es ist $p_0(i)p_1(i) = p$, $0 \le i \le d$. Wegen

$$\begin{aligned}|V(\overline{G}_i)| &= p_0(i) = p/p_1(i) = [p_1(i-1)/p_1(i)]p_0(i-1)\\ &= [p_1(i-1)/p_1(i)]|V(\overline{G}_{i-1})| ,\end{aligned}$$

erhält man einen Knoten von $\overline{G}_{i-1}$ durch Zusammenfassen von $p_1(i-1)/p_1(i)$ Knoten von $\overline{G}_i$. Diese müssen in $\overline{G}_i$ zusammenhängen, da die zu $\overline{G}_{i-1}$ gehörigen Subgraphen von G zusammenhängend sein sollen. D.h. durch das Zusammenfassen der Knoten werden mindestens $[p_1(i-1)/p_1(i) - 1]p_0(i-1)$ Kanten aus $E(\overline{G}_i)$ entfernt. Da G vollständig ist, also $\kappa(\overline{G}_i) \ge p_1(i)$, $0 \le i \le d$, ist damit:

$$\begin{aligned}|E_1(i-1) \cap E_0(i)| &\ge [p_1(i-1)/p_1(i) - 1]p_0(i-1)p_1(i)\\ &= [1 - p_1(i)/p_1(i-1)]p_1(i-1)p_0(i-1) \ge \alpha p.\end{aligned}$$

Also insgesamt: $|E_0(d)| \ge \sum_{i=1}^{d} |E_1(i-1) \cap E_0(i)| \ge d\alpha p.$

□

Wir wenden uns nun d-dimensionalen Graphen zu.

Das Beispiel 3.3.2. verdeutlicht, daß es nicht sinnvoll ist, den Begriff der Dimension im Falle allgemeiner Partitionierungen der Tiefe d zu verwenden, da die möglicherweise unterschiedliche Topologie der miteinander kombinierten Graphen nicht berücksichtigt wird. Der Dimensionsbegriff

soll nur für solche Graphen verwendet werden, die aus Graphen gleicher Struktur gemäß 3.3.1. zusammengesetzt werden. Genauer:

Sei $\Gamma = \{G(n) \mid n \in \mathbb{N}_0\}$ eine Klasse von Graphen, $G(n)$ besitze $p(n)$ Knoten und $|V(G(0))| = 1$.
Beispielsweise bestehe Γ aus allen Gittern mit n^2 Knoten oder aus allen Shuffle-Exchange-Netzen mit 2^n Knoten.

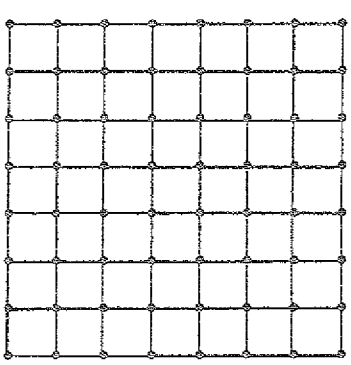
3.3.4. Definition

G heißt *d-dimensionaler Graph bez.* Γ :<=>
G besitzt eine Partitionierung $(\overline{G}_0,\ldots,\overline{G}_d)$ der Tiefe d+1 mit:
$\overline{G}_d = G$ und es gilt:
(i) die durch die Knoten von $\overline{G}_{i-1}$ festgelegten Subgraphen von $\overline{G}_i$ liegen in Γ, $1 \le i \le d$,
(ii) G ist symmetrisch bez. $\overline{G}_i$, $0 \le i \le d$.
Gilt darüberhinaus
(iii) G ist bez. $\overline{G}_i$ vollständig, $0 \le i \le d$,
so heißt G *vollständiger* d-dimensionaler Graph bez. Γ.

□

Die Angabe der Graphenklasse Γ ist für die Bestimmung der Dimension wesentlich, wie das folgende Beispiel verdeutlicht:
Das Gitter

G:



ist bez. der Klasse der linearen Netze ein vollständiger 2-dimensionaler Graph mit Hypernetzen

$\overline{G}_0$:

$\overline{G}_1$: , $\overline{G}_2 = G$.

$\kappa(\overline{G}_1) = 8 = p_1(1)$.

Verzichtet man auf die Vollständigkeit, so ist G bez. derselben Graphenklasse ein 4-dimensionaler Graph.
$(\overline{G}_0, \overline{G}_1, \overline{G}_2, \overline{G}_3, \overline{G}_4)$ ist gegeben durch:

$\overline{G}_0$:

$\overline{G}_1$: $p_1(1)=32$, $\kappa(\overline{G}_1)=8=(1/4)p_1(1)$

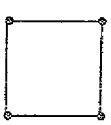
$\overline{G}_2$: $p_1(2)=16$, $\kappa(\overline{G}_2)=8=(1/2)p_1(2)$

$\overline{G}_3$: $p_1(3)=8$, $\kappa(\overline{G}_3)=8=p_1(3))$

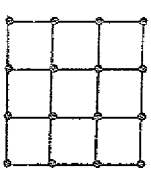
$\overline{G}_4 = G$.

Man erhält hierbei einen Subgraphen für $\overline{G}_i$ durch Zusammenfassen von 2^{3-i} in G benachbarten "Vertikalen".
Legt man mit Γ jedoch die Klasse der quadratischen ebenen Gitter zugrunde, so ist G lediglich ein 3-dimensionaler Graph:

$\overline{G}_0$:

$\overline{G}_1$: 

$p_1(1) = 16$, $\kappa(\overline{G}_1) = 4 = p_1(1)^{1/2}$

$\overline{G}_2$: 

$p_1(2) = 4$, $\kappa(\overline{G}_2) = 2 = p_1(2)^{1/2}$

$\overline{G}_3 = G$.

Das Beispiel läßt sich verallgemeinern zu der Aussage:

3.3.5. Lemma

- Seien $c, d \in \mathbb{N}$, und sei Γ die Klasse der quadratischen Gitter, d.h. $G(n) \in \Gamma$ ist gegeben durch:
  $V(G(n)) := \{v_{ij} \mid 1 \le i,j \le n\}$,
  $E(G(n)) := \{(v_{ij}, v_{i+1,j}), (v_{ji}, v_{j,i+1}) \mid 1 \le i \le n-1,\ 1 \le j \le n\}$.

  Dann ist $G(c^d)$ bez. Γ ein d-dimensionaler Graph, so daß gilt:

  $$\overline{G}_i \simeq G(c^i),\ p_1(i) = c^{2(d-i)},\ \kappa(\overline{G}_i) = c^{d-i} = p_1(i)^{1/2},\ 0 \le i \le d.$$

- Ist Γ' die Klasse der linearen Netze, d.h. $G'(n) \in \Gamma'$ ist gegeben durch
  $V(G'(n)) := \{v_i \mid 1 \le i \le n\}$,
  $E(G'(n)) := \{(v_i, v_{i+1}) \mid 1 \le i \le n-1\}$,

  so ist $G(c^d)$ bez. Γ' ein (d+1)-dimensionaler Graph, so daß gilt:

  $$\overline{G}_{d+1} = G \text{ und } \overline{G}_i \simeq G'(c^i),\ p_1(i) = c^{2d-i},\ \kappa(\overline{G}_i) = c^d,\ 0 \le i \le d.$$

(Der Beweis ergibt sich unmittelbar aus dem angeführten Beispiel.)

□

Es ist zu beachten, daß Definition 3.3.4. es nicht zuläßt, c^d linear miteinander verbundene Shuffle-Netze (vgl. 3.1.14.) analog zu 3.3.5.(ii) als (d+1)-dimensionale Graphen bez. Γ' zu interpretieren: Man kann die 3.3.5.(ii) zugrundeliegende Konstruktionsvorschrift für $\overline{G}_i$ zwar problemlos für $i = 0,\ldots,d$ anwenden. Da jedoch $\overline{G}_{d+1} = G$ verlangt ist, wird Bedingung (i) in 3.3.4. für $\overline{G}_d$ verletzt.

Die Konsistenz des in 3.3.4. eingeführten Dimensionsbegriffes mit dem herkömmlichen Dimensionsverständnis kann am Beispiel eines n-dimensionalen Würfels überprüft werden.

3.3.6. Lemma

Sei Γ'' die Klasse aller Würfel, d.h. $G''(n) \in \Gamma''$ ist gegeben durch:

$$V(G''(n)) := \{v(i_{n-1}\ldots i_0) \mid i_\nu \in \{0,1\},\ 0 \le \nu \le n-1\},$$
$$E(G''(n)) := \{(v(i_{n-1}\ldots i_j\ldots i_0), v(i_{n-1}\ldots i_j\ldots i_0)) \mid i_\nu \in \{0,1\}, 0 \le \nu, j \le n-1\}.$$

Dann ist $G''(n)$ bez. Γ'' ein vollständiger n-dimensionaler Graph.

Beweis:

$G''(n)$ besitzt Hypernetze $\overline{G}_d \simeq G''(d)$, $0 \le d \le n$. Ist für $0 \le k \le 2^d-1$

$$\mathrm{bin}_d k = k_{d-1}\ldots k_0,$$

so repräsentiert ein Knoten $g(k)$ von $\overline{G}_d$ den Subgraphen

$$[\{v(k_{d-1}\ldots k_0 i_{n-d-1}\ldots i_0) \mid i_\nu \in \{0,1\},\ 0 \le \nu \le n-d-1\}].$$

Diese Subgraphen sind isomorph zu $G''(n-d-1)$, so daß $G''(n)$ bez. $\overline{G}_d$ symmetrisch ist.

Ist $(g(k_{d-1}\ldots k_j\ldots k_0), g(k_{d-1}\ldots k_j\ldots k_0))$ eine Kante von $\overline{G}_d$, so enthält $G''(n)$ die Kanten

$$\{(v(k_{d-1}\ldots k_j\ldots k_0 i_{n-d-1}\ldots i_0),\ v(k_{d-1}\ldots k_j\ldots k_0 i_{n-d-1}\ldots i_0)) \mid i_\nu \in \{0,1\}, 0 \le \nu \le n-d-1\}$$

und ist deshalb bez. $\overline{G}_d$ vollständig.

Offensichtlich ist $\overline{G}_{d+1}$ eine Verfeinerung von $\overline{G}_d$. Der durch den Knoten $g(k_{d-1}\ldots k_0)$ repräsentierte Subgraph von $\overline{G}_{d+1}$ besteht aus den beiden in $\overline{G}_{d+1}$ miteinander verbundenen Knoten $g(k_{d-1}\ldots k_0 0)$ und $g(k_{d-1}\ldots k_0 1)$, liegt also selbst wieder in Γ''.

□

3.3.7. Korollar

$G''(n) \in \Gamma''$ ist bez. der Klasse Γ, die nur aus den Graphen $\circ\!\!-\!\!\!-\!\!\!-\!\!\circ$ besteht ein vollständiger n-dimensionaler Graph.

□

Schließlich ist ein 3-dimensionales Gitter bez. Γ' ein vollständiger 3-dimensionaler Graph:

Insgesamt ergibt sich eine Übereinstimmung des intuitiven Dimensionsbegriffes mit unserer Definition eines vollständigen d-dimensionalen Graphen.

Vollständige Netze sind wegen ihrer großen Kantenanzahl kaum von praktischer Relevanz. Vielmehr sind als Kommunikationsgraphen d-dimensionale Graphen G, die selbst eine einfache Struktur besitzen wünschenswert. Eine Möglichkeit, diese Anforderung an G zu formulieren besteht in der Annahme, daß ein bez. Γ d-dimensionaler Graph G selbst in der Klasse Γ liegt. Wir beschränken uns auf folgende Situation:

(i) $c_\Gamma |V(G(n))| \leq |E(G(n))| < 2c_\Gamma |V(G(n))|$, $G(n) \in \Gamma$, $c_\Gamma \geq 2$.

(ii) Für $e \in E(\overline{G}_i)$ sei $\kappa(e) = p_1(i)^\beta$, $0 \leq \beta \leq 1$.

Durch (i) erfolgt eine Einschränkung von Γ, die eine vergleichbare Kantenkomplexität aller Graphen aus Γ gewährleistet; (ii) beinhaltet, daß die Kapazität der Hypernetzkanten von $\overline{G}_i$ nur von der Größe der Subgraphen abhängt.
Mit diesen Voraussetzungen berechnet sich die im Beweis von 3.3.3. angegebene Schranke für $|E_0(d)|$ zu

3.3.8.

$$|E_0(d)| \geq \sum_{i=1}^{d} c_\Gamma (p_1(i-1)/p_1(i)) p_0(i-1) p_1(i)^\beta$$

$$= c_\Gamma p \sum_{i=1}^{d} p_1(i)^{\beta-1}.$$

□

Es liegt nahe, daß für $d \geq 2$ ein vollständiger d-dimensionaler Graph nicht in Γ liegen kann, da er die in (i) geforderte Gradbeschränkung nicht mehr erfüllen kann. Weniger offensichtlich ist, daß ein beliebiger d-dimensionaler Graph nicht in Γ liegen kann, wenn Γ nur reguläre Graphen gleichen Grades enthält:

3.3.9. Satz

Sei $G \in \Gamma$ ein d-dimensionaler Graph bez. Γ, der (i) und (ii) erfüllt. Falls gilt:

(1) G ist vollständig bez. $\overline{G}_i$, $0 \leq i \leq d$, oder falls gilt:

(2) $|E(G(n))| = c_\Gamma |V(G(n))|$, $G(n) \in \Gamma$,

so folgt: $d = 1$.

Beweis:

Die Vollständigkeit von G bedeutet $\beta = 1$ und damit

$c_\Gamma pd \leq |E_0(d)| \leq |E(G)| < 2c_\Gamma p \Rightarrow d \leq 1$.

Im Falle der Bedingung (2), die beispielsweise erfüllt ist, wenn alle $G(n) \in \Gamma$ regulär sind, folgt für $d > 1$ aus 3.3.8. der Widerspruch

$$c_\Gamma p[1 + \sum_{i=1}^{d-1} p_1(i)^{\beta-1}] \leq |E_0(d)| \leq |E(G)| = c_\Gamma p.$$

□

Um die Situation im Falle der Bedingung (2) zu veranschaulichen, wähle man für Γ zum einen die Klasse der 2-dimensionalen Gitter mit n^2 Knoten und zum Vergleich die Klasse der 2-dimensionalen Gitter mit n^2 Knoten und zyklischen Verbindungen. Bedingung (2) aus 3.3.9. ist nur im zweiten Fall erfüllt. Gemäß 3.3.5. können die Gitter ohne zyklische Verbindungen als mehrdimensionale Netze bez. derselben Graphenklasse Γ interpretiert werden. Verlangt man jedoch die zyklischen Verbindungen für alle Graphen in Γ, so wird dies unmöglich.

3.4. ZUSAMMENHANG VON HYPERNETZEN UND NETZWERKEMULATIONEN

Parallele Algorithmen werden in der Regel für Netzwerke $G(n) \in \Gamma$ mit variabler Prozessoranzahl $p(n)$ beschrieben. In der Praxis muß ein Algorithmus jedoch auf einem fest vorgegebenen Netzwerk G mit konstanter Prozessoranzahl p und vorgegebener Topologie realisiert werden. D.h. für jeden Schritt i eines parallelen Algorithmus ist eine Abbildung $\phi_i: V(G(n)) \to V(G)$ anzugeben. Diese Abbildungen legen für jeden Prozessor P aus G(n) fest, welche Prozessoren aus G P in Schritt i des Algorithmus simulieren.

Wir beschränken uns auf die Situation, daß diese Simulationsabbildungen unabhängig vom Algorithmus sind. In diesem Fall wird die Simulation des

Netzwerks $G(n)$ durch das Netzwerk G durch *eine* Abbildung $\phi: V(G(n)) \to V(G)$ festgelegt. ϕ soll dabei so gewählt werden, daß

(i) die Verbindungsstruktur von $G(n)$ erhalten bleibt und

(ii) die Prozessoren von G möglichst effizient verwendet werden können.

Die folgenden Definitionen stützen sich auf [4].

3.4.1. Definition

Seien $G(n)$ und G Graphen mit $p(n)$ bzw. p Knoten, $p(n) \geq p$. Eine *Netzwerkemulation* ist eine Abbildung $\phi: V(G(n)) \to V(G)$ mit

$$(v,w) \in E(G(n)) \Rightarrow \phi(v) = \phi(w) \text{ oder } (\phi(v),\phi(w)) \in E(G).$$

Gilt darüberhinaus

$$|\phi^{-1}(v)| = |\phi^{-1}(w)| = p(n)/p \text{ für alle } v,w \in V(G),$$

so heißt ϕ *berechnungsuniform*.

□

Durch ϕ wird eine Abbildung $\phi': E(G(n)) \to E(G) \cup V(G)$ induziert.

3.4.2. Definition

Eine Emulation $\phi: V(G(n)) \to V(G)$ heißt *uniform* :<=>

ϕ ist berechnungsuniform, und für ϕ' gilt: $|\phi'^{-1}(e_1)| = |\phi'^{-1}(e_2)|$ für alle $e_1, e_2 \in E(G)$.

□

Es besteht ein einfacher Zusammenhang zwischen Netzwerkemulationen und Hypernetzen:

3.4.3. Satz

Falls $G(n)$ ein zu G isomorphes p-Hypernetz $\overline{G}$ besitzt, so gilt:

Es gibt eine berechnungsuniforme Emulation $\phi: V(G(n)) \to V(G)$.

Gilt darüberhinaus $\kappa(e) = \kappa$ für alle $e \in E(\overline{G})$, so ist ϕ eine uniforme Emulation.

Beweis:

Seien $G_1,\ldots,G_p$ die Subgraphen von $G(n)$, die durch die Hypernetzknoten $g_1,\ldots,g_p$ zusammengefaßt werden. Die Knoten von G bezeichnen wir durch $V(G) = \{v_1,\ldots,v_p\}$, so daß $G \approx \overline{G}$ vermöge $g_i \to v_i$. Dann wird ϕ definiert durch

$\phi(v) = v_i$ für $v \in V(G_i)$. Insbesondere ist $|\phi^{-1}(v_i)| = p(n)/p$, $1 \leq i \leq p$. Für $(v,w) \in E_0(G(n))$ folgt $(\phi(v),\phi(w)) \in E(G)$, und für $(v,w) \in E_1(G(n))$

ist $\phi(v) = \phi(w)$. $\kappa(g_i,g_j))$ bezeichnet die Anzahl der Kanten aus $E_0(G(n))$, die G_i und G_j miteinander verbinden.

Daher ist $\kappa((g_i,g_j)) = |\phi'^{-1}((v_i,v_j))|$.

□

Netzwerkemulationen lassen es zu, daß die von $\phi^{-1}(v)$, $v \in V(G)$, erzeugten Subgraphen von G(n) nicht zusammenhängend sind. Schließt man dies jedoch aus, so läßt sich auch die umgekehrte Richtung von 3.4.3. zeigen:

3.4.4. Satz

Sei ϕ: $V(G(n)) \to V(G)$ eine berechnungsuniforme Netzwerkemulation mit:
$[\phi^{-1}(v)]$ ist für alle $v \in V(G)$ zusammenhängend und
$\phi'^{-1}(e) \neq \emptyset$ für alle $e \in E(G)$.
Dann besitzt G(n) ein zu G isomorphes (p(n)/p)-Hypernetz.

Beweis:

Eine berechnungsuniforme Emulation legt eine Äquivalenzrelation in V(G(n)) fest:
$v,w \in V(G(n))$, $v \sim w$:<=> es gibt ein $v_i \in V(G)$ mit $v,w \in \phi^{-1}(v_i)$.

Da $[\phi^{-1}(v_i)]$ in G(n) zusammenhängend ist, legt diese Äquivalenzrelation ein (p(n)/p)-Hypernetz $\overline{G}$ von G(n) fest. Bezeichnet man den zugehörigen Hypernetzknoten mit g_i, so ist $\overline{G} \simeq G$ gegeben durch $g_i \to v_i$. Es gilt:
$(g_i,g_j) \in E(\overline{G}) \Rightarrow (v_i,v_j) \in E(G)$, denn $(g_i,g_j) \in E(\overline{G})$ impliziert:
es gibt $v,w \in V(G(n))$, $(v,w) \in E(G(n))$ mit $\phi(v) = v_i$, $\phi(w) = v_j$. Nach Definition einer Emulation ist damit $(v_i,v_j) \in E(G)$.
Umgekehrt folgt aus $(v_i,v_j) \in E(G)$ auch $(g_i,g_i) \in E(\overline{G})$, denn wegen

$\phi'^{-1}((v_i,v_j)) \neq \emptyset$ gibt es $v,w \in V(G(n))$ mit $\phi(v) = v_i$, $\phi(w) = v_j$ und $(v,w) \in E_0(G(n))$, also $(g_i,g_j) \in E(\overline{G})$. Also ist $\overline{G} \simeq G$.

□

4. PARTITIONIERTE NETZWERKALGORITHMEN

In den ersten Kapiteln haben wir als einzige Restriktion, die eine praktische Realisierung eines parallelen Algorithmus mit sich bringt, eine Beschränkung p der Anzahl verfügbarer Prozessoren angenommen. Durch die Definition 1.1.5. eines parallelen Algorithmus ist implizit ein Rechnermodell mit unbeschränktem gemeinsamem Speicher angenommen worden. Die durch Speicherzugriffe bedingten Zeitkosten wurden nicht untersucht. Es wurde darauf verwiesen, daß die Kommunikationskosten eines parallelen Algorithmus im Hinblick auf eine Multiprozessoranordnung mit statischem Prozessor-Verbindungsnetzwerk untersucht werden sollen. Wir nehmen an, daß alle Prozessoren identisch sind und lokale Speicher besitzen.
Im vorigen Kapitel wurden Kommunikationgraphen betrachtet, die sich für eine Abbildung der Abhängigkeitsstruktur partitionierter Algorithmen eignen. Die Beispiele in 3.1. und die Beschreibung d-dimensionaler Netze in 3.2. haben gezeigt, daß solche Graphen eine sehr komplexe Struktur besitzen können. Man kann deshalb im allgemeinen nicht davon ausgehen, daß der Kommunikationsgraph G eines Algorithmus und das real gegebene Prozessorverbindungsnetzwerk N identisch sind. Ob N eine unmittelbare Wiedergabe von G darstellt, oder ob es erforderlich ist, ein Verfahren zur Simulation von G durch N anzugeben, hängt wesentlich von dem konkreten Parallelrechner ab (vgl. [22],[23]). Eine unmittelbare Wiedergabe eines partitionierten Kommunikationsgraphen liegt beispielsweise im Falle untereinander verbundener VLSI-Chips vor. Die einzelnen Chips können dann nämlich als Subgraphen eines globalen Kommunikationsgraphen interpretiert werden, dessen Hypernetz durch die Verbindungen der Chips untereinander gegeben ist.
Da in unserem Kontext kein konkreter Parallelrechner vorgegeben ist, beschränken wir uns auf die Kommunikationsgraphen G. Da G nicht unabhängig vom Algorithmus ist, sondern dessen Abhängigkeitsstruktur abbilden soll, ist es sinnvoll, die Definition der Realisierung eines parallelen Algorithmus auf einem Netzwerk nicht von der Definition eines parallelen Algorithmus zu trennen. Die folgende Definition eines Netzwerkalgorithmus beinhaltet deshalb sowohl die "parallele Arithmetik" des Algorithmus als

auch dessen Ausführung auf einem Kommunikationsgraphen (vgl. [17], Def.4.1.1.).

In 4.1. übertragen wir die in 1.1. entwickelte Beschreibung eines partitionierten Algorithmus auf die Situation von Netzwerkalgorithmen. Die weitgehende Analogie zu partitionierten PRAMs wird hierbei mit Hilfe des Hypernetzkonzeptes aus Kapitel 3 hergestellt. Die Definition eines partitionierten Netzwerkalgorithmus wird durch das Beispiel der Matrix-Multiplikation, ausgeführt auf einem 3-dimensionalen Gitter, illustriert. In 4.2. zeigen wir, daß sich die Berechnung der Kommunikationszeit eines partitionierten Netzwerkalgorithmus, ähnlich wie die Laufzeit einer partitionierten PRAM, auf die Bestimmung der Kommunikationszeiten des externen Algorithmus und der internen Algorithmen zurückführen läßt. Allerdings ist im Gegensatz zur Laufzeit im allgemeinen keine einfache Faktorisierung der Kommunikationszeit möglich. Dies hat zur Konsequenz, daß sich - bei u.U. gleicher Laufzeit - die Kommunikationskosten eines partitionierten Netzwerkalgorithmus und seiner dualen Version voneinander unterscheiden.

4.1. PARTITIONIERTE NETZWERKE ALS KOMMUNIKATIONSGRAPHEN PARTITIONIERTER ALGORITHMEN

Sei K ein Körper. Ziel ist die Berechnung einer Funktion

$$f\colon (K^n \times K^n)^k \to K^n \times K^n,$$

$k,n \in \mathbb{N}$. Für $m \in \mathbb{N}$ mit $m|n$ sei $\overline{f}_m$ eine m-Partitionierung von f.
In der folgenden Definition verwenden wir den Begriff einer RAM und bezeichnen damit eine PRAM im Sinne von 1.1.3. mit einem Prozessor.

4.1.1. Definition

Sei R ein Ring, $p \in \mathbb{R}[x]$. Ein *(polynomieller) Netzwerkalgorithmus* D(n) über R ist gegeben durch

- einen Kommunikationsgraphen $G(n) = (V(G(n)),E(G(n)))$ mit

$V(G(n)) = \{v_1,\ldots,v_{p(n)}\}$,

- $p(n)$ RAMs $P_1,\ldots,P_{p(n)}$ über R mit Registern $1_i, 2_i, \ldots$, Akkumulatoren 0_i, $1 \le i \le p(n)$, und Operationsmenge S.

Neben den in 1.1.3. aufgelisteten Befehlen, kann prog P_i einen Kommunikationsbefehl com(i,j) mit der Wirkung <0i> ← <0j> enthalten.
Dabei ist com(i,j) definiert für alle i,j mit $(v_i,v_j) \in E(G(n))$.
Bezeichnet τ_i die Anzahl der Kommunikationsbefehle im Programm von P_i, so heißt

$$\tau(n) := \max_i \tau_i$$

die *Kommunikationszeit* von D(n).

□

Die Ein- und Ausgabe eines Netzwerkalgorithmus D(n) erfolge direkt in den P_i und zwar so, daß für $n^2 \ge p(n)$ alle P_i zu Beginn bzw. am Ende der Rechnung gleich viele Ein- bzw. Ausgabedaten enthalten, oder aber höchstens einen Wert, falls $n^2 < p(n)$. Die Anordnung dieser Werte in den P_i wird zunächst nicht festgelegt.

Wir nehmen im folgenden an $n^2 < p(n)$ oder $p(n) | n^2$.

4.1.2. Definition

D(n) berechnet die Funktion $f: (K^n \times K^n)^k \to K^n \times K^n$:<=>
Wird D(n) gestartet mit $x \in (K^n \times K^n)^k$, so sind nach Beendigung aller Programme die Koeffizienten von f(x) gegeben in den Registern

$$1_i,\ldots,(n^2/p(n))_i,\ 1 \le i \le p(n),\ \text{falls } n^2 \ge p(n)$$

bzw. in den Registern

$$1_i,\ 1 \le i \le n^2,\ \text{falls } n^2 < p(n).$$

□

Ein partitionierter Netzwerkalgorithmus kann nun weitgehend analog zu einer partitionierten PRAM definiert werden.
Sei nämlich $\overline{D}(n/m)$ ein Netzwerkalgorithmus über $K^m \times K^m$ mit $p_0(n/m)$ Prozessoren und Kommunikationsgraph $\overline{G}(n/m)$, der eine m-Partitionierung $\overline{f}_m$ von f berechnet und hierfür die Operationen S_m verwendet. $D_g(m)$ seien Netzwerkalgorithmen über K mit $p_1(m)$ Prozessoren und Kommunikationsgraphen $G_1(m)$, welche die Grundoperationen $g \in S_m$ berechnen. Wir be-

schreiben einen Netzwerkalgorithmus $D(n,m)$, der $\overline{D}(n/m)$ schrittweise simuliert. $D(n,m)$ besteht aus $p_0(n/m)p_1(m)$ RAMs P_{ij}, $1 \le i \le p_0(n/m)$, $1 \le j \le p_1(m)$, und einem Kommunikationsgraphen $G(n,m)$. In $G(n,m)$ ist ein $p_1(m)$-Hypernetz festgelegt, das isomorph zu $\overline{G}(n/m)$ ist. Die zugehörigen Subgraphen G_i von $G(n,m)$ sind isomorph zu $G_1(m)$, d.h. $G(n,m)$ ist symmetrisch bez. $\overline{G}(n/m)$. Wir setzen

$V(G(n,m)) := \{v_{ij} \mid 1 \le i \le p_0(n/m),\ 1 \le j \le p_1(m)\}$, $G_i = [\{v_{ij} \mid 1 \le j \le p_1(m)\}]$.

Ist $\overline{P}_i$ eine RAM von $\overline{D}(n/m)$, so wird $\overline{P}_i$ simuliert von $P_{i1},\ldots,P_{ip_1(m)}$.

Dabei übernehmen die Register

$$(\lceil m^2/p_1(m)\rceil+\nu)r_{i1},\ldots,(\lceil m^2/p_1(m)\rceil+\nu)r_{ip_1(m)},\ 1 \le \nu \le \lceil m^2/p_1(m)\rceil,$$

die Funktion von Register $(\overline{r-1})_i$ von $\overline{P}_i$, $r = 1,2,\ldots$.

Damit kann die Simulation eines load- bzw. store-Befehls von $\overline{P}_i$ durch

$2\lceil m^2/p_1(m)\rceil$ parallele load- und store-Befehle der $P_{i1},\ldots,P_{ip_1(m)}$

simuliert werden (vgl. 1.1.).

Führt $\overline{P}_i$ die arithmetische Operation g aus, so simulieren $P_{i1},\ldots,P_{ip_1(m)}$

und der Graph G_i den Netzwerkalgorithmus $D_g(m)$. In einem Kommunikationsschritt com(i,j) von $\overline{D}(n/m)$ wird eine $(m \times m)$-Blockmatrix von $\overline{P}_i$ nach $\overline{P}_j$ übertragen. Der zugehörigen Kante $(g_i,g_j) \in E(\overline{G}(n/m))$ entspricht mindestens eine Kante von $G(n,m)$, welche die Subgraphen G_i und G_j in $G(n,m)$ miteinander verbindet.

Um com(i,j) zu simulieren, werden die m^2 Daten aus den Registern

$$(\lceil m^2/p_1(m)\rceil+\nu)1_{i1},\ldots,(\lceil m^2/p_1(m)\rceil+\nu)1_{ip_1(m)}$$

der $\overline{P}_i$ simulierenden RAMs $P_{i1},\ldots,P_{ip_1(m)}$ entlang der durch (g_i,g_j)

repräsentierten Kanten von $G(n,m)$ in die Register

$$(\lceil m^2/p_1(m)\rceil+\nu)1_{j1},\ldots,(\lceil m^2/p_1(m)\rceil+\nu)1_{jp_1(m)}$$

der $\overline{P}_j$ simulierenden RAMs $P_{j1},\ldots,P_{kp_1(m)}$ übertragen, $1 \le \nu \le \lceil m^2/p_1(m)\rceil$.

(Die Bestimmung der Zeitkosten dieser Simulation gehört zum Gegenstand von Abschnitt 4.2..)

Die Simulation eines (parallelen) Schrittes von $\overline{D}(n/m)$ wird begonnen, sobald die Simulation des vorhergehenden Schrittes beendet worden ist. Da $\overline{D}(n/m)$ $\overline{f}_m$ berechnet, enthalten die Register $1_i,\ldots,(n/m)^2/p_0(n/m)_i$, $1 \le i \le p_0(n/m)$, bzw. die Register $1_1,\ldots,1_{(n/m)^2}$ (vgl. 4.1.2.) von $\overline{D}(n/m)$

am Ende der Rechnung jeweils einen Wert $\overline{f}_m(x)_k$, $1 \le k \le (n/m)^2$.
Ist $\langle r_i \rangle = \overline{f}_m(x)_k$, so sind nach Beendigung der Simulation die Elemente von

$f(x)_k$ im Fall $m^2 \ge p_1(m)$ in den Registern

$$(m^2/p_1(m)+\nu)(r+1)_{i1},\ldots,\ (m^2/p_1(m)+\nu)(r+1)_{ip_1(m)},\ 1 \le \nu \le m^2/p_1(m),$$

von $D(n,m)$ enthalten. Damit enthält für $(n/m)^2 \ge p_0(n/m)$ jedes P_{ij} am Ende der Rechnung

$$n^2/[p_0(n/m)p_1(m)]$$

Ausgabedaten, bzw. für $(n/m)^2 < p_0(n/m)$ enthalten

$$P_{1j},\ldots,P_{(n/m)j} \quad \text{je} \quad m^2/p_1(m)$$

Ausgabedaten, $1 \le j \le p_1(m)$.

Im Falle $m^2 < p_1(m)$ sind die Elemente von $f(x)_k$ in den Registern

$2_{i1},\ldots,2_{im^2}$ enthalten. Dann enthalten für $(n/m)^2 \ge p_0(n/m)$ $P_{i1},\ldots,P_{im^2}$

am Ende der Rechnung je $(n/m)^2/p_0(n/m)$ Ausgabedaten, $1 \le i \le p_0(n/m)$.

Für $(n/m)^2 < p_0(n/m)$ enthalten die Register 2_{ij}, $1 \le i \le (n/m)^2$, $1 \le j \le m^2$, je einen Ausgabewert.

Nach dem stop-Befehl von $\overline{D}(n/m)$ übertragen die P_{ij} die Ausgabewerte in ihre ersten Register.
Im Falle $m^2 < p_1(m)$ und $(n/m)^2 \ge p_0(n/m)$ wird zusätzlich in jedem G_i ein Routing durchgeführt, das die Ausgabedaten gleichmäßig auf die P_{ij}, $1 \le j \le p_1(m)$, verteilt.

Damit haben wir insgesamt:

4.1.3. Satz

Sei $\overline{D}(n/m)$ ein Netzwerkalgorithmus über $K^m \times K^m$, der eine m-Partitionierung $\overline{f}_m$ von f berechnet. Ist $D(n,m)$ ein Netzwerkalgorithmus

über K, der $\overline{D}(n/m)$ in der beschriebenen Weise simuliert, so berechnet D(n,m) die Funktion f.
D(n,m) heißt *partitionierter Netzwerkalgorithmus* zur Berechnung von f.

□

Wir übertragen den Begriff der Partitionierbarkeit auf Netzwerkalgorithmen:

4.1.4. Definition

Seien D'(n) und D"(n) Netzwerkalgorithmen über $K^{m'} \times K^{m'}$ bzw. $K^{m''} \times K^{m''}$ mit RAMs $P_1', \ldots, P'_{p'(m')}$ bzw. $P_1'', \ldots, P''_{p''(n)}$ und Kommunikationsgraphen G'(n) und G"(n).
D'(n) ≈ D"(n) :<=>
G'(n) ≈ G"(n) (also insbesondere auch p'(n) = p"(n)), und zu jedem P_i' gibt es genau ein P_j'' mit $P_i' \approx P_j''$.

□

4.1.5. Definition

Sei $\Gamma = \{G(n) \mid n \in \mathbb{N}_0\}$ eine Klasse von Graphen mit $|V(G(n))| = p(n)$. Ein Netzwerkalgorithmus D(n) über K mit Kommunikationsgraph G(n) zur Berechnung von f heißt *partitionierbar bez.* Γ :<=>
Für alle $m \in \mathbb{N}$ mit $m|n$ existiert ein Netzwerkalgorithmus $\overline{D}(n/m)$ über $K^m \times K^m$ mit Kommunikationsgraph $\overline{G}(n/m)$, der eine m-Partitionierung $\overline{f}_m$ von f berechnet, und es gilt: $\overline{D}(n) = D(n,1) = D(n)$.
Die D(n,m), welche die $\overline{D}(n/m)$ in der beschriebenen Weise simulieren, heißen *Partitionierungen* oder *partitionierte Versionen* von D(n).

□

Im folgenden legen wir wieder eine gemeinsame Operationsmenge S aller RAMs zugrunde (vgl. 1.1.).

4.1.6. Fakt

Sei $f: (K^n \times K^n)^k \to K^n \times K^n$ definiert für alle $n \in \mathbb{N}$, und seien D(n) bez. Γ partitionierbare Netzwerkalgorithmen zur Berechnung von f, $n \in \mathbb{N}$. Dann gilt:

$$\overline{D}(n/m) = D(n/m)$$

für alle $m \in \mathbb{N}$ mit $m|n$.

□

Bevor wir uns in 4.2. den Kommunikationskosten zuwenden, verdeutlichen wir durch ein Beispiel, wie die oben beschriebene Simulation Realisierungen

partitionierter Algorithmen auf symmetrischen (partitionierten) Netzen G(n,m) liefert.

4.1.7. Matrix-Multiplikation

Es soll das Produkt

$$\begin{bmatrix} a_{11} & \cdots & a_{1,15} \\ \vdots & & \vdots \\ a_{15,1} & \cdots & a_{15,15} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1,15} \\ \vdots & & \vdots \\ b_{15,1} & \cdots & b_{15,15} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1,15} \\ \vdots & & \vdots \\ c_{15,1} & \cdots & c_{15,15} \end{bmatrix}$$

auf einem dreidimensionalen Gitter berechnet werden.
Es seien $p = 75$ Prozessoren verfügbar, es ist also $n < p < n^2$. Von den in 2.4. beschriebenen Partitionierungen kommen deshalb die folgenden in Frage:

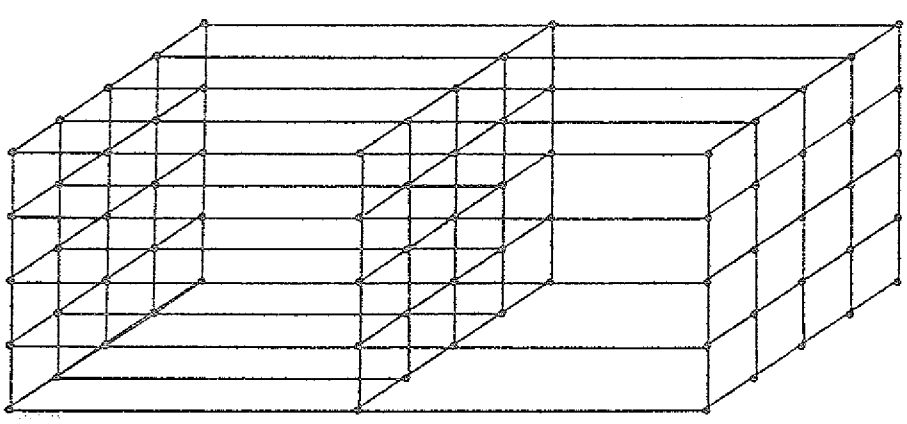
(i) $p_0(n/m) = 1,\ p_1(m) = m^2$

(ii) $p_0(n/m) = (n/m)^2,\ p_1(m) = 1$

(iii) $p_0(n/m) = (n/m)^2,\ p_1(m) = m$

(iv) $p_0(n/m) = n/m,\ p_1(m) = m^2$

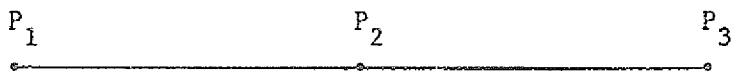
Da 75 keine Quadratzahl ist, kann man m in den Fällen (i) und (ii) nicht geeignet bestimmen. Wir betrachten deshalb nur (iii) und (iv). Für (iii) erhalten wir aus $p(n,m) = n^2/m = p$ die Blockgröße $m = 3$, im Falle (iv) liefert $p(n,m) = nm = p$ die Blockgröße $m = 5$. Der Kommunikationsgraph G sei gegeben durch



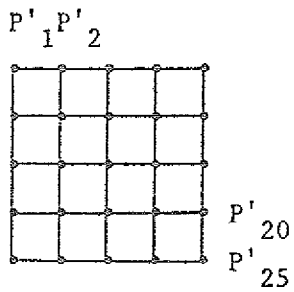
(Wir verzichten auf die vollständige Eintragung aller Kanten.)

G besitzt zwei zueinander duale Hypernetze $\bar{G}$, $\bar{G}'$:

$\bar{G}$:



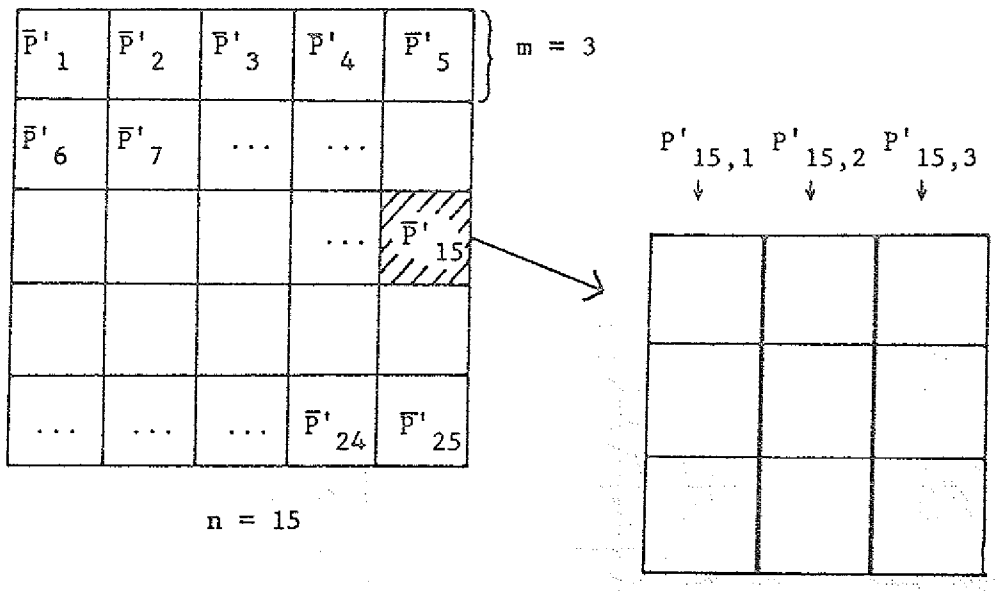
$\bar{G}'$:



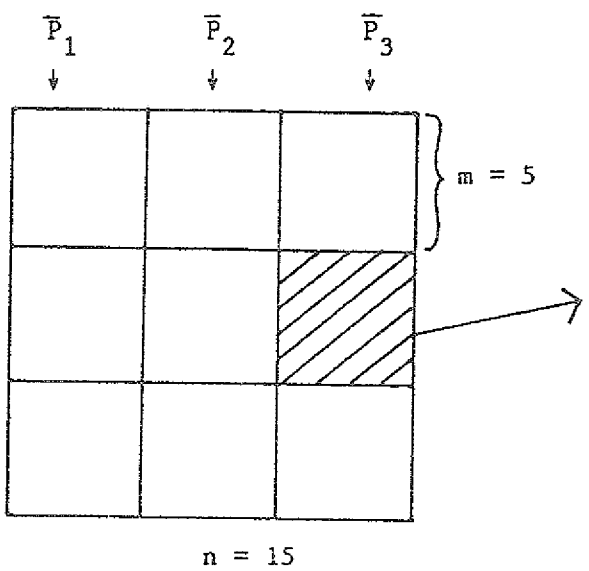
Die durch die Knoten von $\bar{G}$ ($\bar{G}'$) repräsentierten Subgraphen sind also isomorph zu $\bar{G}'$ ($\bar{G}$).

Die Parallelisierungen (iii) und (iv) lassen sich durch folgende Zuordnung der Subgraphen- und Hypernetzprozessoren zu den Matrix-Elementen veranschaulichen:

(iii)

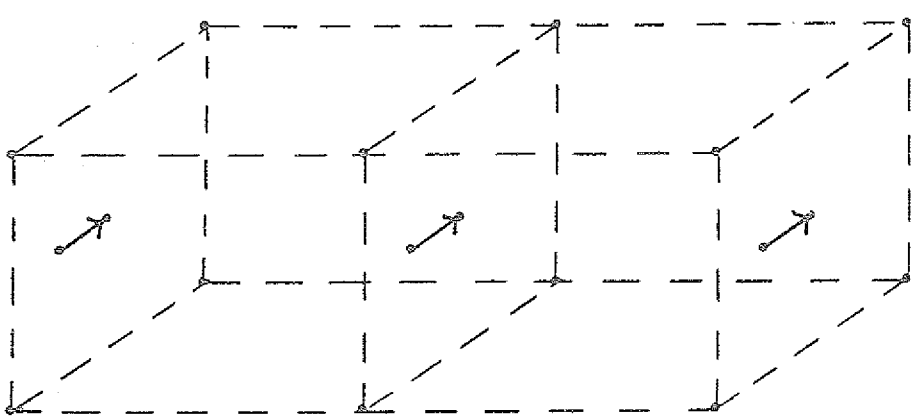


(iv)

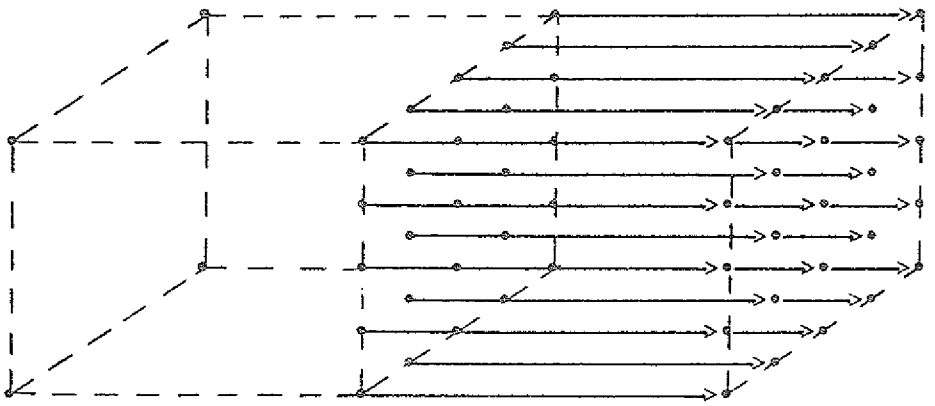


$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$
$P_{3,6}$	$P_{3,7}$	...	...	
		...	$P_{3,24}$	$P_{3,25}$

Für die Realisierung der (partitionierten) Gesamtmultiplikation genügt also die Kenntnis eines Matrix-Multiplikationsalgorithmus M_1 für (3×3)-Matrizen auf 3 linear miteinander verbundenen Prozessoren und eines Matrix-Multiplikationsalgorithmus M_2 für (5 × 5)-Matrizen auf einem quadratischen Gitter mit 25 Prozessoren. Im Fall (iii) liegen die Koeffizienten der von M_2 multiplizierten Matrizen in $K^3 \times K^3$, für (iv) sind die Koeffizienten der von M_1 multiplizierten Matrizen Elemente aus $K^5 \times K^5$. Für M_1 und M_2 können beliebige bekannte Netzwerkalgorithmen (s. z.B. [3], [9]) verwendet werden. Die Übertragung einer (3 × 3)-Blockmatrix durch M_2 im Fall (iii) wird durch 3 Übertragungsschritte der folgenden Form realisiert:



Die Übertragung einer (5 × 5)-Blockmatrix durch M_1 im Fall (iv) kann in einem Schritt erfolgen:



In völlig analoger Weise können Realisierungen für andere partitionierte (Matrix-)Algorithmen auf dreidimensionalen Gittern aus bekannten Netzwerkalgorithmen für lineare Prozessoranordnungen und für zweidimensionale Gitter konstruiert werden. (S. hierzu 6.3.). Insbesondere können M_1 und M_2 natürlich auch systolisch ausgeführt werden.

4.2. BESTIMMUNG DER KOMMUNIKATIONSZEIT PARTITIONIERTER ALGORITHMEN UND IHRER DUALEN VERSIONEN

Die Beispiele verdeutlichen, wie die Kommunikationszeit eines partitionierten Netzwerkalgorithmus zu berechnen ist. Sei $D(n,m)$ ein partitionierter Netzwerkalgorithmus mit externem Algorithmus $D(n/m)$ und internen Algorithmen $D_i(m)$, $1 \leq i \leq r$. Die zugehörigen Kommunikationszeiten werden bezeichnet durch $\tau(n,m)$, $\tau_0(n/m)$ und $\tau_i(m)$, $1 \leq i \leq r$. Wir setzen o.E. $\tau_i(m) \geq \tau_{i+1}(m)$, $1 \leq i \leq r-1$. Dann stellen die Kommunikationskosten der internen Algorithmen eine triviale untere Schranke für $\tau(n,m)$ dar (vgl. 2.1.2.):

4.2.1. Fakt

$$\tau(n,m) \geq \sum_i t_{0,i}(n/m)\tau_i(m).$$

□

Nach Definition von $D(n,m)$ ist in dem Kommunikationsgraphen $G(n,m)$ ein zu $\overline{G}(n/m)$ isomorphes $p_1(m)$-Hypernetz festgelegt. Die Kanten von $G(n,m)$ können deshalb wie in 3.1. in zwei disjunkte Klassen $E_0 = E_0(G(n,m))$ und $E_1 = E_1(G(n,m))$ zerlegt werden.

Während die zu 4.2.1. analoge Aussage 2.1.2. für die Laufzeit $t(n,m)$ eine Gleichheit lieferte, erhalten wir für die Kommunikationszeit:

4.2.2. Lemma

$$\tau(n,m) = \sum_i t_{0,i}(n/m)\tau_i(m) \iff$$

$p_0(n/m) = 1$, oder aber es findet kein Datenaustausch zwischen den Prozessoren von $\overline{D}(n/m)$ statt. ("$\overline{D}(n/m)$ berechnet nur unabhängige Operationen.")

Beweis:

Gemäß der Simulationsvorschrift für $D(n,m)$ werden die Kanten E_1 nur für die Simulation der arithmetischen Operationen und die Kanten E_0 nur für die Simulation der Kommunikationsbefehle von $\overline{D}(n/m)$ verwendet. Da die Simulation synchronisiert ist, werden also niemals gleichzeitig Daten entlang von Kanten aus E_0 und aus E_1 übertragen. Die Gleichheit in 4.2.1. gilt deshalb genau dann, wenn in den Programmen von $\overline{D}(n/m)$ keine Befehle $com(i,j)$ enthalten sind, also wenn $\tau_0(n/m) = 0$.

□

Da in einem Befehl $com(i,j)$ von $\overline{D}(n/m)$ eine $(m \times m)$-Matrix entlang der Kante $(g_i,g_j) \in E(\overline{G}(n/m))$ übertragen wird, erfordert die Simulation dieses Befehls durch $D(n,m)$ also mindestens Zeit $\lceil m^2/\kappa((g_i,g_j))\rceil$.
Mit $\kappa := \max\{\kappa(e) \mid e \in E(\overline{G}(n/m))\}$ erhalten wir:

4.2.3. Fakt

$$\tau(n,m) \geq \sum_i t_{0,i}(n/m)\tau_i(m) + \tau_0(n/m)\lceil m^2/\kappa\rceil.$$

□

Für die weiteren Überlegungen setzen wir wieder voraus, daß die Kommunikationszeiten aller internen Netzwerkalgorithmen gleich sind, nämlich $\tau_1(m)$. Damit wird 4.2.3. zu

4.2.3.'

$$\tau(n,m) \geq t_0(n/m)\tau_1(m) + \tau_0(n/m)\lceil m^2/\kappa \rceil .$$

□

Nachfolgend geben wir einige einfache Eigenschaften von $D(n,m)$ an, die es ermöglichen, daß $\tau(n,m)$ den in 4.2.3.' angegebenen minimalen Wert annimmt. Im allgemeinen ist davon auszugehen, daß die Simulation eines Befehls $com(i,j)$ von $\overline{D}(n/m)$ nicht unmittelbar mit der Übertragung der m^2 Daten von G_i nach G_j begonnen werden kann, sondern daß zuvor ein Routing der m^2 Daten innerhalb von G_i erfolgen muß, welches die Blockmatrixkoeffizienten in Knoten von G_i überträgt, die mit Knoten von G_j verbunden sind. Darüberhinaus ist davon auszugehen, daß nach der Übertragung der Blockmatrix von G_i nach G_j ein Routing der m^2 Daten in G_j erfolgen muß, so daß ihre Verteilung in G_j der Eingabeverteilung des im nächsten Schritt in G_j auszuführenden internen Algorithmus $D_g(m)$ entspricht. Diese zusätzlichen Routingzeiten können vermieden werden, wenn in den $p_0(n/m)$ Subgraphen von $G(n,m)$ eine Ein-/Ausgabeverteilung festgelegt ist, und alle entsprechenden Prozessoren miteinander verbunden sind.
Genauer: Bezeichne A_{in}, $A_{out} \in K^m \times K^m$ die Ein- bzw. Ausgabematrix von $D_g(m)$.

4.2.4. Definition Eine *Eingabe-(Ausgabe-)Verteilung von $D_g(m)$* ist eine Abbildung

$$\sigma_{in}(\sigma_{out}): \{1,\ldots,m\} \times \{1,\ldots,m\} \to \{1,\ldots,p_1(m)\}$$

mit $|\sigma_{in}^{-1}(j)| = |\sigma_{out}^{-1}(j)| = m^2/p_1(m)$, falls $m^2 \geq p_1(m)$,

$|\sigma_{in}^{-1}(j)|$, $|\sigma_{out}^{-1}(j)| \in \{0,1\}$, falls $m^2 > p_1(m)$, und es gilt:

$P_{\sigma_{in}(i,j)}$ enthält $(A_{in})_{ij}$ als Eingabe, $P_{\sigma_{out}(i,j)}$ enthält $(A_{out})_{ij}$ als Ausgabe.

□

4.2.5. Satz

Angenommen alle $D_g(m)$, $g \in S$, besitzen dieselbe Ein- und Ausgabeverteilung, und es gilt: $\sigma_{in} = \sigma_{out} = \sigma$. Sei ferner $G(n,m)$ vollständig bez. $\overline{G}(n/m)$ dann gilt:

$$\tau(n,m) \leq t_0(n/m)\tau_1(m) + \tau_0(n/m)\max\{1,m^2/p_1(m)\}.$$

Beweis:

Sei $(g_k,g_l) \in E(\overline{G}(n/m))$, und bezeichne v_j den Knoten von $G_l(m)$, der P_j von $D_g(m)$ repräsentiert. In einem Kommunikationsschritt von $\overline{D}(n/m)$ werde eine $(m \times m)$-Matrix von g_k nach g_l übertragen. $G(n,m)$ ist symmetrisch bez. $\overline{G}(n/m)$, und durch ι_i: $V(G_l(m)) \to V(G_i)$ werden die zugehörigen Isomorphismen bezeichnet. Wegen der Vollständigkeit von $G(n,m)$ (vgl. 3.1.9.) sind dann alle Ein-/Ausgabeknoten $v_{\sigma(i,j)}$ von G_k und G_l miteinander verbunden: $(\iota_k(v_{\sigma(i,j)}),\iota_l(v_{\sigma(i,j)})) \in E(G(n,m))$. Da nach Voraussetzung die Ein- und Ausgabewerte gleichmäßig auf die Knoten von $G_l(m)$ verteilt sind (vgl. 4.1.2. und 4.1.4.), können in einem parallelen Kommunikationsschritt von $D(n,m)$ also $\min\{m^2,p_l(m)\}$ Blockmatrixkoeffizienten aus G_k an die richtigen Positionen von G_l übertragen werden.

□

Für $\kappa(e) = p_l(m)$, $e \in E(\overline{G}(n/m))$, ist also

4.2.6. Korollar

$$\tau(n,m) = t_0(n/m)\tau_1(m) + \tau_0(n/m)\max\{1,m^2/p_1(m)\}.$$

□

Wir betrachten einen Netzwerkalgorithmus $D(n,m)$, dessen Kommunikationsgraph $G(n,m)$ bez. $\overline{G}(n/m)$ vollständig ist und $\kappa(e) = p_l(m)$, $e \in E(\overline{G}(n/m))$. In Kapitel 3 wurde gezeigt, daß ein bez. $\overline{G}(n/m)$ vollständiges symmetrisches Netzwerk $G(n,m)$ mit $\kappa(e) = p_l(m)$ genau ein zu $\overline{G}(n/m)$ duales $p_0(n/m)$-Hypernetz besitzt. Dies legt es nahe, in Analogie zu 1.1.10. zu $D(n,m)$ duale Netzwerkalgorithmen zur Berechnung von f zu betrachten.

Angenommen also es gibt Netzwerkalgorithmen $\overline{D}^*(m)$ über $K^{n/m} \times K^{n/m}$ zur Berechnung von $\overline{f}_{n/m}$, $D_g^*(n/m)$ über K zur Berechnung der Grundoperationen von

$\overline{D}^*(m)$ mit $p_l(m)$ bzw. $p_0(n/m)$ Prozessoren und Kommunikationsgraphen

$\overline{G}^*(m) \simeq G_l(m)$ bzw. $G_l^*(n/m) \simeq \overline{G}(n/m)$. Der zugehörige partitionierte

Netzwerkalgorithmus $D^*(n,m)$ heißt *dualer Netzwerkalgorithmus* von $D(n,m)$. Er berechnet f mit $p_0(n/m)p_1(m)$ Prozessoren und besitzt denselben (vollständigen) Kommunikationsgraphen $G(n,m)$ wie $D(n,m)$. Wir markieren nachfolgend die Laufzeiten, Kommunikationszeiten und Prozessoranzahlen von $D^*(n,m)$ durch ein $*$.

Wendet man 4.2.5. auf $D^*(n,m)$ an, so erhält man:

4.2.7. Satz

Sei $D^*(n,m)$ ein zu $D(n,m)$ dualer Netzwerkalgorithmus. Angenommen alle $D_g^*(n/m)$, $g \in S$, besitzen dieselbe Ein- und Ausgabeverteilung, und es gilt $\sigma_{in} = \sigma_{out}$. Dann gilt:

$$\tau^*(n,m) = t_0^*(m)\tau_1^*(n/m) + \tau_0^*(m)\max\{1,(n/m)^2/p_0(n/m)\}.$$

□

Besitzt $D(n,m)$ einen dualen Netzwerkalgorithmus, so kann mit Hilfe der Gleichungen in 4.2.6. und 4.2.7. im konkreten Beispiel leicht entschieden werden, welche der beiden Partitionierungen die kürzere Kommunikationszeit (bei Verwendung desselben Kommunikationsgraphen!) liefert. Um dies zu verdeutlichen, nehmen wir speziell an, daß die Laufzeiten und Kommunikationszeiten von der gleichen Größenordnung sind (vgl. 5.), d.h. $\tau_i = \theta(t_i)$, $\tau_i^* = \theta(t_i^*)$, $i \in \{0,1\}$.

Für die Beispiele 1.2.2., 1.2.4.-1.2.6. ist es möglich, in Bezug auf die Funktionen $p_0(n)$, $p_1(n) \in \{n, n^2\}$ duale Algorithmen zu betrachten. Die Untersuchung der Laufzeiten in Kapitel 2 hat in diesen Fällen ergeben:
$t_0(x)p_0(x) = \theta(x^3)$, $t_1(x)p_1(x) = \theta(x^3)$.
Insbesondere folgt also:
$t_0^*(x)p_0^*(x) = t_0^*(x)p_1(x) = \theta(x^3) \Rightarrow t_0^* = \theta(t_1)$

und $t_1^*(x)p_1^*(x) = t_1^*(x)p_0(x) = \theta(x^3) \Rightarrow t_1^* = \theta(t_0)$.

Damit sind die Laufzeiten $t(n,m)$ von $D(n,m)$ und $t^*(n,m)$ von $D^*(n,m)$ von der gleichen Größenordnung. Für die Kommunikationszeiten gilt jedoch:

$$\tau(n,m) = \theta(t(n,m) + t_0(n/m)\max\{1,m^2p_1(m)\},$$

$$\tau^*(n,m) = \theta(t(n,m) + t_1(m)\max\{1,(n/m)^2/p_0(n/m)\}).$$

Wir vergleichen die Terme für die externe Kommunikation für die angegebenen Beispiele:

(i) $p_1(m) = m$, $p_0(n/m) = (n/m)^2$:

$$t_0(n/m)\max\{1,m^2/p_1(m)\} = \theta(n),\ t_1(m)\max\{1,(n/m)^2/p_0(n/m)\} = \theta(m^2).$$

Für $m < n^{1/2}$ erhalten wir also eine kürzere Kommunikationszeit des dualen Algorithmus.

(ii) $p_1(m) = m^2$, $p_0(n/m) = n/m$:

$$t_0(n/m)\max\{1,m^2/p_1\} = \theta((n/m)^2),\ t_1(m)\max\{1,(n/m)^2/p_0(n/m)\} = \theta(n).$$

Auch in diesem Fall liefert der duale Algorithmus für $m < n^{1/2}$ die bessere Kommunikationszeit.

Zusammengefaßt besagen (i) und (ii), daß die Parallelisierung des externen Algorithmus mit einer linearen Anzahl Prozessoren für Blockgrößen $>n^{1/2}$ der Parallelisierung des externen Algorithmus mit einer quadratischen Prozessoranzahl vorzuziehen ist.

5. KOMMUNIKATIONSZEITEN IN NETZEN MIT BESCHRÄNKTER PROZESSORZAHL

Während in 4. Kommunikationszeiten partitionierter Algorithmen lediglich in Bezug auf Kommunikationsgraphen $G(n,m)$ mit variabler Prozessorzahl $p_0(n/m)p_1(m)$ betrachtet worden sind, werden in diesem Kapitel, ähnlich wie in 2.4., Netzwerke G mit beschränkter Prozessoranzahl p zugrundegelegt. Ziel ist es, durch den Vergleich partitionierter und nicht-partitionierter Netzwerkalgorithmen zur Berechnung von f mit p Prozessoren ein Maß für die Kommunikationskomplexität partitionierter Netzwerkalgorithmen zu gewinnen.
Wir gehen aus von einem bez. $\Gamma = \{G(n) \mid n \in \mathbb{N}_0\}$ partitionierbaren Netzwerkalgorithmus $D(n)$ zur Berechnung von f mit $p(n)$ Prozessoren. Durch Bestimmung einer geeigneten Blockgröße m_p läßt sich eine Partitionierung von $D(n)$ gewinnen, die nur p Prozessoren verwendet. Verzichtet man auf eine Partitionierung, so läßt sich $D(n)$ auf G unter Verwendung einer Netzwerkemulation $\phi: V(G(n)) \to V(G)$ (vgl. [4] und 3.4.) realisieren. Die hierfür maximal erforderliche Kommunikationszeit τ vergleichen wir mit der minimalen Kommunikationszeit $\tau(n,m_p)$ einer Partitionierung $D(n,m_p)$ von $D(n)$ mit p Prozessoren. Es wird vorausgesetzt, daß $G(n)$ und G gleichen durchschnittlichen Knotengrad besitzen. Dies ist z.B. für Netze $G(n)$ und G mit gleicher Gitterstruktur gegeben.

Ist $D(n)$ ein Algorithmus mit konstanter Effizienz, so weisen wir für eine große Klasse von Algorithmen, die systolisch ausgeführte Algorithmen beinhaltet, nach: die Kommunikationszeit des partitionierten Algorithmus ist durch die Kommunikationszeit des nicht-partitionierten Algorithmus nach unten beschränkt, $\tau(n,m_p) = \Omega(\tau)$. Falls die internen Algorithmen parallel ausgeführt werden, so verschlechtert sich die Kommunikationszeit des partitionierten Algorithmus sogar um einen Faktor $(p(n)/p)^{1-\beta}$ mit $0 \le 1-\beta \le 1$, d.h. wir zeigen: $\tau(n,m_p) = \Omega((p(n)/p)^{1-\beta}\,\tau)$. Damit wird der in 2.3. geführte Nachweis einer großen Berechnungskomplexität partitionierter Algorithmen ergänzt durch eine entsprechende Aussage über ihre Kommunikationszeit.

Wir nehmen in diesem Kapitel wieder vereinfachend an, daß $D(n,m)$ nur einen internen Algorithmus $D_1(m)$ besitzt.
Sei $G = (V(G),E(G))$ mit $|V(G)| = p$. Für den durchschnittlichen Knotengrad $\gamma(G)$ von G und $\gamma(G(n))$ setzen wir voraus

$$\gamma(G(n)) \leq \gamma(G).$$

Wir beginnen mit der Untersuchung einer nicht-partitionierten Realisierung von $D(n)$ auf G.

5.0.1. Lemma

Sei $\phi: V(G(n)) \to V(G)$ eine uniforme Emulation, d.h. $|\phi^{-1}(v)| = p(n)/p$ für alle $v \in V$ und $\kappa := |\phi'^{-1}(e_1)| = |\phi'^{-1}(e_2)|$ für alle $e_1, e_2 \in E(G)$. Dann gilt:

$$\kappa \leq p(n)/p.$$

Beweis:

Wir zeigen $|E(G(n))| \geq |E(G(n))||V(G)||V(G(n))|^{-1}\kappa$. Wegen $|V(G)| = p$ und $|V(G(n))| = p(n)$ ist dann $\kappa \leq p(n)/p$.

$$\begin{aligned}
|E(G(n))| \geq |E(G)|\kappa &= (1/2) \sum_{v \in V(G)} \mathrm{grad}(v)\ \kappa \\
&= (1/2)\gamma(G)|V(G)|\kappa \geq (1/2)\gamma(G(n))|V(G)|\kappa \\
&= (1/2)\Big(\sum_{v \in V(G(n))} \mathrm{grad}(v)/|V(G(n))|\Big)\ |V(G)\kappa \\
&= |E(G(n))||V(G)|\kappa/|V(G(n))|.
\end{aligned}$$

□

Bezeichne $\tau(n)$ die Kommunikationszeit von $D(n)$ und τ die Kommunikationszeit, welche die Simulation von $D(n)$ auf G vermöge der Emulation ϕ erfordert. Ohne nähere Angaben über diese Simulation läßt sich eine elementare obere Schranke für τ wie folgt angeben:

5.0.2. Lemma

$$\tau \leq \kappa\tau(n).$$

Beweis:

Angenommen $D(n)$ verwendet für die Ausführung eines Kommunikationsschrittes S die Kanten $E_S \subseteq E(G(n))$. Um S auf G zu simulieren ist dann höchstens Zeit

$\max_{e \in E(G)} |\phi'^{-1}(e) \cap E_S| \leq \kappa$ erforderlich, so daß man für die gesamten Kommunikationskosten erhält $\tau \leq \kappa\tau(n)$.

□

Auf der Grundlage von 5.0.1. beschreiben wir κ durch $\kappa = (p(n)/p)^{\beta}$, $0 \leq \beta \leq 1$. Damit wird 5.0.2. zu

$$\tau \leq (p(n)/p)^{\beta}\tau(n).$$

Wir wenden uns nun der Kommunikationszeit der Partitionierung $D(n,m_p)$ von $D(n)$ zu. Die Blockgröße m_p erfülle Bedingung 2.3.1., insbesondere gilt also $|V(G(n,m_p))| = p$. Bei der Abschätzung der Kommunikationszeit $\tau(n,m_p)$ werden die Fälle $p_1(m_p) > 1$ und $p_1(m_p) = 1$ unterschieden. Aus 4.2.3.' erhalten wir:

5.0.3. Fakt

Ist $p_1(m_p) > 1$, so ist $\tau(n,m_p) \geq t_0(n/m_p)\tau_1(m_p)$.

Für $p_1 \equiv 1$ ist $\tau(n,m_p) \geq \tau_0(n/m_p)m_p^2$.

□

Damit wird die Kommunikationszeit des externen Algorithmus nur dann für die Schranke von $\tau(n,m_p)$ berücksichtigt, wenn keine interne Kommunikation erfolgen kann.

Nachfolgend beschreiben wir alle auftretenden Kommunikationszeiten durch die Berechnungszeiten. Wir definieren Funktionen α, α_0, α_1, indem wir setzen

$$\begin{aligned}
\tau(n) &= \alpha(n)t(n) \\
\tau_0(n/m) &= \alpha_0(n/m)t_0(n/m) \\
\tau_1(m) &= \alpha_1(m)t_1(m).
\end{aligned}$$

Diese Darstellung liegt nahe, wenn die Kommunikation der betrachteten Algorithmen analog zum Datenfluß systolischer Algorithmen erfolgt, d.h. wenn jeweils abwechselnd ein paralleler Berechnungsschritt und eine konstante Anzahl paralleler Datenübertragungsschritte ausgeführt werden. (Vgl. 6.3.). Allgemein wird man von Netzwerken, welche die Abhängigkeitsstruktur eines bestimmten Algorithmus im intuitiven Sinn optimal abbilden, erwarten, daß die α-Funktionen konstant sind. Nur für universeller

zu verwendende Netze wird man zulassen, daß α, α_0, α_1 (möglichst langsam) wachsende Funktionen sind.

Wir sind nun in der Lage, die Kommunikationszeit des partitionierten Netzwerkalgorithmus mit Hilfe der Effizienzen von internem und externem Algorithmus auszudrücken. Da die Effizienzdefinition aus 2.3. ausschließlich die Berechnungszeit und die Prozessoranzahl eines parallelen (partitionierten) Algorithmus verwendet, kann sie vollständig für (partitionierte) Netzwerkalgorithmen übernommen werden.
Wie in 2.3. legen wir die sequentiellen Bezugsalgorithmen so fest, daß $t(seq)(n) = T(seq)(n)$. Da $D(n,m_p)$ eine Partitionierung von $D(n)$ ist, ist ferner $t_0(seq)(n/m_p) = t(seq)(n/m_p)$.

5.0.4. Lemma

Falls $p_1(m_p) > 1$, so ist

$$\tau(n,m_p) \geq T(seq)(n)\alpha_1(m_p)/(pE_p(D(n,m_p))).$$

Falls $p_1(m_p) = 1$, so ist

$$\tau(n,m_p) \geq T(seq)(n/m_p)\alpha_0(n/m_p)\ m_p^2/(pE_p(\overline{D}(n/m_p)))$$

Beweis:

Im ersten Fall haben wir

$$\tau(n,m_p) \geq \alpha_1(m_p)t_0(n/m_p)t_1(m_p)$$

$$= \alpha_1(m_p)t_0(seq)\ (n/m_p)t_1(seq)(m_p)/[pE_{p_0}\ (n/m_p)E_{p_1}(m_p)]$$

$$= \alpha_1(m_p)T(seq)(n)/(pE_p(D(n,m_p))).$$

Im zweiten Fall erhalten wir:

$$\tau(n,m_p) \geq \tau_0(n/m_p)m_p^2 = \alpha_0(n/m_p)t_0\ (n/m_p)m_p^2$$

$$= \alpha_0(n/m_p)\ m_p^2 t_0(seq)(n/m_p)/[pE_p(\overline{D}(n/m_p))]$$

$$\geq \alpha_0(n/m_p)m_p^2 T(seq)(n/m_p)/\ [pE_p(\overline{D}(n/m_p))].$$

In gleicher Weise wird die Kommunikationszeit τ des nicht-partitionierten Algorithmus abgeschätzt durch:

5.0.5. Lemma

$\tau \leq \alpha(n)T(seq)(n)/[p^{\beta}p(n)^{1-\beta}E_{p(n)}(D(n))]$

Beweis:

Wir haben

$$\tau \leq (p(n)/p)^{\beta}\alpha(n)t(n)$$

$$= (p(n)/p)^{\beta}\alpha(n)t(seq)(n)/[p(n)E_{p(n)}(D(n))]$$

$$= \alpha(n)T(seq)(n)/[p^{\beta}p(n)^{1-\beta}E_{p(n)}(D(n))].$$

□

Zusammenfassend liefern 5.0.4. und 5.0.5. den angestrebten Vergleich der Kommunikationszeiten:

5.0.6. Satz

Sei $D(n)$ ein bez. $\Gamma = \{G(n) \mid n \in \mathbb{N}_0\}$ partitionierbarer Netzwerkalgorithmus und $D(n,m_p)$ eine Partitionierung von $D(n)$ mit $p_0(n/m_p)p_1(m_p) = p$. Sei ferner für einen Graphen G mit $|V(G)| = p$

$\phi: V(G(n)) \to V(G)$ eine uniforme Netzwerkemulation. Bezeichnet τ die Kommunikationszeit des vermöge ϕ ausgeführten $D(n)$ und $\tau(n,m_p)$ die Kommunikationszeit von $D(n,m_p)$, so gilt:

(i) $\tau(n,m_p)/\tau \geq (\alpha_1(m_p)/\alpha(n))\ [E_{p(n)}(D(n))/E_p(D(n,m_p))]\ (p(n)/p)^{1-\beta}$,
falls $p_1(m_p) > 1$.

(ii) $\tau(n,m_p)/\tau \geq (\alpha_0(n/m_p)/\alpha(n))[T(seq)(n/m_p)m_p^{\ 2}/T(seq)(n)]*$

$$*[E_{p(n)}(D(n))/E_p(\overline{D}(n/m_p))](p(n)/p)^{1-\beta},$$

falls $p_1(m_p) = 1$.

□

Wir wenden dies Resultat an auf den Fall konstanter Funktionen α,α_0,α_1 und einen externen Algorithmus konstanter Effizienz. Dabei haben wir, da $D(n,m_p)$ Partitionierung von $D(n)$ ist: $p_0(n/m_p) = p(n/m_p)$ und somit auch $E_{p_0(n/m_p)} = E_{p(n/m_p)}$.

5.0.7. Korollar

Angenommen die Funktionen α, α_0, α_1 und die Effizienz $E_{p(n)}(D(n))$ des Ausgangsalgorithmus sind konstant. Dann gilt:

(i) $\tau(n,m_p) = \Omega((p(n)/p)^{1-\beta}(E_{p_1(m_p)})^{-1}\tau)$, falls $p_1(m_p) > 1$,

(ii) $\tau(n,m_p) = \Omega((p(n)/p)^{1-\beta}[T(seq)(n/m_p)m_p^{\ 2}/T(seq)(n)]\ \tau)$,
falls $p_1(m_p) = 1$.

□

In keinem der Beispiele wächst die sequentielle Berechnungszeit $T(seq)(n)$ stärker als $\theta(n^3)$, so daß wir 5.0.7.(ii) im Hinblick auf die Beispiele ersetzen können durch

$$\tau(n,m_p) = \Omega((p(n)/p)^{1-\beta}m_p^{\ -1}\tau).$$

(Für $T(seq)(n) = \theta(n^2)$ kann auch noch auf den Faktor $m_p^{\ -1}$ verzichtet werden.)

Unter Berücksichtigung der Tatsache, daß beispielsweise für eine 2-dimensionale Nächste-Nachbar-Verbindung $\beta = 1/2$ und für lineare Netze sogar $\beta = 0$, erweist sich die Kommunikationszeit des partitionierten Netzwerkalgorithmus im ersten Fall als deutlich schlechter. Für $\beta = 1/2$ und $p(n) = n^2$ erhält man im Fall (ii) immer noch $\tau(n,m_p) = \Omega(\tau)$.

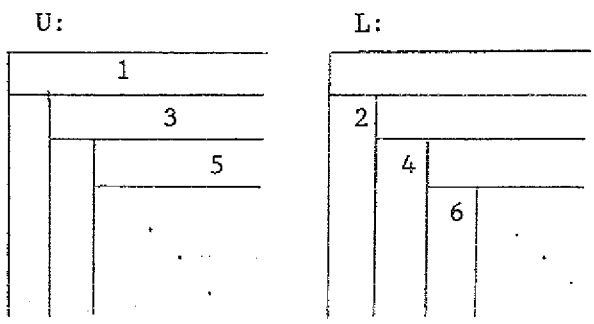
6. DIE PARTITIONIERTE LU-ZERLEGUNG - EIN BEISPIEL

In diesem Kapitel werden einige der Ergebnisse der Arbeit anhand eines Beispiels erläutert. Wir legen hierfür die partitionierte LU-Zerlegung (vgl. 1.2.5.) zugrunde. Bei Eingabe einer Matrix $A \in K^n \times K^n$ werden die untere und die obere Dreiecksmatrix L und U berechnet mit $LU = A$, $l_{ii} = 1$, $1 \leq i \leq n$. Wir setzen voraus, daß A eine Matrix ist, die in dieser Form darstellbar ist.

Das zugrundeliegende mathematische Verfahren ist der Crout-Algorithmus. D.h. die Zeilen 1,...,n (Spalten 1,...,n) von U (L) werden nacheinander berechnet. Nachdem die Zeilen 1,...,k-1 von U und die Spalten 1,...,k-1 von L bestimmt worden sind, berchnet man die Werte aus Zeile k von U und aus Spalte k von L durch

$$u_{ki} = l_{ii}u_{ki} = a_{ki} - (l_{k1}u_{1i} + \ldots + l_{k,k-1}u_{k-1,i}),\ k \leq i,$$
$$l_{ik} = [a_{ik} - (l_{i1}u_{1k} + \ldots + l_{i,k-1}u_{k-1,k})]/u_{kk},\ k \leq i.$$

Die Berechnungsreihenfolge des Algorithmus läßt sich also wie folgt veranschaulichen (da $l_{ii} = 1$, brauchen die Diagonalenelemente von L nicht berechnet zu werden):



Die Menge der internen Operationen ist in diesem Beispiel gegeben durch
S = {Matrix-Multiplikation/Addition, Inversion von Dreiecksmatrizen, LU-Zerlegung}.

Für den externen Algorithmus $\overline{M}(n/m)$ gehen wir aus von den in 1.2.5. dargestellten Algorithmen mit $p_0(n/m) \in \{1, n/m, (n/m)^2\}$.
Alle drei internen Operationen können mit parallelen Algorithmen für $p_1(m) \in \{1, m, m^2\}$ Prozessoren berechnet werden, für die Matrix-Multiplikation haben wir darüberhinaus einen parallelen Algorithmus mit m^3 Prozessoren. Diese (elementaren) Algorithmen erhält man aus den in 1.2.2., 1.2.5. und 1.2.6. dargestellten Algorithmen, indem die Blockgröße 1 gesetzt wird, und n durch m ersetzt wird. Auf diese Weise lassen sich insgesamt also 12 verschiedene parallele partitionierte Algorithmen zur Berechnung der LU-Zerlegung einer Matrix gewinnen, nämlich mit
$p(n,m) = 1, m, m^2, m^3, n/m, (n/m)^2, n, nm, nm^2, n^2/m, n^2$ oder n^2m
Prozessoren. Zunächst bedeutet dies, daß man mit Hilfe der Partitionierung parallele Algorithmen für eine große Klasse verschiedener Prozessoranzahlfunktionen gewinnen kann. Legt man die Blockgröße m beispielsweise fest durch $m = n^{1/4}$ oder durch $m = \log n$, so erhält man parallele Algorithmen zur Berechnung der LU-Zerlegung einer Matrix für folgende Prozessoranzahlen:

$$p(n) \in \{1, \log n, \log^2 n, \log^3 n, n/\log n, n^2/\log^2 n, n, n\log n, n\log^2 n, n^2/\log n, n^2, n^2\log n, n^{1/4}, n^{1/2}, n^{3/4}, n^{3/2}, n^{5/4}, n^{7/4}, n^{9/4}\}.$$

Um parallele Algorithmen für eine beschränkte Prozessorzahl p anzugeben, besteht damit offensichtlich eine Möglichkeit darin, aus einer solchen (möglichst großen) Klasse von Funktionen p(n) diejenige auszuwählen, für deren Wert p(n') bei vorgegebener Problemgröße n' gilt: $p(n') \approx p$.
Die in 2.3.1. formulierte Anforderung, die Blockgröße m in Abhängigkeit von n und p so zu wählen, daß gilt:

$$p_0(n/m)p_1(m) = p,$$

stellt eine Systematisierung des skizzierten Vorgehens dar.
Wir beginnen in 6.1. mit der Untersuchung der Laufzeiten und Effizienzen der verschiedenen Partitionierungen. In 6.2. wird exemplarisch gezeigt, wie sich im Falle der LU-Zerlegung partitionierte Algorithmen der Tiefe 2 gewinnen lassen. In 6.3. werden die Ergebnisse aus Kapitel 4 auf das Beispiel angewendet und verschiedene Realisierungen der partitionierten LU-Zerlegung auf Prozessornetzwerken angegeben. Die Berechnung der Kommunikationskosten bei fester Prozessoranzahl p erfolgt in 6.4.. Dabei steht der Vergleich mit den Kommunikationskosten einer nicht-partitionierten Version der LU-Zerlegung im Vordergrund.

6.1. BERECHNUNGSZEIT

Die Berechnungszeiten der externen Algorithmen $\overline{M}_{LU}(n/m)$ sind in 2.1.4. tabelliert worden. Aus der Laufzeittabelle 2.1.4. geht insbesondere hervor, daß für die angegebenen Parallelisierungen von $\overline{M}_{LU}(n/m)$ - nämlich mit

$p_0(n/m) \in \{1, n/m, (n/m)^2\}$ - die Matrix-Multiplikation die dominierende interne Operation ist, in dem Sinne daß $t_{0,1}(n/m) = \Omega(t_0(n/m))$. Es ist deswegen gerechtfertigt, sich hinsichtlich der Laufzeit auf den Subalgorithmus von $\overline{M}_{LU}(n/m)$ zu beschränken, der nur aus den Blockmatrix-Multiplikationen besteht. Für diesen Subalgorithmus erhält man analog zu 2.1.5. folgende Laufzeittabelle:

6.1.1. Multiplikations-Berechnungszeit der partitionierten LU-Zerlegung

$p_0 \backslash p_1$	1	m	m^2	m^3
1	$n^3/3+a$	$n^3/(3m)+b$	$n^3/(3m^2)+c$	$n^3(\log m)/m^3+d$
n/m	$3n^2m/2+O(nm^2)$	$3n^2/2+O(nm)$	$3n^2/(2m)+O(n)$	$3n^2\log m/m^2+e$
$(n/m)^2$	$3nm^2-1$	$3nm-1$	$3n-1$	$(3n/m-1)\log m+O(n/m)$

$a = O(n^2m - nm^2)$

$b = O(n^2 - nm)$

$c = O(n^2/m - n)$

$d = O((n^2/m^2)\log m)$

$e = O((n/m)\log m)$.

Für die nachfolgenden Berechnungen berücksichtigen wir wie in den vorangegangenen Kapiteln lediglich die dominierenden Terme der Laufzeitpolynome. Wir bestimmen die Effizienzen der verschiedenen Partitionierungen in Bezug auf die in 1.2. angegebenen sequentiellen Algorithmen (vgl. 2.3.) und erhalten folgende Werte:

6.1.2. Effizienzen der partitionierten LU-Zerlegung

$p_0 \backslash p_1$	1	m	m^2	m^3
1	1	1	1	1/log m
n/m	2/9	2/9	2/9	2/(9log m)
$(n/m)^2$	1/9	1/9	1/9	1/(9log m)

Die gleichen Effizienzwerte in den ersten drei Stellen jeder Zeile der Tabelle 6.1.2. sind darauf zurückzuführen, daß die Matrix-Multiplikation mit $p_1(m) \varepsilon \{1, m, m^2\}$ Prozessoren Effizienz 1 besitzt: Die Faktoriserung 2.3.5. besagt in diesem Fall, daß die Effizienzen des partitionierten Algorithmus und des externen Algorithmus gleich sein müssen. An dieser Stelle sei daran erinnert, daß die Korrekturfunktion q(m) (vgl. 2.3.4.) im vorliegenden Beispiel nicht berücksichtigt zu werden braucht. Es ist nämlich

$$t(seq)(n,m) = (1/3)n^3 ,$$

also

$$T(seq)(n) = \min_m t(seq)(n,m) = (1/3)n^3$$

und somit q(m) = 1.

Ebenfalls unmittelbar aus der Tabelle 6.1.2. ersichtlich ist die Aussage des Satzes 2.4.1.: einen parallelen partitionierten Algorithmus zur Berechnung der LU-Zerlegung mit maximaler Effizienz erhält man im Falle der sequentiellen Ausführung des internen oder des externen Algorithmus. Da die Vergleiche der Effizienzen bzw. Laufzeiten in 2.4.1., 2.4.5.-2.4.7. jeweils nur innerhalb einer Zeile oder Spalte der Tabelle durchgeführt worden sind, muß im konkreten Fall noch überprüft werden, welche der Partitionierungen mit $p_0(n/m) = 1$ oder $p_1(m) = 1$ das beste Laufzeit- bzw. Effizienzverhalten aufweist. In unserem Beispiel ist es offenbar optimal, möglichst alle p Prozessoren nur für die (interne) Matrix-Multiplikation zu verwenden falls, $p \leq n^2$.
Bevor wir den Fall $p > n^2$ betrachten, sei darauf hingewiesen (vgl. auch 2.3., S. 39), daß in Abhängigkeit von den Werten n und p nicht immer alle der 12 angegebenen partitionierten Algorithmen zur Verfügung stehen, um die LU-Zerlegung mit p Prozessoren zu berechnen. Auf der Grundlage der Beziehung

$$p(n,m) = p_0(n/m)p_1(m) = p$$

sind folgende drei Fälle zu unterscheiden :

(i) $1 \le p \le n$

(ii) $n \le p \le n^2$

(iii) $n^2 \le p \le n^3$.

Wir ordnen diese drei Fälle den verschiedenen durch $p(n,m)$ charakterisierten Partitionierungsmöglichkeiten zu:

6.1.3.

$p_0 \backslash p_1$	1	m	m^2	m^3
1	$(p = 1)$	(i)	(i), (ii)	(i),(ii),(iii)
n/m	(i)	$(p = n)$	(ii)	(ii), (iii)
$(n/m)^2$	(i), (ii)	(ii)	$(p = n^2)$	(iii)

Stehen also im Verhältnis zur Problemgröße n viele Prozessoren zur Verfügung, d.h. ist $p > n^2$, so ist es nicht möglich, einen in Bezug auf die Effizienz optimalen Algorithmus unter allen partitionierten Algorithmen auszuwählen. Es können lediglich die Partitionierungen mit $p_1(m) = m^3$ untereinander verglichen werden. Die allgemeine Aussage von Satz 2.4.5. beinhaltet allerdings keinen Vergleich dieser drei Effizienzen untereinander, da daß Effizienzprodukt gemäß Lemma 2.4.4. in allen drei Fällen im ganzen Intervall [1,n] monoton fallend ist. Wir berechnen im Beispiel die nicht-konstanten Effizienzen im Fall (iii) unter der Bedingung $p(n,m) = p$ und erhalten:

6.1.4. Effizienzen im Fall $n^2 \le p \le n^3$

$p_1 \backslash p_0$	1	n/m	$(n/m)^2$
m^3	$3/\log p$	$4/(9\log(p/n))$	$1/(9\log(p/n^2))$

Setzt man $p = n^{2+\alpha}$, $0 \le \alpha \le 1$, so überzeugt man sich leicht davon, daß die sequentielle Ausführung des externen Algorithmus immer ein besseres Effizienzverhalten aufweist als die Partitionierung mit $p_0(n/m) = n/m$, da

$$3/[(2+\alpha)\log n] > 4/[9(1+\alpha)\log n],\ 0 \le \alpha \le 1.$$

Jedoch liefert die gemischte Parallelisierung mit $p_0(n/m) = (n/m)^2$ für $\alpha < 1/13$ eine größere Effizienz als die Ausführung mit $p_0(n/m) = 1$:

$$\alpha < 1/13 \quad \Rightarrow \quad 1/(9\alpha \log n) > 3/[(2+\alpha)\log n].$$

6.2. PARTITIONIERUNGEN DER TIEFE 2

Zu Beginn dieses Kapitels wurde darauf verwiesen, daß mit Hilfe von Partitionierungen parallele Algorithmen für neue Prozessoranzahlfunktionen p(n) gewonnen werden können. Es liegt nahe zu überprüfen, ob diese (neuen) Algorithmen selbst wiederum partitionierbar sind (vgl. 1.1.8.). D.h. sie sollen als externe Algorithmen neuer partitionierter Algorithmen verwendet werden. Dies ist gleichbedeutend damit, eine Partitionierung der Tiefe 2 (vgl. 1.3.3.) der LU-Zerlegung anzugeben. Wir betrachten exemplarisch die Partitionierungen mit nm' und n^2m' Prozessoren für $m' = n^{1/2}$ und erhalten parallele Algorithmen $M_1(n)$, $M_2(n)$ zur Berechnung der LU-Zerlegung mit $n^{3/2}$ bzw. $n^{5/2}$ Prozessoren. Diese Algorithmen verwenden lediglich die arithmetischen Operationen $S = \{+,-,*,/\}$, so daß sie als Algorithmen über $K^m \times K^m$ interpretiert werden können. Dies führt noch nicht unmittelbar zu Algorithmen, die eine m-Partitionierung der LU-Zerlegung berechnen. Erweitert man S jedoch um die interne Operation "LU-Zerlegung einer $(m \times m)$-Matrix" (vgl. 1.1.9.), so können ähnlich wie in 1.2.5. durch geringfügige Modifikationen von $M_1(n)$ und $M_2(n)$ Partitionierungen dieser Algorithmen gewonnen werden, die eine m-Partitionierung der LU-Zerlegung berechnen. Wir beschreiben nachfolgend diese Algorithmen.

Dabei verwenden wir in diesem und in den folgenden Abschnitten wiederum Querstriche, um m-Partitionierungen zu kennzeichnen, d.h. für eine beliebige Matrix B bezeichnet $\overline{B}$ eine m-Partitionierung von B und $\overline{b}_{ij} \in K^m \times K^m$ die Koeffizienten von $\overline{B}$. Matrix-Blöcke der Größe m werden immer durch Kleinbuchstaben bezeichnet.

Es seien $\overline{M}_{LU}(n/m)$, $\overline{M}_{/}(n/m)$ die in 1.2.5. und 1.2.6. angegebenen Algorithmen über $\overline{K}$ mit

$$p_0(n/m) = (n/m)^2$$

Prozessoren zur Berechnung von m-Partitionierungen der LU-Zerlegung einer Matrix und der Inversion einer Dreiecksmatrix. Ferner seien $\overline{M}_*^2(n/m)$ und $\overline{M}_*^3(n/m)$ die in 1.2.2. beschriebenen Algorithmen über $\overline{K}$ mit

$$p_0(n/m) = (n/m)^2 \text{ bzw. } p_0(n/m) = (n/m)^3$$

Prozessoren zur Berechnung einer m-Partitionierung von AB + C, $A,B,C \in K^n \times K^n$. Wir bezeichnen die Ergebnismatrizen, die durch Anwendung dieser Prozeduren auf die Eingabematrizen $\overline{A}$, $\overline{B}$, $\overline{C} \in \overline{K}^{n/m} \times \overline{K}^{n/m}$ erhalten werden durch

$$\overline{M}_{LU}(n/m)(\overline{A}),\ \overline{M}_/(n/m)(\overline{A}),\ \overline{M}_*^i(n/m)(\overline{A},\overline{B},\overline{C}),\ i=2,3.$$

Für $A \in K^n \times K^n$ bezeichne $(A_{ij})_{1\le i,j\le\sqrt{n/m}}$ eine $(nm)^{1/2}$-Partitionierung von A. Dann kann eine m-Partitionierung von A beschrieben werden durch $(\overline{A}_{ij})_{1\le i,j\le\sqrt{n/m}}$ mit $\overline{A}_{ij} \in \overline{K}^{\sqrt{n/m}} \times \overline{K}^{\sqrt{n/m}}$, $1 \le i,j \le (n/m)^{1/2}$.

Die Partitionierungen von $M_1(n)$, $M_2(n)$ besitzen die externen Algorithmen $\overline{M}_1(n/m)$, $\overline{M}_2(n/m)$, welche unmittelbar aus den in 1.2.5. angegebenen Algorithmen gewonnen werden können.

$\overline{M}_1(n/m)$ und $\overline{M}_2(n/m)$ berechnen aus der Eingabe $\overline{A} = (\overline{A}_{ij})_{1\le i,j\le\sqrt{n/m}}$ die Ausgabe $\overline{L} = (\overline{L}_{ij})_{1\le i,j\le\sqrt{n/m}}$, $\overline{U} = (\overline{U}_{ij})_{1\le i,j\le\sqrt{n/m}}$.

$\overline{M}_1(n/m)$:

```
begin
  M̄_LU(√nm/m)(Ā_11) = L̄_11 Ū_11;
  L̄_11' := M̄_/(√nm/m)(L̄_11);
  Ū_11' := M̄_/(√nm/m)(Ū_11);
  for all i = 1,...,√n/m pardo
   begin
    Ū_1i := M̄_*^2(√nm/m)(L̄_11',Ā_1i,0̄);

    L̄_i1 := M̄_*^2(√nm/m)(Ā_i1,Ū_11',0̄)
   end
  for k = 2,...,√n/m do
   begin
    for j = 1,...,k-1 do
     Ā_kk := M̄_*^2(√nm/m)(-L̄_kj,Ū_jk,Ā_kk);
    M̄_LU(√nm/m)(Ā_kk) = L̄_kk Ū_kk;
    L̄_kk' := M̄_/(√nm/m)(L̄_kk);
```

```
      U_kk' := M_/(√nm/m)(U_kk);
      for all i = k+1,...,√n/m pardo
       begin
        for j = 1,...,k-1 do
         begin
          A_ki := M_*^2(√nm/m)(-L_kj,U_ji,A_ki);

          A_ik := M_*^2(√nm/m)(-L_ij,U_jk,A_ik)
         end
        U_ki := M_*^2(√nm/m)(L_kk',A_ki,0);

        L_ik := M_*^2(√nm/m)(A_ik,U_kk',0)
       end
     end
   end

M_2(n/m):
 begin
  for k = 1,...,√n/m-1 do
   begin
    M_LU(√nm/m)(A_kk) = L_kk U_kk;
    L_kk' := M_/(√nm/m)(L_kk);
    U_kk' := M_/(√nm/m)(U_kk);
    for all i = k+1,...,√n/m pardo
     begin
      U_ki := M_*^3(√nm/m)(L_kk',A_ki,0);

      L_ik := M_*^3(√nm/m)(A_ik,U_kk',0)
     end
    for all i,j = k+1,...,√n/m pardo
     A_ij := M_*^3(√nm/m)(-L_ik,U_kj,A_ij)
   end
  M_LU(√nm/m)(A_√n/m,√n/m) = L_√n/m,√n/m U_√n/m,√n/m
 end
```

Offensichtlich stellen beide Algorithmen $\overline{M}_1(n/m)$ und $\overline{M}_2(n/m)$ die explizite Ausführung von Partitionierungen der Tiefe 2 der LU-Zerlegung mit n

bzw. n^2 Prozessoren im Sinne von Definition 1.3.3. dar. Wir haben in beiden Fällen die drei folgenden Partitionierungen der Tiefe 2 vorliegen:

$(M_0,M_1) = (\overline{M}_{LU}(n/\sqrt{nm}), \overline{M}_*(\sqrt{nm}/m))$
$(M_0,M_1') = (\overline{M}_{LU}(n/\sqrt{nm}), \overline{M}_/(\sqrt{nm}/m))$
$(M_0,M_1'') = (\overline{M}_{LU}(n/\sqrt{nm}), \overline{M}_{LU}(\sqrt{nm}/m))$.

Wir bestimmen die Laufzeit $t_{0,1}(n/m)$ des Subalgorithmus der Blockmatrix-Multiplikationen für $\overline{M}_1(n/m)$ und $\overline{M}_2(n/m)$ und erhalten im ersten Fall mit $y = (n/m)^{1/2}$:

$$t_{0,1}(n/m) = t_{0,1}(LU)(y) + 2t_{0,1}(/)(y) + 2t_{0,1}(*)(y) + \sum_{k=2}^{y} [(3k+1)t_{0,1}(*)(y) + t_{0,1}(LU)(y) + 2t_{0,1}(/)(y)]$$

und für $\overline{M}_2(n/m)$:

$$t_{0,1}(n/m) = \sum_{k=1}^{y-1} [t_{0,1}(LU)(y) + 2t_{0,1}(/)(y) + 3t_{0,1}(*)(y)] + t_{0,1}(LU)(y).$$

Unter Verwendung der in 2.1.4. tabellierten Laufzeiten also:

$$\overline{M}_1(n/m): t_{0,1}(n/m) = 3y^3/2 + 15y^2/2 + 4y - 5$$
$$= (3/2)(n/m)^{3/2} + O(n/m).$$

$$\overline{M}_2(n/m): t_{0,1}(n/m) = 7y^2 + (y-1)\log y - 6y + 1$$
$$= 7n/m + O((n/m)^{1/2}\log n/m).$$

Es zeigt sich, daß im zweiten Fall durch die Verwendung einer größeren Prozessoranzahl ($p_0(n/m) = (n/m)^{5/2}$) keinerlei Gewinn gegenüber der (Multiplikations-)Laufzeit des Algorithmus mit $p_0(n/m) = (n/m)^2$ erzielt wird, die Kosten sind sogar größer. (Zum Vergleich: der Algorithmus mit $p_0(n/m) = (n/m)^2$ lieferte $t_{0,1}(n/m) = 3n/m - 1$.) Dies ist darauf zurückzuführen, daß die Prozeduren $\overline{M}_{LU}(\sqrt{nm}/m)$ und $\overline{M}_/(\sqrt{nm}/n)$ in $\overline{M}_2(n/m)$ jeweils nur

mit n/m Prozessoren und nicht wie $\overline{M}_*^3(\sqrt{nm}/m)$ mit $(n/m)^{3/2}$ Prozessoren ausgeführt worden sind. In der Folge dominieren die Kosten ihrer Blockmatrix-Multiplikationen den Wert $t_{0,1}(n/m)$ für ganz $\overline{M}_2(n/m)$.

Wir ergänzen aus dem genannten Grund die Laufzeit- und Effizienztabellen 6.1.1. und 6.1.2. lediglich um die Parallelisierung $p_0(n/m) = (n/m)^{3/2}$ und erhalten:

6.2.1. Laufzeit und Effizienz des Multiplikationssubalgorithmus für $p_0(n/m) = (n/m)^{3/2}$

p_1	1	m	m^2	m^3
t	$3(nm)^{3/2}/2 + a$	$3n^{3/2}m^{1/2}/2+b$	$3n^{3/2}/2m^{1/2}+c$	$(3(n/m)^{3/2}/2)\log m + d$
E_p	2/9	2/9	2/9	$2/(9\log m)$

$a = O(nm^2)$, $b = O(nm)$, $c = O(n)$, $d = O((n/m)\log m)$.

6.3. DIE PARTITIONIERTE LU-ZERLEGUNG ALS NETZWERKALGORITHMUS

In 4.1. ist am Beispiel der Matrix-Multiplikation illustriert worden, wie eine Realisierung eines partitionierten Algorithmus auf einer dreidimensionalen Nächste-Nachbar-Verbindung gewonnen werden kann. In diesem Abschnitt werden wir analog zu dem genannten Beispiel die partitionierte LU-Zerlegung als partitionierten Netzwerkalgorithmus mit Kommunikationsgraph G(n,m) beschreiben und die Kommunikationszeiten gemäß 4.2. bestimmen.
Wir beschränken uns auf die drei folgenden Partitionierungen:

- $p_0(n/m) = n/m$, $p_1(m) = 1$. Der Kommunikationsgraph $G_1 = G_1(n,m)$ ist ein lineares Netz mit n/m Knoten:
  $V(G_1) = \{P(1),\ldots,P(n/m)\}$,
  $E(G_1) = \{(P(i),P(i+1)) \mid 1 \le i \le n/m-1\}$.

- $p_0(n/m) = 1$, $p_1(m) = m^2$. Der Kommunikationsgraph $G_2 = G_2(n,m)$ ist ein zweidimensionales Gitter mit m^2 Knoten:
  $V(G_2) = \{P(i,j)=P((i-1)m+j) \mid 1 \le i,j \le m\}$,
  $E(G_2) = \{(P(i,j),P(i+1,j)), (P(j,i),P(j,i+1)) \mid 1 \le i \le m-1,$
  $1 \le j \le m\}$.

- $p_0(n/m) = n/m$, $p_1(m) = m^2$. Der Kommunikationsgraph $G_3 = G_3(n,m)$ ist ein dreidimensionales Gitter mit nm Knoten:
  $V(G_3) = \{P(i,j,k) \mid 1 \le i,j \le m,\ 1 \le k \le n/m\}$,
  $E(G_3) = \{(P(i,j,k),P(i,j,k+1)) \mid 1 \le i,j \le m,\ 1 \le k \le n/m-1\}$
  $\{(P(i,j,k),P(i+1,j,k)),(P(j,i,k),P(j,i+1,k)) \mid 1 \le k \le n/m,$
  $1 \le j \le m,\ 1 \le i \le m-1\}$.

Gemäß der Definition eines partitionierten Netzwerkalgorithmus (vgl. 4.1.) genügt es, Realisierungen der internen Algorithmen auf einem quadratischen Gitter und des externen Algorithmus auf einem linearen Netz anzugeben. Denn das $p_1(m)$-Hypernetz $\overline{G}(n/m)$, welches als Kommunikationsgraph des externen Algorithmus dient, ist im ersten Fall unmittelbar durch G_1 gegeben; für $p_0(n/m) = 1$ besteht es lediglich aus einem Punkt, und im dritten Fall ist $\overline{G}(n/m) \simeq G_1$, wenn wir die Äquivalenzklassen in $V(G_3)$ für $P(i,j,k)$, $P(i',j',k') \in V(G_3)$ festlegen durch

$$P(i,j,k) \sim P(i',j',k') \Leftrightarrow k = k'.$$

Entsprechend bestehen die Subgraphen für die Realisierungen der internen Algorithmen im ersten Fall aus isolierten Punkten; in den anderen beiden Fällen sind sie isomorph zu G_2.

Wir beschreiben nachfolgend Realisierungen des externen Algorithmus auf G_1 und der internen Algorithmen auf G_2 und legen dabei dieselben mathematischen Verfahren zugrunde wie in 1.2.. Insbesondere handelt es sich bei 6.3.1. und 6.3.2. um das Verfahren von Crout zur Bestimmung der LU-Zerlegung einer Matrix. Die Kommunikation der Algorithmen 6.3.1.-6.3.3. ist ähnlich wie bei systolischen Algorithmen organisiert, d.h. sie wird durch einen wellenartigen Datenfluß realisiert (vgl. [16],[24]).

Wir verwenden folgende Bezeichnungen:

Für $P \in V(G)$ bedeutet "P: (arithmetische Operation g)", daß der durch P repräsentierte Prozessor die Operation g ausführt.

Für $(P,P') \in E(G_1)$, $\bar{a} \in K^m \times K^m$, bedeutet "$P \to P'$: $\bar{a}$", daß die Blockmatrix $\bar{a}$ von der durch den Hypernetzknoten P repräsentierten Prozessorgruppe in die durch P' repräsentierte Prozessorgruppe übertragen wird. Analog bezeichnet für $(P,P') \in E(G_2)$, $a \in K$, "$P \to P'$: a" die Übertragung des Werts a von Prozessor P nach Prozessor P'.

σ_{in} und σ_{out} kennzeichnen die Ein- und Ausgabeverteilung im Sinne von 4.2.4.. Im Falle des externen Algorithmus 6.3.1. sind σ_{in}, σ_{out} dabei gegeben durch Abbildungen

$$\{1,\ldots,n/m\} \times \{1,\ldots,n/m\} \to \{1,\ldots,p_0(n/m)\}.$$

6.3.1. $\bar{D}_{LU}(n/m)$ mit Kommunikationsgraph G_1

Eingabe: $\bar{A} = (\bar{a}_{ij})_{1\le i,j\le n/m} \in \bar{K}^{n/m} \times \bar{K}^{n/m}$.
$\sigma_{in}(i,j) = j,\ 1 \le i,j \le n/m$.

Ausgabe: $\bar{L} = (\bar{l}_{ij})_{1\le i,j\le n/m}$, $\bar{U} = (\bar{u}_{i,j})_{1\le i,j\le n/m} \in \bar{K}^{n/m} \times \bar{K}^{n/m}$.
$\sigma_{out}(i,j) = j,\ 1 \le i,j \le n/m$.

```
begin
  for k = 1,...,n/m-1 do
   begin
    P(k): ā_kk = l̄_kk' ū_kk;

    P(k): l̄_kk := l̄_kk'^-1;

    P(k): ū_kk' := ū_kk^-1;
    for i = 1,...,n/m-k do
     begin
      for all j = k,...,k+i-1 pardo
       P(j) → P(j+1): l̄_2k+i-j-1,k;
      for J=k and for all j = k+1,...,k+i pardo
       P(J): l̄_k+i,k := ā_k+i,k ū_kk'
       P(j): ā_2k+i-j,j := ā_2k+i-j,j - l̄_2k+i-j,k ū_k,j
     end
    for i = 1,...,n/m-k do
     begin
      for all j = k+i,...,n/m-1 pardo
       P(j) → P(j+1): l̄_n/m-j-i-k,k;
      for J=k+i and for all j = k+i+1,...,n/m pardo
```

```
      P(J): ū_{k,k+i} := l̄_{kk} ā_{k,k+i}
      P(j): ā_{n/m-j-i-k+1,j} := ā_{n/m-j-i-k+1,j} - l̄_{n/m-j-i-k+1,k} ū_{k,j}
    end
   l̄_{kk} := l̄_{kk}';
   P(n/m): ā_{n/m,n/m} = l̄_{n/m,n/m} ū_{n/m,n/m}
  end
 for all k = 2,...,n/m pardo
  P(k): for i = 1,...,k-1 do
        l̄_{ik} := 0̄;
 for all k = 1,...,n/m-1 pardo
  P(k): for i + k+1,...,n/m do
        ū_{ik} := 0̄
end
```

Die Berechnungs- und die Kommunikationszeit von $\overline{D}_{LU}(n/m)$ sind gegeben durch

$$t_0(n/m) = (n/m)^2 + n/m - 2$$

$$\tau_0(n/m) = (n/m)^2 - n/m.$$

Die Ausführungszeit des Algorithmus kann noch um einen Faktor 1/2 reduziert werden, wenn man eine Überlappung der Iterationen der äußeren k-Schleife zuläßt. Die folgende Illustration des Algorithmus für eine (6×6)-Matrix A mir $(m \times m)$-Blockmatrizen $\overline{a}_{ij}$ (also $A \in K^{6m} \times K^{6m}$) macht von dieser Möglichkeit Gebrauch. Wir verzichten im Schaubild auf die Querstriche zur Markierung der Blockmatrizen, es ist also $l_{ij} = \overline{l}_{ij}$, $u_{ij} = \overline{u}_{ij}$, und verwenden die Bezeichnung
$a_{ij}(k) = \overline{a}_{ij}(k) = \overline{a}_{ij} - \overline{l}_{i1}\overline{u}_{1j} - \ldots - \overline{l}_{ik}\overline{u}_{kj}$.
Die tabellenartige Darstellung gibt den Algorithmus in seinem zeitlichen Ablauf wieder. In der obersten Zeile sind die Prozessoren und Verbindungskanten von G_1 angegeben, letztere als Pfeile, da die Kommunikation nur in der bezeichneten Richtung verläuft. In Zeile i wird angegeben, welche Werte zum Zeitpunkt i in den jeweiligen Prozessoren berechnet werden bzw. entlang der jeweiligen Kante übertragen werden (kursiv).

P(1)	→	P(2)	→	P(3)	→	P(4)	→	P(5)	→	P(6)
l_{11}^{-1} u_{11}^{-1}										
l_{21}	l_{11}^{-1}	u_{12}								
l_{31}	l_{21}	$a_{22}(1)$ l_{22} u_{22}	l_{11}^{-1}	u_{13}						
l_{41}	l_{31}	$a_{32}(1)$	l_{21}	$a_{23}(1)$	l_{11}^{-1}	u_{14}				
l_{51}	l_{41}	$a_{42}(1)$	l_{31}	$a_{33}(1)$	l_{21}	$a_{24}(1)$	l_{11}^{-1}	u_{15}		
l_{61}	l_{51}	$a_{52}(1)$	l_{41}	$a_{43}(1)$	l_{31}	$a_{34}(1)$	l_{21}	$a_{25}(1)$	l_{11}^{-1}	u_{16}
	l_{61}	$a_{62}(1)$	l_{51}	$a_{53}(1)$	l_{41}	$a_{44}(1)$	l_{31}	$a_{35}(1)$	l_{21}	$a_{26}(1)$
		l_{22}^{-1} u_{22}^{-1}	l_{61}	$a_{63}(1)$	l_{51}	$a_{54}(1)$	l_{41}	$a_{45}(1)$	l_{31}	$a_{36}(1)$
		l_{32}	l_{22}^{-1}	u_{23}	l_{61}	$a_{64}(1)$	l_{51}	$a_{55}(1)$	l_{41}	$a_{46}(1)$
		l_{42}	l_{32}	$a_{33}(2)$ l_{33} u_{33}	l_{22}^{-1}	u_{24}	l_{61}	$a_{65}(1)$	l_{51}	$a_{56}(1)$
		l_{52}	l_{42}	$a_{43}(2)$	l_{32}	$a_{34}(2)$	l_{22}^{-1}	u_{25}	l_{61}	$a_{66}(1)$
		l_{62}	l_{52}	$a_{53}(2)$	l_{42}	$a_{44}(2)$	l_{32}	$a_{35}(2)$	l_{22}^{-1}	u_{26}
			l_{62}	$a_{63}(2)$	l_{52}	$a_{54}(2)$	l_{42}	$a_{45}(2)$	l_{32}	$a_{36}(2)$

P(1)	→	P(2)	→	P(3)	→	P(4)	→	P(5)	→	P(6)
				l_{33}^{-1} u_{33}^{-1}	l_{62}	$a_{63}(2)$	l_{52}	$a_{55}(2)$	l_{42}	$a_{46}(2)$
				l_{43}	l_{33}^{-1}	u_{34}	l_{62}	$a_{65}(2)$	l_{52}	$a_{56}(2)$
				l_{53}	l_{43}	$a_{44}(3)$ l_{44} u_{44}	l_{33}^{-1}	u_{35}	l_{62}	$a_{66}(2)$
				l_{63}	l_{53}	$a_{54}(3)$	l_{43}	$a_{45}(3)$	l_{33}^{-1}	u_{36}
					l_{63}	$a_{64}(3)$	l_{53}	$a_{55}(3)$	l_{43}	$a_{46}(3)$
						l_{44}^{-1} u_{44}^{-1}	l_{63}	$a_{65}(3)$	l_{53}	$a_{56}(3)$
						l_{54}	l_{44}^{-1}	u_{45}	l_{63}	$a_{66}(3)$
						l_{64}	l_{54}	$a_{55}(4)$ l_{55} u_{55}	l_{44}^{-1}	u_{46}
							l_{64}	$a_{65}(4)$	l_{54}	$a_{56}(4)$
								l_{55}^{-1} u_{55}^{-1}	l_{64}	$a_{66}(4)$
								l_{65}	l_{55}^{-1}	u_{56}
									l_{65}	$a_{66}(5)$ l_{66} u_{66}

6.3.2. Interne LU-Zerlegung mit Kommunikationsgraph G_2

Eingabe: $\overline{a} = (a_{ij})_{1\le i,j\le m} \in K^m \times K^m$
$\sigma_{in}(i,j) = (i-1)m+j.$

Ausgabe: $\overline{l} = (l_{ij})_{1\le i,j\le m}$, $\overline{u} = (u_{ij})_{1\le i,j\le m} \in K^m \times K^m$
$\sigma_{out}(i,j) = (i-1)m+j.$

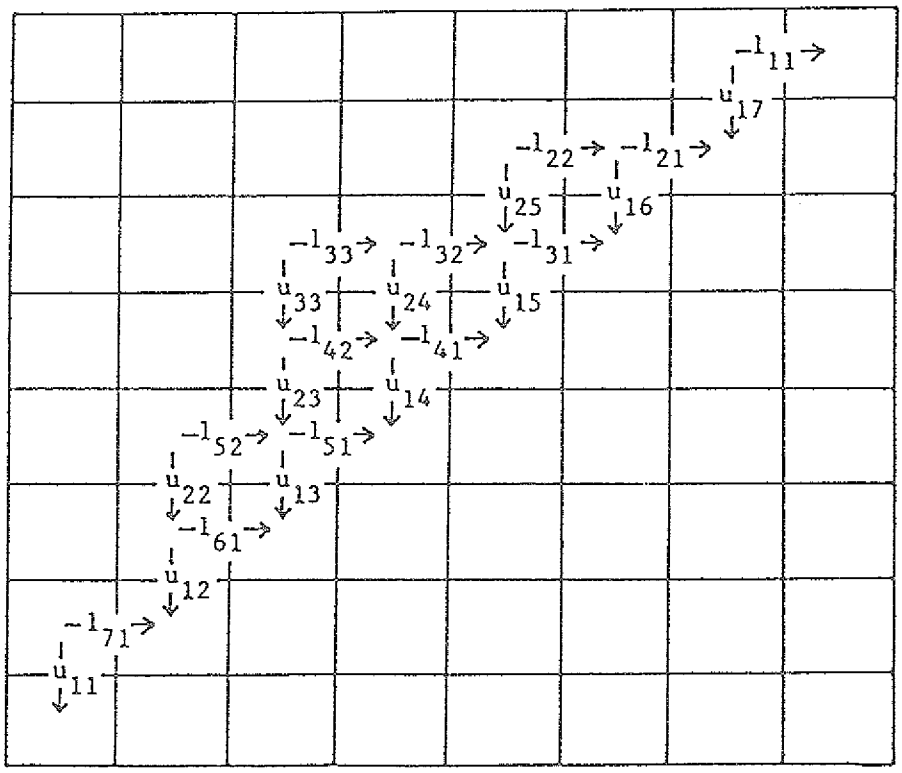
Die folgende Darstellung des Algorithmus wie auch des Algorithmus 6.3.3. läßt Prozessoren P(i,j) mit i,j > m zu. Arithmetische und Kommunikationsbefehle, die solche Prozessoren betreffen, betrachten wir als nicht definiert. Die (parallele) Ausführungszeit des Algorithmus wird durch diese Befehle nicht beeinflußt.

```
begin
 P(1,1): u_11 := a_11;
 for all k = 1,...,m pardo
  P(k,k): l_kk := 1;
 for all i = 1,...,m-1 pardo
  for all j = i+1,...,m pardo
   P(i,j): l_ij := 0
   P(j,i): u_ji := 0;
 for k = 1,...,m-1 do
  begin
   for all i = 1,...,k pardo
    begin
     for all j = i,...,3k-2i pardo
      P(j,3k-i-j) → P(j,3k-i-j+1): l_ji
      P(3k-i-j,j) → P(3k-i-j+1,j): u_ij;
     for J = i and for all j = i+1,...,3k-2i pardo
      P(J,3k-2J+1): u_i,3k-2i+1 := a_i,2k-2i+1
      P(3k-2J+1,J): l_3k-2i+1,i := a_3k-2i+1,i/u_ii
      P(j,3k-i-j+1): a_j,3k-i-j+1 := a_j,3k-i-j+1 - l_ji u_i,3k-i-j+1
    end
   for all i = 1,...,k pardo
    begin
     for all j = i,...,3k-2i+1 pardo
      P(j,3k-i-j+1) → P(j,3k-i-j+2): l_ji
      P(3k-i-j+1,j) → P(3k-i-j+2,j): u_ij;
     for J = i and for all j = i+1,...,3k-2i+1 pardo
```

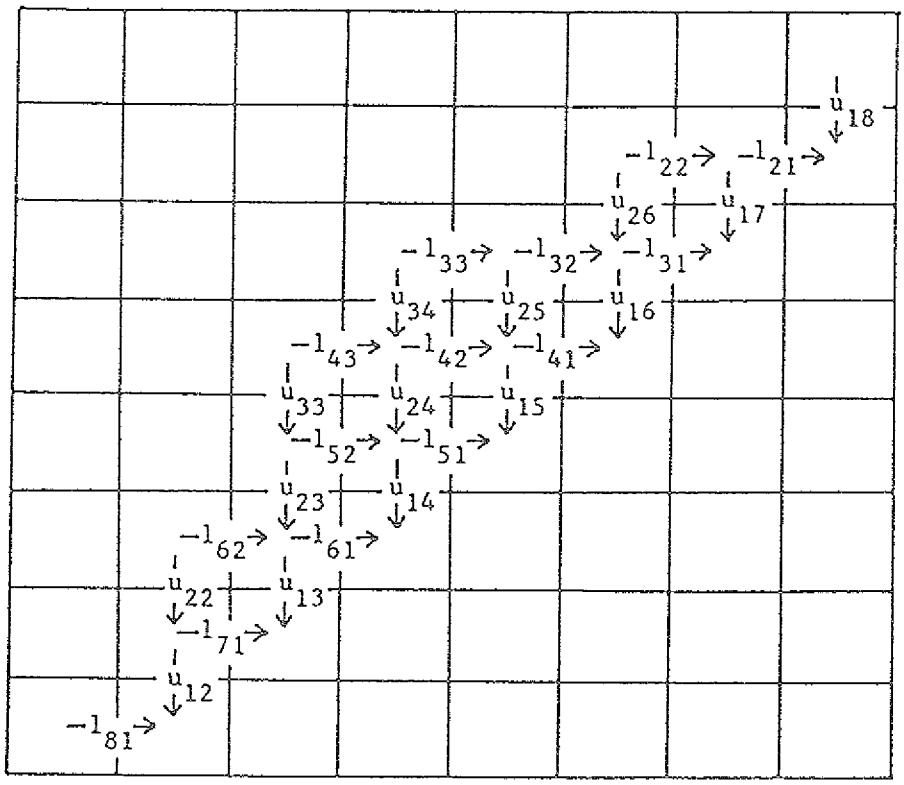
```
      P(J,3k-2J+2): u_{i,3k-2i+2} := a_{i,3k-2i+2}
      P(3k-2J+2,J): l_{3k-2i+2,i} := a_{3k-2i+2,i}/u_{ii}
      P(j,3k-i-j+2): a_{j,3k-i-j+2} := a_{j,3k-i-j+2} - l_{ji}u_{i,3k-i-j+2}
    end
   for all i = 1,...,k pardo
    begin
     for all j = i,...,3k-2i+2 pardo
      P(j,3k-i-j+2) → P(j,3k-i-j+3): l_{ji}
      P(3k-i-j+2,j) → P(3k-i-j+3,j): u_{ij};
     for J = i and for all j + i+1,...,3k-2i+2 pardo
      P(J,3k-2J+3): u_{i,3k-2i+3} := a_{i,3k-2i+3}
      P(3k-2J+3,J): l_{3k-2i+3,i} := a_{3k-2i+3,i}/u_{ii}
      P(j,3k-i-j+3): a_{j,3k-i-j+3} - l_{ji}u_{i,3k-i-j+3}
    end
   P(k+1,k+1): u_{k+1,k+1} := a_{k+1,k+1}
  end
 end
```

Berechnungs- und Kommunikationszeit der internen LU-Zerlegung berechnen sich somit zu $t_6(m) = \tau_6(m) = 3m - 3$.

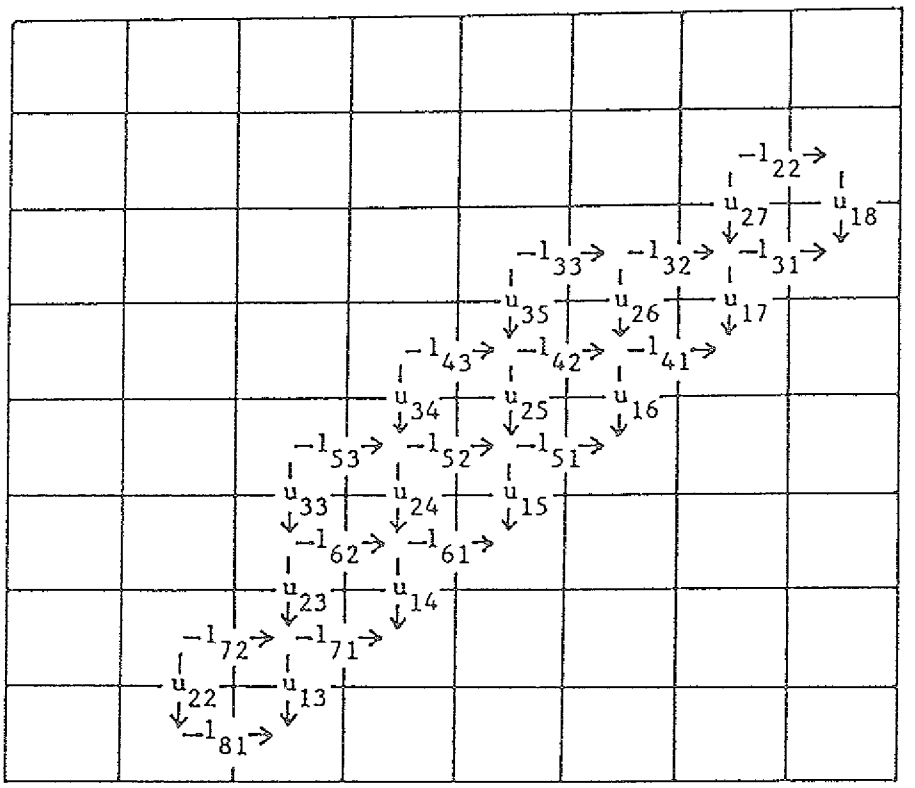
Es ist erforderlich, die angegebenen drei parallelen Kommunikations- und Berechnungsschritte auszuführen, bevor mit der Iteration k+1 begonnen wird, um zu vermeiden, daß Kanten von G_2 in einem Kommunikationsschritt mehrfach verwendet werden. Die folgenden Abbildungen stellen für eine (8 × 8)-Matrix die dritte Iteration des Algorithmus dar. Kommunikations- und Berechnungsschritte wechseln einander dabei ab. Jede der folgenden Abbildungen besteht aus einem Schema, dessen 64 Felder den Matrix-Koeffizienten bzw. den Prozessoren von G_2 entsprechen. Für die Berechnungsschritte ist in jedem Feld der Wert eingetragen, der von dem entsprechenden Prozessor berechnet wird, die Übertragung der Daten ist durch Pfeile markiert.



							u_{18}
					u_{26}	$a_{27}(1)$	
			u_{34}	$a_{35}(2)$	$a_{36}(1)$		
		l_{43}	$a_{44}(2)$	$a_{45}(1)$			
		$a_{53}(2)$	$a_{54}(1)$				
	l_{62}	$a_{63}(1)$					
	$a_{72}(1)$						
l_{81}							



						u_{27}	$a_{28}(1)$
				u_{35}	$a_{36}(2)$	$a_{37}(1)$	
			$a_{44}(3)$	$a_{45}(2)$	$a_{46}(1)$		
		l_{53}	$a_{54}(2)$	$a_{55}(1)$			
		$a_{63}(2)$	$a_{64}(1)$				
	l_{72}	$a_{73}(1)$					
	$a_{82}(1)$						



							u_{28}
					u_{36}	$a_{37}(2)$	
				$a_{45}(3)$	$a_{46}(2)$	$a_{47}(1)$	
			$a_{54}(3)$	$a_{55}(2)$	$a_{56}(1)$		
		l_{63}	$a_{64}(2)$	$a_{65}(1)$			
		$a_{73}(2)$	$a_{74}(1)$				
	l_{82}	$a_{83}(1)$					

6.3.3. Interne Inversion von Dreiecksmatrizen mit Kommunikationsgraph G_2

Eingabe: Obere Dreiecksmatrix $\overline{a} = (a_{ij})_{1\le i,j\le m} \in K^m \times K^m$

$\sigma_{in}(i,j) = (i-1)m+j$.

Ausgabe: Obere Dreiecksmatrix $\overline{a}' = \overline{a}^{-1} = (a_{ij}')_{1\le i,j\le m} \in K^m \times K^m$

$\sigma_{out}(i,j) = (i-1)m+j$.

Der nachfolgende Algorithmus verwendet die Prozessoren $P(i,j)$ mit $i > j$ nicht für die Berechnung, d.h. die untere Dreiecksmatrix $\overline{l}$ mit $l_{ii} = 1$, $1 \le i \le m$, und die obere Dreieckmatrix $\overline{u}$ können simultan auf dem Prozessornetzwerk G_2 invertiert werden.

Im Unterschied zu Datenflußalgorithmen im engeren Sinne lassen wir zu, daß Daten, die einen Prozessor des Netzwerks durchlaufen haben, im lokalen Speicher dieses Prozessors gespeichert werden können. Insbesondere handelt es sich hierbei um die Diagonalenelemente a_{kk}', $2 \le k \le m$, die zu Beginn der Rechnung an alle Prozessoren der k-ten Spalte verteilt werden.

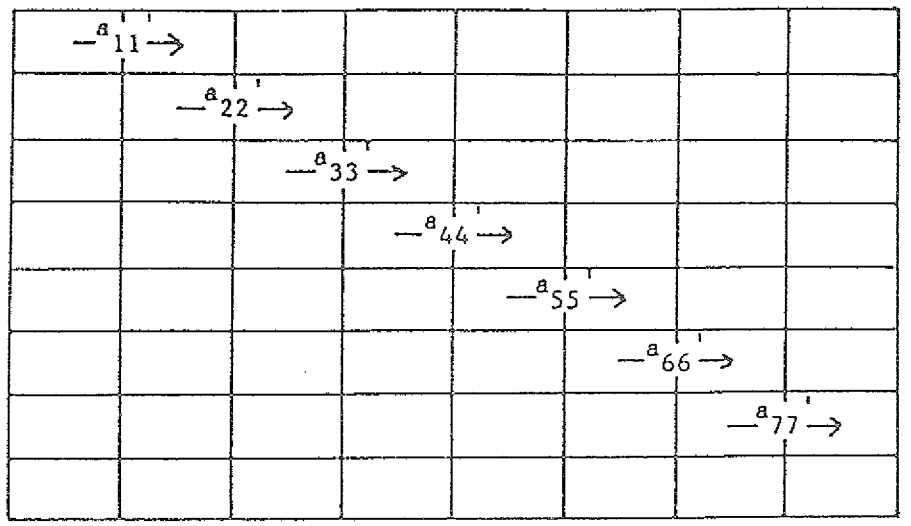
```
begin
 for all k = 1,...,m pardo
  P(k,k): a_kk' := 1/a_kk;
 for all k = 2,...,m pardo
  for i = 1,...,m-1 do
   P(k-i+1,k) → P(k-i,k): a_kk';
 for all k = 1,...,m-1 pardo
  begin
   P(k,k) → P(k,k+1): a_kk';
            P(k,k+1): c_k,k+1 := a_kk' a_k,k+1;
   P(k,k+1) → P(k,k+2): a_kk';
              P(k,k+1): a'_k,k+1 := a'_k+1,k+1 c_k,k+1
              P(k,k+2): c_k,k+2 := a_kk' a_k,k+2
  end
 for i = 1,...,m-2 do
  for all k = 1,...,m-i pardo
   begin
    for all j = 0,...,i-1, r = 0,...,i pardo
     P(k,k+i+r) → P(k,k+i+r+1): a'_k,k+i-r
     P(k+1,k+i+j+1) → P(k,k+i+j+1): a_k+i-j,k+i+j+1;
```

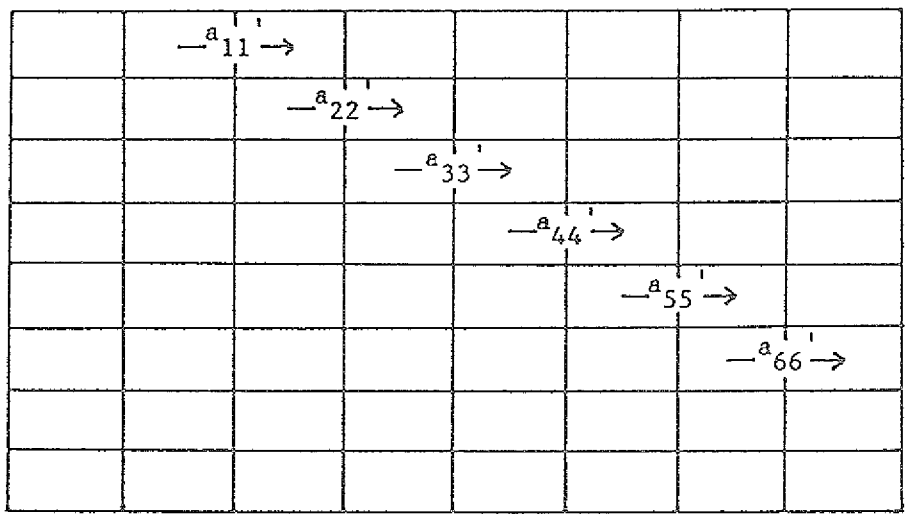
```
      for J=i and for all j = 0,...,i-1 pardo
       P(k,k+2J+1): c_{k,k+2i+1} := a_{kk}' a_{k,k+2i+1}
       P(k,k+i+j+1): c_{k,k+i+j+1} := c_{k,k+i+j+1} + a'_{k,k+i-j} a_{k+i-j,k+i+j+1};
      for all j = 0,...,i-1, r = 0,...,i pardo
       P(k,k+i+r+1) → P(k,k+i+r+2): a'_{k,k+i-r}
       P(k+1,k+i+j+2) → P(k,k+i+j+2): a_{k+i-j,k+i+j+2};
      for J=i and for all j = 0,...,i-1 pardo
       P(k,k+J+1): a'_{k,k+i+1} := -a'_{k+i+1,k+i+1} c_{k,k+i+1}
       P(k,k+2J+2): c_{k,k+2i+2} := a'_{kk} a_{k,k+2i+2}
       P(k,k+i+j+2): c_{k,k+i+j+2} := c_{k,k+i+j+2} + a'_{k,k+i-j} a_{k+i-j,k+i+j+2}
     end
   for all i = 2,...,m pardo
    for all j = 1,...,i-1 pardo
     P(i,j): a_{ij}' := 0
  end
```

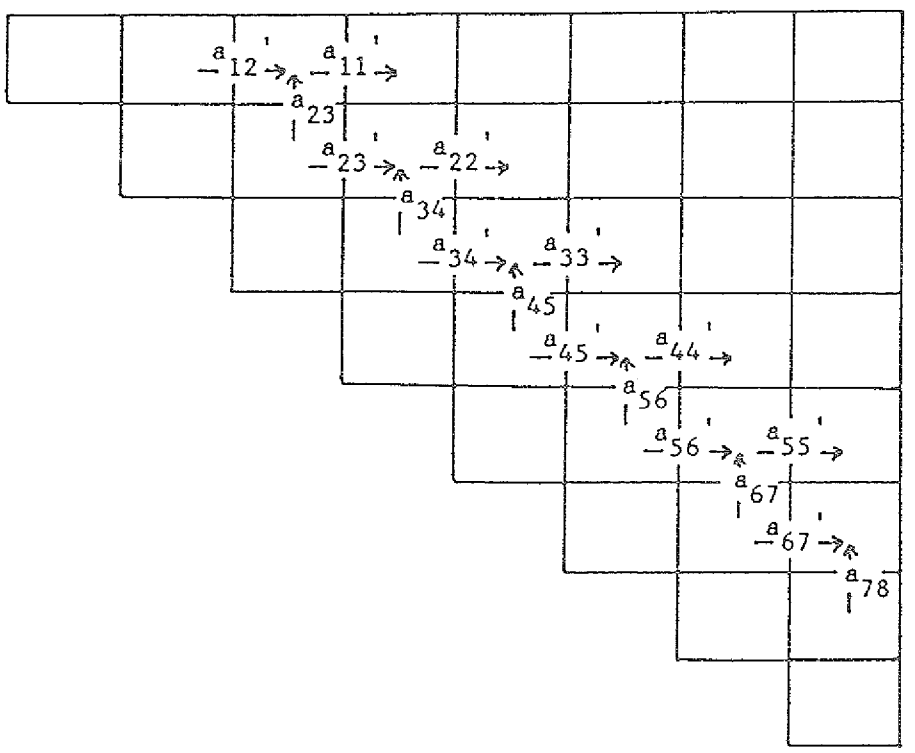
Berechnungs- und Kommunikationszeit der internen Inversion von Dreiecksmatrizen sind damit gegeben durch

$t_4(m) = 2m - 1$, $\tau_4(m) = (m - 1) + (2m - 3) = 3m - 4$.

Wir skizzieren den Datenfluß des Algorithmus (ohne die anfängliche Verteilung der Werte a_{ii}') für eine (8×8)-Matrix. Die Darstellung ist analog zum Beispiel in 6.3.2., wobei auf die Wiedergabe der Berechnungsschritte verzichtet wird.


a11
a22
a33
a44
a55
a66
a77


a11
a22
a33
a44
a55
a66


a12
a11
a23
a23
a22
a34
a34
a33
a45
a45
a44
a56
a56
a55
a67
a67
a78

6.3.4. Interne Matrix-Multiplikation mit Kommunikationsgraph G_2

Eingabe: $\overline{a}, \overline{b} \in K^m \times K^m$

$\sigma_{in}(i,j) = (i-1)m+j.$

Ausgabe: $\overline{c} = \overline{a}\overline{b} \in K^m \times K^m$

$\sigma_{out}(i,j) = (i-1)m+j.$

Für die interne Matrix-Multiplikation auf dem quadratischen Gitter G_2 verwenden wir den in [3] beschriebenen Netzwerkalgorithmus. Diesem Algorithmus liegt der Algorithmus von Cannon für die Matrix-Multiplikation auf einem zyklisch verbundenen Gitter zugrunde. Da G_2 keine zyklischen Verbindungen besitzt, muß ein Kommunikationsschritt des Cannon-Algorithmus geeignet simuliert werden. Hierzu wird zu Beginn der Rechnung jede Zeile i von a und jede Spalte j von b ersetzt durch

$$a_{i1} \cdots a_{i,m/2}\ a_{i,m/2+1}\ a_{i,m/2+2} \cdots a_{im}$$
$$a_{i,m/2}\ \ a_{i,m/2-1} \cdots a_{i1}$$

bzw.

$$b_{1j} \cdots b_{m/2,j}\ b_{m/2+1,j}\ b_{m/2+2,j} \cdots b_{mj}$$
$$b_{m/2,j}\ \ b_{m/2-1,j} \cdots b_{1j}.$$

D.h. für $k \geq m/2$ enthalten die Prozessoren $P(i,k)$ die Werte a_{ik} und $a_{i,m-k+1}$ und die Prozessoren $P(k,j)$ die Werte b_{kj} und $b_{m-k+1,j}$, $1 \leq i,j \leq m$.

Nachdem diese Verteilung der Daten in G_2 einmal erzeugt worden ist, kann jeder Kommunikationsschritt in einem zyklisch verbundenen Gitter durch zwei Kommunikationsschritte in einem Gitter ohne zyklische Verbindungen realisiert werden. Als Berechnungs- und Kommunikationszeit des so gewonnenen Algorithmus erhält man: $t_1(m) = m$, $\tau_1(m) = (m-1) + 2*2(m-1) = 5m - 5$.

Wir sind nun in der Lage, Realisierungen der partitionierten LU-Zerlegung auf den eingangs beschriebenen Netzen G_1, G_2 und G_3 anzugeben.

6.3.5. Partitionierte LU-Zerlegung mit linearem Kommunikationsgraph G_1

Eingabe: Für $1 \leq j \leq n/m$ erhält $P(j)$ als Eingabe die Koeffizienten $a_{i,(j-1)m+1},\ldots,a_{i,jm}$ von A, $1 \leq i \leq n$. Dies entspricht gerade der j-ten Spalte der m-Partitionierung $\overline{A}$ von A.

Ausgabe: Am Ende der Rechnung enthält $P(j)$ die Spalte j von $\overline{L}$ und $\overline{U}$, $1 \leq j \leq n/m$, also die Koeffizienten

$u_{i,(j-1)m+1},\ldots,u_{i,jm}$, $l_{i,(j-1)m+1},\ldots,l_{i,jm}$ von U und L, $1 \le i \le n$.

$D_{LU}(n,m)$ erhält man aus Algorithmus 6.3.1., indem die Befehle "P: (arithmetische Operation g)" ersetzt werden durch einen von P ausgeführten sequentiellen Algorithmus zur Berechnung von g. Für $\bar{a} = (a_{ij})_{1\le i,j\le m}$ wird ein Kommunikationsschritt "$P \to P'$: $\bar{a}$" realisiert

durch die m^2 Datenübertragungen $P \to P'$: a_{ij}, $1 \le i,j \le m$. (vgl. 4.1., 4.2.6.). Für die Kommunikationszeit $\tau(n,m)$ von $D_{LU}(n,m)$ erhalten wir also:

$$\tau(n,m) = \tau_0(n/m)m^2 = n^2 - nm.$$

6.3.6. Partitionierte LU-Zerlegung mit Kommunikationsgraph G_2 (zweidimensionales Gitter)

Eingabe: Für $1 \le i,j \le m$ erhält $P(i,j)$ als Eingabe die $(n/m)^2$ Koeffizienten $a_{rm+i,sm+j}$, $0 \le r,s \le n/m-1$, von A.
Ausgabe: Am Ende der Rechnung enthält $P(i,j)$ die Koeffizienten $l_{rm+i,sm+j}$ und $u_{rm+i,sm+j}$ von L und U, $0 \le r,s \le n/m-1$.

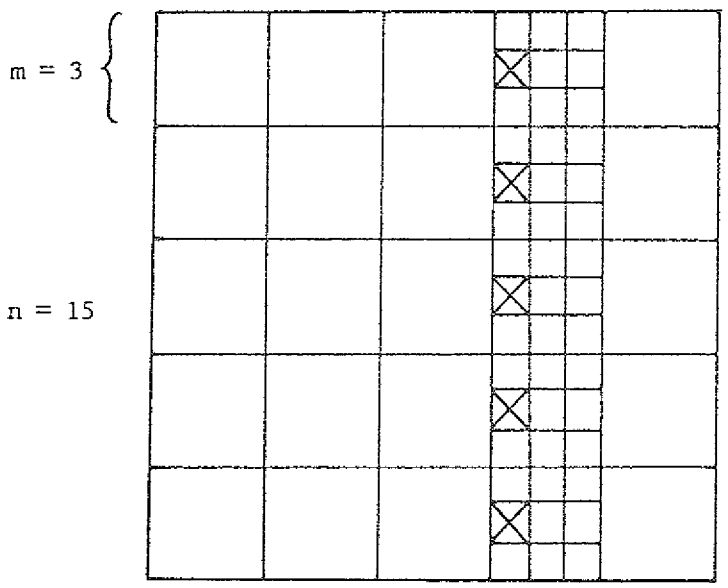
$D'_{LU}(n,m)$ erhält man aus dem in 1.2.5. angegebenen sequentiellen Algorithmus, indem jede arithmetische Operation g (Matrix-Multiplikation, -Inversion, LU-Zerlegung) ersetzt wird durch einen der Netzwerkalgorithmen 6.3.2.-6.3.4. zur Berechnung von g. Da diese drei Algorithmen dieselbe Ein- und Ausgabeverteilung besitzen, sind die Voraussetzungen von Satz 4.2.5. erfüllt, und die Kommunikationszeit von $D'_{LU}(n,m)$ berechnet sich zu

$$\begin{aligned}\tau'(n,m) &= t_{0,1}(n/m)\tau_1(m) + t_{0,4}(n/m)\tau_4(m) + t_{0,6}(n/m)\tau_6(m)\\ &\approx (n^3/(3m^3) + n^2/m^2)(5m - 5) + (2n/m - 2)(3m - 4) + (n/m)(3m - 3)\\ &= (5/3)(n^3/m^2) + 5n^2/m + O(n - (n/m)^3).\end{aligned}$$

6.3.7. Partitionierte LU-Zerlegung mit Kommunikationsgraph G_3 (dreidimensionales Gitter)

Eingabe: Für $1 \le i,j \le m$, $1 \le k \le n/m$ erhält $P(i,j,k)$ als Eingabe die n/m Koeffizienten $a_{rm+i,(k-1)m+j+1}$, $0 \le r \le n/m-1$, von A.
Ausgabe: Am Ende der Rechnung enthält $P(i,j,k)$ die Koeffizienten $l_{rm+i,(k-1)m+j+1}$, $u_{rm+i,(k-1)m+j+1}$, $0 \le r \le n/m-1$, von L und U.

Im folgenden Bild sind für eine (15×15)-Matrix mit Blockgröße $m = 3$ die Koeffizienten gekennzeichnet, die $P(2,1,4)$ als Ein- bzw. Ausgabe erhält:



$D''_{LU}(n,m)$ erhält man aus Algorithmus 6.3.1., indem ein Befehl

$$P(k): \text{(arithmetische Operation } g)$$

ersetzt wird durch einen der Netzwerkalgorithmen 6.3.2.-6.3.4. zur Berechnung von g ausgeführt durch den Subgraphen

$$[\{P(i,j,k) \mid 1 \le i,j \le m\}] \cong G_2$$

von G_3.

Für $\bar{a} = (a_{ij})_{1 \le i,j \le m} \in K^m \times K^m$ wird ein Kommunikationsbefehl

$$P(k) \to P(k+1): \bar{a}$$

von $D''_{LU}(n,m)$ realisiert durch

for all $i,j = 1,\ldots,m$ **pardo**

$P(i,j,k) \to P(i,j,k+1): a_{ij}$;

Diese Datenübertragung kann für jeden Kommunikationsbefehl von $\overline{D}_{LU}(n/m)$ unmittelbar begonnen werden, da die Ein- und Ausgabeverteilungen der in-

ternen Algorithmen identisch sind (vgl. 4.2., S. 80). Damit berechnet sich die Kommunikationszeit von $D''_{LU}(n,m)$ als Summe der internen und externen Kommunikationszeiten wie folgt (vgl. 4.2.3., 4.2.5.):

$$\tau''(n,m) = t_{0,1}(n/m)\tau_1(m) + t_{0,4}(n/m)\tau_4(m) + t_{0,6}(n/m)\tau_6(m) + \tau_0(n/m)$$

$$= (5/3)(n^3/m^2) + 5n^2/m + n^2 - nm + O(n - (n/m)^3).$$

6.4. KOMMUNIKATIONSZEITEN

Setzt man im Algorithmus 6.3.1. die Blockgröße $m = 1$, so erhält man einen (nicht-partitionierten) Netzwerkalgorithmus $D_{LU}(n)$ zur Berechnung der LU-Zerlegung einer $(n \times n)$-Matrix auf einem linearen Prozessornetz $G_1(n)$ mit $p(n) = n$ Prozessoren. Die Algorithmen 6.3.5. und 6.3.7. stellen Partitionierungen dieses Algorithmus im Sinne von Definition 4.1.5. dar. Die Graphen $G(n)$ der Klasse Γ sind in diesem Fall die linearen Netze mit n Knoten.

Im folgenden Abschnitt wird an diesem Beispiel der in Kapitel 5 durchgeführte allgemeine Vergleich der Kommunikationszeiten von partitionierten und nicht-partitionierten Netzwerkalgorithmen für Kommunikationsgraphen mit p Prozessoren verdeutlicht.

Es sei $p < n$, $p|n$. Die nicht-partitionierte Ausführung des Algorithmus erfolge auf einem linearen Netz G mit p Prozessoren, welches vermöge einer Netzwerkemulation ([4], 3.4.) aus $G_1(n)$ gewonnen wird. Für die partitionierten Netzwerkalgorithmen werden die in 6.3.5. und 6.3.7. angegebenen Kommunikationsgraphen zugrundegelegt. Die Blockgröße m_p sei dabei so gewählt, daß $p(n,m_p) = p$.

Ist $V(G) = \{P'(1),\ldots,P'(p)\}$ und $V(G_1(n)) = \{P(1),\ldots,P(n)\}$, so ist die Netzwerkemulation gegeben durch

$$\phi: V(G_1(n)) \to V(G)$$
$$P(i) \to P'(\lceil ip/n \rceil),\ 1 \le i \le n.$$

Diese Emulation ist offensichtlich uniform, und es ist $|\phi'(e)| = 1$, $e \in E(G)$, d.h. der in Kapitel 5 bezeichnete allgemeine Wert β wird in diesem Fall zu 0.
Da die Kommunikationszeiten der in 6.3. beschriebenen Algorithmen von der gleichen Größenordnung sind wie die Berechnungszeiten, sind insgesamt die Voraussetzungen von Korollar 5.0.7. erfüllt. Demnach ist zu erwarten, daß die Kommunikationszeit $\tau''(n,m_p)$ des partitionierten Algorithmus im Falle 6.3.7. um den Faktor n/p größer ist als die Kommunikationszeit τ des nicht-partitionierten Algorithmus ausgeführt auf G. Im Falle 6.3.5. (sequentielle Ausführung der internen Algorithmen) besagt das Korollar immer noch $\tau(n,m_p) = \Omega(n/(pm_p)\tau)$, d.h. die Kommunikationszeit des partitionierten Algorithmus ist durch die Kommunikationszeit des nicht-partitionierten Algorithmus nach unten beschränkt. (Wir haben in diesem Fall $m_p = n/p$.)
Es ist anzumerken, daß der Kommunikationsgraph des Algorithmus 6.3.5. gerade G ist.

Wir überprüfen die Voraussagen des Korollars 5.0.7. in unserem Beispiel. Zunächst ist die Kommunikationszeit des Algorithmus $D_{LU}(n)$ (6.3.1.) mit

$m = 1$ und Kommunikationsgraph $G_1(n)$ gegeben durch $\tau(n) = n^2 - n$. Durch die Abbildung

$$\phi': E(G_1(n)) \to V(G) \quad E(G)$$

werden nun lediglich die Kanten $(P(in/p),P(in/p+1))$, $1 \le i < p$, aus $E(G_1(n))$ auf Kanten von G abgebildet; alle anderen Kanten von $G_1(n)$ werden auf Knoten von G abgebildet. D.h. die Kommunikationszeit des Algorithmus $D_{LU}(n)$ ausgeführt auf dem linearen Netz mit p Prozessoren ist gegeben durch die Anzahl der (parallelen) Kommunikationsschritte von $D_{LU}(n)$ ausgeführt auf $G_1(n)$, die mindestens eine der Kanten $(P(in/p),P(in/p+1))$, $1 \le i < p$, verwenden.
Der erste Kommunikationsbefehl in Zeile 10 des Algorithmus 6.3.1. benötigt nun genau dann eine dieser Kanten, wenn $i \ge p$. Im Falle des Kommunikationsbefehls in Zeile 18 wird eine Kante von G verwendet, wenn $n - p \ge k + i$. Somit berechnet sich τ zu

$$\tau = 2\sum_{k=1}^{n-1}(n - p - k) + (n-1) = n^2 - 2p(n-1) - 1, \text{ also } \tau \approx n^2 - 2pn.$$

Im Falle des Algorithmus 6.3.5. ist wegen $p_0(n/m_p) = n/m_p = p$

$$\tau(n,m_p) = n^2 - n^2/p.$$

Es ist also in der Tat $\tau(n,m_p) = \Omega(\tau)$ und für $p > (n/2)^{1/2}$ ist $\tau < \tau(n,m_p)$. Dieses Beispiel verdeutlicht aber auch, daß bei sequentieller Ausführung der internen Algorithmen die Kommunikationszeit des partitionierten Netzwerkalgorithmus sich keineswegs signifikant von der Kommunikationszeit der nicht-partitionierten Version zu unterscheiden braucht.

Bei paralleler Ausführung der internen Algorithmen (6.3.7.) erhalten wir jedoch für die Kommunikationszeit unter Verwendung von $p(n,m_p) = nm_p = p$ die Abschätzung

$$\tau''(n,m_p) > (5/3)n^5/p^2 \geq (5/3)n^3.$$

Dieser Wert liegt offensichtlich deutlich über den Kommunikationskosten τ. Die allgemeine Aussage aus Kapitel 5 beinhaltete

$$\tau''(n,m_p) = \Omega((n/p)\ \tau) = \Omega(n^3/p).$$

Da p für diese gemischte Partitionierung zwischen n und n^2 liegt, $n \leq p \leq n^2$, liegt der theoretisch bestimmte Schrankenwert für große Prozessoranzahlen nahe am real bestimmten Wert des Beispiels; für $p = n^2$ ist $n^5/p^2 = n^3/p$. Ohne weitere Einschränkungen für p ist im allgemeinen deshalb keine bessere Schranke anzugeben.

7. ZUSAMMENFASSUNG UND AUSBLICK

Im abschließenden Kapitel steht neben einer Zusammenfassung der Ergebnisse dieser Arbeit ein Ausblick auf weiterführende Fragestellungen, sowie auf Anwendungsmöglichkeiten der in den vorangegangenen Kapiteln hergeleiteten Aussagen im Vordergrund. Dabei beschränken wir uns auf einen Rahmen, in dem lediglich eine qualitative Wertung der Ergebnisse erfolgt, ohne hierfür die zuvor entwickelten Details und Formalismen nochmals heranzuziehen.

Den Gegenstand dieser Arbeit bilden parallele partitionierte Algorithmen und insbesondere partitionierte Netzwerkalgorithmen. Dabei wird unter einem partitionierten Algorithmus ein Algorithmus verstanden, der auf partitionierten Datenmengen operiert. Da partitionierte Algorithmen bisher nur in speziellen Zusammenhängen Berücksichtigung finden, fehlt es in der Literatur an einer systematischen Behandlung von Algorithmen- und Netzwerkpartitionierungen. Daraus resultiert eine unzureichende allgemeine Vorstellung über die verschiedenen Bereiche, in denen partitionierte Algorithmen sinnvoll oder aber auch weniger sinnvoll einzusetzen sind. Unsere Arbeit stellt einen Beitrag dazu dar, diese Lücke zu schließen.

Dabei liefern die von uns im ersten Kapitel hergeleiteten Grundlagen und Definitionen die Basis für eine einheitliche und geschlossene Behandlung des Komplexes paralleler partitionierter Algorithmen und ihrer verschiedenartigen Anwendungsbereiche.
Naturgemäß erfordert eine exakte Erfassung der wesentlichen Charakteristika partitionierter Algorithmen zunächst ihre Behandlung in einem Kontext, der weitestgehend unabhängig von solchen Restriktionen ist, die durch konkrete Parallelprozessoren vorgegeben werden. Dies findet in der vorliegenden Arbeit seinen Niederschlag in der zweigeteilten Struktur: im ersten Teil (Kapitel 1 und 2) beschäftigen wir uns ausschließlich auf der Grundlage einer PRAM über beliebigen Ringen R mit partitionierten Algorithmen. Die Laufzeit- und Effizienzuntersuchungen in Kapitel 2 sind hierbei als notwendiger Bestandteil einer systematischen Beschreibung und

Bewertung partitionierter Algorithmen, insbesondere im Vergleich zu nicht-partitionierten Algorithmen zu sehen.
Der zweite Teil der Arbeit (Kapitel 3, 4 und 5) befaßt sich mit partitionierten Algorithmen für Multiprozessoranordnungen mit lokalen Speichern und (statischen) Prozessorverbindungsnetzwerken. Hierzu werden zunächst algorithmenunabhängige Netzwerkpartitionierungen analysiert (Kapitel 3), um dann in Kapitel 4 durch die Herleitung von Realisierungen partitionierter Algorithmen auf partitionierten Prozessorverbindungsnetzwerken den Zusammenhang zum ersten Teil der Arbeit herzustellen.

In der gesamten Arbeit wird als Leitlinie für einen möglichen Anwendungsbereich partitionierter Algorithmen eine Beschränkung der Anzahl verfügbarer parallel arbeitender Prozessoren gewählt. Unter dieser Maßgabe untersuchen wir in Kapitel 2 und in Kapitel 5 die Berechnungs- und Kommunikationskomplexität paralleler partitionierter Algorithmen im Vergleich zu ihrer jeweiligen nicht-partitionierten Version. Dabei ist insbesondere auf das in Kapitel 5 bewiesene interessante Ergebnis hinzuweisen, welches unter bestimmten Voraussetzungen einen nicht-konstanten Zeitverlustfaktor der Kommunikationszeit eines partitionierten Netzwerkalgorithmus gegenüber einem vergleichbaren nicht-partitionierten Netzwerkalgorithmus nachweist.

Die in der Arbeit erfolgten Komplexitätsuntersuchungen haben insgesamt ergeben, daß für partitionierte Algorithmen im allgemeinen eine größere Ausführungszeit zu erwarten ist als für nicht-partitionierte Algorithmen. Dies ist zurückzuführen auf die komplexeren Struktureigenschaften partitionierter Algorithmen. Hieraus erklärt sich auch das Ergebnis aus 2.4., wonach es unter dem Aspekt der Ausführungszeit nicht sinnvoll ist, die partitionierte Struktur des Algorithmus auf den Parallelrechner abzubilden. Gleichzeitig verweisen aber gerade die Überlegungen in 2.4. auf Fälle, in denen es geboten ist, partitionierte Algorithmen zu verwenden, da es nicht immer möglich ist, den externen oder die internen Algorithmen sequentiell auszuführen.

Eine detaillierte Untersuchung der skizzierten sowie aller weitergehenden Fragestellungen erhält ihre Rechtfertigung daraus, daß es zur Lösung

vieler Probleme auch wirklich partitionierte Algorithmen gibt. Wir ordnen deshalb an dieser Stelle die Beispielalgorithmen der Arbeit in die Klasse aller partitionierbaren Algorithmen ein.
In 1.2. sind verschiedenartige partitionierte Algorithmen zusammengestellt worden. Es handelt sich hierbei um mehrere Algorithmen aus dem Bereich der Linearen Algebra (Lösung linearer Gleichungssysteme, Matrix-Operationen und -Transformationen), deren nicht-partitionierte Versionen lediglich die arithmetischen Grundoperationen +,-,*,/ verwenden, sowie um Sortieralgorithmen, deren nicht-partitionierte Versionen lediglich eine Vergleichoperation verwenden. Als typisches Beispiel für einen Algorithmus, der aus einem Rekursionsverfahren gewonnen wird, ist neben dem in 1.3. beschriebenen Matrix-Inversionsalgorithmus die Partitionierung der Fast-Fourier-Transformation anzusehen. Obwohl durch diese Beispiele bereits viele Probleme erfaßt werden, gehen wir davon aus, daß die Klasse partitionierbarer Algorithmen noch erheblich umfassender ist. Grundlage für diese Annahme ist insbesondere das den Beispielen der Linearen Algebra zugrundeliegende gemeinsame Vorgehen. Dieses besteht in der Verwendung eines herkömmlichen Algorithmus M als Algorithmus über dem Matrix-Ring $K^m \times K^m$ unter Hinzuziehung einer zusätzlichen "arithmetischen" Operation, nämlich der Ausführung von M auf $(m \times m)$-Blockmatrizen. Man überzeugt sich im Einzelfall leicht davon, daß viele bekannte Algorithmen auf diese Weise zu partitionieren sind. Neben den in der Arbeit ausgeführten Beispielen sollen hier noch der Chen-Kuck-Algorithmus zur Lösung von Dreieckssystemen ([10]), sowie die Berechnung der Cholesky-Zerlegung einer positiv definiten symmetrischen Matrix ([24]) als partitionierbare Algorithmen angeführt werden. (Im letzten Fall ersetze man die Quadratwurzel-Operation durch die interne Operation "Cholesky-Zerlegung einer $(m \times m)$-Blockmatrix".) Es ist zu vermuten, daß mit Methoden der algebraischen Komplexitätstheorie (Beschreibung von Algorithmen durch Polynome bzw. rationale Funktionen) ein allgemeiner Nachweis der Partitionierbarkeit dieser und ähnlicher Algorithmen zu führen ist.

Eine offenkundige Schwierigkeit besteht in der Verallgemeinerung der Division: generell ist nicht von der Invertierbarkeit der Blockmatrizen auszugehen, so daß ein Algorithmus zwar partitionierbar sein kann, aber nicht für alle Eingaben auch wirklich eine Lösung liefert. Wann immer es

möglich ist, sollten deshalb divisionsfreie Algorithmen die Grundlage für Partitionierungen bilden. Als Beispiel für einen divisionsfreien partitionierten Algorithmus (d.h. der externe Algorithmus benötigt keine Inversionen) ist neben der Matrix-Multiplikation die partitionierte Rücksubstitution 1.2.9. anzusehen.

Die eingangs genannte Beschränkung der Prozessoranzahl verweist sicherlich nur auf eine mögliche Anwendung partitionierter Algorithmen. Wesentlich erscheint uns darüberhinaus die Möglichkeit, vermöge der Abbildung partitionierter Algorithmen auf partitionierte Prozessornetzwerke Algorithmen für mehrdimensionale Netzwerke zu entwickeln. Das hierfür erforderliche Vorgehen wird an Beispielen in 4.1. und in Kapitel 6 beschrieben. Insgesamt liefert die Arbeit Grundlagen für den systematischen Entwurf mehrdimensionaler Netzwerkalgorithmen. Unter Verwendung der in Kapitel 1 erläuterten Strukturierung eines partitionierten Algorithmus durch externen und interne Algorithmen, sowie des in Kapitel 3 entwickelten Hypernetzkonzeptes wird in Kapitel 4 das notwendige Vorgehen für dreidimensionale Netze erörtert: um einen Algorithmus zur Berechnung einer Funktion f für ein im intuitiven Sinne dreidimensionales Netzwerk (also z.B. dreidimensionale Gitter, geschachtelte Perfect-Shuffle-Netze, linear verbundene Bäume, baumartig verbundene Gitter,...,vgl. 3.1.) zu gewinnen, verwende man einen partitionierten Algorithmus zur Berechnung von f und bilde den zugehörigen externen Algorithmus auf das Hypernetz sowie die internen Algorithmen auf die korrespondierenden (isomorphen) Subgraphen des Netzwerks ab.
Obwohl wir auf eine bis ins einzelne gehende Ausführung verzichtet haben, wird in der Arbeit der Weg zur Verallgemeinerung dieses Vorgehens für d-dimensionale Netzwerkalgorithmen gewiesen. In diesem Fall wird man die Operationen partitionierter Algorithmen der Tiefe d (vgl. 1.3.) auf die Knoten d-dimensionaler Kommunikationsgraphen (vgl. 3.3.) abbilden.

In diesem Zusammenhang soll daran erinnert werden, daß partitionierte Algorithmen der Tiefe d den Berührungspunkt zu rekursiven Algorithmen darstellen. Neben der naheliegenden Tatsache, daß mehrfach partitionierbare Algorithmen spezifische Rekursionsverfahren darstellen, haben wir in 1.3. gezeigt, wie aus geeigneten rekursiven Algorithmen partitionierte Algorithmen gewonnen werden können.

Abschließend ist festzustellen: eine sinnvolle Anwendung partitionierter Algorithmen sollte sich neben der Möglichkeit der Beschränkung der Prozessorzahl die besonderen Struktureigenschaften, welche zunächst eine größere Zeitkomplexität bewirken, unter weitergehenden Gesichtspunkten zunutze machen. Dies kann konkret bedeuten

- partitionierte Algorithmen zu verwenden für Parallelrechner, die aus untereinander verbundenen VLSI-Chips aufgebaut sind: die einzelnen Chips können in diesem Fall für die Berechnung der internen Blockoperationen genutzt werden. Die theoretischen Überlegungen in 1.1. weisen darauf hin, daß die Anzahl unterschiedlicher Chips für eine große Klasse verschiedener Algorithmen eng begrenzt ist.

- partitionierte Algorithmen zu verwenden, um einfache Speicherhierarchien günstig zu realisieren: da für die Ausführung der internen Algorithmen im Verlauf des Gesamtalgorithmus jeweils nur die Daten der (kleinen) Blockmatrizen benötigt werden, braucht das *cache* der Prozessoren im Verhältnis zur Gesamtproblemgröße nur wenig Daten speichern zu können.

- partitionierte Algorithmen zu verwenden, um *systolic arrays* variabler Größe g(n) durch ein *systolic array* beschränkter Größe einfach zu simulieren: die Partitionierung ermöglicht es in diesem Fall, daß in jedem Knoten des *array* unabhängig von der Problemgröße n immer dieselbe Operation ausgeführt wird, so daß der in einem Knoten berechnete Wert also jeweils nur von den konkreten Daten abhängt.

Diese Aufzählung erhebt keinen Anspruch auf Vollständigkeit, sondern soll lediglich einen Hinweis auf die Verschiedenartigkeit der möglichen Aspekte einer weitergehenden Untersuchung paralleler partitionierter Algorithmen geben.

BEZEICHNUNGEN

A, B Mengen, f, g Funktionen.

$A \dot{\cup} B$: disjunkte Vereinigung von A und B.

$f = O(g)$:<=> es gibt eine Konstante c mit $f \leq cg$.

$f = \Omega(g)$:<=> $f = O(g)$.

$f = \theta(g)$:<=> $f = O(g)$ und $f = \Omega(g)$.

LITERATUR

[1] Berge,C., Graphs and hypergraphs, Elsevier-North Holland, New York (1973)

[2] Bitton,D., DeWitt,D.J., Hsaio,D.K., Menon,J., A taxonomy of parallel sorting, ACM Comput.Surv. 16,3, 287-318 (1984)

[3] Dekel,E., Nassimi,D., Sahni,S., Parallel matrix and graph algorithms, SIAM J. Comput. 10,4, 657-675 (1981)

[4] Fishburn,J.P., Finkel,R.A., Quotient networks, IEEE Trans. Comp. C-31,4, 288-295 (1982)

[5] Fortune,S., Wyllie,J., Parallelism in random access machines, ACM STOC 10, 114-118 (1978)

[6] Gannon,D.B., v.Rosendale,J., On the impact of communication complexity on the design of parallel numerical algorithms, IEEE Trans. Comp. C-33,12, 1180-1194 (1984)

[7] Gentleman,W.M., Some complexity results for matrix computations on parallel processors, J.ACM 25,1, 112-115 (1978)

[8] Gottlieb,A., Schwartz,J.T., Networks and algorithms for very large scale parallel computation, Computer 1/82, 27-36 (1982)

[9] Haynes,L.S., Lau,R.L. Siewiorek,D.P., Mizell,D.W., A survey of highly parallel computing, Computer 1/82, 9-24 (1982)

[10] Heller,D., A survey of parallel algorithms in numerical linear algebra, SIAM Rev. 20,4, 740-777 (1978)

[11] Hoßfeld,F., Parallele Algorithmen, Informatik Fachberichte, Springer Verlag Berlin-Heidelberg-New York (1983)

[12] Huang,M.-D.A., Solving some graph problems with optimal or near-optimal speedup on mesh-of-tree networks, IEEE FOCS 26, 232-240 (1985)

[13] Hwang,K., Briggs,F.A., Computer architecture and parallel processing, Mc-Graw-Hill Series in Computer Organization and Architecture, New York (1984)

[14] Hwang,K., Cheng,Y.-H., Partitioned matrix algorithms for VLSI arithmetic systems, IEEE Trans. Comp. C-31,12 (Dec. 1982), 1215-1224

[15] Kruskal,C.P., Rudolph,L., Snir,M., The power of parallel prefix,

IEEE Trans. Comp. C-34,10, 965-968 (1985)

[16] Kung,H.T., Highly parallel systolic algorithms and their implementation, 5th Aachen Symp. on Mathematical Methods in Signal Processing, Aachen, W-Germany (1984)

[17] Meyer auf der Heide,F., Lower time bounds for testing the solvability of diophantine equations on several parallel computational models, Univ. Frankfurt, FB Informatik, Interner Bericht 7/84 (1984)

[18] Moldovan,D.I., Fortes,J.A.B., Partitioning and mapping algorithms into fixed size systolic arrays, IEEE Trans. Comp. C-35,1, 1-12 (1986)

[19] Nussbaumer,H.J., Fast Fourier Transform and convolution algorithms, Springer Series in Information Sciences, Springer Verlag Berlin-Heidelberg-New York (1981)

[20] Preparata,F.P., Vuillemin,J., The cube-connected cycles: A versatile network for parallel computation, Comm. ACM 24,5, 300-309 (1981)

[21] Quinn,M.J., Deo,N., Parallel graph algorithms, ACM Comput. Surv. 16,3, 319-348 (1984)

[22] Seban,R.P., Siegel,H.J., Shuffling with the Illiac and PM2I SIMD networks, IEEE Trans. Comp. C-33,7, 619-625 (1984)

[23] Snyder,L., Introduction to the configurable highly parallel computer, Computer 1/82, 47-56 (1982)

[24] Stewart,G.W., O'Leary,D.P., Data-flow algorithms for parallel matrix computations, Comm. ACM, 28,8, 840-853 (1985)