Numerische Algorithmen für parallele Rechnerarchitekturen

Notizen zur Vorlesung WS 01/02

Version 1.02

Überarbeitete Fassung der Notizen zur Vorlesung

im WS 98/99 von P. Buchholz

P. Kemper

24. Januar 2002

Vorwort

Der vorliegende Text beinhaltet Notizen zur Vorlesung "Numerische Algorithmen für parallele Rechnerarchitekturen". Dieser Text stammt im wesentlichen von P. Buchholz und basiert auf Notizen zu einer gleichnamigen Vorlesung, die im WS 1998/99 als 2-stündige Spezialvorlesung von P. Buchholz gehalten wurde. Trotz einer ersten Überarbeitung des Textes steht zu befürchten, daß er noch eine Reihe von Fehlern beinhaltet. Ich bitte dafür schon jetzt um Nachsicht und bin für jegliche Hinweise zur Verbesserung sehr dankbar.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Parallelverarbeitung ist ein relativ neues und schnell wachsendes Gebiet in der Informatik. Mit der Einführung massiv paralleler Rechnerarchitekturen, die mehrere tausend Prozessoren umfassen, werden sehr große Hoffnungen verknüpft. Ob diese Hoffnungen erfüllt werden können, ist heute noch völlig offen. Erfahrungen der letzten Jahre zeigen, daß einzelne Bereiche existieren, in denen sehr effiziente parallele Algorithmen recht einfach entwickelt werden konnten. Auf anderen Gebieten dagegen sind die Fortschritte eher gering.

Praktische Bedeutung erlangte die Parallelverarbeitung erst mit der Verfügbarkeit kostengünstiger Parallelrechner ab Mitte der 80er Jahre. Inzwischen gibt es eine Reihe preisgünstiger Multiprozessorrechner im Workstationbereich, die zwar noch nicht tausende, sondern eher 2 bis 8 Prozessoren beinhalten, eine Vergrößerung der Prozessorzahl ist aber in den nächsten Jahren zu erwarten. Aktuelle parallele Hochleistungsrechner mit 64-256 Prozessoren, sind inzwischen schon relativ weit verbreitet. Des weiteren sind Netze von Workstations oder PC Netze die Standardausstattung jeder Universität. Solche Netze können, bei Bereitstellung einer entsprechenden Programmierumgebung, natürlich auch für die Implementierung paralleler Algorithmen genutzt werden. Die Bereitstellung einer großen Anzahl von Prozessoren ist ein Punkt, ein anderer ist die effiziente Nutzung der angebotenen Rechenkapazität und dies erfordert bei Parallelrechnern und auch auf Rechnernetzen leider oftmals neue oder modifizierte Algorithmen und Konzepte.

Der Hauptgrund für die Einführung von parallelen Rechnerarchitekturen ist natürlich der Wunsch nach mehr Rechenleistung oder anders ausgedrückt, Anwendungen, die mit Standardrechnern nicht genügend schnell ausgeführt werden können. Es gibt zwei verschiedene Ansätze zur Leistungssteigerung von Rechnern. Ein Ansatz ist die schnellere Ausführung von Instruktionen durch die Verwendung von schnellerer Hardware. Hier sind allerdings Grenzen gesetzt, da technologische Obergrenzen für die erreichbare Rechenleistung bestehen und Rechner, die nahe an dieser Obergrenze liegen unverhältnismäßig teuer sind. Daher ist es naheliegend, Instruktionen parallel auszuführen. Die parallele Ausführung von Instruktionen führt noch nicht automatisch zu Parallelrechnern, viel elementarer und auch üblicher ist die Überlappung von Ein-/Ausgabe und Berechnung durch die Verwendung von Kanälen, die Einführung von Speicherhierarchien und Cachespeichern und die Verwendung von Pipeline- oder Vektorrechnern. Alle diese Ansätze der Parallelisierung erfordern keine neuen Algorithmen oder Programmiersprachen (sieht man einmal von Vektorrechnern ab) und sind auf Benutzerebene nicht oder kaum sichtbar. Allerdings sind die Möglichkeiten der Parallelisierung und damit auch Leistungssteigerung auf natürliche Weise beschränkt und in vielen heutigen Systemen ausgereizt (zumindest was E/A-Kanäle und Cache Speicher angeht).

Ein weiterer Schritt in Richtung paralleler Ausführung von Instruktionen ist die Verwendung von Parallelrechnern mit mehreren Prozessoren. Leider führt dies dazu, daß bestehende

Algorithmen und Programme nicht mehr so einfach von seriellen Rechnern übertragen werden können, womit über neue Algorithmen und Programmiersprachen nachgedacht werden muß. Die Verwendung mehrerer Prozessoren macht außerdem die Verwendung einer adäquaten Kommunikationsstruktur zwischen den Prozessoren und die Verfügbarkeit von Speichern mit schnellen Zugriffszeiten notwendig. Wir werden auf diese Problemkreise und Lösungsmöglichkeiten kurz im folgenden Kapitel zurückkommen. Die Parallelisierung durch die Verwendung von mehreren Prozessoren hat gegenüber den obigen Methoden der gleichzeitigen oder überlappenden Instruktionsausführung den Vorteil, daß sie theoretisch beliebig weit getrieben werden kann. Es gibt gleichwohl eine Reihe durchaus ernst zu nehmender Argumente gegen zuviel Euphorie beim Einsatz von Parallelrechnern. Ein wichtiges Argument ist Amdahls Gesetz, welches aussagt, daß die erreichbare Leistungssteigerung durch den sequentiellen Anteil eines Programms beschränkt ist. Formaler kann man das so darstellen: Wenn S_p die Leistungssteigerung durch die Verwendung von p Prozessoren ist und f der Anteil des Programms ist, der sequentiell ausgeführt werden muß, so gilt

$$S_p \leq \frac{1}{f + (1-f)/p} \leq \frac{1}{f}$$

Dadurch ist die Leistungssteigerung durch den Faktor $1/f$ nach oben beschränkt, dieser kann auch durch den Einsatz weiterer Prozessoren nicht überschritten werden. Müssen z.B. 20% eines Programms sequentiell ausgeführt werden, so ist eine maximale Leistungssteigerung um den Faktor 5 möglich. Programme, die einen hohen sequentiellen Anteil beinhalten, sind damit mit parallelen Rechnern nur geringfügig schneller abzuarbeiten und rechtfertigen insbesondere nicht den Einsatz massiv paralleler Architekturen. Es gibt aber zahlreiche Algorithmen, die fast vollkommen ohne sequentiellen Anteil auskommen und damit potentiell beliebig beschleunigt werden können, wenn nur genügend Prozessoren zur Verfügung stehen.

Kommen wir nun zu einigen einführenden Begriffsklärungen und Einordnungen des behandelten Themengebiets. Unter einer parallelen Architektur wollen wir im weiteren eine Menge von Prozessoren verstehen, die kommunizieren und kooperieren, um gemeinsam ein Problem zu lösen. Unter einer massiv- oder hoch-parallelen Architektur wird eine Architektur mit sehr vielen (i.a. mehreren tausend) Prozessoren verstanden. Diese sehr allgemeine Definition wird im Rahmen von Kapitel 2 weiter unterteilt und erläutert.

Der Bereich der parallelen Systeme umfaßt verschiedene Teilgebiete, von denen jedes mehr als eine Vorlesung füllen würde. Als Beispiele seien hier genannt, Aufbau, Entwurf und Bewertung paralleler Rechnerarchitekturen, Betriebssysteme für parallele Rechner, Programmiersprachen für parallele Rechner, Parallelisierung bestehender Software, Entwurf und Analyse paralleler Algorithmen und vieles andere mehr. Im Rahmen dieser Vorlesung werden wir uns mit Algorithmen beschäftigen und auch dort nur mit einem sehr speziellen Teilbereich. Wie schon angedeutet, gibt es eine Reihe von Bereichen, in denen parallele Algorithmen erfolgreich entwickelt wurden. Kandidaten sind insbesondere solche Gebiete, die einerseits eine sehr hohe Rechenleistung erfordern, die von konventionellen Rechner nicht erbracht wird und die ein genügendes Maß an inhärenter Parallelität beinhalten, damit die parallele Rechnerarchitektur ausgenutzt werden kann. Beispiele für Gebiete in denen parallele Rechner in größerem Stil eingesetzt wurden und in denen ein verstärkter Einsatz in Zukunft zu erwarten ist, sind die Wettervorhersage, rechnergestützte Aerodynamikberechnungen, Simulationen, rechnergestütztes VLSI Design, künstliche Intelligenz, Datenbanksysteme, Optimierungsprobleme. Wie man aus dieser sicherlich nicht erschöpfenden Aufzählung sieht, liegt der Schwerpunkt des Einsatzes paralleler Architekturen im Bereich technischer Anwendungen, die i.a. sehr große numerische Probleme beinhalten. Die Vorlesung beschäftigt sich mit numerischen Algorithmen und dort speziell mit Algorithmen zur Lösung von Gleichungssystemen und für Optimierungsprobleme.

Diese Algorithmen haben eine große praktische Relevanz und sind leicht parallelisierbar. Wir werden uns also insbesondere mit Anwendungen aus dem Bereich des "wissenschaftlichen Rechnens" beschäftigen. Wissenschaftliches Rechnen ist eine etwas hölzerne Übersetzung des Begriffs "Scientific Computing". Wir wollen deshalb versuchen, den Begriff "Scientific Computing" etwas genauer einzugrenzen.

Ziel der unter dem Begriff "Scientific Computing" zusammengefaßten Techniken ist die Nachbildung und Analyse komplexer natürlicher und technischer Probleme auf einem Rechner und ihre Analyse mittels numerischer Methoden. Scientific Computing ist damit interdisziplinär und umfaßt Methoden aus der Informatik, der Mathematik und dem jeweiligen Anwendungsgebiet. Im Rahmen dieser Vorlesung werden wir uns primär mit den zur Informatik gehörigen Aspekten beschäftigen. Es läßt sich aber nicht vermeiden, auch einige mathematische Grundlagen zu behandeln und einen kurzen Blick in Anwendungsgebiete zu tun. Typische Anwendungsgebiete des Scientific Computing sind z.B. die Klimamodellierung und die Analyse des menschlichen Genoms. Bei diesen beiden Beispielen und auch auch bei vielen weiteren ähnlichen Beispielen sind die auftretenden Probleme von einer immensen Komplexität, die heute nur ansatzweise beherrschbar ist. Damit ergibt sich natürlich direkt ein potentielles Anwendungsgebiet für Parallelrechner.

Der Entwurf paralleler Algorithmen kann auf verschiedenen Ebenen erfolgen. Typischerweise unterscheidet man

- Die Parallelisierung sequentieller Algorithmen, ohne Bezug auf eine spezielle Rechnerarchitektur.
- Den Entwurf neuer und einfach zu parallelisierender Algorithmen, unabhängig von der parallelen Rechnerarchitektur.
- Den Entwurf effizienter paralleler Algorithmen für einen speziellen Typ von Parallelrechner (insbesondere für eine spezielle Prozessortopologie).
- Den Entwurf eines parallelen Algorithmus und einer ganz speziellen Architektur für diesen Algorithmus.

Im Rahmen dieser Vorlesung werden uns die ersten drei Arten von Algorithmen interessieren. Ein Schwerpunkt liegt dabei auf Algorithmen, die sich zwischen den Punkten 1 und 2 bewegen, d.h. wir untersuchen die Parallelisierung vorhandener, effizienter numerischer Algorithmen, die u.U. zur besseren Parallelisierung leicht modifiziert werden. Die Realisierung der Algorithmen wird Prozeduren zur Kommunikation und für numerische Basisoperationen verwenden, die für spezifische Architekturen optimiert wurden. Diese Vorgehen hat den Vorteil, daß die Algorithmen nicht direkt von einer speziellen Rechnerarchitektur abhängen und daß bestehende Software automatisch oder semi-automatisch auf parallele Rechner übertragen werden kann. Der letzte Punkt ist vom praktischen Standpunkt her von sehr großer Bedeutung, da die Neuentwicklung von Software zeitaufwendig und teuer ist. Gleichwohl ist es sinnvoll, auch Algorithmen, die speziell für spezifische Prozessortopologien entwickelt wurden, zu betrachten. An einigen Stellen werden deshalb "optimale" Algorithmen für weitverbreitete Typen von Topologien (z.B. Netze, Hyperwürfel) vorgestellt. Algorithmen dieser Art zeigen, welche Leistung auf parallelen Rechnern eines bestimmten Typs prinzipiell erreichbar ist und wie weit der parallelisierte sequentielle Algorithmus von diesem Optimum entfernt ist.

Es wird wahrscheinlich überraschen, daß Algorithmen für spezielle Architekturen entworfen werden. Es gibt aber bisher noch kein allgemein einsetzbares praktisches Konzept zum automatischen Entwurf paralleler Programme unabhängig von der verwendeten parallelen Architektur. Dies zeigt auch, daß der praktische Einsatz der Parallelverarbeitung erst am Anfang steht.

Die Vorlesung gliedert sich in acht Teile. Im ersten Teil (Kapitel 2) werden wir uns relativ allgemein mit parallelen Architekturen und ihrer Beschreibung beschäftigen. Auch wenn die Vorlesung sich primär mit Algorithmen beschäftigt, können diese, wie schon erwähnt, nicht völlig losgelöst von der Architektur des parallelen Rechners untersucht werden. Wir werden deshalb konkrete Architekturen kurz kennenlernen und ein etwas abstrakteres Modell einführen, welches die Beschreibung der Kommunikationsstruktur erlaubt. Auf Basis dieses Architekturmodells werden später Basisalgorithmen entwickelt. Nachdem parallele Architekturen eingeführt wurden, wollen wir ein Algorithmenmodell entwerfen. Ziel ist es, den Ablauf eines Algorithmus auf einer abstrakten Ebene so zu beschreiben, daß der sequentielle und potentiell parallele Anteil leicht sichtbar sind. Des weiteren werden Bewertungsmaße für parallele Algorithmen eingeführt, um Algorithmen zu vergleichen und festzustellen, inwieweit sich eine Parallelisierung lohnt. Weiterhin werden wir in sehr kurzer Form, parallele Programmiersprachen und Bibliotheken zur plattformunabhängigen Realisierung paralleler Programme vorstellen. Im dritten Teil werden wir uns der Numerik zuwenden. Es werden numerische Basisalgorithmen, die als Unterprozeduren in komplexeren Algorithmen verwendet werden, eingeführt und die zugehörigen Datenstrukturen erläutert. Daran anschließend beschäftigen wir uns einige Zeit mit der Lösung von Gleichungssystemen. Wir werden dabei hauptsächlich lineare Gleichungssysteme betrachten. Auch wenn die Lösung konzeptuell einfach ist, so ist die Lösung linearer Gleichungssysteme ein oft auftretendes und nicht zu unterschätzendes Problem in der Praxis. Dies liegt daran, daß reale Gleichungssysteme meist sehr groß und oftmals numerisch sehr problematisch sind. Es gibt eine Vielzahl von effizienten und parallelen Algorithmen zur Analyse solcher Gleichungssysteme. Die meisten parallelen Algorithmen sind synchron, d.h. die Berechnungsschritte auf den einzelnen Prozessoren werden in irgend einer Form synchronisiert. Ein solches Vorgehen erlaubt oftmals eine relativ detaillierte Analyse der resultierenden Algorithmen, hat aber den Nachteil, daß der langsamste Berechnungsschritt die Geschwindigkeit des gesamten Algorithmus bestimmt. Dies führt in vielen Fällen zu einer deutlichen Verlangsamung des Algorithmus, insbesondere wenn ein Rechnernetz als Plattform verwendet wird. Eine Alternative zu synchronen Algorithmen sind asynchrone Algorithmen, bei denen Synchronisation nicht oder nur sehr selten ausgeführt wird. Asynchrone Algorithmen können deutlich besser auf unterschiedliche Lasten auf den einzelnen Prozessoren oder Schwankungen in den Kommunikationszeiten reagieren. Damit sind asynchrone Algorithmen insbesondere für Rechnernetze geeignet. Leider sind aber Entwurf und Analyse asynchroner Algorithmen schwieriger als im synchronen Fall. Wir werden uns in Kapitel 6 mit asynchronen Algorithmen und den bei der Entwicklung und Analyse auftretenden Problemen beschäftigen. Als konkretes Anwendungsgebiet wird wieder die Lösung linearer Gleichungssysteme dienen. Im Kapitel 7 wird sehr kurz die parallele Analyse nicht-linearer Gleichungssysteme behandelt. Ein weiteres wichtiges Gebiet, für das wir parallele Algorithmen untersuchen werden, sind Optimierungsprobleme. Wir werden einige parallele Algorithmen kennenlernen, das Gebiet aber nicht so ausführlich wie die Lösung von Gleichungssystemen untersuchen. Im Anhang sind einige Grundlagen der linearen Algebra zu finden.

Kapitel 2

Parallele Architekturen und deren Topologie

Bevor konkrete Algorithmen und deren Parallelisierung vorgestellt werden, soll in diesem Kapitel eine Einführung in den Bereich der parallelen Rechnerarchitekturen gegeben werden. Wir werden uns dabei im wesentlichen damit begnügen, Rechnerarchitekturen auf einer relativ abstrakten Ebene zu untersuchen, die aber realistisch genug ist, reale Architekturen widerzugeben. Begonnen wird mit einem kurzen historischen Abriß und der Betrachtung unterschiedlicher Klassifizierungen paralleler Architekturen. Daran anschließend werden detaillierter Multiprozessorsysteme mit verteiltem Speicher und Workstationnetze (NOWs) vorgestellt. Es werden verschiedene Aspekte untersucht, die die Leistungsfähigkeit paralleler Architekturen beschreiben.

2.1 Parallele Rechnerarchitekturen und ihre Klassifizierung

Parallelität wird seit langem in Rechnern verwendet, um die Leistungsfähigkeit zu steigern. Gerade im Bereich großer numerischer Probleme tauchte immer schon das Verlangen nach mehr Rechenleistung auf als aktuell verfügbar war oder ist. Insofern sind numerische Algorithmen eines der Anwendungsgebiete, welches die Entwicklung von Hochleistungsrechnern vorantreibt. Um die Leistung eines Systems zu erhöhen, können zwei verschiedene Wege beschritten werden. Einmal ist es möglich, die Geschwindigkeit eines einzelnen Prozessors zu erhöhen und so eine höhere Rechenleistung zu erzielen. Weiterhin können einfach viele langsame Prozessoren verwendet werden, die eine Aufgabe gemeinsam lösen. Zwischen diesen beiden Extremen gibt es natürlich Zwischenlösungen, so besitzen heutige Höchstleistungsrechner oft relativ wenige aber sehr leistungsstarke Prozessoren. Die Tendenz scheint allerdings dahin zu gehen, daß zukünftige parallele Systeme aus sehr vielen Standardprozessoren bestehen. Diese können als Workstation Cluster oder als Parallelrechner betrieben werden.

2.1.1 Flynns Klassifizierung

Eine sehr alte und trotzdem verbreitet Art Rechnerarchitekturen zu klassifizieren, ist der Ansatz von Flynn. Dieser Ansatz betrachtet den Fluß von Informationen durch das System, wobei Instruktionen und Daten unterschieden werden. Es gibt die im folgenden vorgestellten vier Klassen von Systemen. **SISD** (*single instruction single data*), **SIMD** (*single instruction multiple data*), **MISD** (*multiple instruction single data*) und **MIMD** (*multiple instruction multiple* *data*).

Konventionelle Rechnerarchitekturen werden als **SISD** Systeme bezeichnet. Ein Instruktionsstrom und ein Datenstrom werden bearbeitet, d.h. Instruktionen werden konsekutiv auf Daten angewendet. Praktisch alle Einprozessorsysteme fallen in die Kategorie SISD.

SIMD Maschinen besitzen eine Instruktionseinheit, die Instruktionen parallel auf mehrere Datenströme anwendet. Typische Vertreter sind die Illiac IV oder die CM-1/2 von Thinking Machines. Die CM2 ist ein moderner Vertreter der SIMD Architekturform. Im Maximalausbau besteht die Maschine aus 4 Quadranten mit 16K Rechenelementen. Jeder Quadrant hat ein eigenes Mikroprogrammsteuerwerk, welches die Recheneinheiten mit Instruktionen versorgt. Die Recheneinheiten sind 1-Bit-ALUs, d.h. sie arbeiten bitseriell. Ursprünglich wurde die CM2 für Anwendungen der künstlichen Intelligenz und die Programmierung mit Lisp entworfen. Es zeigte sich allerdings, daß die Parallelisierung von Lisp Programmen deutlich schwieriger als erwartet war. Außerdem war und ist der Markt für nicht-numerische Supercomputer, wie die ursprüngliche CM2, praktisch nicht vorhanden. Aus diesem Grund wurde die Architektur um Gleitpunktprozessoren ergänzt, wobei je 32 Recheneinheiten ein Gleitpunktprozessor zugeordnet wurde. Damit entstand ein sehr leistungsfähiger Rechner für numerische Anwendungen, der mit anderen Vektorrechnern vergleichbar oder diesen überlegen ist. Womit ein weiterer Typ von Maschinen genannt ist, die unter den Oberbegriff SIMD gefaßt werden, nämlich die Vektorrechner. Es sollte erwähnt werden, daß diese Klassifizierung nicht allgemein akzeptiert ist, da Vektorrechner auch mit guten Argumenten unter SISD Systeme gefaßt werden könnten. Wir wollen die mehr philosophische Frage, ob Vektorrechner nun SISD oder SIMD Maschinen sind nicht weiter vertiefen und die Funktionsweise von Vektorrechnern kurz erläutern. Betrachten wir dazu drei Vektoren mit jeweils n Elementen, die von 0 bis $n-1$ numeriert sind. Berechnet werden soll die Funktion

$$\begin{array}{l} for\ i = 0\ to\ n-1\ do \\ \quad \mathbf{d}(i) = \mathbf{a}(i) * \mathbf{b}(i) + \mathbf{c}(i)\ ; \end{array}$$

Solche und ähnliche Operationen tauchen sehr oft in numerischen Berechnungen auf. In üblichen SIMD Maschinen, wie der Illiac IV, würde die Berechnung parallel für eine Reihe von Vektorelementen auf den einzelnen Prozessoren erfolgen. D.h. im ersten Schritt führt ein Prozessor die Multiplikation, im zweiten Schritt die Addition der ihm zugewiesenen Vektorelemente aus. Die einzelnen Rechenelemente/Prozessoren sind in der Lage verschiedene Operationen auszuführen. In einer Vektormaschine sind die einzelnen Rechenelemente auf feste Operationen spezialisiert, d.h. es handelt sich um Addierer oder Multiplizierer. Die obige Berechnung wird nach dem Pipelineprinzip ausgeführt. D.h. ein Multiplizierer bildet das Produkt $\mathbf{a}(i) * \mathbf{b}(i)$ und ein Addierer addiert $\mathbf{c}(i)$ dazu, während der Multiplizierer schon schon den nächsten Schritt ausführt. Diese Operationsprinzip ist meist mit Vektorregistern (d.h. Registern die Vektoren statt Skalare speichern) kombiniert. Da die verwendeten Gleitpunktrecheneinheiten mit einem starken Pipelining arbeiten, werden extrem hohe Taktraten erreicht. Dies bedingt, daß die rechtzeitige Verfügbarkeit der Daten sichergestellt werden muß. Um dies zu erreichen, sind Speicherverschränkung und Speicherbusse mit hoher Bandbreite notwendig. Insbesondere muß auch der Speicherzugriff auf die Vektoroperationen abgestimmt sein, damit der Speicher nicht zum Flaschenhals wird. Im wesentlichen eignen sich Vektormaschinen für die schnelle Ausführung von Fließkommaoperationen auf regulär strukturierten Daten, d.h. Daten die im Speicher an konsekutiven Adressen gespeichert sind. Der Einsatz lohnt sich nur in Höchstleistungsrechnern. Die Leistung hängt stark von der Vektorlänge und dem gleichmäßigen Zugriff auf Vektorelemente ab. Die Programmierung von Vektorrechnern ist nicht einfach, es wird dazu meist FORTRAN oder in Ausnahmefällen C verwendet. Vektorrechner sind insbesondere mit dem Produktnamen Cray verbunden, führen aber eine Nischenexistenz. Eine gewisse Bedeutung könnten Vektorrechner als Spezialprozessoren in zukünftigen parallelen oder verteilten System erlangen, eine weite

Verbreitung ist allerdings auf Grund des eingeschränkten Einsatzbereichs nicht zu erwarten.

Die Kategorie MISD (*Multiple Instruction Single Data*) ist nur schwer mit bekannten Rechnern zu füllen. Manchmal werden Pipeline Rechner oder systolische Arrays in diese Klasse eingeordnet, ein Datum wird von mehreren Instruktionen bearbeitet.

Die letzte Klasse MIMD (*Multiple Instruction Multiple Data*) umfaßt die im Rahmen dieser Vorlesung relevanten parallelen Architekturen. Unterschiedliche Prozessoren bearbeiten unterschiedliche Instruktionen auf unterschiedlichen Daten. Dies bedingt i.a. auch, daß keine globale Kontrollinstanz existiert, viele Entscheidungen müssen also lokal im Prozessor auf Basis der dort verfügbaren Information getroffen werden. MIMD Rechner beinhalten verteilte und parallele Systeme. Die Klasse MIMD ist relativ breit und umfaßt eine Vielzahl unterschiedlicher Architekturkonzepte. So werden Systeme mit geteiltem oder verteiltem Speicher unter MIMD Systeme gefaßt. Weiterhin werden Systeme unabhängig von ihrer Prozessor-Prozessor und Prozessor-Speicher Verbindungsstruktur in diese Kategorie gefaßt. Beide Aspekte, Speicherorganisation und Kommunikationsstruktur, haben aber einen großen Einfluß auf das Programmiermodell, welches für die Architektur benötigt wird.

Die kurz angesprochenen Kritikpunkte zeigen auch die Grenzen und Schwächen der Taxonomie von Flynn. Parallele Architekturen können nicht genügend genau klassifiziert werden und verschiedene Architekturkonzepte sind nicht vernünftig einzuordnen (z.B. Vektorrechner). Auf der anderen Seite ist die Kategorie MISD nur schwer mit realen Architekturen zu füllen. Wir werden deshalb im weiteren einige feinere Kategorien für Parallelrechner betrachten, die eine detailliertere Einordnung erlauben.

2.1.2 Andere Klassifizierungen

Die Klassifizierung nach Flynn ist weit verbreitet, zeigt aber deutliche Mängel, wenn sie auf heutige Rechner angewendet wird. Hier sind deshalb weitere Kriterien nach denen Rechner klassifiziert werden können und die insbesondere Multiprozessorsysteme genauer klassifizieren. Einige dieser Kriterien werden später wieder aufgegriffen.

- *Eigenschaften der verwendeten Prozessoren:* Es wird nach Anzahl und Typ der verwendeten Prozessoren unterschieden. Systeme mit mehreren Prozessoren kann man dahingehend unterteilen, ob diese Prozessoren homogen oder inhomogen sind.

- *Speicherzugriff:* Ist ein globaler Speicher im System vorhanden oder existieren einzelne Speicherbereiche, die jeweils einzelnen Prozessoren exklusiv zugeordnet sind. Diese Entscheidung bestimmt natürlich auch die Art der Kommunikation zwischen den Prozessoren. Im ersten Fall wird über den gemeinsamen Speicher kommuniziert, im zweiten Fall durch den Austausch von Nachrichten. Diese Art der Klassifizierung wird uns in 2.2 noch näher beschäftigen.

- *Struktur der Verbindung zwischen Prozessoren:* Prozessoren in Multiprozessorsystemen sind durch eine Kommunikationshardware verbunden. Diese kann aus geschalteten oder festen Verbindungen bestehen und beeinflußt das Systemverhalten und die Programmierung in starkem Maße. Verbindungsstrukturen sind Inhalt von 2.3.

- *Art des verwendeten Betriebssystems:* Wird die parallele Hardware durch das Betriebssystem verdeckt oder ist sie für den Benutzer sichtbar? Weiterhin gibt es unterschiedliche Arten, parallele Prozessoren zu organisieren. Schlagworte in diesem Zusammenhang sind "Master-Slave" oder "Client-Server" Strukturen.

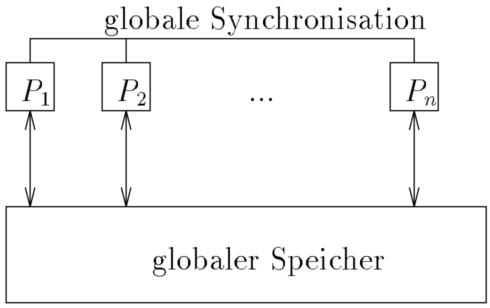


Abbildung 2.1: Struktur einer PRAM.

- *Ebenen der Parallelisierung:* Die parallele Abarbeitung kann auf unterschiedlichen Ebenen erfolgen, angefangen von der Parallelisierung einzelner Befehle, bis zur Abarbeitung paralleler Programme. In 3.2 werden verschiedene Ebenen oder Granularitäten näher betrachtet.

2.1.3 PRAMs: ein theoretisches Modell für parallele Rechner

Das wohl bekannteste Parallelrechnermodell der theoretischen Informatik ist die "parallel random access machine" (PRAM) als Erweiterung der random access machine (RAM). PRAMs beschreiben MIMD Systeme mit geteiltem Speicher. Eine p-Prozessor PRAM besteht aus p Prozessoren und einem globalen gemeinsamen Speicher (shared memory) siehe Abb. 2.1. Jeder Prozessor ist eine Registermaschine mit lokalem Speicher. Kommunikation zwischen Prozessoren findet über den globalen Speicher statt. Der globale Speicher beinhaltet m individuell adressierbare Speicherbereiche. In jedem Rechenschritt können Prozessoren lesend oder schreibend auf den globalen Speicher zugreifen oder lokale Berechnungen durchführen. Die Prozessoren arbeiten synchron mit identischem Takt und identischer Geschwindigkeit. Konflikte treten auf, wenn zwei Prozessoren gleichzeitig auf einen identischen Bereich des globalen Speichers zugreifen wollen. Es wurden verschiedene Untermodelle zur Lösung von Speicherzugriffskonflikten publiziert.

Das PRAM Modell ist hervorragend zur Analyse von parallelen Algorithmen geeignet. Man braucht sich nicht um Verbindungsstrukturen und die Verteilung der Daten zu kümmern. Die Anzahl der ermittelten Rechenschritte für einen parallelen Algorithmus ist ein Maß für die Rechenzeit des Algorithmus.

Diese vom theoretischen Standpunkt sehr angenehmen Eigenschaften stellen aber gleichzeitig das größte Problem bei der Realisierung von PRAMs dar. Gemeinsame Speicher in der geforderten Form sind für Multiprozessorsysteme mit vielen Prozessoren bisher nicht effizient realisierbar. Die naheliegende Methode, die Segmente des gemeinsamen Speichers einfach auf die lokalen Speicher zu verteilen, führt natürlich dazu, daß Zugriffe auf nicht lokal vorhandene Daten durch Nachrichtenaustausch simuliert werden müssen. Ist dies der Fall, so müssen natürlich auch die Kommunikationszeiten bei der Algorithmenanalyse einbezogen werden. Dies erschwert die Analyse deutlich, wie wir später noch sehen werden. Es gibt eine Reihe von Ansätzen, um die Simulation von PRAMS auf Architekturen mit verteiltem Speicher möglichst effizient durchzuführen. Trotzdem beeinflußt die Lage der Daten klarerweise die Geschwindigkeit des Algorithmus, ein Punkt der im PRAM Modell nicht erfaßbar ist.

Es gibt eine weitere, wenn auch nicht so verbreitete, parallele Variante der random access machine, die MP-RAM. Eine MP-RAM basiert auf dem message passing Paradigma. p RAMs bilden eine MP-RAM, im Gegensatz zur PRAM existiert nun aber kein gemeinsam zugreifbarer Speicher mehr. Die einzelnen RAMs besitzen lokale Speicher und kommunizieren mit anderen

RAMs über den Austausch von Nachrichten. Dazu wird der Befehlssatz um Primitive zum Versenden und Empfangen von Nachrichten erweitert. RAMs können nur mit bestimmten anderen RAMs, nämlich den direkten Nachbarn, direkt kommunizieren. Die Kommunikation mit weiter entfernten RAMs muß über Zwischenknoten erfolgen. Die Kommunikation zwischen den RAMs erfolgt blockierend, d.h. falls eine RAM eine Nachricht empfangen will, blockiert sie so lange, bis eine Nachricht eintrifft. Wird eine zweite Nachricht von einem Sender an einen Empfänger geschickt, ohne daß die erste gelesen wurde, so blockiert der Sender.

Das MP-RAM Modell beschreibt viele parallele Architekturen sehr viel genauer als das PRAM Modell, hat aber den Nachteil, daß es schwieriger zu analysieren ist und die Analyse von der Verbindungstopologie abhängt. Wir werden bei der Analyse konkreter Topologien implizit eine vereinfachte Form der MP-RAM zugrundelegen.

2.2 Systeme mit geteiltem und verteiltem Speicher

Eine wichtige Klassifizierung realer Systeme ist die Unterscheidung in Systeme mit geteiltem und mit verteiltem Speicher. Erstere werden manchmal als *Multiprocessor Systems*, letztere als *Multicomputer Systems* bezeichnet. Es sind natürlich auch Realisierungen möglich und üblich, bei denen einige Prozessoren sich einen Speicherbereich teilen, der von anderen Prozessoren nicht zugreifbar ist. Eine Beispiel für eine solche Architektur wäre ein Workstation Cluster, in dem Workstations mit mehreren Prozessoren und geteiltem Speicher existieren.

2.2.1 Systeme mit geteiltem Speicher

In Systemen mit geteiltem Speicher existiert ein globaler Speicher, der physikalisch natürlich aus mehreren Segmenten bestehen darf. Jeder Prozessor kann auf jedes Segment lesend und schreibend zugreifen. Es wird angenommen, daß die Speicherzugriffszeiten aller Prozessoren ungefähr identisch sind. Zwei Prozessoren kommunizieren, indem Prozessor 1 eine Speicherzelle schreibt, die Prozessor 2 anschließend liest. Zugriffskonflikte werden auf Betriebssystemebene gelöst. Architekturen mit geteiltem Speicher erlauben ein einfaches Design der Algorithmen, da die Kommunikation zwischen Prozessoren und zwischen Prozessor und Speicher unabhängig von der Prozessor- und Speicheradresse ist. Damit entfällt das Problem einer effizienten Zuordnung von Prozessen zu Prozessoren und Daten zu Speicherbereichen. Die Zuordnung wird i.a. automatisch vorgenommen. Eine abstrakte Sicht der Realisierung einer Architektur mit geteiltem Speicher ist in Abbildung 2.2 gezeigt. p Prozessoren werden über ein Verbindungsnetz mit m Speichern verbunden. Der kritische Punkt ist das Verbindungsnetz. Werden nur wenige Prozessoren (zwischen 2 und 8) verwendet, so reicht ein schneller Bus als Verbindungsnetz aus. Bei einer Erhöhung der Prozessorzahl wird der Bus aber zwangsläufig zum Flaschenhals und verhindert damit eine effiziente Prozessornutzung. Deshalb müssen zur Verbindung größerer Prozessorzahlen komplexe Schaltnetze verwendet werden, die Verbindungen zwischen Prozessor und Speicher herstellen. Zwangsläufig wächst die Komplexität solcher Schaltnetze sehr schnell mit der Prozessor- und Speicherzahl. Dies führt dazu, daß die Speicherzugriffszeit und Fehleranfälligkeit des Verbindungsnetzes steigt. Weiterhin sind solche Systeme nicht skalierbar, d.h. die Anzahl der Prozessoren kann nicht einfach durch Hinzufügen weiterer Prozessoren erhöht werden. Aus diesen Gründen scheint der Ansatz für massiv parallele Architekturen mit sehr großen Prozessorzahlen nicht geeignet zu sein.

Systeme mit geteiltem Speicher, die wie oben beschrieben realisiert sind, werden oftmals auch als UMA (uniform memory access) Systeme bezeichnet. Da diese Systeme eine relativ konventionelle Programmierung paralleler Anwendungen erlauben, sind sie vom Standpunkt

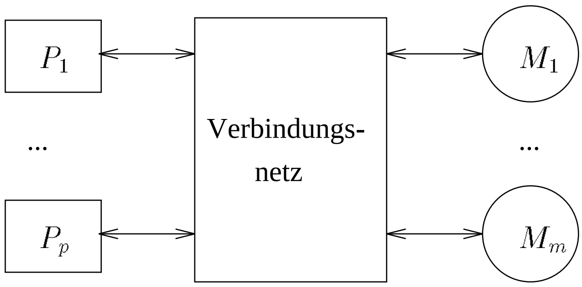


Abbildung 2.2: Architektur mit geteiltem Speicher.

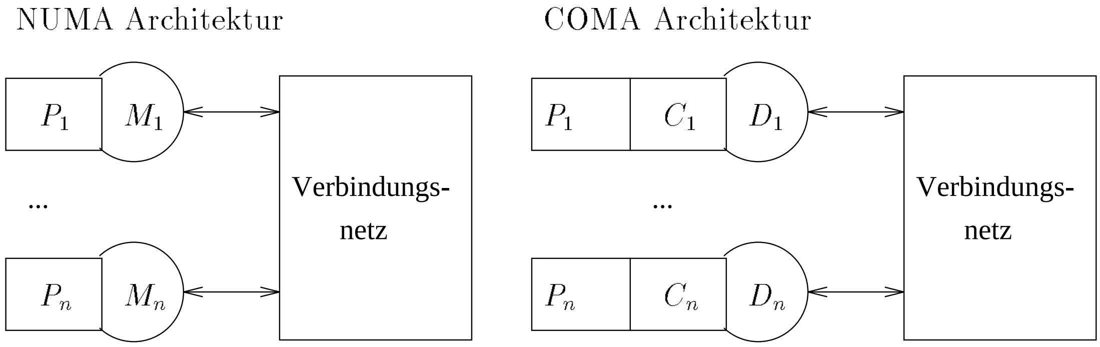


Abbildung 2.3: NUMA und COMA Systeme mit virtuell geteiltem Speicher.

der Programmierung sehr beliebt. Leider sind der physikalischen Realisierung von UMA Systemen enge Grenzen gesetzt. Aus diesem Grund wird versucht, einen geteilten Speicher auf Systemen mit verteiltem Speicher zu simulieren. Die beiden möglichen Realisierungen sind in 2.3 schematisch angedeutet und sollen kurz erläutert werden.

Beginnen wir mit den NUMA (non-uniform memory access) Systemen. Speichermodule sind einzelnen Prozessoren zugewiesen, es existiert aber ein globaler Adreßraum, der alle Speichermodule umfaßt. Ein Prozessor kann damit auf eine beliebige Speicherzelle unabhängig von deren physikalischer Lage zugreifen. Diese Sichtweise erlaubt die Programmierung des Systems wie bei Systemen mit real geteiltem Speicher. Leider ist aber die Zugriffszeit auf einzelne Speicherbereiche von deren Lage abhängig. So ist der Zugriff auf den lokalen Speicher um ein Vielfaches schneller als der Zugriff auf einen physikalisch weit entfernten Speicherbereich. Die Geschwindigkeit, mit der Programme abgearbeitet werden, hängt ganz entscheidend davon ab, wo sich die Daten befinden. Die vollständige Abstraktion von der physikalischen Speicherstruktur kann deshalb zu extrem ineffizienten Programmen führen.

Die zweite Variante des virtuell geteilten Speichers sind COMA (cache only memory architectures) Architekturen. Die statische Struktur dieser Systeme, wie sie in Abb. 2.3 gezeigt wird, ist ähnlich zu NUMA Systemen. Der dynamische Ablauf unterscheidet sich aber deutlich. Während in NUMA Systemen ein Datum einen festen Platz im Speicher hat, werden in COMA Architekturen Daten in lokale Speicher (Caches) geladen. Jeder Prozessor besitzt einen lokalen Cache und ein Cache Directory (D). Der logische Adreßraum ist wie bei NUMA Systemen homogen, logische Adressen werden aber auf physikalische Adressen umgerechnet durch Zugriff auf das Cache Directory. Daten werden i. a. zu dem Prozessor transferiert, der zuletzt auf das Datum zugegriffen hat. Damit ist sichergestellt, daß Daten zur Berechnung auch lokal vorhanden sind. COMA Architekturen sind deshalb in vielen Anwendungen deutlich schneller als NUMA Architekturen. Allerdings haben COMA Architekturen auch Nachteile. So ist die Verwaltung der Directories und die Migration von Daten sehr komplex und kann in manchen Situationen zu einem extremen Verwaltungsaufwand führen, der die Leistungsvorteile zunichte macht. Weiterhin werden Daten klarerweise in Blöcken transferiert. Tritt nun eine Situation auf, in der ein Block Daten beinhaltet, auf die von unterschiedlichen Prozessoren aus zugegriffen wird, so wird der entsprechende Block immer wieder zwischen den Speicherbereichen hin und her übertragen, was zu einer deutlichen Leistungsminderung führt.

2.2.2 Systeme mit verteiltem Speicher

In einem System mit verteiltem Speicher hat jeder Prozessor einen lokalen Speicher und kann auch nur seinen lokalen Speicher adressieren. Einzelne Prozessoren können als eigenständige Computer interpretiert werden. Prozessoren sind auf unterschiedliche Weise verbunden. Im allgemeinen existiert kein globaler Bus, sondern eine feste Topologie, in der nur einzelne Prozessoren verbunden sind. Diese Struktur hat gegenüber dem geteilten Speicher verschiedene Vorteile. Insbesondere sind Architekturen mit verteiltem Speicher (fast) beliebig skalierbar. Da keine globalen und gemeinsam genutzten Komponenten existieren, treten auch keine Flaschenhälse bzgl. der verfügbaren Bandbreite oder Speicherkapazität auf. Das Hinzufügen neuer Prozessoren mit lokalem Speicher ist in den meisten Topologien durch eine einfache Erweiterung der bestehenden Topologie möglich. Des weiteren bestehen auch keine Konsistenzprobleme bzgl. des Speicher- oder Cacheinhalts, da jeder Prozessor für seinen eigenen lokalen Speicher verantwortlich ist.

Der entscheidende Nachteil von Systemen mit verteiltem Speicher ist der Kommunikationsaufwand, der durch die Interprozeßkommunikation bedingt ist. Wenn ein Prozessor Daten benötigt, die im Speicher eines anderen Prozessors liegen, so muß dies durch den Austausch von Nachrichten realisiert werden. Dies bedeutet zusätzlichen Aufwand aus zwei Gründen. Zum einen muß eine Nachricht beim Sendeprozeß generiert und zum Empfängerprozeß geschickt werden. Zum anderen muß der Empfängerprozeß unterbrochen werden, damit die angeforderten Daten aus dem Speicher geholt und übertragen werden. In modernen Systemen wird ein Großteil der beschriebenen Funktionalität von der Hardware abgewickelt und damit relativ effizient realisiert. Trotzdem ist der resultierende Overhead nicht zu vernachlässigen.

Die Programmierung von Systemen mit verteiltem Speicher ist deutlich schwieriger als die Programmierung sequentieller Architekturen. Dies ist insbesondere dadurch bedingt, daß Programme in unabhängige Tasks unterteilt werden müssen, die auf den verschiedenen Prozessoren ablaufen. Ziel einer solchen Aufteilung muß es sein, den Kommunikationsaufwand zwischen Tasks zu minimieren und Kommunikation möglichst auf solche Tasks zu beschränken, die auf benachbarten Prozessoren liegen. Damit kommen einige neue Aspekte bei der Programmerstellung hinzu. Programme müssen in kommunizierende Tasks mit möglichst hoher Lokalität partitioniert werden und die Zuordnung von Tasks zu Prozessoren beeinflußt die Laufzeit des Programms. Eine sinnvolle Zuordnung kann nur in Abhängigkeit von der Topologie gewählt werden. Letzterer Aspekt macht eine Programmierung unabhängig von der verwendeten Rechnerarchitektur sehr schwierig, wenn nicht sogar unmöglich. Im Rahmen von Unterkapitel 2.3.2 werden wir uns mit einigen typischen und weitverbreiteten Topologien und deren Eigenschaften beschäftigen.

Viele Konzepte sequentieller Programmierung sind auf parallelen Architekturen mit verteiltem Speicher nicht oder nur sehr ineffizient einsetzbar. Anderseits ist der Austausch von Nachrichten aber auf natürliche Weise in objektorientierten Ansätzen realisierbar.

2.2.3 Workstation Netze

Die am weitesten verbreiteten Architekturen, auf denen parallele Programme ablaufen können sind Cluster von Workstations oder PCs. Heutige Workstation Netze (NOWs) basieren im allgemeinen auf einer 100 MBit/s Ethernet Kommunikationsstruktur, die zumindest von der Bandbreite her für viele Anwendungen ausreichend ist. Weiterhin ermöglicht die Verfügbarkeit von Kommunikationsbibliotheken, die (teilweise) plattformunabhängige Programmierung von parallelen Algorithmen. Damit können insbesondere parallele Algorithmen auf NOWs entwickelt und getestet werden, was die Entwicklungskosten deutlich senkt, da der niedrigere Preis einer der Hauptvorteile von NOWs gegenüber Multiprozessorsystemen ist. Es muß allerdings angemerkt werden, daß die Leistung von NOWs im allgemeinen nicht die Leistung von MPs erreicht. Dies gilt insbesondere für synchrone Algorithmen mit größerem Kommunikationsbedarf. Die Vor- und Nachteile von Workstation Netzen gegenüber Parallelrechnern werden in Tabelle 2.1 zusammengefaßt.

	Workstation Netze	Multiprozessorsysteme
Preis	vergleichsweise preiswert	teuer
Architektur	standard (hohe Stückzahlen)	spezielle (kleine Stückzahlen)
Rechenleistung	theoretisch hoch	theoretisch hoch
	praktisch deutlich niedriger	praktisch meist auch hoch
Fehlertoleranz	hoch	niedriger
Kopplung	lose	eng
Betriebssystem	Standard (i.a. Unix)	speziell
Flexibilität	hoch da Mehrbenutzerbetrieb und	niedrig, da Prozessoren oft nur im
	Standardsoftware auf jedem Rechner	exklusiven Betrieb, wenig Software
Skalierbarkeit	groß durch einfache Integration	eingeschränkt durch Grenzen
	neuer WS	der Hardware,
		nur ein Prozessortyp verwendbar
Wartungskosten	gering durch Standard-Soft- und Hardware	hoch durch Spezialkomponenten
Vorteile	Flexibilität und Preis	Rechenleistung
Anwendung	allgemeiner Einsatz	"Grand Challenges´´ und Forschung

Tabelle 2.1: Vergleich von Workstation Netzen und Multiprozessorsystemen.

Man unterscheidet zwei grundsätzliche Ansätze, Workstation Cluster als Plattform für parallele Programme einzusetzen. Der naheliegende Ansatz ist es, parallele Algorithmen auf vorhandenen Clustern zu realisieren. In den meisten Fällen sind genügend Ressourcen in einem Netz vorhanden, so daß zumindest zu Randzeiten eine größere Anzahl von Workstations exklusiv genutzt werden kann. Dieses Vorgehen hat den Vorteil, daß keine zusätzlichen Investitionen in Hard- oder Software getätigt werden müssen, da Kommunikationsbibliotheken wie PVM oder MPI frei verfügbare Software sind. Allerdings wird für die Realisierung paralleler Programme auf NOWs Softwareunterstützung benötigt, die die Tasks eines Programms den Workstations zuweist. Dabei ist die Last auf den einzelnen Rechnern zu beachten, wobei als zusätzliche Schwierigkeit hinzukommt, daß die Last sich während eines Programmlaufs ändern kann. Weitere Probleme bei der Verwendung allgemeiner NOWs als Plattform für parallele Algorithmen sind heterogene Workstations im Netz und die langsame und durch Hintergrundlast beeinflußte Kommunikation. Zusammenfassend kann man sagen, daß konventionelle NOWs sich nur in eingeschränktem Maße als Plattform für parallele Algorithmen eignen. Effizient können nur sehr wenige Algorithmen mit einem geringen Kommunikationsanteil auf diesen Architekturen realisiert werden. Gleichwohl eignen sich solche NOWs als Plattform zur prototypischen Realisierung von parallelen Algorithmen, wenn es weniger auf die tatsächlich erreichbare Leistung als auf die Untersuchung des Verhaltens ankommt. Dies trifft oftmals auf Algorithmen zu, die im Rahmen von Lehrveranstaltungen realisiert werden. Wir werden uns mit einigen Aspekten

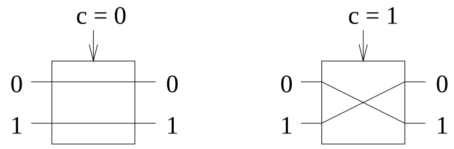


Abbildung 2.4: Binärer Switch mit Control Bit c=0 für Through State und c=1 für Cross State.

der parallelen Realisierung von numerischen Algorithmen in Kapitel 6 beschäftigen.

Eine zweite Form von NOWs, die sich besser als Plattform für parallele Algorithmen eignet, sind sogenannte *dedicated cluster*. Solche NOWs bestehen im allgemeinen aus homogenen Workstations, die lokal eng beieinander plaziert werden und schnelleren Kommunikationsmedien verbunden werden. Solche NOWs werden als Alternative zu Parallelrechnern eingesetzt und sind deshalb auch nicht frei zugänglich und haben nur eine eingeschränkte Softwarekonfiguration. NOWs dieser Form sind eine ernsthafte Konkurrenz für Parallelerechner, da sie billiger aber (fast) genauso leistungsfähig sind.

2.2.4 Eine kurze Zusammenfassung

Wir haben uns in diesem Kapitel mit unterschiedlichen Typen von Parallelrechnern beschäftigt. Dabei zeigt sich, daß Systeme mit verteiltem Speicher am weitesten verbreitetet sind und wohl auch die in Zukunft am meisten verwendete Architekturform sein werden. Aus diesem Grund werden wir uns auf parallele Algorithmen für Systeme mit verteiltem Speicher beschränken.

2.3 Verbindungsstrukturen paralleler Systeme

Wir haben bereits gesehen, daß die Verbindungsstruktur von Prozessoren in einem parallelen System einen großen Einfluß auf die Programmierung und die Laufzeit von Algorithmen haben kann. Im Rahmen dieses Kapitels werden einige typische Verbindungsstrukturen vorgestellt. Es werden dabei statische Strukturen, in denen nur einige Prozessoren verbunden sind, und dynamische Strukturen, in denen Verbindungen zwischen beliebigen Prozessoren geschaltet werden können, unterschieden.

2.3.1 Dynamische Verbindungsstrukturen

Die einfachste dynamische Verbindungsstruktur ist ein Bus, an dem alle Komponenten angeschlossen sind. Ein Bus bildet zwar eine statische Verbindung zwischen den Komponenten, Kommunikation wird aber dadurch realisiert, daß eine feste Verbindung zwischen zwei oder mehreren Komponenten durch das Kommunikationsprotokoll sichergestellt wird. Wir wollen deshalb den Bus als dynamische Verbindungsstruktur interpretieren. Diese Verbindungsart ist relativ einfach und erlaubt insbesondere auch das Senden einer Nachricht an alle anderen Komponenten (Broadcast). Das größte Problem einer Busverbindung ist aber die limitierte Bandbreite und die dadurch fehlende Skalierbarkeit. Der Bus ist immer eine globale Ressource, die zum Engpaß wird, wenn die Anzahl oder Übertragungsrate der angeschlossenen Komponenten erhöht wird. Busse eignen sich damit nur zur Verbindung weniger Prozessoren und Speicher.

Eine Alternative zur festen Busverbindung sind geschaltete Verbindungen. Schaltungen werden meist über binäre Schaltungselemente (Switches) realisiert, wie in Abb. 2.4 gezeigt. Durch Zusammenschalten mehrerer Switches läßt sich ein Verbindungsnetz für eine theoretisch beliebige Anzahl von Komponenten generieren. Routing von einer Quelle zu einem Ziel erfolgt

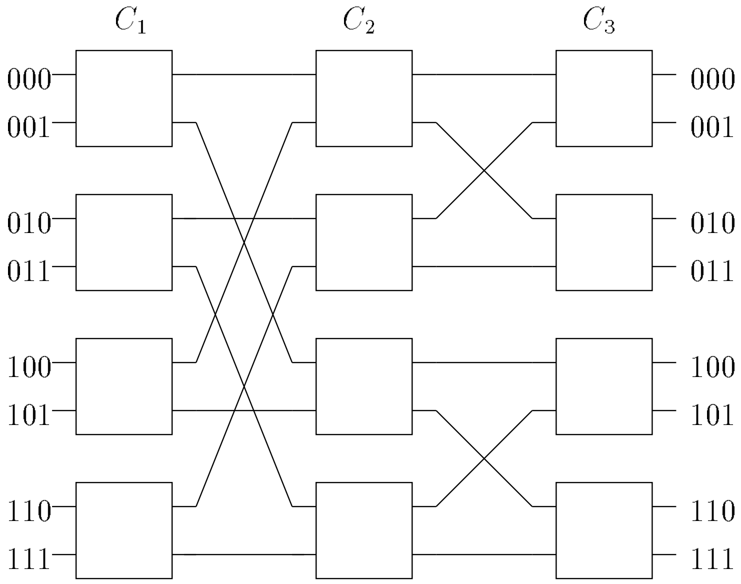


Abbildung 2.5: 8×8 Banyan Netzwerk.

über binäre Adressen, wie durch die Numerierung der Ein-/Ausgänge am Switch angedeutet. Es gibt unterschiedliche Möglichkeiten, geschaltete Netze durch Kombination binärer Elemente zu generieren. Abb. 2.5 zeigt ein Banyan Netz zur Verbindung von 8 Komponenten. Die gezeigte Architektur kann dazu eingesetzt werden, bis zu 8 Prozessoren mit bis zu 8 Speichern zu verbinden. Alternativ können auch bis zu 8 Prozessoren mit lokalem Speicher untereinander verbunden werden. In diesem Fall sind die Ein- und Ausgänge mit identischer Adresse mit dem Ein-/Ausgangsport eines Prozessors verbunden.

Routing in Banyan Netzen erfolgt durch Interpretation der Zieladresse. Auf jeder Stufe (C_1, C_2, C_3) wird ein Bit der Adresse interpretiert. In Stufe C_i wird das i-te Bit interpretiert, i.e. als Control Bit verwendet. In Abhängigkeit vom Wert dieses Bits wird die Nachricht durch den oberen Ausgang (das i-te Bit ist 0) oder durch den unteren Ausgang (i-te Bit ist 1) geleitet. Diese Art des Routings kann effizient in der Hardware realisiert werden, was sehr wichtig für die Leistungsfähigkeit der Verbindungsstruktur ist. Blockierungen treten auf, wenn Pakete über identische Verbindungen geschickt werden sollen. Bei steigender Zahl der zu verbindenden Komponenten steigt auch die Wahrscheinlichkeit der Blockierung von Nachrichten, da die Anzahl der Stufen zunimmt, auf denen Blockierungen auftreten können. Dies beeinflußt natürlich die Übertragungszeit und maximale Bandbreite. Die Anzahl der Elemente zur Verbindung von N Komponenten ist $\lceil N/2 \rceil \lceil \log_2 N \rceil$. Da wir im wesentlichen den Logarithmus zur Basis 2 benötigen, wird im folgenden log für $\log_2$ verwendet.

Als letzte geschaltete Verbindungsstruktur zur Verbindung von Prozessoren und Speichern in Systemen mit geteiltem Speicher betrachten wir den sogenannten "Crossbar-Switch". Die zugehörige Struktur ist in Abb. 2.6 gezeigt. Für jeden Prozessor und jeden Speicher existiert ein lokaler Bus. Jeder Prozessor Bus ist durch ein Schaltelement mit jedem Speicherbus verbunden. Der Zugriff auf einen Speicherbereich erfolgt über den Prozessorbus des zugreifenden Prozessors und den Speicherbus des entsprechenden Speichermoduls. Konflikte treten nur auf, wenn auf gleiche Speicherbereiche zugegriffen werden soll. Die Anzahl der Busse in einem System mit P Prozessoren und M Speichermodule ist $P + M$, die Anzahl der Schalter PM. Für Architekturen mit relativ wenigen Prozessoren und Speichern ist diese Verbindungsart gut geeignet. Für Systeme mit großer Prozessor- und Speicherzahl werden allerdings schnell die Grenzen des technologisch Machbaren erreicht.

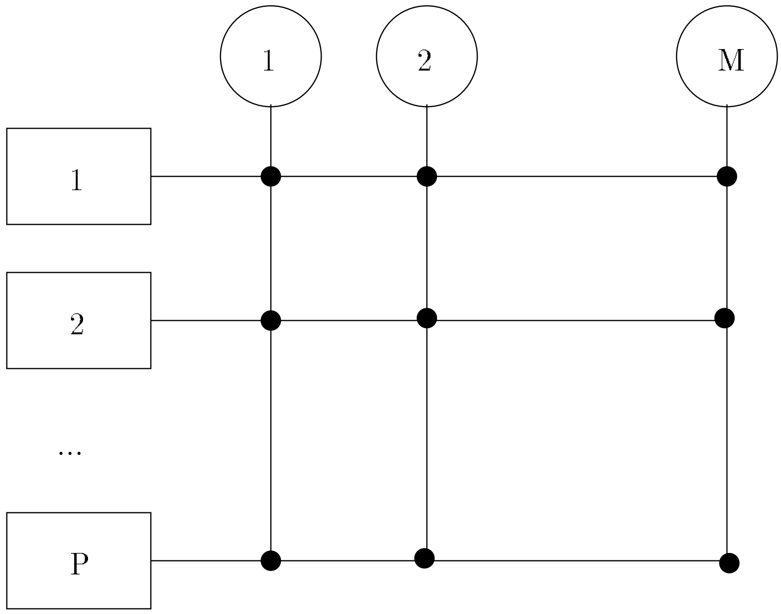


Abbildung 2.6: Crossbar-switch zur Verbindung von Prozessoren und Speichern.

2.3.2 Statische Verbindungsstrukturen

Bei den meisten Multiprozessoren mit verteiltem Speicher ist die Verbindungsstruktur statisch. Die Prozessoren mit ihren lokalen Speichern können als Knoten in einem Graph interpretiert werden. Die Kanten des Graphs beschreiben die Kommunikationspfade. Aus Komplexitätsgründen ist der aus einer Multiprozessorarchitektur resultierende Graph im allgemeinen nicht voll vermascht. Auch wenn die Prozessoren als Knoten des Graphen interpretiert werden können, so wird die Kommunikation in den meisten Fällen nicht vom Prozessor abgewickelt. Üblicherweise steht spezielle Kommunikationshardware zur Verfügung.

Wenn nicht direkt verbundene Prozessoren kommunizieren wollen, so müssen Nachrichten über Zwischenknoten versandt werden; um dies zu realisieren ist ein Routing Algorithmus notwendig. Es gibt verschiedene Möglichkeiten, Routing Algorithmen zu realisieren. In Multiprozessorsystemen muß das Routing sehr schnell, also in der Hardware realisiert sein und es muß dezentral sein, damit die Architektur skalierbar ist.

Im folgenden werden wir erst einige verbreitete Topologien vorstellen, dann Bewertungsmaße definieren und die vorgestellten Topologien auf Grund dieser Maße bewerten. Zum Abschluß des Kapitels betrachten wir die Einbettung verschiedener Topologien in andere Topologien. Eine solche Einbettung kann benutzt werden, um einen Algorithmus von einer Topologie auf eine andere zu transferieren und so eine teilweise Architekturunabhängigkeit herzustellen.

Verbreitete Topologien

Aus rein pragmatischen Gründen sind die meisten Multiprozessorsysteme regulär strukturiert. Wir betrachten im folgenden einige übliche Strukturen. Eine verbreitete Topologie, die insbesondere für Probleme des wissenschaftlichen Rechnens geeignet ist, sind ein- oder mehrdimensionale Arrays. Die Leitungen zwischen Prozessoren sind meist bidirektional. In der Praxis werden oft ein- oder zweidimensionale Arrays von Prozessoren verwendet, da diese physikalisch am einfachsten zu realisieren sind. Zweidimensionale Prozessorarrays haben den Vorteil, daß ihre Struktur auf natürliche Weise zu zweidimensionalen Matrizen korrespondiert, die für viele mathematische Probleme die Basisdatenstruktur bilden. Arrays werden auch als Gitter bezeichnet.

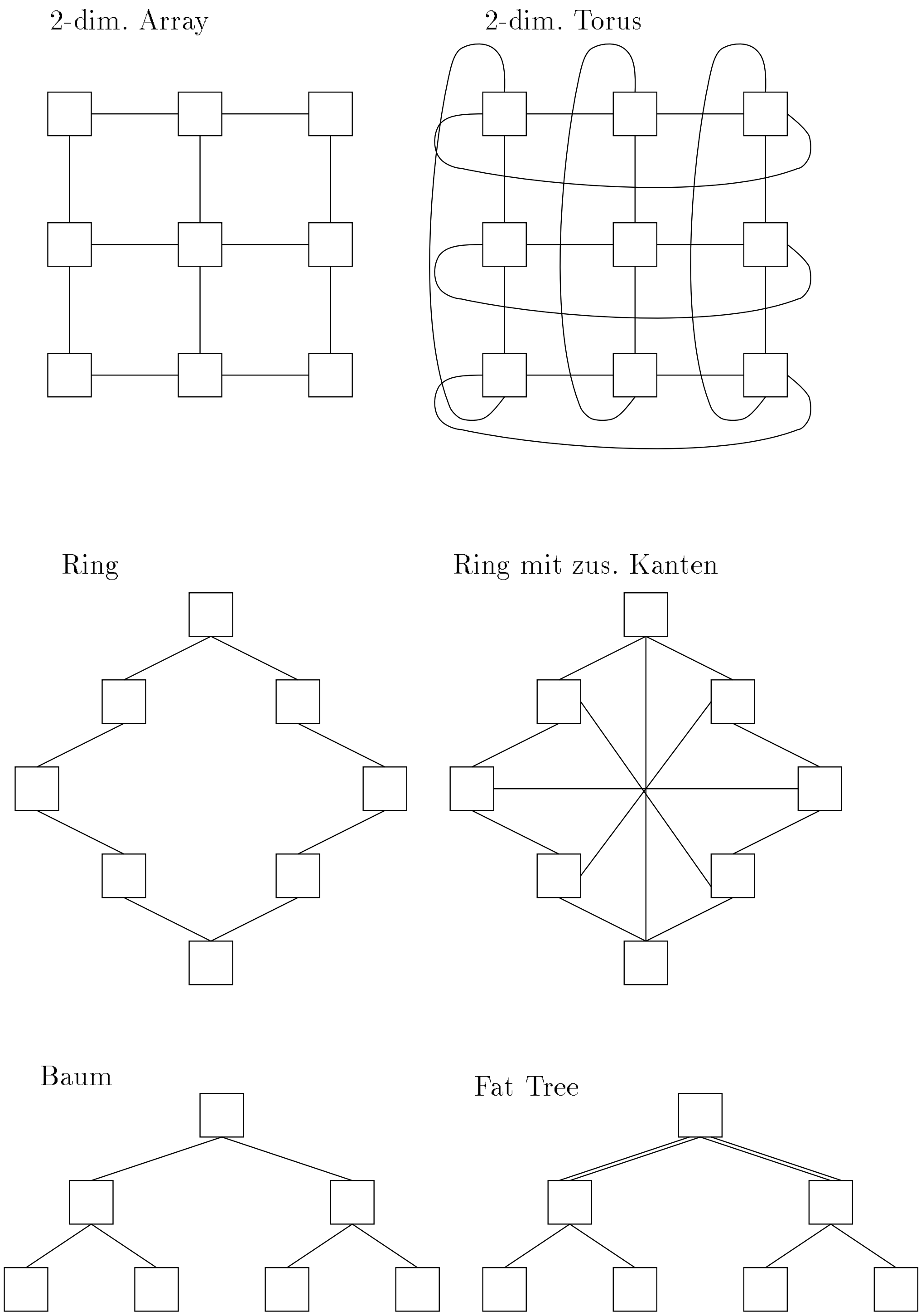


Abbildung 2.7: Array, Ring, Torus Netz und Baumstrukturen.

Eine weitere Klasse von Topologien sind Ringe. Diese können aus uni- oder bidirektionalen Leitungen gebildet werden. Da in einem Ring die Distanz zwischen Prozessoren sehr groß werden kann, werden manchmal Leitungen zu "weiter entfernten" Prozessoren eingefügt.

Als Verbindung zwischen Arrays und Ringen sind Torus Netze anzusehen. Ausgehend von einem Array werden der erste und letzte Prozessor in jeder Dimension miteinander verbunden. Wie auch bei Arrays werden im wesentlichen zwei- oder maximal dreidimensionale Torus Netze realisiert.

Nicht so weit verbreitet, aber grundsätzlich möglich ist der Aufbau von Baumstrukturen. Dabei sind besonders Binärbäume von Interesse. Diese Architektur ist natürlich besonders geeignet für Algorithmen, die auf Binärbäumen als Basisdatenstruktur basieren. Ein Nachteil der Baumstruktur ist die ungleiche Auslastung der Kommunikationspfade. Sämtliche Kommunikation zwischen rechtem und linkem Teilbaum läuft über die Wurzel, die dadurch zum Flaschenhals der Architektur wird. Abhilfe schafft hier der Einsatz schnellerer Leitungen in den oberen Ebenen der Baumhierarchie. Dies führt zum sogenannten "Fat-Tree". Abb. 2.7 zeigt die Graphen der bisher vorgestellten Architekturen. Für den Fat Tree sind die schnelleren Leitungen durch zwei parallele Verbindungen angedeutet.

Eine weitere verbreitete Topologie sind Hyperwürfel. Ein Hyperwürfel ist im Prinzip ein d-dimensionales Gitter, wobei in jeder Dimension die Anzahl der Prozessoren 2 ist. Ein Hyperwürfel der Dimension 1 ist die Verbindung von zwei Prozessoren, ein Hyperwürfel der Dimension 2 ist ein 2×2 Gitter, ein Hyperwürfel der Dimension 3 ist ein Würfel mit Kantenlänge 2 usw. Eine räumliche Darstellung von Hyperwürfeln ist natürlich nur bis zu drei Dimensionen möglich. Hyperwürfel können aber auch flach dargestellt werden. Abb. 2.8 zeigt, wie ein Hyperwürfel der Dimension 4 aus zwei Hyperwürfeln der Dimension 3 gebildet wird. Das zugrundeliegende Konstruktionsprinzip ist allgemein und kann verwendet werden, um aus zwei Hyperwürfeln der Dimension $d-1$ einen Hyperwürfel der Dimension d zu erzeugen. Prozessoren im Hyperwürfel bekommen eine binäre Adresse, die aus d Bits in einen d-dimensionalen Hyperwürfel besteht. Prozessoren, deren Adressen sich in nur einem Bit unterscheiden, sind durch eine Leitung verbunden. Durch dieses einfache Schema ist die Wegewahl von Nachrichten sehr einfach realisierbar. Ein Prozessor kann auf Grund der Differenz seiner eigenen Adresse und der Zieladresse einer Nachricht auf einfache Art den nächsten Knoten wählen. Die *Hamming Distanz* zweier Prozessoren ist die Anzahl der Bits, um die sich die Adressen der Prozessoren unterscheiden. Die physikalische Distanz zwischen zwei Prozessoren im Hyperwürfel entspricht gerade der Hamming Distanz. Aus den Adressen zweier Knoten läßt sich auch ein direkter (d.h. möglichst kurzer) Weg konstruieren, indem nacheinander die unterschiedlichen Bits ausgehend von der Startadresse modifiziert werden und die Nachricht zu dem entsprechenden Prozessor geleitet wird. In einem dreidimensionalem Hyperwürfel verläuft ein kürzester Weg von 001 nach 010 über 011 oder 000. Damit wird auch klar, daß zwischen zwei Prozessoren mit Hamming Distanz h $h!$ viele kürzeste Wege existieren und die maximale Weglänge im Hyperwürfel gleich d ist. Die durchschnittliche Weglänge in einem Hyperwürfel ist ungefähr $d/2$. Es gibt eine Reihe kommerzieller Systeme mit einer Hyperwürfel Struktur. Als Beispiele seien hier der "Cosmic Cube", als historische auf 8086 Prozessoren basierende Architektur, und dessen kommerzielle Variante, der iPSC-1, mit 80286 Prozessoren genannt. Der iPSC-1 konnte mit 16 bis 128 Prozessoren konfiguriert werden, was einem 3 bis 7 dimensionalen Hyperwürfel entspricht.

Hyperwürfel sind sehr mächtige Topologien, auf denen viele parallele Algorithmen eine "optimale" Laufzeit haben. Weiterhin lassen sich die meisten anderen Topologien auf einem Hyperwürfel einbetten oder zumindest effizient simulieren. Aus diesen Gründen erlangten Hyperwürfel ein großes Interesse in der Literatur. Problematisch ist, daß Hyperwürfel mit vielen Prozessoren relativ komplex sind. Die Zahl der Leitungen im System und auch die Zahl der Ports pro Prozessor wächst mit der Zahl der Prozessoren und ist ab einer bestimmten Dimensi-

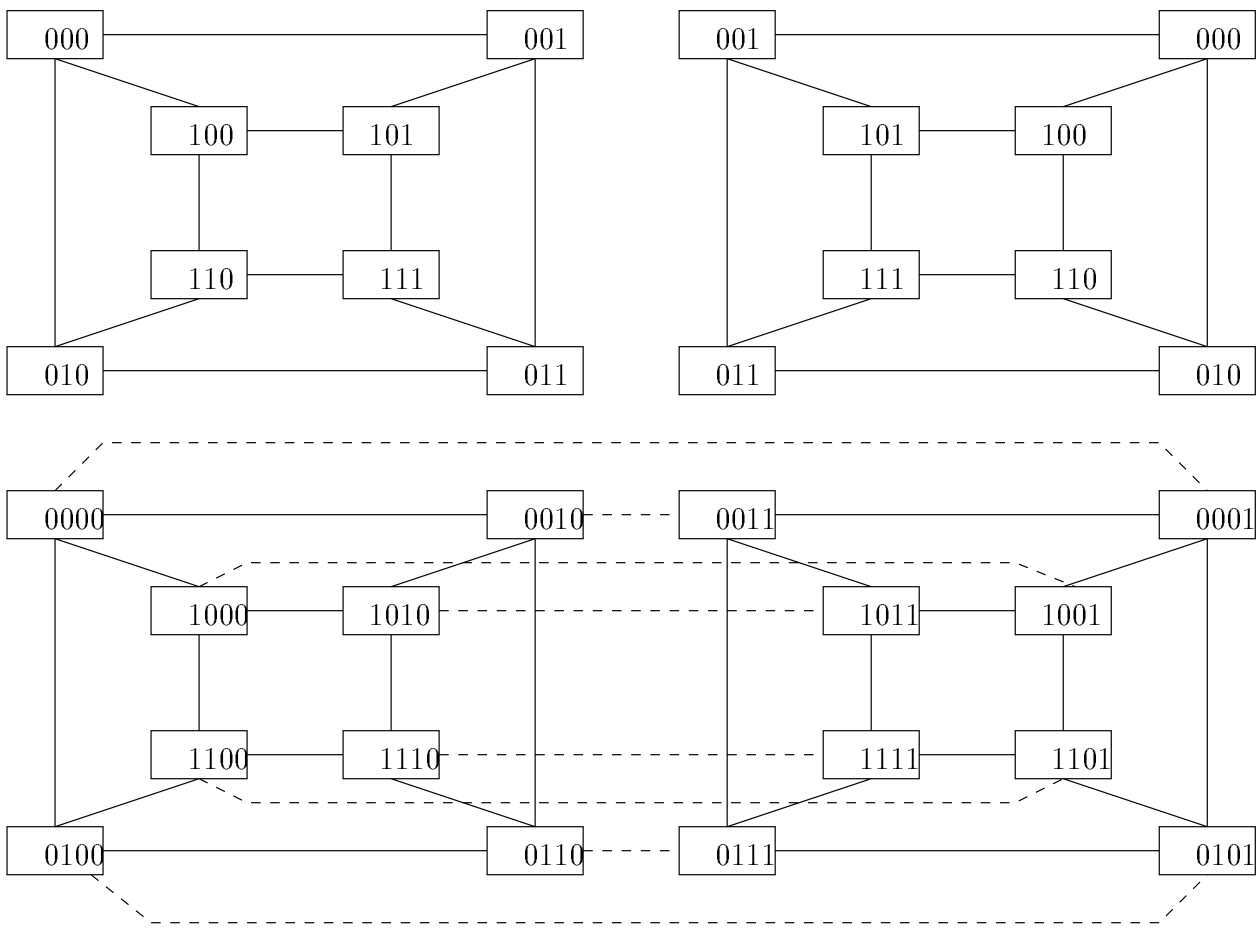


Abbildung 2.8: Konstruktion eines Hyperwürfels der Dimension 4.

on nur noch mit hohem Aufwand physikalisch realisierbar. Neuere hochparallele Architekturen verwenden die Verbindungsstruktur deshalb nicht mehr. Aus diesem Grund wurden alternative Verbindungsstrukturen realisiert, die Hyperwürfel mit einem noch akzeptablem Zusatzaufwand simulieren können und weniger komplex sind. Ein Beispiel für eine solche Architektur ist der Cube Connected Cycle (CCC). Ein CCC der Dimension d entsteht aus einem Hyperwürfel der Dimension d, indem jeder Prozessor durch einen Ring von d Prozessoren ersetzt wird. Jeder Prozessor ist mit seinen Nachbarn im Ring verbunden und realisiert eine der ursprünglichen Verbindungen des Hyperwürfels. Damit besitzt jeder Prozessor nur 3 Ports, was die physikalische Realisierung deutlich vereinfacht. Der Preis für die einfachere Struktur ist ein erhöhter Kommunikationsaufwand.

Fazit

Wir werden im folgenden im wesentlichen Systeme mit statischen Verbindungsstrukturen betrachten, auch wenn in neuerer Zeit geschaltete Verbindungen durch neuere Hardware eine größere Bedeutung erlangen. In den nächsten Kapiteln werden einige Aspekte statischer Verbindungsstrukturen erläutert und verglichen.

An dieser Stelle soll das zugrundeliegende Modell noch einmal konkretisiert werden. Wir nehmen an, daß ein festes Verbindungsnetz aus uni- oder bidirektionalen Leitungen zwischen den einzelnen Prozessoren existiert. Topologien lassen sich als (gerichtete) Graphen (N, A) interpretieren. Die Knoten beschreiben die einzelnen Prozessoren, eine Kante $(i, j) \in A$ bezeichnet eine uni- oder bidirektionale Verbindung zwischen den Prozessoren i und j. Wenn nicht anders gesagt, werden wir im folgenden immer von bidirektionalen Verbindungen zwischen Prozessoren ausgehen. Der resultierende Graph ist damit ungerichtet.

2.4 Bewertungsaspekte paralleler Architekturen

In diesem Kapitel werden unterschiedliche Aspekte der Bewertung paralleler Architekturen vorgestellt. Begonnen wird mit einigen Maßen, die die potentielle Leistung eines konkreten Multiprozessors wiedergeben und bei der Vorstellung solcher Architekturen zu finden sind. Daran anschließend betrachten wir zwei Aspekte, die eine etwas abstraktere Sichtweise einnehmen. Es werden Bewertunsaspekte bzgl. des Topologiegraphen vorgestellt. In 2.4.2 werden Bewertungskriterien für die Graphstruktur eingeführt. Abschließend wird der Aspekt der Einbettung einer Topologie in eine andere beschrieben.

2.4.1 Leistungsmaße für Parallelrechner

Es gibt eine Reihe von Maßen für die Darstellung der Leistung von Rechnerarchitekturen. Leistungsmaße werden extensiv in der Werbung eingesetzt, sind aber immer zu hinterfragen. Der Komplex der Leistungsbeschreibung und Messung kann und soll an dieser Stelle nicht vollständig behandelt werden. Es wird nur ein kleiner Ausschnitt dargestellt, der insbesondere für die Vergleichbarkeit verschiedener Architekturen bzgl. der Laufzeit numerischer Algorithmen bedeutsam ist. Die Bewertung einer Rechnerarchitektur bzgl. anderer Anwendungsgebiete kann zu vollkommen anderen Ergebnissen führen.

Am weitesten verbreitet ist die Taktfrequenz eines Prozessors, die die Anzahl der Zyklen pro Sekunde angibt. Je höher die Taktfrequenz ist, desto mehr Operationen pro Sekunde können durchgeführt werden. Taktfrequenzen können verwendet werden, um Prozessoren eines Typs zu vergleichen. Es ist allerdings nicht oder nur sehr eingeschränkt möglich, verschiedene Systeme anhand der Taktfrequenz zu vergleichen. Dies hat zwei Gründe. Zum einen ist die Leistung eines

Systems nicht nur von der Prozessorleistung, sondern auch von der Speicherzugriffszeit und der Übertragungsbandbreite der Kommunikationspfade im System abhängig. Zum anderen sagt die Taktfrequenz nichts darüber aus, was während eines Zyklus passiert. Insbesondere wenn sich die Instruktionssätze deutlich unterscheiden, sind Vergleiche anhand der Taktfrequenz sinnlos. Vergleiche sind nur sinnvoll, wenn die verschiedenen Instruktionssätze in die Betrachtung einbezogen werden und ein Maß für die Anzahl der Zyklen per abstrakter Instruktion definiert wird. Dies setzt voraus, daß ein abstrakter Instruktionssatz existiert und verschiedene Systeme auf Basis dieses Instruktionssatzes verglichen werden.

In ähnlicher Form ist der Vergleich von Systemen anhand von *MIPS* (millions instructions per second) sinnlos, ohne die Art der Instruktionen genauer zu definieren. Eine solche Definition kann z.B. durch die Ausrichtung an einer bekannten Architektur erfolgen. Verwendet wird so die Definition von "VAX-Mips", welche die Leistung einer VAX 11/780 als Basis ansieht. Der Vergleich zu anderen Architekturen erfolgt i.a. durch den Laufzeitvergleich einiger Programme und hängt damit auch von der Auswahl der Programme ab.

Für den Vergleich der numerischen Leistung von Prozessoren wird oft der Vergleich von *MFLOPS* (millions of floating point operations per second) herangezogen. Die sogenannte "peak rate" in MFLOPS gibt an, wieviele Fließkommaoperationen das System theoretisch maximal durchführen kann. Zur Berechnung wird der Kehrwert der Ausführungszeit einer Fließkommaoperation verwendet. Auch dieses Maß ist sehr künstlich, da nur die reine Berechnungszeit verglichen wird. Die tatsächliche Leistung hängt aber auch von der Abarbeitung nicht numerischer Operationen und der Speicherzugriffszeit ab. Die angegebene peak Rate in MFLOPS ist deshalb in allen Fällen deutlich größer als die tatsächlich erreichbare Zahl von Fließkommaoperationen.

Eine etwas detailliertere Analyse erlaubt den Vergleich verschiedener Architekturen, indem die Laufzeit standardisierter Programme verglichen wird. Für diesen Zweck werden sogenannte Benchmark Programme verwendet. Zum Vergleich der numerischen Leistungsfähigkeit von Architekturen wird insbesondere LINPACK (linear algebra package) verwendet. LINPACK ist eine Sammlung numerischer Programme für Probleme der linearen Algebra und ist ursprünglich in FORTRAN geschrieben. Es wird i.a. die Zeit zur Lösung eines linearen Gleichungssystems mit 100 oder 1000 Variablen unter Verwendung des Algorithmus zur Gauß Elimination aus LINPACK verwendet. Eine größere Sammlung von Benchmark Programmen ist unter dem Namen SPEC (System Performance Evaluation Cooperative) zu finden. Oft wird die Leistung eines Rechners in SPECs angegeben. Dieser Wert wird aus der Zeit berechnet, die für die Abarbeitung einer vorgegebenen Auswahl des SPEC Benchmarks notwendig ist. Der Vergleich auf Grund der Laufzeit bestimmter Programme ist natürlich deutlich aussagekräftiger als die Angabe der peak Rate in MFLOPS oder der Taktfrequenz des Prozessors. Trotzdem muß man auch bei dieser Art von Leistungsangaben vorsichtig sein. So zeigt sich, daß die Leistung oft stark vom verwendeten Compiler abhängt, und daß verschiedene Hersteller Compiler und auch Architekturen bzgl. bestimmter Benchmarks optimieren, womit ein fairer Leistungsvergleich nicht möglich ist. Es ist deshalb notwendig, den verwendeten Compiler und die Compileroptionen zusätzlich zu den Laufzeiten anzugeben. Weiterhin spielt die verwendete Zahlendarstellung eine Rolle. Standardisierte Vergleiche basieren i.a. auf einer 64 Bit Darstellung für reelle Zahlen. Ein weiteres Problem, welches beim Vergleich paralleler Architekturen auftaucht liegt darin begründet, daß LINPACK und auch der SPEC Benchmark für sequentielle Maschinen entworfen wurden. Für den Vergleich paralleler Architekturen müssen die Programme parallelisiert werden. Diese ist nicht einfach und führt natürlich zu nicht standardisierten Versionen und unter Umständen schwer vergleichbaren Leistungsmaßen. Aus diesen Gründen sind Vergleichsergebnisse für parallele Rechner rar und nur unter Vorbehalt zu interpretieren. In Tab. 2.2 wird die Leistung einiger Rechnerarchitekturen verglichen. Die erste Spalte beinhaltet den Namen der

Architektur, die zweite Spalte die Prozessorzahl, die dritte Spalte den Rechnertyp, die vierte Spalte die Rate an Floating Point Operationen (in *MFLOPS* (= Millionen Floating Point Operationen) pro Sekunde), die bei der Lösung eines Gleichungssystems mit 100 Variablen mittels der Gauß Elimination aus LINPACK erreicht wurde. Dabei darf die Standardimplementierung aus LINPACK nicht verändert werden. Parallelisierung wird dadurch erreicht, daß die Gauß-Elimination auf Matrix- und nicht auf Elementebene realisiert wird (siehe auch Kapitel 5). Die fünfte Spalte gibt die Rate an Floating Point Operationen (in MFLOPS pro Sekunde) bei der Lösung eines Gleichungssystems mit 1000 Variablen mit einem optimierten Algorithmus an. Der Algorithmus darf speziell für die Architektur optimiert werden, muß aber, wie die Gauß Elimination, $2n^3/3 + 2n^2$ Operationen durchführen und muß mit 64 Bit Darstellung für reelle Zahlen arbeiten. Die letzte Spalte zeigt die peak Rate an Floating Point Operationen des entsprechenden Rechners. Alle Raten werden in *MFLOPS* pro Sekunde angegeben. In der Tabelle sind Resultate für ältere und neuere Architekturen gemischt, damit wird auch die Leistungssteigerung im Rechnerbereich in den letzten Jahren deutlich.

Rechner	Proz.	Typ	Linpack	Optimiert	$MFLOPS_{peak}$
Compaq 486	1	SISD	1.4		
Pentium Pro (200MHz)	1	SISD	38		200
Sun UltraSparc(167Mhz)	1	SISD	70	237	333
Sun UltraSparc(167Mhz)	24	MISD		3566	8000
Sun UltraSparc(336Mhz)	1	SISD	154		672
Sun UltraSparc(336Mhz)	2	MIMD get. Sp.		843	1344
Sun UltraSparc(336Mhz)	4	MIMD get. Sp.		1438	2688
Sun UltraSparc(336Mhz)	8	MIMD get. Sp.		2481	5376
Sun UltraSparc(336Mhz)	16	MIMD get. Sp.		3981	10752
Sun UltraSparc(336Mhz)	24	MIMD get. Sp.		4755	16128
Fujitsu VP-200	1	Vektorproz.	18	422	533
Cray C90	2	MIMD get. Sp.	387	1780	1905
Cray C90	16	MIMD get. Sp.	479	10780	15238
Meiko CS 2	64	MIMD vert. Sp.		652	11520
Intel iPSC/860	128	MIMD vert. Sp.		219	5120
Intel iPSC/Delta	128	MIMD vert. Sp.		393	5120

Tabelle 2.2: Leistungsfähigkeit einiger Rechnerarchitekturen bei der Lösung linearer Gleichungssysteme.

Die einzelnen Daten sollten nicht überinterpretiert werden, interessant ist aber ein Vergleich der Werte in einer Zeile. Man sieht, daß die peak Leistung gerade bei Multiprozessorsystemen sehr weit von der tatsächlich nutzbaren Leistung entfernt ist. Das Verhältnis zwischen maximaler und tatsächlich erreichbarer Leistung wird natürlich besser wenn die Problemgröße zunimmt und der Algorithmus für eine Architektur optimiert wird. Dies wird in Tabelle 2.3 deutlich, in der die maximale Leistung in GFLOPS für die Lösung eines linearen Gleichungssystems mit einem Verfahren, welches $2n^3/3 + 2n^2$ Schritte für ein System mit n Variablen benötigt, angegeben ist. Ein solches Lösungverfahren wäre die Gauß-Elimination mit Pivotisierung. In der Tabelle gibt R_{max} die maximal erreichte Zahl von GFLOPS an und n_{max} die Problemgröße, bei der R_{max} erreicht wird. $n_{1/2}$ gibt die Problemgröße an, bei der die Hälfte von R_{max} erreicht wird. R_{peak} ist wieder die peak Rate des Systems.

Die Ergebnisse verdeutlichen, daß bei genügend großen Problemen die maximale Rate zumindest im Bereich der Hälfte der peak Rate liegt. Interessant ist der Vergleich der beiden Intel

Rechner	Proz.	Typ	R_{max}	n_{max}	$n_{1/2}$	R_{Peak}
Intel ASCI (200MHz Pentium Pro)	9152	MIMD vert. Sp.	1338	235000	63000	1830
Cray T3E-1200E	1080	MIMD vert. Sp.	891.5	259200	53400	1296
SunUltra HPC-Cluster	256	MIMD vert. Sp.	100.4	80640	22528	128
Fujitsu VPP500	128	MIMD vert. Sp.	149	40960	13120	205
Meiko CS 2	64	MIMD vert. Sp.	5	18688	6144	11.5
Intel iPSC/860	128	MIMD vert. Sp.	2.6	12000	4500	5.12
Intel iPSC/Delta	128	MIMD vert. Sp.	3.5	12500	3500	5.12

Tabelle 2.3: Maximal erreichbare Leistung einiger Parallelrechner (in GFLOPS/sec).

Architekturen, die sich nur durch das Verbindungsnetz unterscheiden. Im iPSC/860 findet eine Hyperwürfelstruktur Verwendung, die sich allerdings als recht langsam herausgestellt hat. Im iPSC/Delta wurde der Hyperwürfel durch ein Gitter ersetzt, welches schnellere Kommunikation erlaubt. Durch diese Änderung treten signifikante Unterschiede in der tatsächlichen Leistungsfähigkeit auf.

Die dargestellten Leistungsmaße resultieren aus der Rechen- und der Kommunikationszeit, die der Algorithmus verbraucht. Daß die Kommunikationszeit einen signifikanten Teil der Gesamtzeit ausmachen kann, sieht man am Vergleich der beiden Intel Architekturen. Beide Architekturen haben identische Prozessorleistungen und der iPSC/860 bietet als Hyperwürfel sogar eine vom theoretischen Standpunkt bessere Verbindungsstruktur als der iPSC/Delta. Trotzdem liefert der iPSC/Delta durch die schnellere Kommunikation eine deutlich bessere Leistung. Kommunikationszeiten werden, im Gegensatz zur Prozessorleistung, meist nicht angegeben und hängen auch vom verwendeten Betriebssystem ab. Im allgemeinen setzt sich die Kommunikationszeit T_K aus zwei Komponenten zusammen, der Startzeit T_S, die zum Aufbau der Verbindung benötigt wird, und der Übertragungszeit T_U zur Datenübertragung. T_S ist unabhängig von der Menge zu übertragender Daten und macht für kleinere Datenmengen einen großen Teil der Kommunikationszeit aus. Für moderne Multiprozessorsysteme liegt T_S im Bereich von $70 - 300\mu sek$. Systeme mit single-user Betrieb auf den einzelnen Prozessoren sind dabei meist deutlich schneller als multi-user Systeme. Es sind aber auch durchaus Zeiten im Millisekundenbereich für T_S anzutreffen, wenn konventionelle Betriebssysteme wie Unix verwendet werden. Die Übertragungszeit T_U hängt von der Verbindungsbandbreite ab, üblich sind heute Bandbreiten von $50 - 100 Mbit/sek$, in Zukunft sind auch noch deutlich größere Bandbreiten möglich. Wenn man nun die Rechenleistung in Fließkommaoperationen, die bei heutigen Prozessoren bei ungefähr $500 MFLOPs$ liegt, in Bezug zur Übertragungszeit setzt, so sieht man, daß die Übertragung einer Fließkommazahl soviel Zeit wie 500000 Fließkommaoperationen benötigen kann, wenn die Relation der Parameter ungünstig ist. Dieses Verhältnis ändert sich, wenn große Datenmengen übertragen werden, zeigt aber die Bedeutung einer minimierten Kommunikation zur Optimierung der Leistung einer parallelen Anwendung.

2.4.2 Bewertung von Topologien

Bewertet wird an dieser Stelle im wesentlichen die Graphstruktur, die meist als Funktion der Knotenzahl gesehen werden kann. Wir definieren zuerst einige Grapheigenschaften, die im Zusammenhang mit parallelen Architekturen von Interesse sind.

a) Die *Kantenzahl* im Graph gibt die Anzahl der benötigten Leitungen an.

b) Die *Symmetrie* gibt an, ob der Graph aus Sicht jedes einzelnen Knotens gleich aussieht.

c) Die Anzahl der *Ports* pro Prozessor beschreibt die maximale Zahl von Leitungen, die von einem Prozessor ausgehen. Die Zahl der Ports ist aus technologischen Gründen beschränkt.

d) Die *maximale Weglänge* (diameter) beschreibt die Länge des kürzesten Weges zwischen zwei beliebigen Prozessoren. Manchmal wird auch die *durchschnittliche Weglänge* untersucht, die sich aus der Summe über die Längen der kürzesten Wege zwischen allen möglichen Prozessorpaaren dividiert durch die Anzahl der Prozessorpaare ergibt. Die Maße beschreiben implizit die maximale/durchschnittliche Übertragungszeit einer Nachricht ohne Berücksichtigung möglicher Wartezeiten .

e) Die *Konnektivität* (connectivity) gibt die Anzahl der Leitungen an, die mindestens ausfallen müssen, damit zwei beliebige Prozessoren nicht mehr kommunizieren können. Die Konnektivität ist ein Maß für die Fehlertoleranz der Topologie und entspricht der minimalen Anzahl von unabhängigen Wegen zwischen zwei beliebigen Prozessoren. Es wird manchmal auch die maximale Anzahl der Leitungen untersucht, die ausfallen dürfen, ohne die Kommunikation zwischen beliebigen Prozessoren zu unterbrechen. Auf Ebene des Topologiegraphen beschreiben die Maße die minimale/maximale Anzahl von Kanten, die entfernt werden können und der Graph trotzdem stark zusammenhängend bleibt. Neben dem Ausfall von Kanten können in analoger Weise auch Knotenausfälle untersucht werden.

f) Die *Skalierbarkeit* beschreibt die Möglichkeit, die Anzahl der Prozessoren zu erhöhen. In skalierbaren Topologien ist dies mit geringem Aufwand möglich. Die Anzahl der Ports pro Prozessor sollte in skalierbaren Topologien unabhängig von der Anzahl der Prozessoren im Gesamtsystem sein. Zumindest sollte aber das logische Verhalten und die maximale Weglänge nicht oder kaum von der Prozessorzahl abhängen.

g) Das *Routingverhalten* beschreibt die Komplexität des Routingalgorithmus. Es ist anzustreben, daß der Routingalgorithmus möglichst einfach ist und sich der nächste zu wählende Prozessor auf einem Pfad direkt aus der Zielprozessoradresse ableiten läßt.

h) Die *Flexibilität* beschreibt die Möglichkeit, andere Topologien einzubetten. Ein Graph $G' = (N', A')$ kann in einen Graph $G = (N, A)$ eingebettet werden, wenn eine Funktion $\sigma : N' \to N$ existiert, mit $\sigma(i) \neq \sigma(j)$ für $i \neq j$ und $(\sigma(i), \sigma(j)) \in A$ für alle $(i, j) \in A'$. Existiert eine Einbettung von G' in G, so kann jeder für G' definierte Algorithmus mit gleichem Aufwand auf G laufen (wobei identische Prozessorgeschwindigkeiten und Übertragungsraten vorausgesetzt werden). Einige Einbettungen werden im nächsten Abschnitt vorgestellt.

i) Die *Kommunikationszeit* einer Datenübertragung in einer Topologie ist natürlich vom Problem abhängig, d.h. welche Daten zwischen welchen Prozessoren übertragen werden sollen. Die maximale Übertragungszeit eines einzelnen Pakets zwischen zwei beliebigen Prozessoren ist durch die maximale Weglänge gegeben. Komplexere Kommunikationsprobleme müssen auch berücksichtigen, daß u.U. Wartezeiten auftreten, da mehr als ein Paket zu einem Zeitpunkt über eine Leitung übertragen werden soll. Wir werden später einige Basiskommunikationsprobleme definieren, die in vielen numerischen Algorithmen auftreten.

Tab. 2.4 zeigt die Eigenschaften einiger Topologien bzgl. der oben genannten Kriterien. Es werden Ergebnisse für eine Topologie mit N Knoten dargestellt, dabei zeigen sich deutliche Unterschiede zwischen den einzelnen Topologien.

Topologie	Kantenzahl	Ports	Diameter	Konnektivität	Symmetrie
1D-Array	$N-1$	2	$N-1$	1	N
Ring	N	2	$\lfloor N/2 \rfloor$	2	Y
Bin. Baum	$N-1$	3	$2((\log_2 N)-1)$	1	N
2D-Array	$2N-2\sqrt{N}$	4	$2(\sqrt{N}-1)$	2	N
2D-Torus	$2N$	4	$2\lfloor \sqrt{N}/2 \rfloor$	4	Y
Hyperwürfel	$(N \log_2 N)/2$	$\log_2 N$	$\log_2 N$	$\log_2 N$	Y
vol. vermascht	$N(N-1)/2$	$N-1$	1	$N-1$	Y

Tabelle 2.4: Eigenschaften einiger typischer Topologien.

2.4.3 Einbettung von Topologien

Einer der Bewertungsaspekte, die genannt wurden, war die Flexibilität einer Topologie, d.h. die Möglichkeit, in einer Topologie eine andere einzubetten. Wenn dieses möglich ist, so können Algorithmen, die für die eingebettete Topologie entworfen wurden, direkt auf die einbettende Topologie übernommen werden. Damit wird zumindest eine partielle Unabhängigkeit der Algorithmen von der Architektur erreicht. Die Einbettung von Topologien ist im wesentlichen ein Graph Problem und es existieren zahlreiche Ergebnisse in der Literatur. Wir werden hier nur einige wenige Beispiele untersuchen, um das prinzipielle Vorgehen kennenzulernen.

Eine offensichtliche Einbettung, ist die Einbettung eines linearen Arrays in ein d-dimensionales Array mit gleicher oder größerer Prozessorzahl. Als Beispiel betrachten wir ein lineares Array mit $p = n^d$ Prozessoren und ein Array der Dimension d wobei die Indizes in jeder Dimension zwischen 1 und n variieren. Jeder Prozessor im d-dimensionalen Array bekommt eine d-dimensionale Adresse $(x_1, \ldots, x_d)$ mit $1 \leq x_i \leq n$. Wir wählen nun die folgende Anordnung der Prozessoren

$$(1, \ldots, 1), (1, \ldots, 1, 2), \ldots, (1, \ldots, 1, n), (1, \ldots, 2, n), \ldots, (1, \ldots, 2, 1), (1, \ldots, 3, 1), \ldots, (n, \ldots, n)$$

und ordnen ihnen konsekutiv die Prozessoren des linearen Arrays zu.

Das Umkehrproblem zur Einbettung eines niedrig dimensionalen Array auf einem höher dimensionalen Array ist die Simulation des höher dimensionalen Arrays auf dem Array niedrigerer Dimension. Eine Einbettung ist in diesem Fall natürlich nicht möglich, da das höher dimensionale Array mehr Verbindungen definiert, als im Array niedriger Dimension vorhanden sind. Sei $\mathcal{A}_s$ ein s-dimensionales $n \times n \times \ldots \times n$ Array. $\mathcal{A}_s$ beinhaltet n^s Prozessoren. Wir wollen $\mathcal{A}_s$ auf einem Array $\mathcal{A}_r$ mit $r < s$ simulieren. Die Simulation ist recht einfach. Prozessor $(x_1, x_2, \ldots, x_r)$ simuliert die Prozessoren $(y_1, y_2, \ldots, y_s)$, falls $x_i = y_i$ für $1 \leq i \leq r$. Das niedriger dimensionale Array benötigt maximal n^{s-r} Schritte, um einen Schritt auf dem höher dimensionalen Array zu simulieren. Dies liegt darin begründet, daß benachbarte Prozessoren durch den selben oder benachbarte Prozessoren simuliert werden. Dadurch wird das Versenden einer Nachricht durch das Versenden keiner oder einer Nachricht simuliert, woraus folgt, daß der Kommunikationsaufwand maximal um den Faktor n^{r-s} höher sein kann, gleiches gilt für die Anzahl der Rechenschritte. Da das ursprüngliche Array n^{r-s} mal soviele Prozessoren hat und die Geschwindigkeit sich gleichzeitig maximal um den Faktor n^{r-s} erhöht, ist die Effizienz besser oder identisch. Klarerweise müssen die Prozessoren über genügend Speicher verfügen, um die Daten der n^{r-s} Prozessoren gleichzeitig zu speichern. Die vorgestellte Simulation beschreibt einen allgemeinen Ansatz zur Lösung großer hoch-dimensionaler Probleme auf kleineren niedriger-dimensionalen Topologien.

Offensichtlich kann in einen voll vermaschten Graph jede andere vorgestellte Topologie eingebettet werden. Der voll vermaschte Graph ist aber keine realisierbare Topologie, wenn größere

Anzahlen von Prozessoren verbunden werden sollen. Realistischer ist die Verwendung von Hyperwürfeln, auch wenn diese für große Prozessorzahlen viele Ports pro Prozessor benötigen. Wir stellen deshalb Einbettungen anderer Topologien in Hyperwürfel vor und betrachten dazu einen Hyperwürfel der Dimension d mit $p = 2^d$ Prozessoren.

Wir beginnen mit der Einbettung der einfachsten Topologien, dem linearen Array und dem Ring. Jeder Hyperwürfel mit $p \geq 4$ Prozessoren beinhaltet ein lineares Array und einen Ring mit p Prozessoren als Untergraphen. Um die beiden Topologien in einen Hyperwürfel einzubetten, müssen $p = 2^d$ unterschiedliche Binärzahlen erzeugt werden, die sich jeweils in einem Bit unterscheiden. Solche Sequenzen sind aus der Kodierungstheorie bekannt und werden als *Gray Codes* bezeichnet. Wir werden hier im speziellen den reflektierenden *Gray Code* (*RGC*) betrachten, dessen Konstruktion ähnlich zur Konstruktion eines d-dimensionalen Hyperwürfels aus zwei $d-1$-dimensionalen ist.

Ein 1 Bit *RGC* ist die Sequenz $\{0,1\}$. Durch Einfügen einer führenden 0 und einer führenden 1 erhalten wir die Sequenzen $\{00,01\}$ und $\{10,11\}$. Die zweite Sequenz wird gespiegelt und anschließend werden beide Sequenzen aneinandergehängt, wodurch $\{00,01,11,10\}$, der 2 Bit *RGC*, entsteht. Sei $\{b_1, b_2, \ldots, b_p\}$ ein $d-1$ Bit *RGC* ($p = 2^{d-1}$), dann ist $\{0b_1, \ldots, 0b_p, 1b_p, \ldots, 1b_1\}$ ein d Bit *RGC*.

Offensichtlich unterscheiden sich aufeinanderfolgende Elemente eines *RGC* in genau einem Bit und auch das letzte und erste Element unterscheiden sich in genau einem Bit. Zur Einbettung des Arrays werden die Prozessoren konsekutiv mit den Bitsequenzen des RGCs numeriert und dem durch die jeweilige Bitsequenz adressiertem Prozessor eines Hyperwürfels zugeordnet. Damit kann jedes lineare Array mit p Prozessoren in einen Hyperwürfel der Dimension $\lceil \log p \rceil$ eingebettet werden. Bei Ringen ist die Situation etwas schwieriger. Ein Ring mit $p = 2^d$ Prozessoren kann in einen Hyperwürfel der Dimension d eingebettet werden, falls $d > 1$. Man kann zeigen, daß Ringe mit einer geraden Anzahl Prozessoren, die größer als 4 ist, in einen Hyperwürfel mit $2^{\lceil \log N \rceil}$ Prozessoren eingebettet werden können. Die Einbettung von Ringen mit 2 Prozessoren oder einer ungeraden Anzahl von Prozessoren ist dagegen nicht möglich. Die Einbettung eines Rings mit 16 Prozessoren in einen Hyperwürfel der Dimension 4 wird in Abb. 2.9 gezeigt. Für die Einbettung werden nur die fett gedruckten Verbindungen benötigt.

Zur Einbettung höher dimensionaler Arrays wollen wir das Konzept der *RGC* etwas erweitern. Seien d_a und d_b positive ganze Zahlen. Sei ferner $d = d_a + d_b$, sowie $\{a_1, \ldots, a_{p_a}\}$ und $\{b_1, \ldots, b_{p_b}\}$ die d_a, d_b *RGC* Sequenzen ($p_a = 2^{d_a}$, $p_b = 2^{d_b}$). Aus dem Kreuzprodukt der beiden Sequenzen kann eine Matrix der Bit Sequenzen $\{a_i b_j | 1 \leq i \leq p_a, 1 \leq j \leq p_b\}$ geformt werden.

$$\begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_{p_b} \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_{p_b} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p_a} b_1 & a_{p_a} b_2 & \cdots & a_{p_a} b_{p_b} \end{pmatrix}$$

Diese Matrix kann nun sequentiell von links nach rechts und oben nach unten durchlaufen werden, wie im folgenden Beispiel gezeigt.

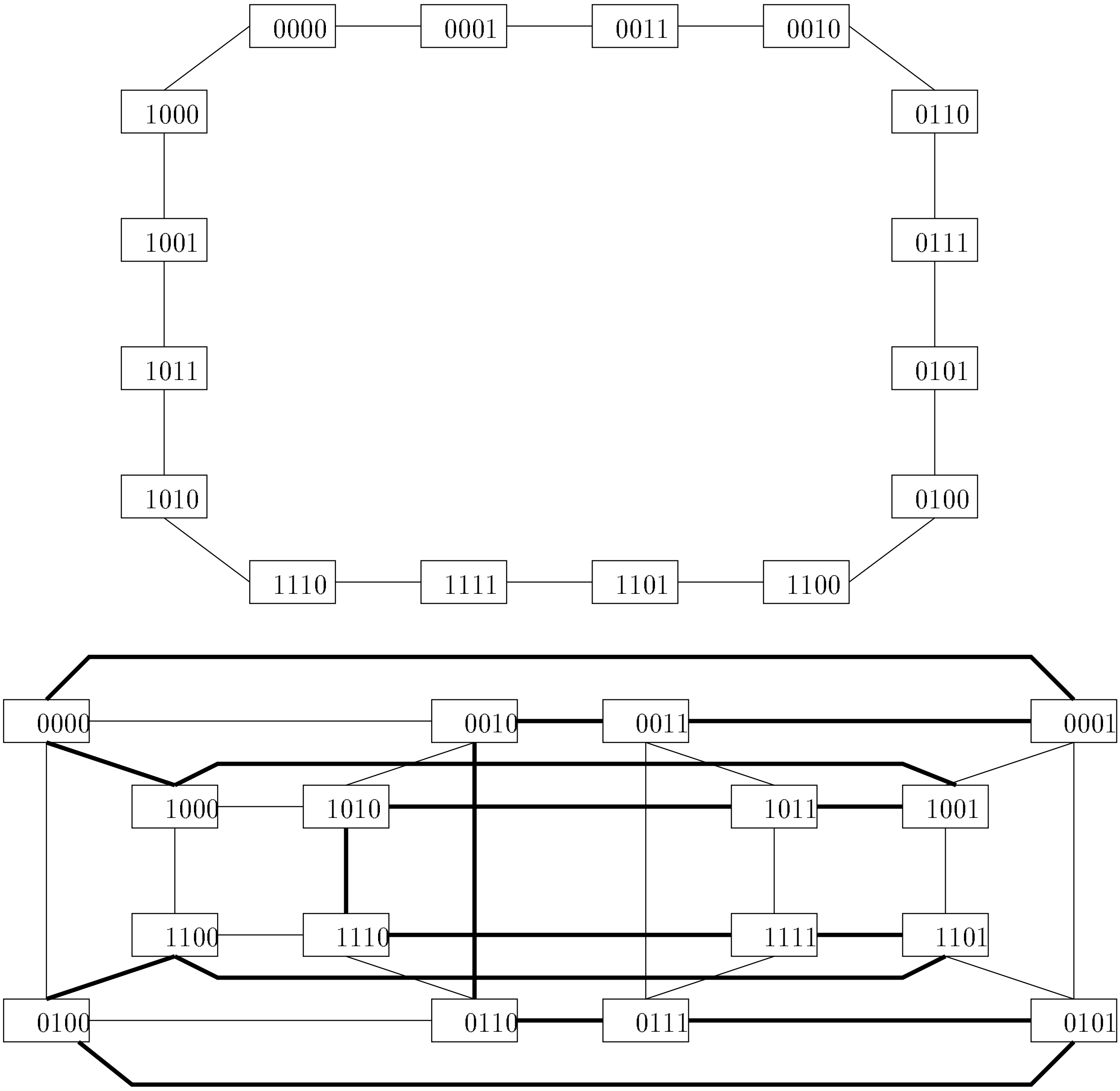


Abbildung 2.9: Einbettung eines Rings in einen Hyperwürfel.

$$\begin{pmatrix} a_1b_1 & \rightarrow & a_1b_2 & \rightarrow & \cdots & \rightarrow & a_1b_{p_b} \\ & & & & & & \downarrow \\ a_2b_1 & \leftarrow & a_2b_2 & \leftarrow & \cdots & \leftarrow & a_2b_{p_b} \\ \downarrow & & & & & & \\ a_3b_1 & \rightarrow & a_3b_2 & \rightarrow & \cdots & \rightarrow & a_3b_{p_b} \\ & & & & & & \downarrow \\ \vdots & & \vdots & & \ddots & & \vdots \\ & & & & & & \downarrow \\ a_{p_a}b_1 & \leftarrow & a_{p_a}b_2 & \leftarrow & \cdots & \leftarrow & a_{p_a}b_{p_b} \end{pmatrix}$$

Das Ergebnis des Durchlaufs ist ein d-Bit *RGC*, da sich sukzessive Bit Sequenzen in höchstens einem Bit unterscheiden. Die obige Methode kann klarerweise auf das Kreuzprodukt von n *RGC* Sequenzen erweitert werden, d.h. aus n *RGC* Sequenzen kann, durch wiederholte Kreuzproduktbildung, eine *RGC* Sequenz der Dimension $d = \sum_{i=1}^{n} d_i$ (d_i ist die Dimension von Sequenz i) gebildet werden.

Aus der allgemeinen Konstruktion von *RGCs* läßt sich eine Methode zur Einbettung von k-dimensionalen Arrays in Hyperwürfel herleiten. Betrachten wir zuerst ein k-dimensionales Array, in dem jede Komponente Werte zwischen 1 und $n_i = 2^{d_i}$ annehmen kann. Einen Prozessor mit Adresse $(x_1, \ldots, x_k)$ $(1 \leq x_i \leq n_i)$ im Array bilden wir auf den Prozessor mit Adresse $s_1 s_2 \ldots s_k$ im Hyperwürfel ab. s_i ist der d_i Bit *RGC* welcher zum isolierten Element x_i gehört. Im Array benachbarte Prozessoren differieren in genau einer Koordinate x_i um genau 1. Wie für lineare Arrays gezeigt, differiert in diesem Fall auch der *RGC* um genau ein Bit in der Sequenz s_i. Damit differieren auch die *RGCs* der Prozessoren $(x_1, \ldots, x_{i-1}, n_i, x_{i+1}, \ldots, x_k)$ und $(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_k)$ in genau einem Bit. Abb. 2.10 zeigt die Einbettung eines 4×4 Arrays in einen Hyperwürfel der Dimension 4. Es ist beachtenswert, daß jedes Unterarray, welches dadurch entsteht, daß einzelne Koordinaten der Prozessoradressen im Array festgehalten werden, in einen eigenen Hyperwürfel eingebettet wird. Diese Hyperwürfel sind Unterwürfel des ursprünglichen Hyperwürfels und sind disjunkt für die Unterarrays, die aus der Zerlegung eines Arrays entstehen.

Betrachten wir nun den allgemeinen Fall, in dem n_i keine Potenz von 2 ist. Ein erster Versuch wäre die Einbettung in einer Hyperwürfel der Dimension $d = \lceil \log(\prod_{i=1}^{k} n_i) \rceil$, der offensichtlich genug Prozessoren beinhaltet. Leider ist eine solche Einbettung nicht allgemein möglich, ein Gegenbeispiel wäre der Versuch ein 3×5 Array in einen 4-dimensionalen Hyperwürfel einzubetten. Stattdessen wird ein Hyperwürfel der Dimension $d = \sum_{i=1}^{k} \lceil \log n_i \rceil$ benötigt. Die Einbettung erfolgt durch Bildung des $d_i = \lceil \log n_i \rceil$ Bit *RGCs* für die Koordinate x_i der Prozessoradresse im Array. Das Vorgehen wurde oben für $d_i = 2^{n_i}$ beschrieben. Dies bedeutet, daß u.U. ein Hyperwürfel mit einer deutlich größeren Prozessorzahl als das ursprüngliche Array benötigt wird. Zur Einbettung eines 3×5 Arrays mit 15 Prozessoren wird ein 5-dimensionaler Hyperwürfel mit 32 Prozessoren benötigt. Nehmen wir in Kauf, daß einzelne Verbindungen, die im Array zwischen benachbarten Prozessoren verlaufen, im Hyperwürfel über einen Zwischenknoten geleitet werden, so ist eine Einbettung in einen Hyperwürfel der Dimension 4 möglich. Man kann allgemein zeigen (ohne, daß wir es tun), daß jedes zweidimensionale Array in einen Hyperwürfel der Dimension $d = \lceil \log(n_1 n_2) \rceil$ eingebettet werden kann und Wege zwischen benachbarten Prozessoren im Array über maximal einen Zwischenknoten im Hyperwürfel verlaufen. In welcher Form sich dieses Resultat auf höhere Dimensionen übertragen läßt ist nicht bekannt.

Als nächstes wollen wir die Einbettung eines Binärbaums in einen Hyperwürfel betrachten. Auf Grund der Struktur des Hyperwürfels liegt die Vermutung nahe, daß ein Hyperwürfel mit p Prozessoren einen Binärbaum mit $p - 1$ Prozessoren beinhaltet. Überraschenderweise ist dies

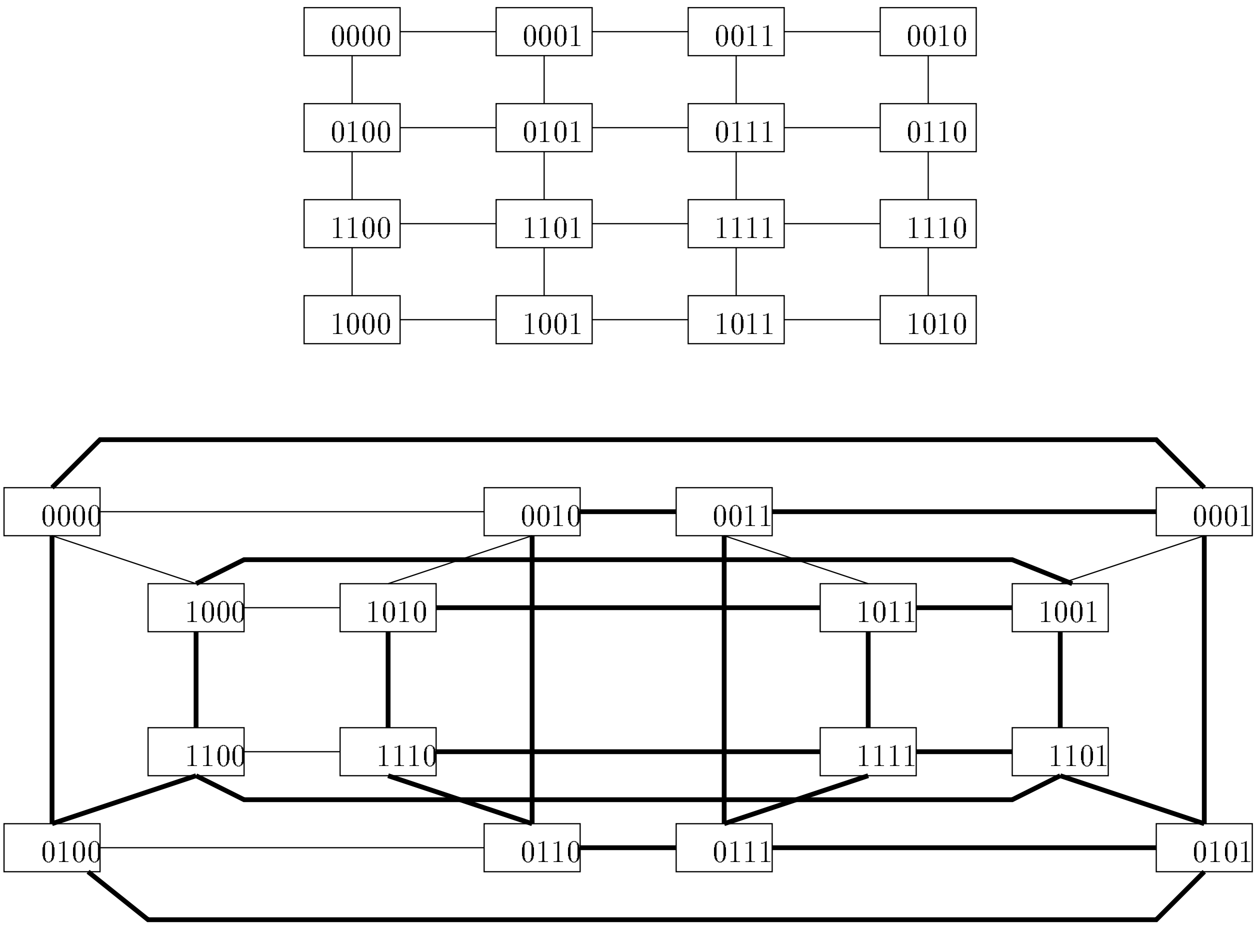


Abbildung 2.10: Einbettung eines 4×4 Arrays in einen Hyperwürfel.

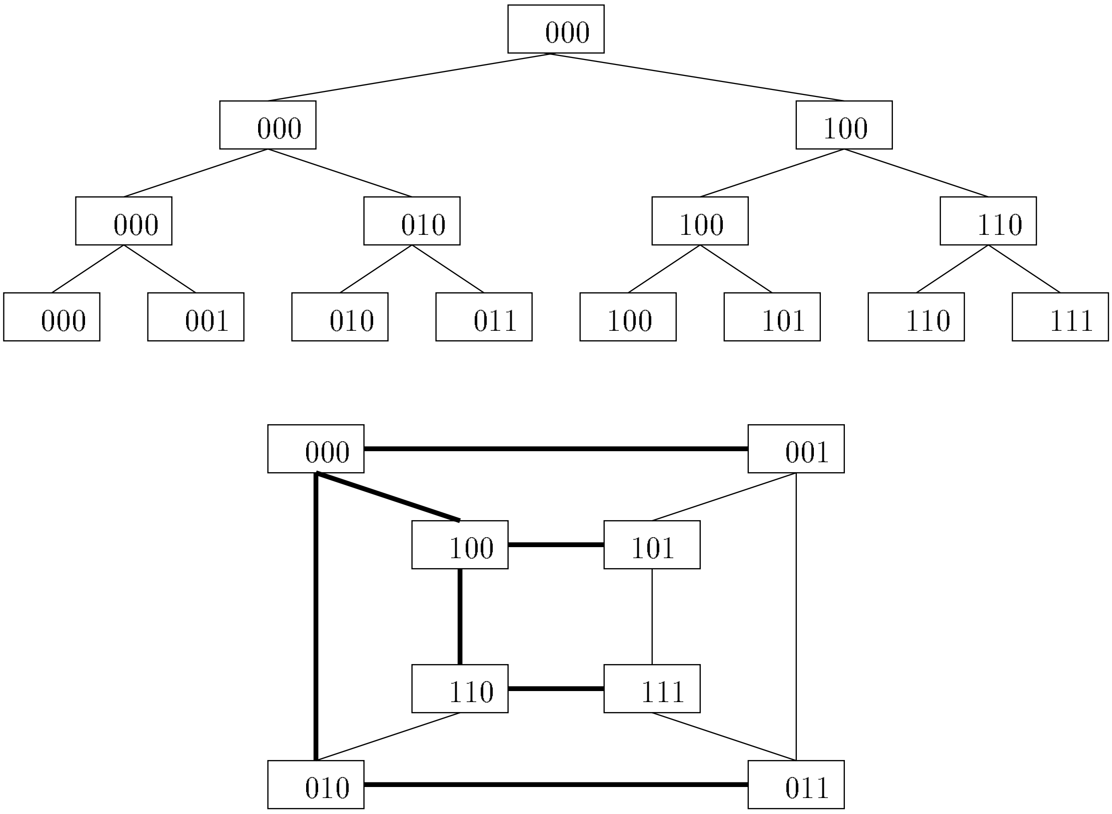


Abbildung 2.11: Einbettung eines Binärbaums in einen Hyperwürfel.

nicht, oder zumindest nicht ganz, der Fall.

Jeder Prozessor in einem Hyperwürfel hat eine gerade Parität wenn seine Binäradresse eine gerade Anzahl von *Einsen* beinhaltet und eine ungerade Parität wenn seine Binäradresse eine ungerade Anzahl von *Einsen* beinhaltet. Es sind nur Prozessoren mit unterschiedlicher Parität verbunden. Außerdem enthält ein Hyperwürfel mit p Prozessoren $p/2$ Prozessoren mit ungerader Parität und $p/2$ Prozessoren mit gerader Parität. Nehmen wir nun an, daß ein Binärbaum mit $p-1$ Prozessoren einzubetten sei und der Wurzelknoten in einem Prozessor mit gerader Parität liegt (wir können genauso ungerade Parität annehmen). Klarerweise müssen die Kinder des Wurzelknotens in Prozessoren mit ungerader Parität liegen, deren Kinder in Prozessoren mit gerader Parität usw. Daraus folgt, daß alle Prozessoren aus den Blättern des Binärbaums in Knoten mit identischer Parität abgebildet werden müssen. Dies bedeutet, daß alle $p/2$ Knoten mit gleicher Parität (gerade oder ungerade) für die Blätter des Binärbaums benötigt werden und damit keine Prozessoren mit dieser Parität für Knoten aus dem Inneren des Baumes zur Verfügung stehen. Damit ist klar, daß (bis auf die Einbettung eines Binärbaums mit 3 Prozessoren in einen Hyperwürfel der Dimension 2) keine weitere Einbettung von Binärbäumen mit $p-1$ Prozessoren in Hyperwürfel mit p Prozessoren möglich ist. Durch eine leichte Modifikation am Binärbaum kann eine Topologie erzeugt werden, die in einem Hyperwürfel enthalten ist. Die Modifikation besteht darin, den Wurzelknoten zu duplizieren und die beiden Wurzelknoten zu verbinden. Die so erzeugt Struktur bezeichnet man als Binärbaum mit doppelter Wurzel und kann in einen Hyperwürfel eingebettet werden. Der Binärbaum mit doppelter Wurzel ist zwar kein eigentlicher Binärbaum, Algorithmen für Binärbäume können aber leicht übertragen werden. Der einzige Unterschied besteht darin, daß die Aufgaben des ursprünglichen Wurzelprozessors nun von den beiden Wurzelprozessoren übernommen werden, dadurch ist keine wesentliche Änderung in der Laufzeit eines Algorithmus zu erwarten.

Viele Algorithmen für Binärbäume benutzen zu einem Zeitpunkt nur Prozessoren in einer Ebene, die übrigen Prozessoren sind dann untätig. Die Ausnutzung dieser Eigenschaft erlaubt eine einfache Einbettung von Binärbäumen. Es reicht aus, die Prozessoren eines Astes im Binärbaum auf einen festen Prozessor abzubilden, wie in Abbildung 2.11 gezeigt. Damit kann eine Binärbaum mit $2^d - 1$ Prozessoren in einen Hyperwürfel der Dimension $d - 1$ mit 2^{d-1} Prozessoren eingebettet werden. Eine weitere Eigenschaft der vorgestellten Einbettung ist, daß die Kommunikation nach jedem Berechnungsschritt über Leitungen in einer neuen konsekutiv folgenden Dimension erfolgt; nach Schritt 1 über Verbindungen der Dimension 1, nach Schritt 2 über Verbindungen der Dimension 2 usw. Algorithmen mit dieser Eigenschaft werden auch als *normal* bezeichnet und können besonders effizient auf CCCs simuliert werden.

Neben der Einbettung anderer Topologien können Hyperwürfel auch für eine effiziente Simulation von Architekturen mit geteiltem Speicher verwendet werden. Insbesondere der Simulation von PRAMs wurde in der Literatur große Aufmerksamkeit gewidmet, um so eine Verwendung von Algorithmen für geteilten Speicher auf Architekturen mit verteiltem Speicher zu ermöglichen und die Ergebnisse der mehr theoretischen Betrachtungen über PRAM Algorithmen auf Hyperwürfel zu übertragen. Wir wollen zugehörige Ansätze hier allerdings nicht weiter betrachten.

2.5 Kommunikationsaspekte in verschiedenen Topologien

Die tatsächliche Laufzeit eines parallelen Programms setzt sich aus der Rechenzeit und der Kommunikationszeit zusammen. Die Kommunikationszeit kann dabei je nach Problem einen beachtlichen Teil der Gesamtlaufzeit ausmachen.

Bei den folgenden Analysen werden wir von den vereinfachten, wenn auch nicht unrealistischen, Annahmen ausgehen, daß alle Leitungen gleiche Kapazität haben und die Übertragung einer Nachricht zwischen zwei direkt verbundenen Prozessoren genau eine Zeiteinheit dauert. Müssen mehrere Nachrichten gleichzeitig über die selbe Verbindung verschickt werden, so treten Wartezeiten auf. Wir nehmen an, daß ein Prozessor quasi gleichzeitig Nachrichten über alle Ausgangsleitungen versenden kann. Wollen zwei nicht direkt verbundene Prozessoren kommunizieren, so erfolgt diese Kommunikation über Zwischenknoten und dauert damit entsprechend länger. Die Auswahl des Weges zwischen zwei Prozessoren erfolgt durch den Routing Algorithmus, dieser benutzt im allgemeinen einen der kürzesten Wege zwischen den Prozessoren (d.h. einen Weg mit einer minimalen Anzahl an Zwischenknoten).

2.5.1 Einige Basiskommunikationsprobleme

Die folgende Kommunikationsprobleme sind in vielen parallelen Algorithmen implizit enthalten und werden in den nachfolgenden Abschnitten auf den verschiedenen Topologien analysiert.

Einfacher Broadcast: Ein einzelnes Paket soll von einem Knoten an alle anderen verschickt werden. Zur effizienten Lösung dieses Problems muß das Paket entlang eines minimalen Spannbaums, mit Wurzel in dem Knoten, der den Broadcast initiiert, übertragen werden. In einem Netz mit maximaler Weglänge l liegt die Dauer der Übertragung in $O(l)$. Zum einfachen Broadcast gibt es ein duales Problem, das allerdings nicht so häufig auftritt, die **einfache Akkumulation**. Es werden hierbei Pakete von allen anderen Knoten an einen Zielknoten geschickt und diese Pakete werden unterwegs verschmolzen. Auch für dieses Problem wird ein minimaler Spannbaum gebildet. Die Knoten in den Blättern beginnen mit der Übertragung, falls ein Knoten im Spannbaum mehr als eine Eingangskante hat, so werden die ankommenden Pakete verschmolzen und ein einzelnes Paket weitergereicht. Eine mögliche Anwendung ist die

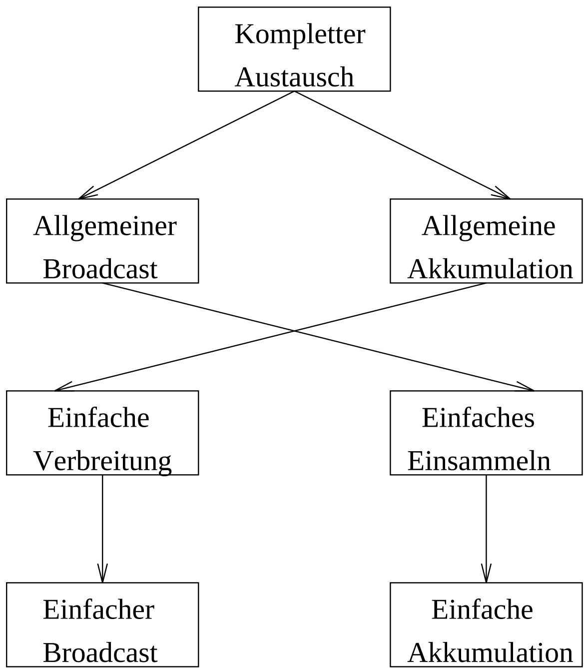


Abbildung 2.12: Hierarchie der Kommunikationsprobleme.

Berechnung einer Summe von Zahlen, wobei jeder Knoten eine Zahl beisteuert. Ein Knoten addiert seine Zahl mit den hereinkommenden Zahlen und reicht die Summe weiter.

Allgemeiner Broadcast: Ein einfacher Broadcast wird simultan von allen Knoten ausgeführt. Es muß für jeden Knoten ein minimaler Spannbaum mit dem entsprechenden Knoten als Wurzel erzeugt werden. Das Problem bei der Analyse ist, daß die verschiedenen Spannbäume gleiche Kanten benutzen können und damit Wartezeiten für Pakete auftreten. Auch zum allgemeinen Broadcast gibt es ein duales Problem, die **allgemeine Akkumulation**. Es wird simultan eine einfache Akkumulation für alle Knoten als Zielknoten durchgeführt.

Einfache Verbreitung: Bei der einfachen Verbreitung sendet ein Knoten jeweils ein Paket an alle anderen. Im Gegensatz zum Broadcast erhält nun aber jeder Knoten ein individuelles Paket. Auch hier gibt es wieder ein duales Problem, das **einfache Einsammeln**. Alle Knoten senden simultan ein individuelles Paket zu einem Ziel.

Kompletter Austausch: Der komplette Austausch stellt das komplexeste betrachtete Kommunikationsproblem dar. Jeder Knoten sendet ein individuelles Paket an jeden anderen. Zu diesem Problem gibt es kein duales Problem, da ein Knoten sowohl Quelle als auch Ziel ist.

Die einzelnen Problem formen eine Hierarchie wie in Abbildung 2.12 gezeigt. Ein gerichteter Pfeil bedeutet jeweils, daß ein Algorithmus der das Problem der oberen Ebene löst auch das Problem der darunter liegenden Ebene löst, auch wenn dies natürlich nicht unbedingt die effizienteste Lösung ist. Probleme in einer Ebene lassen sich mit gleichem Zeitaufwand durch leichte Modifikation am Algorithmus lösen (siehe den Zusammenhang zwischen dem einfachen Broadcast und der einfachen Akkumulation).

2.5.2 Kommunikation in linearen Arrays, Ringstrukturen und Bäumen

Wir betrachten zuerst die einzelnen Kommunikationsprobleme und deren Zeitaufwand auf einem linearen Array mit p Prozessoren. Offensichtlich kann der einfache Broadcast in $\Theta(p)$ Schritten durchgeführt werden. Die exakte Zeit hängt von der Adresse des initiierenden Pro-

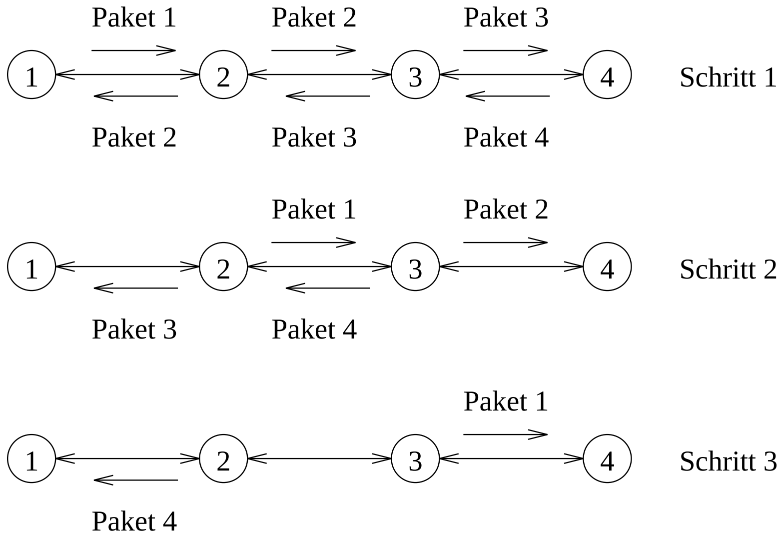


Abbildung 2.13: Allgemeiner Broadcast auf einem linearen Array.

zessors ab und lautet $\max(k, p-k)$ für Prozessor k. Ein allgemeiner Broadcast benötigt $p-1$ Zeiteinheiten. Der Algorithmus ist für $p = 4$ in Abbildung 2.13 dargestellt. Die Leitungen werden parallel benutzt und die Pakete nacheinander durchgeschoben. Eine einfache Verbreitung von Prozessor k initiiert benötigt ebenfalls $\max(k, p-k)$ Zeiteinheiten, wie man sich leicht überlegen kann). Der komplette Austausch benötigt einen Zeitaufwand in $\Theta(p^2)$. Um dies nachzuvollziehen, betrachten wir eine Verbindung zwischen den Prozessoren k und $k+1$. Über diese Verbindung müssen $k(p-k)$ Pakete in jede Richtung übertragen werden. Da dies für alle Kanten gilt, ist $\max_k(k(p-k))$ $(= \lceil (p^2-1)/4 \rceil)$ eine untere Schranke für die Übertragungszeit. Andererseits können wir einen Algorithmus in $\Theta(p^2)$ angeben, indem wir den Algorithmus für die einfache Verbreitung sequentiell für alle Prozessoren anwenden.

Der Aufwand für die Basiskommunikationsprobleme auf einem Ring bewegt sich zwischen dem Aufwand auf einem linearem Array und der Hälfte dieses Wertes. Bei den baumartigen Topologien betrachten wir exemplarisch einen ausgeglichenen Binärbaum mit $p = 2^k - 1$ Prozessoren. Auf einer Binärbaumtopologie ist der einfache Broadcast und die einfache Akkumulation in $\Theta(\log p)$ Schritten auszuführen. Ein einfacher Broadcast vom Wurzelknoten initiiert benötigt $\log p$ Schritte. Ein von einem Knoten im Blatt initiierter einfacher Broadcast benötigt $2 \log p$ Schritte, die Zeiten für Knoten im Inneren des Binärbaums liegen zwischen diesen Werten. Für die einfache Verbreitung und den allgemeinen Broadcast benötigt man $\Theta(p)$ Schritte auf einer Binärbaumtopologie. Der komplette Austausch benötigt $\Theta(p^2)$ Schritte. Aus den Analysen wird deutlich, daß die Verwendung eines Binärbaums statt eines linearen Arrays nur bei solchen Algorithmen Vorteile bringt, deren Kommunikation auf dem einfachen Broadcast oder der einfachen Akkumulation basieren.

2.5.3 Mehrdimensionale Arrays und Torus Netze

In einem d-dimensionalem Array bekommt jeder Prozessor eine d-stellige Adresse $(x_1, \ldots, x_d)$. Der Wert von Koordinate x_i liegt zwischen 1 und n_i. Wir gehen im folgenden vom symmetrischen Fall aus, d.h. $n_i = n$ für alle $1 \leq i \leq d$. Die Anzahl der Prozessoren ist damit $p = n^d$.

Die Zeiten für die Basiskommunikationsprobleme werden für festes d in Abhängigkeit von p ermittelt. Ein einfacher Broadcast benötigt offensichtlich einen Zeitaufwand in $\Theta(p^{1/d})$, wie sich aus der maximalen Weglänge ergibt. Eine einfache Verbreitung und ein allgemeiner Broadcast können nicht mehr als $O(p)$ Zeitaufwand benötigen, da dies die Zeiten für ein lineares Array, welches eingebettet werden kann, sind. Andererseits kann diese Zeit auch nicht unterschritten werden, da jeder Knoten $p-1$ Pakete über seine Ausgangs- bzw. Eingangsleitungen übertragen muß. Die Zahl der Ausgangsleitungen liegt aber zwischen d und $2d$, womit der Zeitaufwand in $\Omega(p)$ liegt. Ein kompletter Austausch benötigt einen Zeitaufwand in $\Omega(p^{(d+1)/d})$, wie man sich anhand der folgenden Argumentation überlegen kann. Seien p und n gerade Zahlen. Wir unterteilen das Array in zwei gleiche Hälften mit $p/2$ Prozessoren, indem alle Prozessoren mit $x_1 \leq n/2$ in die erste Hälfte und alle Prozessoren mit $x_1 > n/2$ in die zweite Hälfte eingeordnet werden. Zwischen beiden Hälften gibt es $p^{(d-1)/d}$ Leitungen. Über diese Leitungen müssen $(p/2)^2$ Pakete übertragen werden, wozu ein Zeitaufwand in $\Omega(p^{(d+1)/d})$ nötig ist. Diese untere Schranke kann tatsächlich erreicht werden, wie wir durch Konstruktion eines $O(p^{(d+1)/d}) = O(n^{d+1})$ Algorithmus zeigen. Angenommen wir hätten einen Algorithmus für ein $d-1$-dimensionales Array mit gleichem n, der einen kompletten Austausch in $O(n^d)$ realisiert[1]. Aus diesem Algorithmus wollen wir nun einen Algorithmus für ein d-dimensionales Array herleiten.

Der Algorithmus läuft in 2 Phasen ab und benutzt die Algorithmen für einen kompletten Austausch in $d-1$-dimensionalen und in linearen Arrays.

Phase 1: Die Pakete werden in Unterarrays der Dimension $d-1$ verteilt. Diese Unterarrays entstehen dadurch, daß die erste Adreßkomponente x_1 festgehalten wird. In dieser Phase sendet Prozessor $(x_1, \ldots, x_d)$ an Prozessor $(x_1, y_2, \ldots, y_d)$ alle Pakete, die für Prozessoren $(*, y_2, \ldots, y_d)$ bestimmt sind. Dies erfordert n komplette Austausche in jedem Unterarray. Da die Unterarrays disjunkt sind, können die Austausche parallel in den Unterarrays ablaufen und der Gesamtaufwand für Phase 1 liegt in $O(n \cdot n^d) = O(n^{d+1}) = O(p^{(d+1)/d})$.

Nach Beendigung von Phase 1 hat jeder Prozessor $(n-1)n^{d-1} \approx n^d$ Pakete, die er an Prozessoren mit differierender erster Adreßkomponente verteilen muß. Für jeden Prozessor $(*, x_2, \ldots, x_d)$ sind n^{d-1} Pakete in Prozessor $(x_1, \ldots, x_d)$ vorhanden. Weiterhin hat Prozessor $(x_1, \ldots, x_d)$ alle Pakete empfangen, die von Prozessoren $(x_1, y_2, \ldots, y_d)$ an ihn "persönlich" gerichtet sind.

Phase 2: Die Verteilung der restlichen Pakete erfolgt durch mehrfachen kompletten Austausch in n^{d-1} linearen Arrays. Die linearen Arrays entstehen, indem die Adreßkomponenten 2 bis d festgehalten werden und nur Komponente 1 variiert. Da die resultierenden linearen Arrays disjunkt sind, können die folgenden Schritte parallel in allen Arrays durchgeführt werden. Ein kompletter Austausch im linearen Array benötigt $\Theta(n^2)$ Schritte und verteilt $n-1$ Pakete pro Prozessor. Damit sind zur Verteilung aller Pakete n^{d-1} komplette Austausche notwendig und der Gesamtaufwand liegt in $O(n^2 n^{d-1}) = O(n^{d+1}) = O(p^{(d+1)/d})$.

Da beide Phasen einen Aufwand in $O(p^{(d+1)/d})$ benötigen und dies identisch zur unteren Schranke ist, kann der komplette Austausch auf einem d-dimensionalem Array in $\Theta(p^{(d+1)/d})$ Schritten ausgeführt werden.

Am Beispiel einer Matrix ($d = 2$) werden die Eigenschaften der Phasen recht gut deutlich: Phase 1 bewirkt einen Austausch der Pakete innerhalb von Zeilen. Nach Abschluß der Phase 1 befinden sich alle Pakete in der zu ihrer Zieladresse gehörenden Spalte. In Phase 2 findet dann die Umverteilung parallel innerhalb jeder Spalte statt, damit die Pakete bzgl. Zeilen- und Spaltenposition an ihrer Zieladresse ankommen.

In einem Torus liegt der Aufwand der verschiedenen Kommunikationsprobleme zwischen dem ursprünglichen Aufwand und der Hälfte dieses Wertes.

[1] Für $d = 2$ ist der bereits bekannte Algorithmus für lineare Arrays in $\Theta(n^2)$ geeignet.

2.5.4 Hyperwürfel und Varianten

Einen Hyperwürfel der Dimension d kann man auch als d-dimensionales Array betrachten, in dem jede Koordinate nur die Werte 0 und 1 annehmen kann. Die Zeit für einen einfachen Broadcast ist gleich der Dimension d bzw. $\log p$. Dies folgt einfach aus der Länge des kürzesten Weges zwischen zwei Prozessoren in einem Hyperwürfel. Für einen allgemeinen Broadcast berechnen wir zunächst eine untere Schranke der benötigten Schritte. Da ein einzelner Prozessor $2^d - 1$ Nachrichten empfangen muß und d Eingangsleitungen besitzt, sind zumindest $\lceil (2^d - 1)/d \rceil$ Schritte nötig. Ein Algorithmus, der die untere Schranke erreicht, kann auch tatsächlich konstruiert werden. Auf die Vorstellung des Algorithmus soll hier allerdings verzichtet werden, da eine recht aufwendige Notation notwendig ist.

Die einfache Verbreitung benötigt ebenfalls $\lceil (2^d - 1)/d \rceil$ Schritte. Da $2^d - 1$ Pakete über d Ausgangsleitungen übertragen werden müssen, geht es offensichtlich nicht schneller. Gleichzeitig ist die Schranke mit Hilfe des Algorithmus für den allgemeinen Broadcast erreichbar (auf Grund der Hierarchie der Kommunikationsprobleme).

Für einen kompletten Austausch unterteilen wir den Hyperwürfel der Dimension d in zwei Unterwürfel der Dimension $d - 1$. Die beiden Unterwürfel sind durch 2^{d-1} Leitungen verbunden. Da beim kompletten Austausch $(2^{d-1})^2$ Pakete zwischen den beiden Unterwürfeln in jede Richtung übertragen werden müssen, kann der komplette Austausch nicht weniger als 2^{d-1} Schritte benötigen. Wir werden nun einen Algorithmus herleiten, der bis auf den Faktor 2 an die untere Schranke herankommt. Sei T_d die Zeit für den kompletten Austausch auf einem d-dimensionalem Hyperwürfel. Rekursiv wird nun ein Algorithmus konstruiert für den $T_d \leq 2^d - 1$ ist. Für $d = 1$ ist T_1 offensichtlich gleich 1. Angenommen wir hätten einen Algorithmus, der den kompletten Austausch in $T_d \leq 2^d - 1$ ausführt. Aus diesem soll nun ein Algorithmus für $d + 1$ Dimensionen mit $T_{d+1} \leq 2^{d+1} - 1$ konstruiert werden. Seien C_1 und C_2 zwei Hyperwürfel der Dimension d, die zu einem Hyperwürfel der Dimension $d + 1$ verbunden werden. Der komplette Austausch wird in drei Phasen ausgeführt. In Phase 1 wird ein kompletter Austausch auf den Unterwürfeln C_1 und C_2 durchgeführt. Da keine gemeinsamen Leitungen benutzt werden, ist dieser nach $T_d \leq 2^d - 1$ Schritten abgeschlossen. In Phase zwei überträgt jeder Prozessor seine 2^d Pakete, die für den anderen Hyperwürfel bestimmt sind, zu dem Prozessor mit identischer Adresse im anderen Hyperwürfel. Zwischen zwei Prozessoren mit identischer Adresse in den beiden Hyperwürfeln existiert genau eine exklusive Leitung, die von keinem anderen Prozessorpaar benutzt wird und auch nicht für den kompletten Austausch in einem Unterwürfel benötigt wird. Die Übertragung der Pakete über diese Leitung kann in 2^d Schritten parallel zum kompletten Austausch in den Unterwürfeln ausgeführt werden. Nach 2^d Schritten ist damit ein kompletter Austausch in den beiden Unterwürfeln ausgeführt und jeder Prozessor hat 2^d Pakete empfangen, von denen er 2^{d-1} an die anderen Prozessoren in seinem Unterwürfel verteilen muß. Das letzte und noch ausstehende Kommunikationsproblem ist aber gerade wieder ein kompletter Austausch im Unterwürfel mit Zeitaufwand T_d. Damit gilt folgende Ungleichung $T_{d+1} \leq T_d + 2^d \leq 2^{d+1} - 1$.

2.5.5 Zusammenfassung und Vergleich

Im nun folgenden Abschnitt sollen die Ergebnisse bzgl. der untersuchten Topologien noch einmal zusammengefaßt und einige zusätzliche Aspekte betrachtet werden. Wir werden uns dabei auf die Basistopologien lineares Array, mehrdimensionales Array und Hyperwürfel beschränken. Die anderen vorgestellten Topologien stellen marginale Modifikationen dar und lassen sich in eine der Basistopologien (fast) einbetten. Für die Basistopologien hatten wir die Zeitaufwände aus Tabelle 2.5 für die Basiskommunikationsprobleme ermittelt.

Kommunikationsproblem	lineares Array	d-dimensionales Array	Hyperwürfel
Einfacher Broadcast	$\Theta(p)$	$\Theta(p^{1/d})$	$\Theta(\log p)$
Einfache Verbreitung	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Allgemeiner Broadcast	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Kompletter Austausch	$\Theta(p^2)$	$\Theta(p^{(d+1)/d})$	$\Theta(p)$

Tabelle 2.5: Zeitaufwand der Basiskommunikationsprobleme auf einigen Topologien.

Bei der Analyse der Basiskommunikationsprobleme wurde bisher immer davon ausgegangen, daß ein Prozessor quasi gleichzeitig über alle seine Ausgangsleitungen Nachrichten übertragen kann. Läßt man diese Annahme weg und geht davon aus, daß in jedem Schritt nur genau ein Paket pro Prozessor übertragen werden kann, so beeinflußt dies den Aufwand eines Kommunikationsproblems auf einer Topologie, bei der die Portanzahl pro Prozessor nicht mit der Prozessorzahl wächst, offensichtlich nur um einen konstanten Faktor. Damit ergeben sich Unterschiede nur für Hyperwürfel. Unter der obigen Annahme ist der Aufwand für die einfache Verbreitung $\Theta(p)$, für den allgemeinen Broadcast $\Theta(p)$ und für den kompletten Austausch $\Theta(p \log p)$ in einem Hyperwürfel.

Für alle Basiskommunikationsprobleme wurde ein bis auf einen konstanten Faktor optimaler Algorithmus erreicht. Bis auf den kompletten Austausch auf dem Hyperwürfel, der $p-1$ Schritte statt der theoretischen unteren Schranke $p/2$ benötigt, war der konstante Faktor 1, d.h. die Algorithmen waren optimal. Untere Schranken für den Zeitaufwand zur Lösung allgemeiner Kommunikationsprobleme können, wie auch in diesem Abschnitt, dadurch gewonnen werden, daß der Flaschenhals der Kommunikation bestimmt und die Zeit zur Überwindung des Flaschenhalses berechnet wird. Bei der einfachen Verbreitung ist z.B. die Anzahl der Ports pro Prozessor ein Flaschenhals, da $p-1$ Pakete über die Ausgangsleitungen übertragen werden müssen.

Mit Hilfe der Basiskommunikationsprobleme werden wir im weiteren Verlauf der Vorlesung die Kommunikationsprobleme lösen, die bei vielen numerischen Algorithmen auftreten. Aus diesem Grund hat die effiziente Realisierung der hier abstrakt formulierten Kommunikationsprobleme eine große praktische Bedeutung für die effiziente Realisierung komplexer numerische Algorithmen.

2.6 Literaturhinweise

Die in diesem Kapitel vorgestellten Ergebnisse sind in zahlreichen Büchern über parallele Architekturen oder parallele Algorithmen zu finden. Beispiele für Lehrbücher, die Einführungen in parallele Architekturen enthalten sind [1, 7, 18, 24, 31, 35]. Weiterhin werden in Kapitel 3 von [8] und in Kapitel 4 von [33] parallele Architekturen und deren Struktur vorgestellt. Ich habe mich im wesentlichen auf [18, 8] gestützt, dies ist aber eine subjektive Auswahl. PRAMs werden in vielen Büchern über theoretische Informatik vorgestellt. Als Beispiele seien Kapitel 8 aus [32] und [37] genannt.

Unterschiedliche Topologien, deren Eigenschaften und die Einbettung von Topologien wird in [27] vorgestellt. Der Teil über die Basiskommunikationsprobleme und deren Realisierung ist im wesentlichen aus [4] entnommen. Leistungsmaße zahlreicher Rechner auf Basis von LINPACK werden in [9] beschrieben.

Kapitel 3

Parallele Algorithmen - Modelle, Klassifikation und Bewertungsmaße

Wir beginnen dieses Kapitel über parallele Algorithmen mit der Definition einiger Bewertungsmaße für (parallele) Algorithmen. Dann betrachten wir verschiedene Ebenen auf denen ein Algorithmus parallelisiert werden kann. Es wird ein einfaches Modell für parallele Algorithmen vorgestellt und auf Basis dieses Modells werden Bewertungskriterien hergeleitet. Das beschriebene Modell erlaubt nur eine sehr abstrakte Analyse, insbesondere wird die Kommunikation zwischen Teilprozessen als ideal (d.h. ohne Zeitverzögerung) angenommen. Kommunikationszeiten haben aber einen großen Einfluß auf die Leistungsfähigkeit eines parallelen Algorithmus, wie in der Praxis immer wieder schmerzlich festgestellt wird, wenn parallele Implementierungen weit unterhalb der erhofften Leistung liegen. Die Einbeziehung von Kommunikationszeiten in die Analyse eines Algorithmus erfordert gleichzeitig die Einbeziehung der Rechnertopologie. Es wird sich zeigen, daß sich die benötigte Kommunikation oft auf die bereits vorgestellten elementaren Kommunikationsprobleme zurückführen läßt. Ein weiterer Aspekt, der in diesem Kapitel behandelt wird, ist der Aspekt der Synchronisation von Algorithmen. Man unterscheidet die Klasse der synchronen und der asynchronen Verfahren. Zum Abschluß des Kapitels werden wir noch sehr kurz auf die Möglichkeiten der parallelen Programmierung eingehen. Für weitere Aspekte zu diesem wichtigen Gebiet sei auf spezielle Vorlesungen zum Thema parallele Programmierung und auf die Übungen verwiesen.

3.1 Bewertungsmaße für Algorithmen

Die Komplexität von Algorithmen wird normalerweise bezüglich der Laufzeit und dem Speicherplatzbedarf untersucht. Wir betrachten im folgenden zuerst die allgemeine Darstellung der Komplexität eines Algorithmus, wie sie in der Informatik üblich ist und hier nur der Vollständigkeit halber dargestellt wird. Daran anschließend werden Maße speziell für parallele Algorithmen und deren Bewertung eingeführt.

3.1.1 Maße für die asymptotische Komplexität

Ziel der Analyse von Algorithmen ist es, möglichst allgemeine Aussagen bzgl. der Laufzeit und des Speicherplatzbedarfs zu machen. Dies bedeutet, daß insbesondere von der konkreten Eingabe abstrahiert werden soll. Weiterhin reichen Aussagen über das asymptotische Verhalten oftmals aus. Solche Aussagen erhält man dadurch, daß die Laufzeit des Algorithmus mit dem Wachstum einer bekannten Funktion verglichen wird. Es werden also Aussagen in der Form, "die Laufzeit von Algorithmus A verhält sich asymptotisch wie $f(n)$", gemacht, wobei $f(n)$

eine bekannte Funktion und n ein Maß für die Größe der Eingabe ist. Die Laufzeit oder auch der Speicherplatz des Algorithmus werden mit einer bekannten Funktion verglichen und für $n \to \infty$ wachsen beide bis auf einen konstanten Faktor gleich schnell. Die obige Aussage wurde in der sogenannten O-Notation formalisiert.

Definition 3.1 *Seien $f(n)$ und $g(n)$ zwei Funktionen mit $n \in \mathbb{N}$ und $f(n)$, $g(n) \in \mathbb{R}_+$, dann ist*

- *$f(n) = O(g(n))$ falls $f(n) \leq cg(n)$ für $n > n_0$ und $c \in \mathbb{R}_+$.*
- *$f(n) = \Omega(g(n))$, falls $g(n) = O(f(n))$.*
- *$f(n) = \Theta(g(n))$, falls $g(n) = O(f(n))$ und $f(n) = O(g(n))$.*

Als Vergleichsfunktionen verwendet man im allgemeinen möglichst einfache Funktionen wie $\log(n)$, n, n^x oder x^n. Die O-Notation beschreibt nur das asymptotische Wachstum einer Funktion, konstante Faktoren werden nicht betrachtet und eine "große" Eingabe vorausgesetzt. Oftmals wählt man eine noch gröbere Einteilung, indem die Laufzeit eines Algorithmus den Klassen konstant, linear, polynomiell und nicht polynomiell zugeordnet wird.

Die betrachteten Algorithmen werden, wenn nicht anders gesagt, bezüglich des durchschnittlichen Verhaltens untersucht (average case). Alternativen dazu wären die Untersuchung des schlechtesten (worst case) oder besten Falls (best case). In vielen der hier betrachteten Algorithmen spielt diese Unterscheidung keine Rolle, alle drei Fälle zeigen asymptotisch identisches Verhalten.

Wir werden uns bei der Analyse der Algorithmen im wesentlichen auf die Laufzeitanalyse konzentrieren. Bei einem sequentiellen Algorithmus auf einem Einprozessorsystem hängt die Laufzeit allein von der Größe der Eingabe ab. Bei einem parallelen Algorithmus auf einem Mehrprozessorsystem spielt neben der Eingabegröße auch die Anzahl der verwendeten Prozessoren und die Topologie des Rechners eine Rolle. Dies macht Komplexitätsaussagen deutlich schwieriger. Im folgenden Abschnitt werden einige üblich Maße für die Bewertung paralleler Algorithmen eingeführt.

3.1.2 Bewertungsmaße paralleler Algorithmen

Für die Bewertung paralleler Algorithmen soll, wie im sequentiellem Fall, die absolute Laufzeit ermittelt werden. Zusätzlich sind aber auch Aussagen über den effizienten Einsatz der Prozessoren von Interesse. Die typische Fragestellung ist: Wieviel Leistungssteigerung bringt der Einsatz zusätzlicher Prozessoren? Um diese Frage zu beantworten, werden Maße definiert, die die Laufzeit des Algorithmus auf einer parallelen Architektur in Relation zur zusätzlich zur Verfügung stehenden Rechenleistung setzen. Die folgenden Leistungsmaße hängen deshalb neben der Eingabegröße n auch von p, der Zahl der Prozessoren ab. Es wird davon ausgegangen, daß p homogene Prozessoren verwendet werden.

Sei $T_p(n)$ die Laufzeit eines Algorithmus für Eingabegröße n und p Prozessoren. Interessant ist natürlich bei der Analyse paralleler Algorithmen der Vergleich mit sequentiellen Algorithmen für dieses Problem. Sei $T^*(n)$ die Laufzeit des besten sequentiellen Algorithmus für das Problem. Der Speedup des parallelen Algorithmus ist dann als

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

definiert. $S_p(n)$ beschreibt den Geschwindigkeitszuwachs, den man erhält, wenn ein p Prozessorsystem statt eines Einzelprozessors verwendet wird. Eine zweite Bewertungsgröße ist

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{pT_p(n)}$$

die Effizienz des parallelen Algorithmus. Die Effizienz beschreibt den Anteil sinnvoller Arbeit eines Prozessor in einem p Prozessorsystem bei Abarbeitung des parallelen Algorithmus. Idealerweise möchte man $S_p(n) = p$ und damit $E_p(n) = 1$ erreichen. Dies bedeutet, daß die Erhöhung der Prozessorzahl die Geschwindigkeit um den gleichen Faktor erhöht. Wir werden im Rahmen der Vorlesung Algorithmen theoretisch analysieren und Prozeßwechselzeiten, Ein-/Ausgabe-Wartezeiten und Speicherzugriffszeiten nicht betrachten. Damit ist $S_p(n) = p$ der maximal erreichbare Speedup. Dies bedeutet, daß $T^*(n) \leq pT_p(n)$ ist, wie leicht gezeigt werden kann, indem der parallele Algorithmus auf einem Einprozessorsystem abgearbeitet wird. Der einzelne Prozessor simuliert die Funktion der p parallelen Prozessoren. Die Zeit für eine solche Simulation ist durch $pT_p(n)$ nach oben beschränkt, da jeder Prozessor für den parallelen Algorithmus maximal $T_p(n)$ Zeiteinheiten aufwendet (dann ist der gesamte Algorithmus abgearbeitet). Der Idealfall $E_p(n) = 1.0$ ist meist nicht erreichbar. Stattdessen müssen wir uns oft damit zufriedengeben, daß $E_p(n) \geq c$ mit $c > 0$ für $n \to \infty$ ist.

In der Literatur wurden auch Messungen, Analysen und Simulationen von Algorithmen mit $S_p(n) > p$ dokumentiert. Diese Fälle rühren daher, daß der Overhead durch Prozeßwechsel und der beschränkte Speicher mit in die Betrachtung einbezogen wird. Damit kann nicht davon ausgegangen werden, daß der parallele Algorithmus auf einem Einprozessor in maximal p-facher Zeit simuliert werden kann und die oftmals auch als superlinear bezeichnete Beschleunigung ist möglich. Die so getroffenen Aussagen sind allerdings meist sehr stark implementierungs- und architekturabhängig und deshalb für allgemeinere Betrachtungen nur eingeschränkt brauchbar.

Noch wurde der optimale sequentielle Algorithmus zur Berechnung von $T^*(n)$ nicht weiter definiert. Ein solcher ist für die meisten Probleme nicht bekannt, oder es kann zumindest nicht nachgewiesen werden, daß er bekannt ist. Es gibt nun die folgenden Möglichkeiten $T^*(n)$ zu definieren.

- $T^*(n)$ ist die Laufzeit des besten bekannten sequentiellen Algorithmus.
- $T^*(n)$ ist die Zeit, die ein sequentieller Benchmark Algorithmus benötigt. Als Benchmark wird meist ein für dieses Problem üblicher sequentieller Algorithmus gewählt.
- $T^*(n)$ wird als $T_1(n)$, also die Laufzeit des parallelen Algorithmus auf einem Einprozessorsystem gewählt. Dieser Vergleichswert sagt natürlich nur wenig über die Parallelisierbarkeit aus, da u.U. sehr viel schnellere sequentielle Algorithmen bekannt sind, die sich aber nicht parallelisieren lassen.

Die beiden obigen Maße betrachten die Auswirkungen der Verwendung zusätzlicher Prozessoren bei einer festen Problemgröße n. Diese Sichtweise, auf der auch die pessimistische Aussage von Amdahl's Gesetz basiert, wird in letzter Zeit zunehmend kritisiert. Der Hauptkritikpunkt ist, daß Multiprozessorsysteme nicht dafür eingesetzt werden, um Problemgrößen zu behandeln, die auf sequentiellen Maschinen behandelt werden. Vielmehr wird die Zahl der eingesetzten Prozessoren erhöht, um größere Probleme zu lösen. Insofern geht eine Analyse, die die Problemgröße n konstant läßt und nur die Prozessorzahl p erhöht, an der Wirklichkeit vorbei.

Eine Möglichkeit das Problemwachstums in die Betrachtungen einzubeziehen, ist die Beobachtung der Problemgröße als Funktion der Prozessorzahl bei fester Laufzeit. Es wird also für festes p und feste Laufzeit $T^*(n)$ die Problemgröße m bestimmt, so daß $T^*(n) = T_p(m)$ gilt. In unserem Modell gilt klarerweise $pn/m \geq 1$. Je näher der Wert bei 1 liegt, desto besser ist der Algorithmus skalierbar. Eine andere Möglichkeit der Bewertung, die in eine ähnliche Richtung geht, ist die Isoeffizienzfunktion, die zum Abschluß kurz erläutert wird. Wir definieren dazu den Overhead, der bei Verwendung von p Prozessoren für ein Problem der Größe n entsteht, als

$$H_p(n) = pT_p(n) - T^*(n) \ .$$

Mit Hilfe des Overheads können wir die Effizienz durch

$$E_p(n) = \frac{T^*(n)}{T^*(n) + H_p(n)}$$

ausdrücken. Für viele parallele Algorithmen wächst $H_p(n)$ langsamer als $T^*(n)$, wenn p fest und n variabel ist. Diese Algorithmen sind skalierbar, d.h. für ein gegebenes p existiert ein n, so daß die Effizienz konstant bleibt. Die Isoeffizienzfunktion betrachtet nun das Wachstum von n im Verhältnis zu p, um eine konstante Effizienz zu erhalten. Wenn wir in der vorherigen Gleichung für die Effizienz den konstanten Wert E einsetzen und die Gleichung nach T^* freistellen, so erhalten wir

$$T^*(m) = \frac{E}{1-E} H_p(m) \ .$$

Es muß nun m bestimmt werden, damit die Gleichung gilt. Der Verlauf von m für festes E und wachsendes p wird durch die Isoeffizienzfunktion $f_E(p)$ beschrieben. Eine Algorithmus ist gut skalierbar, wenn $f_E(p)$ eine Konstante oder ein Polynom kleinen Grades ist.

3.2 Ebenen der Parallelisierung

In diesem kurzen Abschnitt soll der Grad oder die Granularität einer Parallelisierung kurz klassifiziert werden. Man unterscheidet grob die folgenden vier Ebenen, auf denen eine Parallelisierung eingeführt werden kann.

- *Parallelität auf Programmebene:* Mehrere Programme laufen parallel ab.
- *Parallelität auf Prozeßebene:* Kommunizierende und u. U. konkurrierende Prozesse laufen parallel ab.
- *Parallelität auf Anweisungsebene:* Anweisung eines Programms werden parallel ausgeführt.
- *Parallelität auf Operationsebene:* Operationen eines zusammengesetzten Befehls werden parallel ausgeführt.

Parallelität auf Operationsebene und auch teilweise auf Anweisungsebene wird durch den Compiler erkannt und direkt umgesetzt. Dadurch bedingt ist die Leistungsfähigkeit einer Architektur stark abhängig von der Qualität des verwendeten Compilers. Parallelität auf diesen Ebenen wird auch als feinkörnige Parallelität bezeichnet und im Rahmen der Vorlesung nicht direkt behandelt, d.h. es wird davon ausgegangen, daß diese Art von Parallelität automatisch genutzt wird.

Interessanter in unserem Kontext ist die grobkörnige Parallelität auf Prozeßebene. Dies ist auch die Parallelisierungsebene nachrichtenorientierter MIMD Systeme. Die Parallelisierung auf Prozeßebene wird nach dem heutigen Stand der Technik vom Programmierer festgelegt. Eine Automatisierung wird zwar angestrebt, scheint aber sehr schwierig zu sein. Insofern ist auch in Zukunft zu erwarten, daß die Basisstruktur grobkörniger paralleler Algorithmen vom Programmierer festgelegt werden muß.

Für die Synchronisation und Kommunikation paralleler Prozesse stehen unterschiedliche Konzepte bereit, die wir hier nicht detailliert behandeln können. Als Schlagworte seien nur *remote procedure calls* (RPCs), *Rendezvous* und *asynchrone Kommunikation* genannt. Im Kontext objektorientierter Programmierung wird auch an Methoden des *remote object invocation* gearbeitet.

Bei der Parallelisierung ist zu beachten, daß es einfacher ist, einen Algorithmus, der für viele Prozessoren entworfen wurde, mit weniger Prozessoren zu realisieren, als umgekehrt. Die Verwendung zusätzlicher Prozessoren erfordert oftmals Veränderungen am Algorithmus oder auch einen vollständig neuen Ansatz. Insofern werden wir versuchen, Algorithmen mit möglichst hoher Parallelität und damit auch relativ feiner Granularität zu beschreiben. Die Realisierung dieser Algorithmen auf Systemen mit weniger Prozessoren ist kein Problem, ein Prozessor übernimmt die Aufgaben mehrerer anderer. Dies gilt insbesondere für regulär strukturierte Algorithmen auf Matrizen und Vektoren. So lassen sich parallele Algorithmen auf Ebene einzelner Elemente beschreiben. Die Verwendung von weniger Prozessoren führt dazu, daß die Operationen nicht mehr auf einzelnen Elementen sondern auf Untermatrizen oder -vektoren ausgeführt werden.

3.3 Modelle paralleler Algorithmen

Es gibt zahlreiche Ansätze, Modelle paralleler Algorithmen zu definieren. In diesem Kapitel wird ein einfacher Ansatz vorgestellt, in dem Algorithmen durch gerichtete Graphen dargestellt werden. Dieses Modell ist sehr einfach und vernachlässigt einige durchaus wichtige Punkte, erlaubt aber einen Einblick in das Verhalten paralleler Algorithmen und eine grobe Analyse des maximal erreichbaren Speedups und der maximal erreichbaren Effizienz.

3.3.1 Präzedenzgraphen als Modelle paralleler Algorithmen

Die gewählte Darstellungsform sind Präzedenzgraphen, gerichtete azyklische Graphen. Sei $G = (N, A)$ ein gerichteter azyklischer Graph, mit $N = \{1 \dots |N|\}$, der Menge der Knoten und A der Menge der gerichteten Kanten. Knoten beschreiben Operationen und Kanten definieren Datenabhängigkeiten zwischen den Operationen. $(i, j) \in A$ bedeutet, daß Operation i vor j ausgeführt werden muß. Knoten können Operationen auf ganz verschiedenen Ebenen repräsentieren. Im Bereich der Betriebssysteme sind Operationen oftmals ganze Prozesse, hier wollen wir Operationen meist als elementare arithmetische Operationen betrachten. Ein Beispiel für einen einfachen Präzedenzgraphen wird in Abbildung 3.1 gegeben.

Sei der Eingangsgrad eines Knotens die Anzahl der hereinkommenden Kanten, der Ausgangsgrad die Anzahl der herausgehenden Kanten. Knoten ohne Eingangskanten werden als Eingangsknoten, Knoten ohne Ausgangskanten als Ausgangsknoten bezeichnet. Sei N_0 die Menge aller Knoten, die nicht Eingangsknoten sind. Die Menge aller Vorgängerknoten für einen Knoten i ist die Menge aller Knoten j mit $(j, i) \in A$. Die Tiefe D eines Präzedenzgraphen ist definiert als der längste Pfad $i_0 \dots i_D$ mit $(i_{d-1}, i_d) \in A$ $(1 \leq d \leq D)$. Klarerweise muß i_0 ein Eingangsknoten und i_D ein Ausgangsknoten sein.

Sei x_i das Ergebnis der Operation, die durch den i-ten Knoten beschrieben wird, dann gilt $x_i = f_i(x_j | (j, i) \in A)$.
f_i beschreibt die Operation des i-ten Knotens. Auf Grund seiner Struktur ist ein Präzedenzgraph nur für die Darstellung einer Sequenz von Operationen geeignet, Verzweigungen und

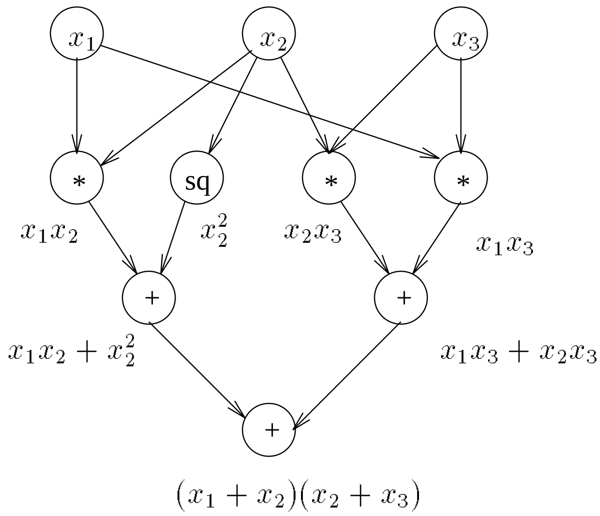


Abbildung 3.1: Präzedenzgraph für $(x_1 + x_2)(x_2 + x_3)$ (sq bedeutet Quadrieren).

Schleifen können mit diesem Modell nicht erfaßt werden. Gleichwohl erlauben Präzedenzgraphen einen ersten Einblick in den Ablauf einfacher Algorithmen und gerade im Bereich der numerischen Algorithmen liegen sehr oft solche Sequenzen von elementaren arithmetischen Operationen vor, auch wenn diese im Programm als *for*-Schleifen programmiert sind. So läßt sich die Schleife

$for\ (i = 1;\ i <= n;\ i++)\ x[i] = y[i] + z[i];$

auch als Sequenz

$x[1] = y[1] + z[1];$
$\ldots$
$x[n] = y[n] + z[n];$

intern darstellen.

Zur Analyse von mittels Präzedenzgraphen beschriebenen Algorithmen werden einige vereinfachende Annahmen getroffen. Wir nehmen an, daß die Ausführung einer Operation, die durch einen Knoten $i \in N_0$ beschrieben wird, genau eine Zeiteinheit benötigt. Diese Annahme ist zu rechtfertigen, wenn Knoten einfache arithmetische Operationen beinhalten, die alle ungefähr gleich lange dauern. Beschreiben Knoten dagegen komplexere Operationen, so sind identische oder fast identische Ausführungszeiten natürlich nicht realistisch und die spezifischen Ausführungszeiten sind in den folgenden Überlegungen zu berücksichtigen. Die Ausführungszeit der Operation eines Eingangsknotens, die im Lesen einer Variablen besteht, sei 0.

Zur Abarbeitung auf einem parallelen Rechner müssen die einzelnen Operationen Prozessoren zugeordnet werden. Seien p Prozessoren verfügbar und sei $P_i \in \{1 \ldots p\}$ der Prozessor, der Operation i ausführt. Weiterhin sei für jeden Knoten $i \in N_0$ $t_i \in \mathbb{N}$ die Zeit, zu der Operation i beendet wird. Da ein Prozessor zu einem Zeitpunkt nur eine Operation durchführen kann und für die Initiierung einer Operation die Resultate der Vorgängeroperation benötigt werden, ergeben sich folgende Restriktionen.

- Für alle $i, j \in N_0$ mit $i \neq j$ und $t_i = t_j$ ist $P_i \neq P_j$.

- Falls $(i, j) \in A$, dann gilt $t_j \geq t_i + 1$.

Eine Menge $\{(i, P_i, t_i) | i \in N_0\}$, die obigen Restriktionen genügt, bezeichnen wir als ein Schedule.

Die obigen Beschreibung eines Algorithmus erlaubt unterschiedliche Implementierungen. Insbesondere werden keine Angaben über die Kommunikationsstruktur zwischen Prozessoren gemacht. Realistischerweise benötigt die Übertragung von Daten zwischen Prozessoren Zeit, deren Einbeziehung zu folgender Verschärfung der zweiten Restriktion führt

$$t_j \geq t_i + 1, \text{ falls } P_i = P_j \text{ und } t_j \geq t_i + \tau(P_i, P_j) + 1 \text{ falls } P_i \neq P_j.$$

$\tau(P_i, P_j)$ umfaßt die Übertragungszeit der Daten zwischen den Prozessoren P_i und P_j und ist zusätzlich noch von der Menge der zu übertragenden Daten abhängig. Insbesondere sind die Übertragungszeiten von der Realisierung und Topologie des verwendeten Rechensystems abhängig. Wir wollen deshalb bei den folgenden Analysen annehmen, daß die Übertragungszeiten vernachlässigbar sind, um so erste allgemeine Resultate zu gewinnen. Später werden Übertragungszeiten in verschiedenen Topologien untersucht und in die Analyse einbezogen.

Kommen wir nun zur Analyse von parallelen Algorithmen zurück, indem wir die Laufzeit T_p eines Algorithmus, der mit einem Präzedenzgraphen spezifiziert wurde, bestimmen. Sei nun $\{(i, P_i, t_i) | i \in N_0\}$ ein Schedule für den Präzedenzgraphen, welcher p Prozessoren benutzt. Die Laufzeit ist dann offensichtlich $\max_{i \in \mathbf{N}_0} t_i$. Wir definieren dann T_p als das Minimum der $\max_{i \in \mathbf{N}_0} t_i$ über alle möglichen Schedules. Weiterhin definieren wir

$$T_\infty = \min_{p \geq 1} T_p$$

T_p ist eine monoton nicht steigende Funktion und es gilt $T_p = T_\infty$ für $p \geq p^*$. T_∞ beschreibt damit die optimale Laufzeit des durch den Präzedenzgraphen spezifizierten Algorithmus, wenn genügend Prozessoren zur Verfügung stehen. T_1 beschreibt die Laufzeit des Algorithmus auf einem Einprozessorsystem. Sowohl T_1 als auch T_∞ sind einfach zu bestimmen. T_1 entspricht der Anzahl der Knoten die nicht Eingangsknoten sind, also $|N_0|$ und T_∞ entspricht gerade der Tiefe D des Präzedenzgraphen.

Nachdem wir nun die beiden Extremwerte der Laufzeit kennen, stellt sich die Frage, wie T_p für ein beliebiges p aussieht. Diese Fragestellung ist realitätsnah, da im allgemeinen nicht beliebig viele, sondern nur eine feste Anzahl von Prozessoren zur Verfügung stehen. Offensichtlich gilt $T_1 \geq T_p \geq T_\infty$. Leider ist die exakte Bestimmung von T_p nicht einfach. Die Zuordnung von Prozessoren zu Aufgaben ist ein NP-hartes kombinatorisches Problem (d.h. kein Algorithmus in polynomieller Zeit ist bekannt). Gleichwohl kann man einfache und wichtige Schranken für T_p angeben. Wie in den folgenden Sätzen gezeigt wird.

Satz 3.1 *Sei c ein positiver ganzzahliger Wert und $q = cp$, dann gilt $T_p \leq cT_q$.*

Bew. Betrachten wir ein Schedule mit q Prozessoren, welches T_q Zeiteinheiten benötigt. In jedem Schritt werden maximal q parallele Operationen ausgeführt, diese können in $q/p = c$ Zeiteinheiten mit p Prozessoren erledigt werden. Damit haben wir ein Schedule erreicht welches maximal cT_q Zeiteinheiten benötigt. □

Satz 3.2 *Für jedes $p \geq 1$ gilt $T_p < T_\infty + \frac{T_1}{p}$.*

Bew. Sei S ein Schedule mit Laufzeit T_∞ und für jedes $\tau \in \{1 \ldots T_\infty\}$ sei n_τ die Anzahl der Knoten i mit $t_i = \tau$. Für p Prozessoren definieren wir ein Schedule S', welches aus S abgeleitet wird, indem in Phase τ die Operationen von S ausgeführt werden, die zum Zeitpunkt τ enden.

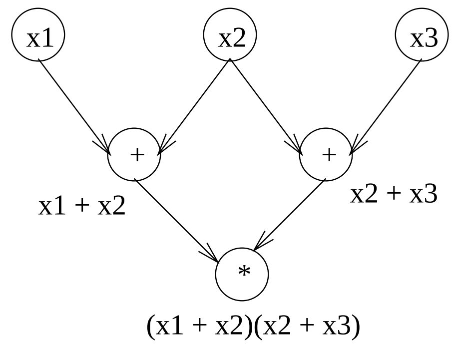


Abbildung 3.2: Zweiter Präzedenzgraph für $(x_1 + x_2)(x_2 + x_3)$.

Da p Prozessoren zur Verfügung stehen, benötigt Phase τ $\lceil n_\tau/p\rceil$ Zeiteinheiten. Da T_p nicht größer als die Zeit die Schedule S' benötigt sein kann, gelten folgende Ungleichungen.

$$T_p \leq \sum_{\tau=1}^{T_\infty} \lceil \tfrac{n_\tau}{p} \rceil < \sum_{\tau=1}^{T_\infty} (\tfrac{n_\tau}{p} + 1) = \tfrac{T_1}{p} + T_\infty.$$

Die letzte Umformung folgt aus $\sum_{\tau=1}^{T_\infty} n_\tau = |N_0| = T_1$. □

Satz 3.3
a) Falls $p \geq T_1/T_\infty$, dann gilt $T_p < 2T_\infty$. Allgemeiner gilt, wenn $p = \Omega(T_1/T_\infty)$, dann ist $T_p = O(T_\infty)$.
b) Falls $p \leq T_1/T_\infty$, dann gilt $\frac{T_1}{p} \leq T_p < 2\frac{T_1}{p}$.
Allgemeiner gilt, wenn $p = O(T_1/T_\infty)$, dann ist $T_p = \Theta(T_1/p)$.

Bew. a) Der Beweis folgt aus Satz 3.2.
b) Aus $p \leq T_1/T_\infty$ folgt $T_\infty \leq T_1/p$ womit nach Satz 3.2 $T_p < 2T_1/p$ folgt. Aus Satz 3.1 folgt $T_1 \leq pT_p$ und damit $T_p \geq T_1/p$. □

Gerade der letzte Satz ist von großer praktischer Relevanz. Es zeigt sich, daß $\Omega(T_1/T_\infty)$ Prozessoren ausreichen, um bis auf einen konstanten Faktor an die minimale Laufzeit heranzukommen. Weiterhin wird im Beweis von Satz 3.2 sogar ein Schedule definiert, um diese Annäherung mit p Prozessoren zu erreichen.

Damit liegt folgende Methode der Parallelisierung nahe. Entwickle den Algorithmus für eine unendliche Anzahl von Prozessoren und modifiziere ihn anschließend für p verfügbare Prozessoren. Die notwendige Modifikation wird im Beweis von von Satz 3.2 angegeben. Solange $p = O(T_1/T_\infty)$ gilt, ist ein Speedup im Bereich von p und eine Effizienz in $\Theta(1)$ zu erwarten. Dies sind, bis auf konstante Faktoren, optimale Werte.

Wir haben bisher die Bearbeitungszeit für einen gegebenen Präzedenzgraphen untersucht und optimiert. Nun existieren aber für ein Problem im allgemeinen verschiedene Präzedenzgraphen, die unterschiedliche Algorithmen beschreiben und unterschiedliche Laufzeiten haben. Die Berechnung von $(x_1+x_2)(x_2+x_3)$ kann mit dem durch den Präzedenzgraph aus Abbildung 3.1 definierten Algorithmus, aber auch mit dem in Abbildung 3.2 spezifizierten Algorithmus ausgeführt werden. Der Algorithmus aus Abbildung 3.1 liefert $T_1 = 7$ und $T_\infty = 3$, während der Algorithmus aus Abbildung 3.2 $T_1 = 3$ und $T_\infty = 2$ Zeiteinheiten benötigt.

Die Erzeugung eines optimalen Präzedenzgraphen für eine gegebene Anzahl von Prozessoren ist schwierig. Für zahlreiche Problemklassen existieren aber Methoden zur Konstruktion quasi optimaler Präzedenzgraphen. Wir werden diese Methoden im Rahmen der Vorlesung nicht vertiefen.

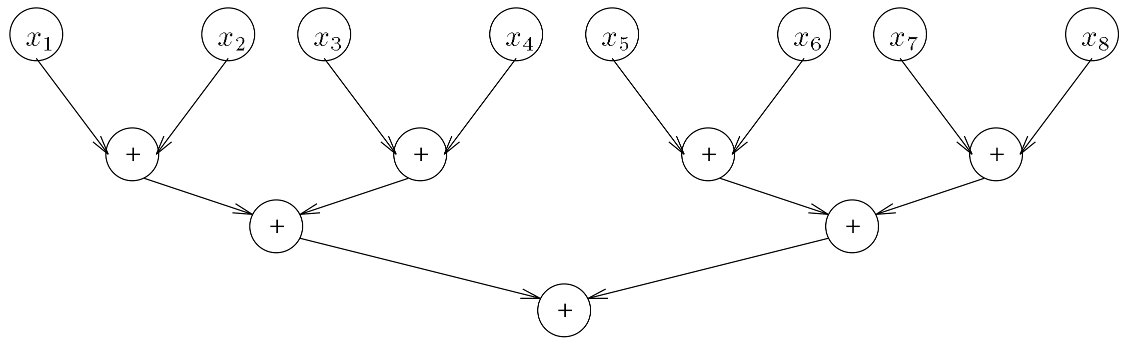


Abbildung 3.3: Präzedenzgraph zur Skalar-Addition.

3.3.2 Beispiele paralleler Algorithmen

In diesem Abschnitt werden einige einfache numerische Algorithmen untersucht. Diese Algorithmen werden in den späteren Kapiteln als Teile der komplexeren numerischen Algorithmen immer wieder auftauchen. Alle Algorithmen lassen sich mit Präzedenzgraphen beschreiben und analysieren. Wir gehen wieder von der vereinfachten Annahme aus, daß die Kommunikation zwischen Prozessoren zeitlos erfolgt. Die so ermittelten Analyseergebnisse sind damit "best case" Resultate. Im nächsten Kapitel werden die vorgestellten Algorithmen auf speziellen Multiprozessortopologien unter Einbeziehung der Kommunikationszeiten analysiert und die Resultate mit den hier ermittelten optimalen Resultaten verglichen.

Addition von Skalaren

Eine einfache Aufgabe ist die Addition von n Skalaren. Ein sequentieller Algorithmus benötigt dafür offensichtlich $n-1$ Schritte, d.h. $T^*(n) = n-1$ oder die Laufzeit liegt in $\Theta(n)$. Ein naheliegender Algorithmus, der $\lfloor n/2 \rfloor$ Prozessoren benutzt, besteht darin, im ersten Schritt mit jedem Prozessor jeweils 2 benachbarte Skalare zu addieren. Im nächsten Schritt werden parallel jeweils benachbarte Summen addiert, d.h. im Schritt 1 wird $x_1 + x_2$ auf Prozessor 1, $x_3 + x_4$ auf Prozessor 2 berechnet. In Schritt 2 wird $(x_1 + x_2) + (x_3 + x_4)$ auf Prozessor 1 berechnet. Nach $\lceil \log n \rceil$ Schritten liegt offensichtlich das Ergebnis vor. Der zum Algorithmus gehörige Präzedenzgraph wird in Abbildung 3.3 gezeigt.

Betrachten wir nun die Effizienz des Algorithmus. Diese ist $\frac{n-1}{\lfloor n/2 \rfloor \lceil \log n \rceil}$. Für $n \to \infty$ geht die Effizienz gegen 0. Diese schlechte Effizienz ist dadurch zu erklären, daß schon im zweiten Schritt die Hälfte und im dritten Schritt drei Viertel der Prozessoren untätig sind.

Ein alternativer Algorithmus ist in Abbildung 3.4 dargestellt. Wir nehmen zur Vereinfachung an, daß $\log n$ eine ganze Zahl und $n/\log n$ eine Potenz von 2 ist. Die n Zahlen werden nun in $n/\log n$ Gruppen mit jeweils $\log n$ Elementen unterteilt. Die Zahlen jeder Gruppe werden von einem Prozessor sequentiell in $\log n - 1$ Schritten addiert, es werden dazu $n/\log n$ Prozessoren verwendet. Nun müssen noch die $n/\log n$ Gruppensummen addiert werden. Dies kann mit dem Algorithmus aus Abbildung 3.3 in $\log(n/\log n) \leq \log n$ Schritten unter Verwendung von $n/(2\log n)$ Prozessoren erfolgen. Dieser "Zweiphasen" Algorithmus benötigt ungefähr doppelt so lange wie der Algorithmus aus Abbildung 3.3, wobei aber deutlich weniger Prozessoren notwendig sind. Dadurch ist die Effizienz ungefähr 0.5, dies ist ein akzeptabler Wert. Die Anzahl der im letzten Algorithmus verwendeten Prozessoren entspricht ungefähr $T_1(n)/T_\infty(n)$, was nach den Aussagen von Satz 3.3 einen effizienten Algorithmus liefert.

Dieses einfache Beispiel zeigt einen sehr oft anzutreffenden Effekt. Indem eine geringfügig längere Laufzeit in Kauf genommen wird, kann die Zahl der benötigten Prozessoren drastisch verringert und damit die Effizienz des Algorithmus deutlich erhöht werden.

Für die Skalar-Addition leiten wir nun noch kurz die Isoeffizienzfunktion her. Sei dazu

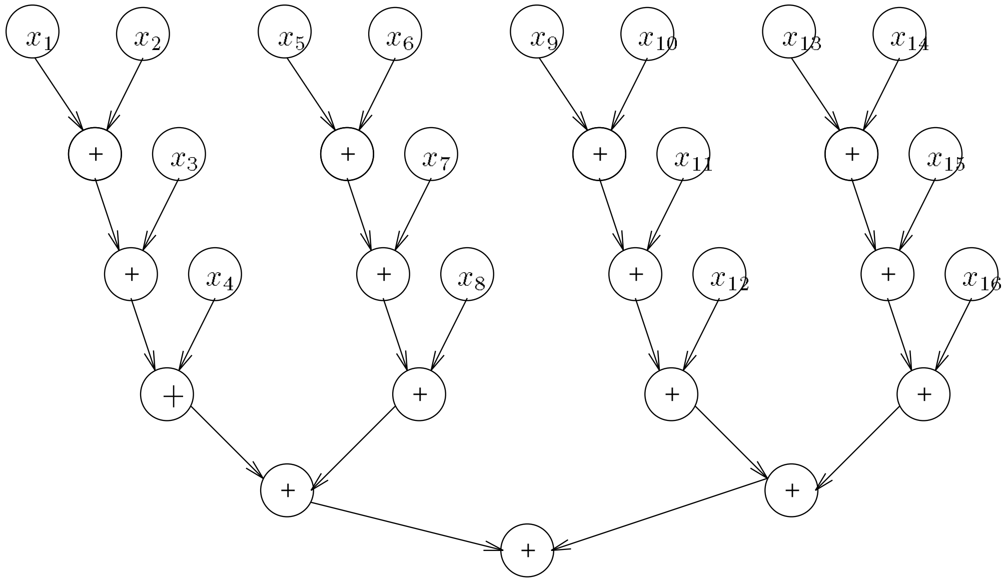


Abbildung 3.4: Alternativer Präzedenzgraph zur Skalar-Addition.

$n = 2^x$, $p = 2^y$ und $x > y \geq 0$. Wir betrachten folgenden Algorithmus, der in 2 Phasen abläuft. In Phase 1 werden auf jedem Prozessor $2^x/2^y = 2^{x-y}$ Elemente addiert. Dies dauert $2^{x-y} - 1$ Zeiteinheiten, wenn eine Addition eine Zeiteinheit benötigt und die Kommunikation vernachlässigt wird. In Phase 2 benutzen wir den ersten Algorithmus zur Addition der 2^y Teilsummen. Dazu sind y Schritte notwendig. Für $y = x - 1$ ist der Algorithmus äquivalent zu Algorithmus 1, für $y = x - \log x$ erhalten wir Algorithmus 2. Die Gesamtlaufzeit für den verallgemeinerten Algorithmus lautet

$$T_p(n) = T_{2^y}(2^x) = 2^{x-y} + y - 1 \ .$$

Speedup und Effizienz berechnen sich zu

$$S_p(n) = S_{2^y}(2^x) = \frac{2^x - 1}{2^{x-y} + y - 1} \text{ und } E_p(n) = E_{2^y}(2^x) = \frac{2^x - 1}{2^y(2^{x-y} + y - 1)} \approx \frac{2^{x-y}}{(2^{x-y} + y)} \ .$$

Der Overhead berechnet sich zu

$$H_p(n) = H_{2^y}(2^x) = 2^y(2^{x-y} + y - 1) - (2^x - 1) = 2^y(y-1) + 1 \approx y2^y \ .$$

Für große Werte von p und n können wir die Approximation für den Overhead verwenden und für eine feste Effizienz folgende Gleichung.

$$2^x = \frac{E}{1-E} y 2^y \implies n = \frac{E}{1-E} p \log p$$

Die Isoeffizienzfunktion ist damit in $\Theta(p \log p)$. Dies bedeutet, daß bei einer Vergrößerung der Prozessorzahl von p nach p' die Zahl der Summanden um den Faktor $(p' \log p')/(p \log p)$ wachsen muß, um eine gleiche Effizienz zu erreichen. Einige Beispiele für die Effizienz des Algorithmus bei unterschiedlichen Werten für p und n sind in Tab. 3.1 zu sehen.

Vektor-Produkt

Das Produkt zweier n-dimensionaler Vektoren x, y ist definiert als $\sum_{i=1}^{n} x_i y_i$. Es kann in $\lceil \log n \rceil +$ 1 Schritten mit n Prozessoren berechnet werden. Im ersten Schritt berechnet Prozessor i das

	p			
n	2	4	8	16
16	0.889	0.667	0.400	0.200
64	0.970	0.889	0.727	0.500
256	0.992	0.970	0.914	0.800
1024	0.998	0.992	0.977	0.941
4096	1.000	0.998	0.994	0.985

Tabelle 3.1: Effizienz der Skalar-Addition für einige Werte von n und p.

Produkt $x_i y_i$, anschließend wird der $\lceil \log n \rceil$ Algorithmus aus Abbildung 3.3 zur Addition der Produkte verwendet. Es gilt dann $S_p(n) = \frac{n+(n-1)}{\lceil \log n \rceil + 1} \approx \frac{2n}{\log n}$ und $E_p(n) \approx \frac{2}{\log n}$. Auch in diesem Fall ist die Effizienz des Algorithmus sehr schlecht.

Ein effizienterer Algorithmus kann ähnlich zur Skalar-Addition definiert werden. Es wird dazu der Algorithmus aus Abbildung 3.4 zur Addition der n Produkte verwendet, wobei der Zeitaufwand ungefähr $2 \log n$ beträgt. Die Bildung der n Produkte mit $n/\log n$ Prozessoren dauert ungefähr $\log n$ Zeiteinheiten, dadurch liegt die Gesamtlaufzeit bei ungefähr $3 \log n$ und die Effizienz bei $2/3$.

Matrix-Addition

Die Summe von n-Matrizen der Dimension $m \times m$ kann in $\lceil \log n \rceil$ Schritten mit $m^2 \lfloor n/2 \rfloor$ Prozessoren berechnet werden. Eine Gruppe von $\lfloor n/2 \rfloor$ Prozessoren berechnet jeweils einen Eintrag in der resultierenden Matrix. Ansonsten wird analog zur Skalar-Addition vorgegangen. Da der Algorithmus zur Skalar-Addition unverändert übernommen wird, gelten auch die Effizienzüberlegungen. Durch Verwendung des zweiten Algorithmus zur Skalar-Addition kann mit Hilfe von $m^2 n/\log n$ Prozessoren die Summe von n $m \times m$ Matrizen in ungefähr $2 \log n$ Schritten gebildet werden, wobei die Effizienz ungefähr 0.5 ist.

Matrix-Multiplikation

Die Multiplikation zweier Matrizen der Dimensionen $m \times n$ und $n \times l$ kann in ml Vektorprodukte n-dimensionaler Vektoren zerlegt werden. Mit Hilfe der parallelen Vektor-Multiplikation kann das Produkt in $\lceil \log n \rceil + 1$ Schritten auf nml Prozessoren berechnet werden. Auch dieser Algorithmus ist zwar extrem schnell, aber auch sehr ineffizient. Falls $n = m = l$ benötigen wir n^3 Prozessoren, was für größere Matrizen eine immense Zahl ist. Die Effizienz des Algorithmus ist ungefähr $2/\log n$.

Es können auch für dieses Problem sehr viel ökonomischere Algorithmen definiert werden. Da die Matrix-Multiplikation auf die Vektor-Multiplikation zurückgeführt wurde, kann der zweite Algorithmus zur Vektor-Multiplikation verwendet werden. Dazu sind $n^3/\log n$ Prozessoren notwendig, um das Produkt in ungefähr $3 \log n$ Schritten zu berechnen. Allgemein ausgedrückt benötigt der Algorithmus einen Zeitaufwand in $\Theta(\log n)$, um mit $\Theta(n^3/\log n)$ Prozessoren das Produkt zweier $n \times n$ Matrizen zu bestimmen. Die Effizienz liegt in $\Theta(1)$. Wird die Anzahl der Prozessoren weiter verringert, so benötigt ein Algorithmus mit n^2 Prozessoren eine Laufzeit in $\Theta(n)$ und eine Algorithmus mit n Prozessoren eine Laufzeit in $\Theta(n^2)$. In beiden Fällen liegt die Effizienz in $\Theta(1)$.

Des weiteren gibt es noch grundsätzlich andere Ansätze der Matrizen-Multiplikation, die das Matrix Produkt nicht über Vektor Produkte definieren. Wir werden uns mit einem solchen Algorithmus kurz im folgenden Kapitel beschäftigen.

Potenzen einer Matrix

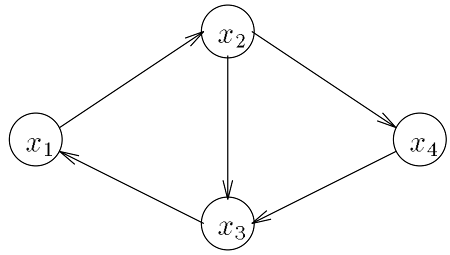


Abbildung 3.5: Beispiel eines Abhängigkeitsgraphen.

Die Potenzen einer $n \times n$ Matrix $\mathbf{A}$ können sehr effizient durch wiederholtes Quadrieren gebildet werden. Betrachten wir zuerst $\mathbf{A}^k$ für $k = 2^l$. Durch l-maliges ($l = \log k$) Quadrieren der Matrix $\mathbf{A}$ erreichen wir $\mathbf{A}^k$. Die Anzahl der benötigten Prozessoren und die Laufzeit hängen vom verwendeten Multiplikationsalgorithmus ab. Mit Hilfe des schnellsten vorgestellten Algorithmus benötigen wir n^3 Prozessoren und $\log k(\lceil \log n \rceil + 1)$ Schritte.

Falls k keine Potenz von 2 ist, so kann k als $\sum_{l=0}^{\lfloor \log k \rfloor} \alpha_l 2^l$ mit $\alpha_l = 0$ oder 1 dargestellt werden. Im ersten Teil des Algorithmus werden die Matrizen $\mathbf{A}^l$ mit $l = 2^0 \ldots 2^{\lfloor \log k \rfloor}$ durch wiederholtes Quadrieren berechnet. Dafür sind $\lfloor \log k \rfloor$ Matrix-Multiplikationen notwendig. Im zweiten Schritt werden diejenigen Matrizen $\mathbf{A}^l$, für die $\alpha_l = 1$ gilt, miteinander multipliziert. Dazu sind maximal $\lfloor \log k \rfloor$ Matrix-Multiplikationen notwendig. Die optimale Rechenzeit zur Berechnung beliebiger Potenzen $\mathbf{A}^k$ liegt damit in $\Theta(\log k \log n)$.

Iterative Algorithmen

Viele der später behandelten Algorithmen sind iterativ, d.h. ausgehend von einem Initialwert werden konsekutiv neue Approximationen für die Lösung berechnet, bis die Approximationen "nahe genug" bei der Lösung liegen. Iterative Verfahren haben alle die folgende Struktur.

$$\mathbf{x}(t+1) = f(\mathbf{x}(t)) \tag{3.1}$$

$\mathbf{x}$ ist n-dimensionaler Vektor, f ist eine kontinuierliche Funktion vom $\mathbb{R}^n$ in den $\mathbb{R}^n$. Gesucht wird die Lösung (der Fixpunkt) $\mathbf{x}^*$ für die gilt $\mathbf{x}^* = f(\mathbf{x}^*)$. Wir können obige Gleichung auch umschreiben, indem wir die Berechnung eines einzelnen Vektorelements $\mathbf{x}_i(t+1)$ betrachten.

$$x_i(t+1) = f_i(\mathbf{x}(t)) \tag{3.2}$$

Die obigen Verfahren bezeichnet man auch als Verfahren vom Jacobi Typ . Eine naheliegende Methode der Parallelisierung ist die Verwendung von n Prozessoren, wobei Prozessor i den Wert von $x_i(t+1)$ berechnet. Anschließend wird der berechnete Wert $x_i(t+1)$ an die übrigen Prozessoren weitergeleitet und der nächste Iterationsschritt kann folgen.

In realen Problemen benutzt f_i meist nicht den gesamten Vektor $\mathbf{x}$, sondern nur einige Komponenten x_j. Dies hat zur Folge, daß Prozessor i sein Resultat $\mathbf{x}_i(t+1)$ nicht an alle anderen Prozessoren versenden muß, sondern nur an Prozessoren j, die Funktionen f_j berechnen, in denen x_i benötigt wird. Wir werden im Laufe der Vorlesung Beispiele kennenlernen, in denen die Funktionen f_i nur von sehr wenigen Komponenten abhängen und diese Abhängigkeit auf sehr regelmäßigen Strukturen beruht. Die Struktur der Datenabhängigkeit läßt sich durch einen Abhängigkeitsgraphen darstellen. Die Knoten beschreiben die Variablen, eine gerichtete Kante von Knoten i nach j sagt aus, daß x_i in f_j benötigt wird. In Abbildung 3.5 wird ein Beispiel für einen Abhängigkeitsgraph gegeben.

Aus dem Abhängigkeitsgraphen läßt sich der Präzedenzgraph des iterativen Algorithmus einfach gewinnen, indem der Abhängigkeitsgraph quasi auseinander gefaltet und für jeden Ite-

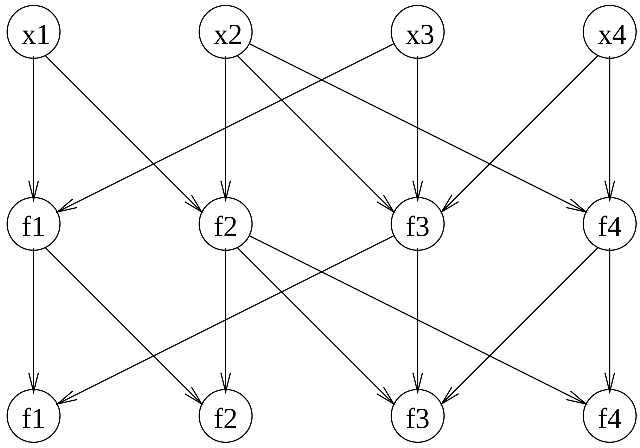


Abbildung 3.6: Präzedenzgraph für den vorherigen Abhängigkeitsgraphen.

rationsschritt dupliziert wird. Abbildung 3.6 zeigt den Präzedenzgraph für den Abhängigkeitsgraph aus Abbildung 3.5 und 2 Iterationen.

In realen Anwendungen ist die Anzahl der Funktionen und Variablen deutlich größer als die Zahl der zur Verfügung stehenden Prozessoren. In einem solchen Fall wird eine Menge von Funktionen von einem einzelnen Prozessor berechnet. Es ist natürlich sinnvoll, möglichst geringe Datenabhängigkeiten zwischen Funktionen, die auf unterschiedlichen Prozessoren berechnet werden, zu erreichen. Eine solche Aufteilung von Funktionen auf Prozessoren ist allgemein nicht trivial.

Die exklusive Zuordnung von Funktionen zu Prozessoren ist ineffizient, wenn in jeder Funktion quasi das selbe berechnet wird. Betrachten wir z.B. den Fall $f_i(\mathbf{x}) = x_i + \sum_{j=1}^{n} x_j^2$. Die Berechnung einer isolierten Funktion auf einem Prozessor führt dazu, daß die Summe, die die meiste Rechenzeit verbraucht, auf allen Prozessoren berechnet wird. Die Prozessoren sind damit zwar ausgelastet, machen aber alle identische Berechnungen. In einem solchen Fall sind natürlich andere Arten der Parallelisierung sinnvoller und effizienter.

Bisher haben wir Iterationen vom Jacobi Typ betrachtet, eine Alternative, die oftmals effizienter bezüglich der Iterationszahl ist, sind Iterationen vom Gauß-Seidel Typ . Gleichung 3.2 wird dann zu

$$x_i(t+1) = f_i(x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t)) \tag{3.3}$$

In der Funktion werden also die Komponenten aus dem aktuellen und nicht aus dem vorherigen Lösungsvektor verwendet. Gauß-Seidel Verfahren sind effizienter weil sie die aktuellste Information zur Berechnung benutzen. Dies ist bezüglich der Parallelisierbarkeit aber auch der größte Nachteil, denn die neuste Information ist noch nicht im Prozessor vorhanden. Betrachten wir zuerst den allgemeinsten Fall, jedes f_i ist vom gesamten Vektor $\mathbf{x}$ abhängig. Im t-ten Iterationsschritt benötigt Prozessor i die Komponenten $1, \ldots, i-1$ aus $\mathbf{x}(t+1)$ und die Komponenten $i, \ldots, n$ aus $\mathbf{x}(t)$. Wenn wir nun wieder davon ausgehen, daß zu Beginn des t-ten Iterationsschrittes jeder Prozessor den Vektor $\mathbf{x}(t)$ kennt, so kann Prozessor 1 $x_1(t+1)$ aus $\mathbf{x}(t)$ berechnen. Prozessor 2 benötigt aber $x_1(t+1)$, muß also auf Prozessor 1 warten, Prozessor 3 muß auf 1 und 2 warten usw. Im Extremfall läuft der Algorithmus damit komplett sequentiell ab und kann in keiner Weise parallelisiert werden.

Glücklicherweise beschreiben die obigen Überlegungen den worst case, der in der Praxis relativ selten vorkommt. Es gibt zwei Situationen, die in fast allen realen Problemen mehr oder weniger ausgeprägt auftreten, die eine Parallelisierung von Gauß-Seidel Verfahren erlauben. Zum einen kann meist mit der Berechnung von f_i schon begonnen werden, wenn noch nicht

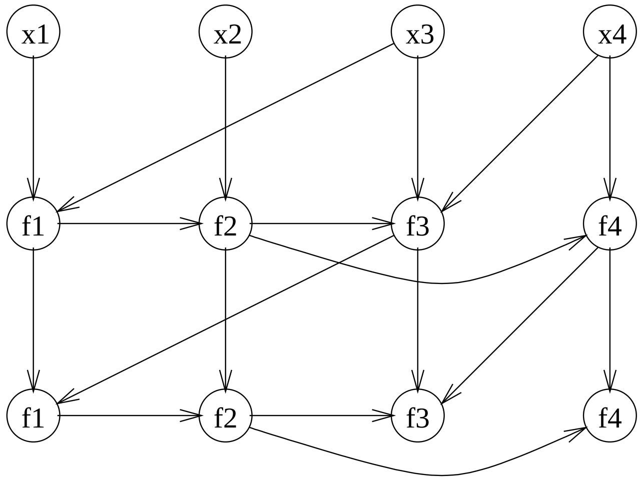


Abbildung 3.7: Präzedenzgraph für die Gauß-Seidel Iteration.

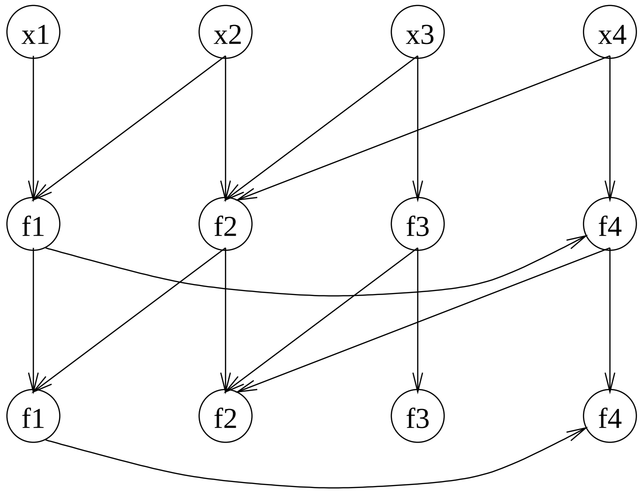


Abbildung 3.8: Präzedenzgraph für die Gauß-Seidel Iteration nach Variablenumordnung.

alle benötigten Komponenten $x_j(t+1)$ $(j < i)$ vorhanden sind. Zum anderen benutzen die Funktionen f_i meist nicht alle Komponenten des Vektors $\mathbf{x}$, wie bereits oben erwähnt. Ist die Datenabhängigkeit der Funktionen bekannt, so läßt sich daraus eine Zuteilung von Funktionen auf Prozessoren ableiten, die oftmals eine recht effiziente Parallelisierung erlaubt.

Betrachten wir noch einmal den Abhängigkeitsgraphen in Abbildung 3.5. Auf Grund der Datenabhängigkeiten müssen die Funktionen f_1, f_2 und f_3 nacheinander ausgeführt werden und f_4 muß nach f_2 ausgeführt werden. Funktion f_4 kann allerdings parallel zu f_3 ausgeführt werden. Wir können dies im Präzedenzgraphen dadurch darstellen, daß für alle $j < i$ mit $(i,j) \in A$ im Abhängigkeitsgraphen, eine gerichtete Kante von f_j nach f_i im Präzedenzgraphen gezogen wird. Dies führt im Beispiel zu dem in Abbildung 3.7 gezeigten Präzedenzgraphen für 2 Iterationen.

Das Gauß-Seidel Verfahren für eine Funktion hängt von der Numerierung der Variablen ab, somit ergeben verschiedene Numerierungen unterschiedliche Iterationsvorschriften mit unterschiedlichen Parallelisierungsmöglichkeiten. Setzen wir z.B. Funktion 2 an den Schluß, d.h. aus 2 wird 4, aus 4 3 und aus 3 2, so können die neuen Funktionen 1,2 und 3 parallel ausgeführt werden, nur Funktion 4 benötigt das aktuelle Ergebnis von 1. Wie im resultierenden Präzedenzgraphen in Abbildung 3.8 gezeigt.

Auch die Konvergenzgeschwindigkeit des Gauß-Seidel Verfahrens hängt von der Variablen-

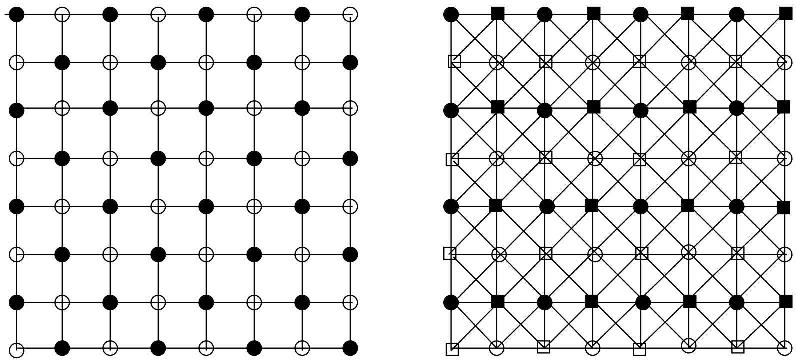


Abbildung 3.9: Zwei- und Viergefärbtes Gitter.

anordnung ab. Da aber a priori nicht bekannt ist, welche Anordnung die schnellste Konvergenz liefert, ist die Variablenanordnung oftmals bzgl. der besten Parallelisierbarkeit zu wählen. Das Problem der optimalen Variablenanordnung läßt sich als Färbeproblem in Graphen auffassen; die Knoten des Abhängigkeitsgraphen werden so gefärbt, daß parallel auszuführende Funktionen identische Farben bekommen. Im Abhängigkeitsgraph darf kein Zyklus identisch gefärbter Knoten existieren, da dann eine parallele Ausführung der zugehörigen Funktionen nicht möglich ist. Andererseits kann man auch zeigen, daß falls obige Bedingung erfüllt ist, eine Variablenanordnung existiert, die gleich gefärbte Funktionen parallel auswertet. Ziel ist es nun, den Abhängigkeitsgraphen mit möglichst wenigen Farben zu färben. Leider ist das zugehörige Problem aus der Graphentheorie als NP-vollständig bekannt. Für spezielle Abhängigkeitsstrukturen lassen sich aber weitergehende Aussagen treffen. So ist bekannt, daß Graphen, in denen jeder Knoten maximal N Nachbarn hat, mit maximal $N+1$ Farben gefärbt werden können.

Zum Abschluß dieses Abschnitts werden wir eine spezielle, aber sehr oft auftauchende Abhängigkeit der Variablen betrachten und eine optimale Färbung vorstellen. Die Variablen seien in einem zweidimensionalem Array angeordnet, d.h. jede Variable ist durch zwei Koordinaten i, j mit $1 \leq i, j \leq M$ bestimmt. Weiterhin seien nur benachbarte Variablen, also solche, die sich in einer Koordinate unterscheiden, voneinander abhängig. Je nach Art der Diskretisierung ist $x_{i,j}$ von $x_{i-1,j}$, $x_{i,j-1}$, $x_{i+1,j}$ und $x_{i,j+1}$ (Fünfpunktdiskretisierung) oder zusätzlich noch von den Variablen $x_{i-1,j-1}$, $x_{i-1,j+1}$, $x_{i+1,j-1}$ und $x_{i+1,j+1}$ (Neunpunktdiskretisierung) abhängig, sofern die zugehörigen Koordinaten existieren. Solche Abhängigkeiten treten auf natürliche Weise bei der Diskretisierung partieller Differentialgleichungen auf. In Abbildung 3.9 sind die Abhängigkeitsgraphen für $M = 8$ und die Färbungen mit zwei bzw. vier Farben angegeben. Benachbarte und unterschiedlich gefärbte Knoten werden einem Prozessor zugewiesen, jeder Prozessor führt die Berechnung der Variablen einer Farbe durch, verschickt die Ergebnisse, berechnet die nächste Variable usw. Durch diese Zuordnung von Funktionen zu Prozessoren ist das Gauß-Seidel Verfahren genauso "parallel" wie das Jacobi Verfahren mit gleicher Zuordnung auszuführen. Ein weiterer Vorteil, der diese Art von Problemen besonders geeignet für parallele Algorithmen macht, ist, daß nur benachbarte Prozessoren miteinander kommunizieren müssen. Da bei den meisten praktischen Problemen die Anzahl der Variablen/Funktionen die Prozessorzahl sowieso um ein Vielfaches übersteigt, ist die Anforderung, jeweils zwei oder vier Funktionen mit einem Prozessor zu bearbeiten, keine Einschränkung.

3.4 Synchronisationsaspekte in parallelen Algorithmen

In einem parallelen Algorithmus ist es notwendig, Aktivitäten einzelner Prozesse zumindest bis zu einem gewissen Grad zu koordinieren. Für eine solche Koordination wird der Algorithmus oftmals in Phasen unterteilt. Während einer Phase führt ein Prozessor eine Reihe von Operationen autonom aus, am Ende der Phase werden die Ergebnisse der Berechnung mit anderen Prozessoren ausgetauscht. Diese Art von Algorithmen bezeichnen wir als synchron. Im Gegensatz dazu sind asynchrone Algorithmen zu sehen, bei denen die Unterteilung in Phasen nicht oder zumindest nicht in der strengen Form existiert. Bisher haben wir nur synchrone Algorithmen vorgestellt und untersucht, im Rahmen dieses Abschnitts werden wir uns mit Aspekten der Synchronisation von Algorithmen befassen und dabei synchrone und asynchrone Algorithmen klassifizieren. Eine ausführliche Betrachtung von asynchronen Algorithmen findet sich in Kapitel 6, dort wird ein formales Modell für asynchrone Algorithmen eingeführt und einige konkrete asynchrone Algorithmen vorgestellt. Die Beobachtungen basieren auf Systemen mit verteiltem Speicher, die über den Austausch von Nachrichten kommunizieren, lassen sich aber auch auf Systeme mit geteiltem Speicher übertragen.

3.4.1 Synchrone Algorithmen

Bei jedem synchronen Algorithmus wird der Ablauf in Phasen unterteilt. Innerhalb einer Phase sind die einzelnen Prozessoren völlig unabhängig, am Ende einer Phase findet die Synchronisation zwischen den Prozessoren statt. Am Ende einer Phase sendet Prozessor i Nachrichten an eine Menge von anderen Prozessoren. Dabei kann es auch vorkommen, daß einem Prozessor nicht bekannt ist, von welchen anderen Prozessoren er Nachrichten bekommt. In synchronen Algorithmen muß davon ausgegangen werden, daß eine Phase erst startet, wenn die vorherige abgeschlossen ist. Um dies sicherzustellen, ist ein Synchronisationsmechanismus notwendig. Man unterscheidet globale und lokale Synchronisation.

Bei der globalen Synchronisation ist die zentrale Idee, daß jeder Prozessor entdeckt, wenn alle Nachrichten in einer Phase gesendet wurden und erst dann mit der nächsten Phase beginnt. Falls obere Schranken für die Berechnungszeiten jedes Prozessors und für die Zeit zur Nachrichtenübertragung bekannt sind, so kann jeder Prozessor bis zum Ablauf dieser beiden Zeiten warten und dann mit der nächsten Phase beginnen. Dieses Vorgehen ist zwar einfach, scheitert aber daran, daß obere Schranken i.a. nicht bekannt oder aber nur als sehr konservative Schätzungen verfügbar sind. In jedem Fall ist das Vorgehen äußerst ineffizient. Eine andere Möglichkeit besteht darin, daß jeder Prozessor nach Beendigung einer Phase eine Endnachricht an alle anderen Prozessoren versendet. Dieses Vorgehen führt zu einem zusätzlichem allgemeinem Broadcast am Ende jeder Phase. Besser ist es, einen Spannbaum in der Topologie zu definieren. Prozessoren senden ihre Endnachrichten an ihren Elterknoten, nachdem sie die Phase beendet haben und die Endnachrichten aller Kinderknoten empfangen haben. Hat die Endnachricht den Wurzelknoten des Spannbaums erreicht, verteilt dieser eine Endnachricht für diese Phase an alle Prozessoren entlang des Spannbaums. Ein Prozessor beginnt nach Empfang der Endnachricht mit der nächsten Phase. Die benötigte zusätzliche Kommunikation entspricht der einfachen Akkumulation und einem anschließenden einfachen Broadcast. Der Aufwand einer globalen Synchronisation ist damit proportional zur maximalen Weglänge in der verwendeten Topologie.

Die zentrale Idee bei der lokalen Synchronisation besteht darin, daß ein Prozessor zum Beginn der neuen Phase nur wissen muß, daß er alle Nachrichten empfangen hat, nicht aber ob für alle anderen Prozessoren dies auch gilt. Unproblematisch ist der Fall, daß jedem Prozessor bekannt ist, welche Nachrichten er am Ende einer Phase erwarten muß. Ein Prozessor beginnt mit der nächsten Phase, wenn er alle erwarteten Nachrichten empfangen hat. Prozessoren, die sich noch in der vorherigen Phase befinden, beeinflussen das Verhalten nicht. Schwieriger wird das Ganze wenn nicht bekannt ist, welche Nachrichten am Ende einer Phase eintreffen. Ist die Menge von Prozessoren bekannt, von denen potentiell Nachrichten zu erwarten sind, so kann das Problem gelöst werden, indem jeder Prozessor aus dieser Menge entweder eine Nachricht mit neuen Ergebnissen oder ein Dummy sendet. Dieses Vorgehen ist natürlich nur sinnvoll, wenn in den meisten Fällen Nachrichten mit Inhalt gesendet werden, sonst erhöht sich der Kommunikationsoverhead durch die Dummy Nachrichten zu sehr und ein globales Synchronisationsverfahren ist vorzuziehen.

Ein eindeutiger Nachteil aller synchronen Verfahren ist, daß die gesamte Berechnung nicht schneller als der langsamste Prozeß sein kann, da am Ende einer Phase auf diesen gewartet werden muß. Auch wenn dies nur für einzelne Prozessoren gilt, so kann doch davon ausgegangen werden, daß der Graph der Kommunikation alle Prozesse umfaßt. Dadurch bedingt beeinflußt die Geschwindigkeit des langsamsten Prozessors nach und nach alle anderen Prozessoren. Geschwindigkeitsunterschiede können verschiedene Gründe haben. Das Problem kann schlecht zerlegt worden sein, so daß einzelne Prozessoren mehr Arbeit verrichten müssen als andere. Es kann aber auch vorkommen, daß ein System Prozessoren unterschiedlicher Leistungsfähigkeit beinhaltet oder daß Prozessoren unterschiedlich ausgelastet sind.

3.4.2 Asynchrone Algorithmen

Der notwendige Synchronisationsaufwand und die Laufzeit vieler Algorithmen kann reduziert werden, indem eine asynchrone Implementierung vorgenommen wird. Ein asynchroner Algorithmus hat nicht die strenge Einteilung in Phasen, wie wir sie bei synchronen Algorithmen kennengelernt haben. In einem total asynchronem Algorithmus führt jeder Prozeß seine Berechnungen durch, ohne sich um andere Prozesse zu kümmern. Es werden zur Berechnung jeweils die aktuell verfügbaren Resultate der anderen Prozesse verwendet. Dies bedeutet natürlich auch, daß der Ablauf des gesamten Algorithmus von der Geschwindigkeit der einzelnen Prozesse abhängig ist und im allgemeinen nicht reproduziert werden kann. Neben total asynchronen Algorithmen gibt es noch die abgeschwächte Form der partiell asynchronen Algorithmen, bei denen davon ausgegangen wird, daß eine obere Schranke B für ein Zeitintervall existiert, in dem jeder Prozeß mindestens einmal seine Resultate auf den neuesten Stand gebracht hat. Die verfügbare Information jedes Prozesses ist damit maximal B Zeiteinheiten alt.

Wir können synchrone und asynchrone Algorithmen auch noch aus einer etwas anderen Sicht betrachten, nämlich als eine Menge einzelner Algorithmen, die auf den Prozessoren ablaufen und miteinander kommunizieren und sich synchronisieren. Im synchronen Algorithmus wird eine globale Uhr dadurch eingeführt, daß am Ende einer Phase alle Algorithmen synchronisiert werden. Im asynchronen Algorithmus existiert keine globale Uhr, sondern nur die lokale Zeit, die für andere Prozessoren keine Bedeutung hat. Dies hat auch zur Folge, daß der Ablauf synchroner Algorithmen reproduzierbar ist, während der Ablauf asynchroner Algorithmen i.a. nicht reproduzierbar ist, da neben den Eingabedaten auch die lokalen Laufzeiten und die Kommunikationszeiten den Ablauf beeinflussen. Andererseits paßt das Paradigma asynchroner Algorithmen besser zur Sichtweise paralleler Architekturen ohne globale Kontrolle.

In Abbildung 3.10 wird der prinzipielle Ablauf eines synchronen und eines asynchronen iterativen Algorithmus dargestellt. Der horizontale Ablauf beschreibt die Zeit, Pfeile beschreiben das Senden und Empfangen von Nachrichten. Es wird deutlich, daß ein asynchroner Algorithmus deutlich mehr Iterationen durchführt, wobei die Iterationen aber ältere Resultate des anderen Prozessors berücksichtigen. Es ist deshalb nicht klar, inwieweit die größere Iterationszahl tatsächlich eine schnellere Konvergenz bewirkt, oder ob vielleicht überhaupt keine Konvergenz beim asynchronen Algorithmus vorliegt. Wir werden uns mit diesen Problemen später beschäftigen, wobei Aussagen natürlich immer nur für spezielle Verfahren getroffen werden können. Im Moment reicht es aus hervorzuheben, daß die Geschwindigkeitsunterschiede zwischen synchronen und asynchronen Algorithmen drei Gründe haben. Erst einmal unter-

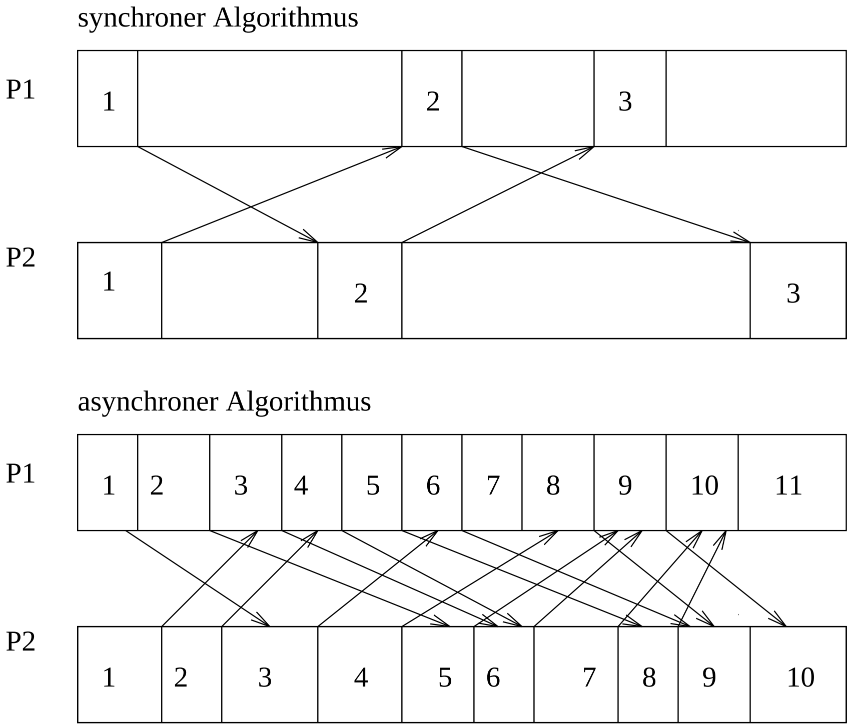


Abbildung 3.10: Schematischer Ablauf des synchronen und asynchronen Algorithmus.

schiedliche Geschwindigkeiten in der Funktionsausführung, dann die Laufzeit der Nachrichten zwischen den Prozessen und zum dritten Schwankungen in den Nachrichtenlaufzeiten hervorgerufen durch Übertragungsfehler oder höher belastete Verbindungsleitungen. Dadurch bedingt kann es im Extremfall auch vorkommen, daß Nachrichten in einer anderen Reihenfolge beim Empfänger ankommen als sie beim Sender abgesendet wurden, auch wenn dies bei Multiprozessorarchitekturen i.a. nicht der Fall sein wird. Selbst im günstigsten Fall, wenn beide Prozessoren gleich schnell ihre Funktion bearbeiten und die Kommunikationszeiten deterministisch sind, treten durch die Kommunikationszeit bedingt Verzögerungen und Untätigkeitsintervalle für die Prozesse auf. Die Unterschiede zwischen synchronen und asynchronen Algorithmen sind dann am größten, wenn die Unterschiede in den einzelnen Zeiten besonders groß sind.

3.5 Parallele Programmierung

Es gibt im wesentlichen zwei Ansätze parallele Programme zu realisieren. Es können spezielle parallele Programmiersprachen verwendet werden, oder es werden allgemeine sequentielle Programmiersprachen wie C, C++ oder Fortran verwendet und die Kommunikation wird auf Basis von Standardbibliotheken realisiert. Beide Ansätze werden im folgenden kurz behandelt.

3.5.1 Parallele Programmiersprachen

Es gibt eine große Zahl paralleler Programmiersprachen, im Bereich des wissenschaftlichen Rechnens und der Computer Numerik werden aber im wesentlichen prozedurale Sprachen und dort speziell Fortran oder C Dialekte eingesetzt. Inwieweit sich dies in Zukunft ändert bleibt abzuwarten. Insbesondere objektorientierte Sprachen bieten im konzeptuellen Bereich interessante Ansätze zur Realisierung paralleler und verteilter Numerikanwendungen. In der Praxis werden objektorientierte Sprachen aber, mit Ausnahme von ersten Ansätzen in C++, kaum zur Entwicklung numerische Software eingesetzt. Dies könnte sich aber ändern, wie die Verfügbarkeit von Linpack in JAVA zeigt.

Selbst wenn man die Betrachtung paralleler Sprachen auf Erweiterungen von Fortran und C/C++ einschränkt, stößt man auf eine Vielzahl von Dialekten, denen unterschiedlichste Konzepte zu Grunde liegen. So gibt es Dialekte, die Systeme mit geteiltem Speicher und andere, die Systeme mit verteiltem Speicher unterstützen. Im ersten Fall müssen Synchronisationsmechanismen verfügbar sein, im zweiten Fall sind Basisoperationen zum Nachrichtenaustausch notwendig. So gibt es für fast jede Multiprozessorlinie ein spezielles Fortran Dialekt. Der Nachteil dieser Vielzahl von Dialekten ist offensichtlich, da Portabilität von Programmen praktisch nicht vorhanden ist.

Wir werden uns hier kurz mit einer recht allgemeinen und verbreiteten parallelen Programmiersprache beschäftigen, nämlich pC++, eine Erweiterung von C++, die *High Performance Fortran* nachempfunden wurde. pC++ ist eine ganze Reihe von Architekturen verfügbar, es gibt inzwischen sogar eine Variante für Workstation Netze auf Basis der Bibliothek PVM. pC++ unterstützt Systeme mit verteiltem und mit geteiltem Speicher. pC++ unterstützt im wesentlichen datenparallele Programmierung Sprachkonstrukt *Collection Class*. Eine Programmierung im SPMD Stil (<u>S</u>ingle <u>P</u>rogramm <u>M</u>ultiple <u>D</u>ata) wird durch die explizite Definition von Threads in einem Programm ebenso unterstützt.

Die Parallelität, die von pC++ unterstützt wird geht über die Datenparallelität vieler anderer Programmiersprachen hinaus und wird als Objektparallelität bezeichnet. Es besteht die Möglichkeit *Collection Classes* aus beliebigen Klassen, auch anderen *Collection Classes*, zu bilden. Operationen werden dann parallel auf allen Elementen einer *Collection Class* ausgeführt. Es gibt eine Klassen Bibliothek mit *Collection Classes* zur Beschreibung von Vektoren, Matrizen, Gittern und ähnlichen Strukturen, die in numerischen Algorithmen benötigt werden. Die Verteilung der einzelnen Klassenelemente erfolgt in einem Zweischrittverfahren, welches von HPFF Fortran übernommen wurde. Es wird ein Template definiert, welches ein ein- oder mehrdimensionales Array darstellt. Dieses Template wird auf die Prozessoren abgebildet. Anschließend werden die Elemente der *Collection Class* auf das Template abgebildet. Die Zuordnung von Daten zu Prozessoren ist damit vom Programmierer vorzunehmen. Die Operationen auf den Daten werden anschließend ohne Berücksichtigung der Datenverteilung programmiert, die eventuell notwendige Kommunikation zur Operationsausführung wird automatisch realisiert. Damit ist die Programmierung unabhängig von der Datenverteilung, nicht aber die Laufzeit des Programms.

3.5.2 Bibliotheken zur Realisierung paralleler Programme

Als Alternative zur Verwendung paralleler Programmiersprachen bieten sich Bibliotheken zur parallelen Programmierung an. Insbesondere *PVM* und *MPI* sind weit verbreitet. *PVM* (Parallel Virtual Machine) ist eine Bibliothek mit deren Entwicklung 1989 begonnen wurde. Ziel war es, insbesondere numerische Anwendungen auf heterogenen Netzwerken zu unterstützen. *PVM* ist im Rahmen der *netlib* frei verfügbar. *MPI* (Message Passing Interface) ist ein Standard, der in Zusammenarbeit unterschiedlicher Gruppen entworfen wurde. *MPI* selber ist damit noch keine Programmierumgebung, es gibt aber inzwischen verschiedene Implementierungen des Standards, die teils frei verfügbar und teils kommerziell sind.

Die Programmierbibliotheken lassen sich in Kombination mit vielen Programmiersprachen verwenden. Sie stellen dem Anwender Operationen zum Nachrichtenaustausch auf Basis einer virtuellen Maschine zur Verfügung. Damit sind die Programme über eine Vielzahl von Plattformen portabel. Kommunikation und Synchronisation müssen explizit im Programm realisiert werden. Sowohl PVM als auch MPI bieten verschiedene Kommunikationssemantiken an. Diese umfassen blockierende und nicht-blockierende Kommunikation. Weiterhin werden mächtige Synchronisationskonzepte, wie Broadcasts und Synchronisationen zwischen Prozeßgruppen unterstützt. Es existieren für PVM und MPI verschieden Tools, die das Prozeßmanagement und -monitoring unterstützen. Damit bieten sich diese Bibliotheken zur Entwicklung paralleler Anwendungen eigentlich an. Das praktische Problem ist allerdings, daß der Funktionsumfang der Bibliotheken seinen Preis in einem nicht zu vernachlässigenden Overhead hat. So haben Messungen auf einer Intel Paragon ergeben, daß die Kommunikation über PVM je nach Einstellung und Paketgröße 4 bis 20 mal langsamer ist als die Kommunikation mit den Basiskommunikationsfunktionen des Rechners ist. Es gibt deshalb Bestrebungen, optimierte Varianten von PVM zu realisieren. Der Preis für mehr Leistung ist aber ein eingeschränkter Funktionsumfang und eingeschränkte Portabilität. Es soll hier nicht auf die Spezifika von MPI oder PVM eingegangen werden, es zeigt sich aber, daß MPI deutlich mächtiger ist und dem Benutzer mehr Freiheitsgrade bietet. PVM ist dagegen in der Benutzung einfacher zu handhaben.

3.6 Literaturhinweise

Die O-Notation kann man in jedem Buch finden, das sich mit der Komplexität von Algorithmen beschäftigt. Bewertungsmaße für parallele Algorithmen sind in den meisten Büchern, die sich mit parallelen Algorithmen beschäftigen. Exemplarisch sei auf [4, 27] verwiesen. Meist beschränkt sich die Einführung von Komplexitätsmaßen für parallele Algorithmen aber auf die Effizienz und den Speedup. Maße wie die Isoeffizienz sind seltener zu finden. Eingeführt wird die Isoeffizienzfunktion meines Wissens nach in einer Konferenzversion von [19], sie wird aber auch in verschiedenen Versionen in [33] behandelt. Präzedenzgraphen sind ein bekannter Ansatz zur Darstellung von Algorithmen, der auch in [4, 17] zu finden ist. Resultate über synchrone und asynchrone Algorithmen sind in [4, 13] dargestellt. Für parallele Programmiersprachen sei auf [5, 30, 38] verwiesen, Kommunikationsbibliotheken MPI und PVM werden in [6, 11, 29] behandelt.

Kapitel 4

Numerische Basisalgorithmen

In den verschiedenen numerischen Verfahren, die in den folgenden Kapiteln vorgestellt werden, wird eine Menge generischer Basisoperationen verwendet. Es lohnt sich deshalb, für diese Operationen effiziente Basisalgorithmen einzuführen und später auf diese Realisierungen zu verweisen. In diesem Kapitel beschäftigen wir uns mit generischen Operationen auf Vektoren und Matrizen. Wir betrachten dazu die folgenden Basisoperationen.

- Das Produkt zweier n-dimensionaler Vektoren $\mathbf{x}$ und $\mathbf{y}$, welches durch $\mathbf{x}\mathbf{y}^T$ oder $\sum_{i=1}^{n} x_i y_i$ definiert ist.

- Das Produkt eines n-dimensionalen Vektors $\mathbf{x}$ mit einer $m \times n$ Matrix $\mathbf{A}$, welches einen m-dimensionalen Vektor $\mathbf{y} = \mathbf{A}\mathbf{x}$ mit $y_j = \sum_{i=1}^{n} A_{j,i} x_i$ ergibt.

- Das Produkt einer $n \times l$ Matrix $\mathbf{A}$ mit einer $l \times m$ Matrix $\mathbf{B}$, welches eine $n \times m$ Matrix $\mathbf{C} = \mathbf{AB}$ mit $C_{i,j} = \sum_{k=1}^{l} A_{i,k} B_{k,j}$ liefert.

Weitere Operationen, die wir aber nicht gesondert behandeln werden, sind die Addition von Matrizen und Vektoren. Wir betrachten im allgemeinen quadratische Matrizen, wie sie in den meisten Anwendungen üblich sind. Für die vorgestellten Algorithmen ist eine Erweiterung auf den allgemeinen Fall der $n \times m$ Matrizen aber problemlos möglich.

In den folgenden Abschnitten werden zuerst Datenstrukturen für Vektoren und Matrizen eingeführt. Daran anschließend betrachten wir Algorithmen für dicht besetzte Matrizen und im letzten Abschnitt werden die Algorithmen für dünn besetzte Matrizen beschrieben.

4.1 Datenstrukturen für Matrizen und Vektoren

In diesem Abschnitt sollen kurz die Basisdatenstrukturen für die numerische Analyse eingeführt werden. Die Datenstrukturen sind unabhängig von einer konkreten Programmiersprache. In Bereich der Numerik wird immer noch FORTRAN als Standardsprache eingesetzt, die meisten frei oder kommerziell verfügbaren Numerikpakete sind in FORTRAN implementiert. In letzter Zeit findet man auch vereinzelte C Implementierungen und, noch seltener, C++ Realisierungen. Vom Standpunkt der strukturierten Programmentwicklung sind alle genannten Sprachen nicht optimal, dies wird aber leider oftmals bei der Realisierung numerischer Algorithmen als von untergeordneter Bedeutung gesehen. Wir wollen uns mit den Gründen für die weite Verbreitung der genannten Sprachen im Bereich des wissenschaftlichen Rechnens hier nicht auseinandersetzen. Man sollte allerdings folgende Rechtfertigungen für den Einsatz von insbesondere FORTRAN zur Kenntnis nehmen. Es kommt bei den meisten numerischen Anwendungen auf höchst mögliche Geschwindigkeit und Speichereffizienz an, dies rechtfertigt oftmals den Einsatz von

FORTRAN und C. Des weiteren werden viele numerische Programme zumindest teilweise aus Bausteinen zusammengesetzt, die aus Programmbibliotheken entnommen werden. Die Wiederverwendung von Software ist in diesem Bereich also durchaus üblich. Dies hat zur Folge, daß eine einmal festgelegte Sprache nur schwer zu verdrängen ist, hat aber auch den Vorteil, daß die verwendeten Bausteine meist korrekt und effizient sind. Leider gilt letzteres uneingeschränkt nur für sequentielle Implementierungen. Im parallelen Bereich sind viele Teile eines Algorithmus abhängig von der verwendeten Rechnerarchitektur und damit nicht mehr so einfach aus Standardbibliotheken zu entnehmen. Ob es in Zukunft möglich sein wird, auch standardisierte Bibliotheken für parallele numerische Algorithmen zu entwerfen, bleibt abzuwarten und ist zum jetzigen Zeitpunkt nicht vorhersehbar. Das Thema der Programmiersprachen soll mit einer allgemeinen Bemerkung beendet werden. Für numerische Anwendungen sollte die höchst mögliche Genauigkeit für Gleitpunktzahlen gewählt werden, auch wenn dies zusätzlichen Speicher kostet. In vielen Anwendungen und den meisten Algorithmen treten numerische Rundungsfehler auf, die bei einer genaueren Gleitpunktarithmetik zwar nicht vermieden, aber oftmals reduziert werden können.

Die Speicherung von Vektoren ist im allgemeinen einfach. Es wird ein n-dimensionales Array verwendet. Interessanter ist die Darstellung von Matrizen. Wenn alle Elemente einer Matrix gespeichert werden, so spricht man von einer dicht besetzten Darstellung. Eine solche Darstellung wird durch ein zweidimensionales Array realisiert. Eine quadratische Matrix der Dimension n erfordert die Speicherung von n^2 Elementen.

Viele Systeme, die durch mathematische Modelle analysiert werden, benötigen eine große Anzahl von Variablen und beschreiben lokale Wechselwirkungen zwischen Variablen. Wechselwirkungen zwischen den Variablen werden üblicherweise in quadratischen Matrizen dargestellt. Liegen nun nur Wechselwirkungen zwischen einigen Variablen vor, so ist ein großer Teil der Matrix Elemente 0. D.h. die Anzahl der Nichtnullelemente ist oft im Bereich von cn für ein festes c, was für große n deutlich kleiner als n^2 ist. Solche Matrizen werden als dünn, schwach oder spärlich besetzte Matrizen bezeichnet. Benutzt man zur Abspeicherung dünn besetzter Matrizen zweidimensionale Arrays, so wird nicht zwischen Null- und Nichtnullelementen unterschieden. Dadurch wird Speicherplatz und auch Rechenzeit verschwendet, da Nullelemente keinen Einfluß auf das Ergebnis bei den vorgestellten Operationen haben. Naheliegend ist es deshalb, nur die Nichtnullelemente abzuspeichern. Die resultierenden Speicherstrukturen nennt man spärlich oder dünn besetzte Matrixformate. Es gibt zahlreiche Varianten, von denen einige kurz vorgestellt werden. Allen Varianten haben gegenüber dicht besetzten Strukturen den Vorteil, daß der Speicherplatz proportional zur Anzahl der Nichtnullelemente in der Matrix ist. Die kompakte Speicherung wird allerdings damit bezahlt, daß auf einzelne Matrixelemente nicht mehr direkt zugegriffen werden kann und die Matrix je nach Speicherstruktur nicht oder nur eingeschränkt modifizierbar ist. Beide Aspekte müssen bei der Realisierung von Algorithmen für dünn besetzte Matrizen beachtet werden.

Man unterscheidet drei Gruppen von dünn besetzten Matrixstrukturen. Die erste Gruppe umfaßt Datenstrukturen für Matrizen, die einmal gespeichert und anschließend nicht mehr geändert werden. Solche Strukturen können in iterativen Verfahren Verwendung finden; nachdem die Iterationsmatrix erzeugt wurde, wird sie zur Berechnung von Vektor-Matrix-Produkten verwendet, bleibt aber konstant. Die zweite Gruppe umfaßt Speicherstrukturen, die spezifische Veränderungen erlaubt, wie sie bei der direkten Lösung linearer Gleichungssysteme notwendig sind. In der dritten Gruppe werden Datenstrukturen für dünn besetzte Matrizen mit spezieller Besetzungsstruktur behandelt.

4.1.1 Dünn besetzte Matrixstrukturen für iterative Verfahren

Die verschiedenen Speicherstrukturen werden anhand der folgenden Matrix erläutert[1]. Zur besseren Lesbarkeit sei darauf hingewiesen, daß Nichtnulleinträge mit Werten $A_{i,j} = ij$ belegt wurden.

$$\mathbf{A} = \begin{pmatrix} 11 & 12 & 0 & 14 & 0 & 0 \\ 0 & 22 & 23 & 0 & 0 & 0 \\ 31 & 0 & 33 & 34 & 0 & 0 \\ 0 & 42 & 0 & 44 & 45 & 46 \\ 0 & 0 & 0 & 0 & 55 & 56 \\ 0 & 0 & 0 & 0 & 65 & 66 \end{pmatrix}$$

Wir definieren $nz(\mathbf{A})$ als die Anzahl der Nichtnullelemente in Matrix **A**. In unserem Beispiel ist $nz(\mathbf{A}) = 16$. Der Besetzungsgrad ist 44%, was i.a. nicht mehr als spärlich besetzt gilt.

COO-Format: Koordinatenformat

Dies ist ein sehr einfaches Format, daß für jedes Nichtnullelement zwei Integer Werte zur Speicherung des Zeilen- und Spaltenindex, sowie ein Fließkommazahl zur Speicherung des Matrixeintrags verwendet. Die Werte werden in Arrays der Länge $nz(\mathbf{A})$ gespeichert. Für die gesamte Matrix müssen $2 * nz(\mathbf{A})$ Integer und $nz(\mathbf{A})$ Fließkommazahlen gespeichert werden. Nichtnullelemente werden in beliebiger Reihenfolge gespeichert. Damit erfordert das Suchen eines Elements einen Aufwand in $O(nz(\mathbf{A}))$. Zeilen- oder spaltenweise Durchläufe durch die Matrix sind unmöglich, bzw. erforderten das Suchen jedes einzelnen Elements. Dies ändert sich auch nicht, wenn Elemente zeilen- oder spaltenweise sortiert gespeichert werden, da die Anzahl der Nichtnullelemente in einer Spalte oder Zeile nicht a priori bekannt ist. Das Einfügen von Elementen ist im COO-Format recht einfach, da neue Elemente nur hinten an das Array angehängt werden müssen. Die Beispielmatrix wird wie folgt im COO-Format dargestellt.

wert	65	56	46	34	44	45	22	66	11	12	31	55	23	33	14	42
row	6	5	4	3	4	4	2	6	1	1	3	5	2	3	1	4
col	5	6	6	4	4	5	2	6	1	2	1	5	3	3	4	2

Der Speicherplatz kann noch etwas verringert werden, wenn Zeilen und Spaltenindex in einem Integer kodiert werden. Der Index i, j wird dann zu $(i-1)n + j$.

CRS- und CCS-Format: Komprimierte Zeilen-/Spaltenspeicherung

Die beiden Formate unterscheiden sich dadurch, daß in einem Fall zeilenweise, im anderen Fall spaltenweise gespeichert wird. Wir betrachten hier die zeilenweise Speicherung. Dies entspricht der spaltenweisen Speicherung der transponierten Matrix. Werte der Nichtnullelemente und deren Spaltenindizes werden in zwei Arrays konsekutiv gespeichert werden. In einem zusätzlichen Integer Array *row* der Länge $n+1$ werden die Indizes des ersten Elements der korrespondierenden Zeile in den beiden anderen Arrays gespeichert. Im letzten Element des Arrays wird der Wert $nz(\mathbf{A}) + 1$ gespeichert. Die Speicherstruktur wird am Beispiel deutlich.

wert	11	12	14	22	23	31	33	34	42	44	45	46	55	56	65	66
col	1	2	4	2	3	1	3	4	2	4	5	6	5	6	5	6

row	1	4	6	9	13	15	17

[1] Das Beispiel wurde aus [36] entnommen.

Die Nichtnullelemente der Zeile i sind also an den Positionen $row(i)$ bis $row(i+1)-1$ zu finden. Damit ist ein zeilenweiser Durchlauf durch die Matrix leicht zu realisieren und auch der Zugriff auf ein Element wird deutlich schneller, da nur im Intervall $[row(i), row(i+1)-1]$ gesucht werden muß, wenn auf ein Element aus Zeile i zugegriffen wird. Ein spaltenweiser Durchlauf oder Modifikationen der Matrix sind allerdings aufwendig. Aufwendig ist auch das Einfügen neuer Elemente, da die gesamte Struktur zu modifizieren ist.

Zur Speicherung einer Matrix im CRS-Format werden jeweils ein Vektor der Länge $nz(\mathbf{A})$ mit Integer und Reals benötigt. Zusätzlich wird ein Integer Vektor der Länge n gebraucht. Da normalerweise $n < nz(\mathbf{A})$ gilt, benötigt das CRS-Format weniger Speicherplatz als das COO Verfahren.

Eine Variante des CRS-Formats ist das MCRS-Format, bei dem die Diagonalelemente separat gespeichert werden. Eine solche Speicherung ermöglicht einen schnellen Zugriff auf die Diagonalelemenete, der in manchen Algorithmen notwendig ist. Im MCRS-Format werden ein Integer Array und ein Real Array der Länge $nz(\mathbf{A}) - \sum_{i=1}^{n} \delta(A_{i,i}) + n + 1$ benötigt[2]. An den ersten n Positionen des Real Ararys werden die Werte der Diagonalelemente gespeichert. Die ersten n Elemente des Integer Arrays beinhalten die Elemente des Arrays row im CRS-Format, wobei Diagonalelemente nicht berücksichtigt werden und der Index für das erste Nichtnull- und Nichtdiagonalelement $n+2$ ist. Element $n+1$ im Real Array bleibt leer und enthält imInteger Array die Länge des gesamten Arrays plus 1. Ab Element $n+2$ werden wie im CRS-Format die Nichtnull- und Nichtdiagonalelemente gespeichert. Für unsere Beispielmatrix ergibt sich folgende Struktur.

wert	11	22	33	44	55	66		12	14	23	31	34	42	45	46	56	65
index	8	10	11	13	16	17	18	2	4	3	1	4	2	5	6	6	5

4.1.2 Dünn besetzte Matrixstrukturen für direkte Verfahren

Die bisher vorgestellten Speicherstrukturen für dünn besetzte Matrizen sind nicht geeignet wenn die Matrix modifiziert werden muß. Eine spezielle Art von Modifikation tritt bei der direkten Lösung linearer Gleichungssysteme auf. Die Matrix wird zeilenweise bearbeitet, dabei entstehen in einer Zeile unter Umständen neue Nichtnullelemente. Dies bezeichnet man als *Fill In*. Eine Datenstruktur, die für direkte Verfahren verwendet wird, muß also die Veränderung einzelner Zeilen der Matrix ermöglichen, ohne daß die gesamte Speicherstruktur geändert werden muß. Des weiteren ist ein zeilenweiser Durchlauf durch die Matrix notwendig.

Um Zeilen einzeln zu bearbeiten, betrachten wir zwei Ansätze. Der erste Ansatz benutzt zwei Integer und ein Real Array der Länge $n + nz(\mathbf{A})$ und ist von den bereits vorgestellten Speicherstrukturen abgeleitet. Der zweite Ansatz basiert auf dynamischen Datenstrukturen und Zeigern.

Wir beginnen mit dem ersten Ansatz. Wie in den bereits vorgestellten Formaten werden die Integer Arrays mit *row* und *col* bezeichnet, das Real Array wird mit *wert* bezeichnet. Die ersten n Elemente des Arrays *val* sind mit 0 belegt, in den folgenden Positionen werden die Werte der Nichtnullelemente in beliebiger Reihenfolge gespeichert. Das Array *row* enthält in Position $i \leq n$ den Index, an dem das erste Nichtnullelement aus Zeile i gespeichert ist. Sei k dieser Index. An der Stelle *row(k)* steht der Index des nächsten Nichtnullelements aus Zeile i usw. Falls k das letzte Nichtnullelement der i-ten Zeile ist, so enthält *row(k)* den Wert i. Im Array *col* wird vollkommen analog bzgl. der Spalten verfahren. Zeilen und Spaltendurchläufe sind in diesem Format möglich. Die Elemente einer Zeile oder Spalte werden aber nicht unbedingt konsekutiv durchlaufen. Insofern ist das Suchen eines Elements relativ aufwendig. Das Einfügen eines

[2] $\delta(A_{i,i}) = 1$ falls $A_{i,i} \neq 0$ und 0 sonst.

Elements erfordert nur das Anhängen des Wertes und die Veränderung von je zwei Einträgen in den Arrays *row* und *col*.

Unsere Beispielmatrix kann wie folgt in diesem Format dargestellt werden. Die Darstellung ist nicht eindeutig, da die Anordnung der Nichtnullelemente beliebig ist.

	1	2	3	4	5	6	7	8	9	10	11
wert	0	0	0	0	0	0	65	56	46	34	44
row	15	13	10	9	8	7	14	18	11	17	12
col	15	13	19	10	7	8	12	9	14	11	21

	12	13	14	15	16	17	18	19	20	21	22
wert	45	22	66	11	12	31	55	23	33	14	42
row	22	19	6	16	21	20	5	2	3	1	4
col	18	16	6	17	22	1	5	20	3	4	2

Die bisher vorgestellten Matrixstrukturen sind in verschiedenen Programmbibliotheken zu finden und richten sich insbesondere auf FORTRAN als Implementierungssprache aus. Als letztes betrachten wir eine zeigerorientierte Darstellung, die eher den Programmiersprachen C oder C++ zuzuordnen ist. Die Matrix wird zeilen- oder spaltenweise dargestellt, wobei in einer Zeile/Spalte nur die Nichtnullelemente gespeichert werden. Wir betrachten hier eine zeilenweise Speicherung, die spaltenweise Speicherung erfolgt vollkommen analog. Es wird ein Array *row* der Länge n benötigt, dessen Elemente Zeiger auf zwei Arrays sind, nämlich ein Real Array *wert* und ein Integer Array *col*. Weiterhin wird ein Integer Array *nz* der Länge n benötigt. *nz(i)* enthält die Anzahl der Nichtnullelemente in Zeile i. Die Arrays *wert* und *col*, auf die *row(i)* zeigt, haben die Länge *nz(i)*. Die Arrays enthalten die Werte und Spaltenindizes der Nichtnullelemente in Zeile i. Man geht im allgemeinen davon aus, daß die Elemente nach steigendem Spaltenindex geordnet sind. Ist dies der Fall, so ist ein zeilenweiser Durchlauf durch die Matrix einfach zu realisieren. Das Suchen eines Elements erfordert Suchen in der entsprechenden Zeile. Der spaltenweise Durchlauf ist aufwendig. Einfügen eines Elements erfordert die Modifikation der Zeile. Sind die Elemente nicht geordnet, so kann das neue Element hinten angehängt werden. Bei geordneten Spalten muß es in die Zeile eingefügt werden. Bei der Verwendung der Struktur in direkten Lösungsverfahren erfolgen die Modifikationen zeilenweise. Diese Vorgehen ist am besten dadurch zu realisieren, daß die Werte der entsprechende Zeile temporär in einem Array der Länge n gespeichert werden, die Modifikationen auf diesem Array ausgeführt werden und für die resultierenden Nichtnullelemente die Arrays *wert* und *col* erzeugt und in die Struktur eingehängt werden. Erfordert der Algorithmus separaten Zugriff auf die Diagonalelemente, so können diese in einem separaten Real Array der Länge n gespeichert werden. Abb. 4.1 zeigt die Datenstruktur für die Beispielmatrix ohne separate Behandlung der Diagonalelemente.

4.1.3 Dünn besetzte Matrixstrukturen für Matrizen mit spezieller Besetzungsstruktur

Matrizen für viele Probleme haben spezifische Strukturen, die bei der Speicherung ausgenutzt werden können. Betrachten wir zuerst symmetrische Matrizen. Da für eine symmetrische Matrix $\mathbf{A} = \mathbf{A}^T$ gilt, können wir eine additive Zerlegung von $\mathbf{A}$ in eine untere Dreicksmatrix $\mathbf{L}$, eine obere Dreiecksmatrix $\mathbf{U}$ und eine Diagonalmatrix $\mathbf{D}$ betrachten. Es gilt $\mathbf{L} = \mathbf{U}^T$. Damit reicht es aus, nur $\mathbf{D}$ und $\mathbf{L}$ zu speichern. Dies reduziert den Speicheraufwand um fast 50%. Für die

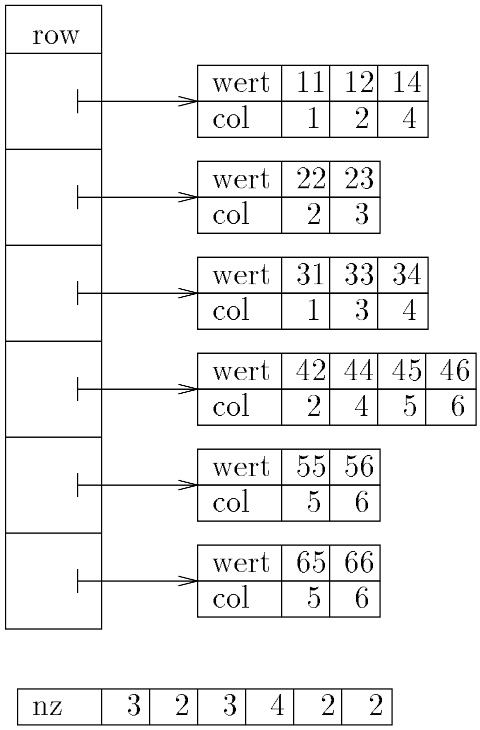


Abbildung 4.1: Verzeigerte dünn besetzte Matrixstruktur.

Speicherung von **D** und **L** kann eines der vorgestellten Verfahren für dünn besetzte Matrizen verwendet werden.

Einen weiteren Typ spezieller Matrixstrukturen bilden die Bandmatrizen. Bei einer Bandmatrix verteilen sich die Nichtnullelemente auf ein Band um die Hauptdiagonale, d.h. $A_{i,j} \neq 0$ bedingt $|i - j| \leq b$ für ein $b < n$ oder $-b_L \leq i - j \leq b_U$, wenn die Bandbreite unter- und oberhalb der Diagonalen unterschiedlich ist. Das **BND-Format** benutzt zur Speicherung einer Bandmatrix ein zweidimensionales Real Array *wert*. Der erste Index läuft von 1 bis $b_L + b_U + 1$, der zweite Index von 1 bis n. An der Stelle wert(i,j) steht das Element $A_{j-i+b_L,j}$, wobei $A_{i,j} = 0$ falls i oder j größer n oder kleiner 1 sind. Mit dieser Speicherstruktur werden alle Elemente innerhalb des Bandes gespeichert, auch wenn diese Null sind. Eine Alternative zum BND-Format ist das **CDS-Format**, bei dem die $b_L + b_U$ Nebendiagonalen die Nichtnullelemente enthalten, und die Hauptdiagonale jeweils als Real Arrays der Länge n gespeichert werden, d.h. die Nebendiagonalen werden mit Nullen auf n Elemente aufgefüllt. Da bei der direkten Lösung linearer Gleichungssysteme, die durch Bandmatrizen beschrieben werden, neue Nichtnullelemente (Fill-In) nur innerhalb des Bandes entstehen, können die vorgestellten Datenstrukturen auch für Algorithmen zur direkten Lösung von Gleichungssystemen eingesetzt werden.

Als letzte Struktur sollen Blockmatrizen angesprochen werden. Diese Matrizen enthalten Nichtnullelemente nur in bestimmten Unterblöcken und entstehen z.B. bei Gleichungssystemen, die aus der Diskretisierung partieller Differentialgleichungen resultieren. Blockmatrizen können effizient in einer zweistufigen Struktur gespeichert werden. Blöcke, die Nichtnullelemente enthalten, werden als dicht besetzte Matrizen gespeichert.

4.1.4 Einsatz der Datenstrukturen auf Parallelrechnern

Werden Parallelrechner mit verteiltem Speicher zur Analyse von Matrixproblemen eingesetzt, so wird die gesamte Matrix nicht in allen lokalen Speichern gehalten. Vielmehr bearbeitet jeder Prozessor einen Teil des Gesamtproblems und benötigt dazu nur einen Teil der gesamten

Matrix. Wie in den folgenden Algorithmen deutlich wird, wird eine Matrix zeilen-, spalten- oder blockweise auf die lokalen Speicher verteilt. Zur lokalen Speicherung dieser Substrukturen können die vorgestellten Datenstrukturen verwendet werden. Eine Teilmenge von $m < n$ Zeilen einer $n \times n$ Matrix kann als eigenständige $m \times n$ Matrix interpretiert werden. Eine Teilmenge von $m < n$ Spalten kann als $n \times m$ Matrix interpretiert werden. In der selben Weise kann ein Unterblock der die Elemente $A_{i,j}$ mit $r_1 \leq i < r_2$, $c_1 \leq i < c_2$ als $r_2 - r_1 \times c_2 - c_1$ Matrix interpretiert und behandelt werden.

4.2 Operationen auf dicht besetzten Matrizen

In diesem Kapitel werden Algorithmen für die vorgestellten Basisprobleme beschrieben. Wir haben die bestmögliche Effizienz und den bestmöglichen Speed up solcher Algorithmen bereits anhand von Präzedenzgraphen untersucht. Bei dieser Untersuchung wurde die Kommunikationszeit nicht beachtet. Wenn wir nun Algorithmen für konkrete Architekturen vorstellen, muß die Kommunikationszeit natürlich in die Analyse einfließen. Damit ist zu erwarten, daß die realen Laufzeiten schlechter als die "best case" Resultate aus dem vorherigen Kapitel sind. In den folgenden Abschnitten werden Algorithmen für die Topologien lineares Array, Gitter/Torus und Hyperwürfel vorgestellt. Viele der hier vorgestellten Algorithmen lassen sich auf Architekturen mit anderen Topologien oder geschalteten Verbindungen übertragen.

Die Kommunikationszeit wird in den folgenden Analysen einbezogen, wir verwenden aber ein vereinfachtes Modell. Dazu sei α die Anzahl der Zeiteinheiten zur Durchführung einer Operation (i.a. einer Addition oder Multiplikation). Die Übertragung von k Werten zwischen zwei verbundenen Prozessoren dauert $\beta + k\gamma$ Zeiteinheiten. Falls in allen Schritten eine konstante Datenmenge übertragen wird, benutzen wir β für die gesamte Übertragungszeit zur Vereinfachung der Analysen. Wir haben bereits gesehen, daß β deutlich größer als α sein kann. Im folgenden werden meist asymptotische Analysen durchgeführt. Dies ist detailliert genug, da das zugrundeliegende Modell einige vereinfachende Annahmen beinhaltet und deshalb klarerweise approximative Ergebnisse liefert.

4.2.1 Algorithmen für lineare Arrays

Wir haben bei der Analyse der Basiskommunikationsprobleme bereits gesehen, daß die Kommunikationsmöglichkeiten durch die große Entfernung des ersten und letzten Knotens eingeschränkt sind. Dies wird insbesondere durch die Zeit für den einfachen Broadcast bzw. die einfache Akkumulation deutlich und hat natürlich auch Einfluß auf die Laufzeit der folgenden Algorithmen.

Vektor-Produkte

Zur Berechnung des Produkts $\mathbf{xy}^T$ zweier Vektoren der Dimension n auf einem linearen Array mit p Prozessoren nehmen wir zur Vereinfachung an, daß $k = n/p$ eine ganze Zahl ist. Es werden dann auf jedem Prozessor k Elemente jedes Vektors gespeichert, wobei Prozessor i die Elemente $x_{(i-1)k+1}, \ldots, x_{ik}$ und $y_{(i-1)k+1}, \ldots, y_{ik}$ speichert. Zur Analyse seien die Daten auf den Prozessoren vorhanden.

Der Algorithmus ist sehr einfach und hat die folgende Struktur:

procedure for processor i
    berechne $z_i = \sum_{j=(i-1)k+1}^{ik} x_j y_j$;
    if $(i = 1)$

sende z_i *an Prozessor* $i+1$ *; stop ;*
if $(1 < i < \lfloor (p+1)/2 \rfloor)$
warte auf Ergebnis z_{i-1} *von Prozessor* $i-1$ *;*
$z_i = z_i + z_{i-1}$ *;*
sende z_i *an Prozessor* $i+1$ *; stop ;*
if $(\lfloor (p+1)/2 \rfloor < i < p)$
warte auf Ergebnis z_{i+1} *von Prozessor* $i+1$ *;*
$z_i = z_i + z_{i+1}$ *;*
sende z_i *an Prozessor* $i-1$ *; stop ;*
if $(i = p)$
sende z_i *an Prozessor* $i-1$ *; stop ;*
if $(i = \lfloor (p+1)/2 \rfloor)$
warte auf Ergebnisse z_{i-1} *und* z_{i+1} *von den Prozessoren* $i-1$ *und* $i+1$ *;*
berechne das Endergebnis $z_i + z_{i-1} + z_{i+1}$ *;*
end ;

Das Endergebnis liegt anschließend im Prozessor $\lfloor (p+1)/2 \rfloor$ vor. Betrachten wir nun die für die Berechnung nötige Zeit. Da nur einzelne Werte übertragen werden, sei β die Zeit zur Übertragung eines Wertes zwischen zwei verbundenen Prozessoren und α sei die Zeit für eine Multiplikation oder Addition. Nach ungefähr $2k\alpha$ Schritten haben alle Prozessoren ihre lokalen Summen berechnet. Im nächsten Schritt übertragen die Prozessoren 1 und p die Ergebnisse an ihre Nachbarn, wo die Ergebnisse β Zeiteinheiten später eintreffen. Die Nachbarn summieren ihre lokalen und die empfangenen Ergebnisse, was α Zeiteinheiten dauert, und senden die Resultate weiter. Abschließend führt der letzte Prozessor zwei Additionen durch. Der Zeitaufwand für die gesamte Berechnung lautet damit

$$T_p(n) = 2k\alpha + \beta + (\alpha + \beta)(\lfloor \frac{p}{2} \rfloor - 1) + 2\alpha = \frac{2n\alpha}{p} + (\alpha + \beta)\lfloor \frac{p}{2} \rfloor + \alpha$$

Im Gegensatz zur Situation ohne Einbeziehung der Kommunikationszeit ist $T_p(n)$ nun nicht mehr monoton nicht fallend. Die Erhöhung der Prozessorzahl erfordert gleichzeitig einen erhöhten Kommunikationszeitbedarf, dieser kann größer sein als die Einsparung bei der Berechnungszeit. Ein Optimum für die Prozessorzahl liegt bei $p \approx \sqrt{\frac{4\alpha n}{\alpha+\beta}}$, wobei ignoriert wird, daß p eine ganze Zahl sein muß. Der Zeitaufwand für die Vektorproduktberechnung liegt damit in $\Theta(\sqrt{n})$ im Gegensatz zu $\Theta(\log n)$ für einen optimalen Algorithmus ohne Berücksichtigung der Kommunikationszeit. Die asymptotische Analyse ist unabhängig von α und β. Die Größe der Konstanten beeinflußt aber klarerweise die reale Laufzeit. So ist es für $\beta > (n-1)\alpha$ nicht sinnvoll, den Algorithmus überhaupt zu parallelisieren, da eine parallele Implementierung mehr Zeit als eine sequentielle benötigen würde.

Matrix-Vektor-Produkte

Als nächstes untersuchen wir das Matrix-Vektor-Produkt $\mathbf{y} = \mathbf{A}\mathbf{x}$. Wobei $\mathbf{A}$ eine $n \times n$ Matrix ist. Sei p die Anzahl der verfügbaren Prozessoren und $k = n/p$ eine ganze Zahl. Es gibt zwei Möglichkeiten, die Matrix $\mathbf{A}$ zu speichern. Ein Prozessor kann Zeilen oder Spalten speichern. Dies beeinflußt den Ablauf des Algorithmus.

- Bei der *zeilenweisen Speicherung* speichert Prozessor i die Zeilen $(i-1)k+1$ bis ik der Matrix $\mathbf{A}$ und den gesamten Vektor $\mathbf{x}$. Aus diesen Daten berechnet der Prozessor die

Komponenten $y_{(i-1)k+1}, \ldots, y_{ik}$ des Lösungsvektors. Wenn wir davon ausgehen, daß der Lösungsvektor von allen Prozessoren benötigt wird, wie es z.B. in iterativen Verfahren der Fall ist, so müssen die k ermittelten Resultatwerte an alle anderen Prozessoren verteilt werden. Da jeder Prozessor diese Verteilung vornimmt, entspricht das Kommunikationsproblem einem allgemeinem Broadcast von k Werten.

- Bei der *spaltenweisen Speicherung* speichert Prozessor i die Spalten $(i-1)k+1$ bis ik der Matrix $\mathbf{A}$ und die Komponenten $(i-1)k+1$ bis ik des Vektors $\mathbf{x}$. Prozessor i berechnet dann $y_l^i = \sum_{j=(i-1)k+1}^{ik} A_{l,j} x_j$ für alle $l = 1, \ldots, n$. Da sich die l-te Komponente des Lösungsvektors $\mathbf{y}$ $(= \mathbf{Ax})$ aus der Summe $y_l = \sum_{i=1}^{p} y_l^i$ zusammensetzt, müssen die lokal berechneten Teilvektoren von allen Prozessoren an den Prozessor geschickt werden, der entsprechenden Teilvektor für die nächste Iteration benötigt und dort addiert werden. Man kann das zugehörige Kommunikationsproblem als allgemeine Akkumulation betrachten.

Die zeilenweise Speicherung wird meist bevorzugt, da die Implementierung etwas einfacher ist. Für dicht besetzte Matrizen haben zeilenweise und spaltenweise Speicherung ungefähr identische Laufzeiten. Beim Speicherplatz ist zu beachten, daß jedes Element der Matrix auf genau einem Prozessor gespeichert wird, während alle Vektor Elemente auf allen Prozessoren vorhanden sein müssen. Bei einer sequentiellen Implementierung werden insgesamt n^2+2n Real Werte gespeichert, während bei der parallelen Implementierung pro Prozessor $n((n+1)/p+1)$ Real Werte und insgesamt $n^2+n(1+p)$ Real Werte gespeichert. Es wird der folgende Algorithmus auf den einzelnen Prozessoren verwendet.

procedure for processor i
    for $(j = (i-1)k+1$ *to* $ik)$ *do*
        for $(l = 1$ *to* $n)$ *do*
            $y_j = y_j + A_{j,l} x_l$;
    sende Vektor $(y_{(i-1)k+1}, \ldots, y_{ik})$ *an alle anderen Prozessoren ;*
end ;

Analysieren wir nun die Laufzeit des Algorithmus. Sei α wieder die Zeit für eine Multiplikation oder Addition und $\beta + k\gamma$ die Zeit für die Übertragung einer Nachricht mit k Werten zu einem Nachbarknoten. In diesem Fall hängt die Anzahl der zu verteilenden Daten von der Anzahl der Elemente auf einem Prozessor ab. Die Übertragung aller Resultate beginnt nachdem der Prozessor alle Berechnungen durchgeführt hat. Der Zeitbedarf ist

$$T_p(n) = \frac{2\alpha n^2}{p} + (p-1)(\beta + \frac{n\gamma}{p}). \tag{4.1}$$

Der erste Summand beschreibt die Zeit, die zur Berechnung der Resultate benötigt wird, der zweite Term die Zeit für den anschließenden Broadcast. Auch hier können wir wieder unter Weglassen der Voraussetzung, daß p ganzzahlig ist, als Optimum $p \approx n(\sqrt{\frac{2\alpha-\gamma/n}{\beta}})$ berechnen. Für große n liefert dies $k = n/p \approx \sqrt{\beta/2\alpha}$. Interessant ist, daß der Zeitaufwand in $\Theta(n)$ liegt. Dies ist der Wert, der auch unter der Annahme zeitloser Kommunikation berechnet wurde und damit natürlich auf keiner Topologie unterboten werden kann.

Durch die Erhöhung der Zahl der verwendeten Prozessoren auf n^2 kann ohne Berücksichtigung der Kommunikationszeiten ein Algorithmus in $O(\log n)$ beschrieben werden. Ein solcher existiert allerdings nicht für lineare Arrays, wie man sich leicht überlegen kann, da schon das Vektor-Produkt, welches einfacher als das Matrix-Vektor-Produkt ist, $O(\sqrt{n})$ Zeitaufwand benötigt.

Matrix-Matrix-Multiplikation

Die Matrix-Matrix-Multiplikation kann als n-faches Matrix-Vektor-Produkt angesehen werden und damit in $\Theta(n^2)$ Schritten auf einem linearen Array mit n Prozessoren berechnet werden. In diesem Fall kann der Zeitaufwand dadurch verringert werden, daß Berechnung und Kommunikation überlagert werden, die ermittelten asymptotischen Werte bleiben aber gültig. Bei der Untersuchung des Speicherplatzes ist zu beachten, daß eine der beiden zu multiplizierenden Matrizen auf allen Prozessoren vorhanden sein muß. Damit speichert ein Prozessor eine Matrix (n^2), eine Teilmatrix (n^2/p) und n Teilvektoren (n/p), somit $n^2(2/p+1)$ Real Werte und insgesamt werden $n^2(2+p)$ Real Werte gespeichert. Im sequentiellen Fall müssen $3n^2$ Real Werte gespeichert werden.

4.2.2 Algorithmen für Gitterstrukturen

Wir betrachten nun Algorithmen für zweidimensionale $\sqrt{p} \times \sqrt{p}$ Gitter ($\sqrt{p}$ ist natürlich eine ganze Zahl). Da ein lineares Array mit p Prozessoren in ein solches Gitter eingebettet werden kann, können alle Algorithmen für das lineare Array auch auf dem Gitter realisiert werden. Zusätzlich gibt es aber zahlreiche effizientere Algorithmen zur Matrix-Matrix-Multiplikation auf Gittern. Einige diese Algorithmen sollen vorgestellt und analysiert werden. Für die Berechnung von Vektor-Matrix-Produkten können auf Gittern keine asymptotisch schnelleren Algorithmen realisiert werden. Es ist aber möglich eine speichereffizientere Version der Vektor-Matrix-Multiplikation zu realisieren.

Für die folgenden Algorithmen werden die Matrizen in Blöcke der Größe $n/\sqrt{p} \times n/\sqrt{p}$ zerlegt. Wir nehmen zur Vereinfachung an, daß $k = n/\sqrt{p}$ eine ganze Zahl ist. Damit liegt die folgende Blockstruktur vor

$$\begin{pmatrix} \mathbf{C}_{1,1} & \cdots & \mathbf{C}_{1,\sqrt{p}} \\ \vdots & \ddots & \vdots \\ \mathbf{C}_{\sqrt{p},1} & \cdots & \mathbf{C}_{\sqrt{p},\sqrt{p}} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,\sqrt{p}} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{\sqrt{p},1} & \cdots & \mathbf{A}_{\sqrt{p},\sqrt{p}} \end{pmatrix} \begin{pmatrix} \mathbf{B}_{1,1} & \cdots & \mathbf{B}_{1,\sqrt{p}} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{\sqrt{p},1} & \cdots & \mathbf{B}_{\sqrt{p},\sqrt{p}} \end{pmatrix}$$

wobei

$$\mathbf{C}_{i,j} = \sum_{l=1}^{\sqrt{p}} \mathbf{A}_{i,l}\mathbf{B}_{l,j} \ .$$

Diese Blockstruktur kann auf natürliche Weise einem Gitter zugeordnet werden. Prozessor i,j speichert initial die Untermatrizen $\mathbf{A}_{i,j}$ und $\mathbf{B}_{i,j}$.

Matrix-Vektor-Produkte

Wir gehen davon aus, daß die Matrix wie oben beschrieben auf den Prozessoren gespeichert ist. Der Vektor $\mathbf{x}$ wird in $\sqrt{p}$ Untervektoren $\mathbf{x}^i$ der Länge $k = n/\sqrt{p}$ zerlegt. Untervektor $\mathbf{x}^i$ umfaßt die Elemente $x_{(i-1)k+1}, \ldots, x_{ik}$. Auf den Prozessoren (i,j) $(i = 1, \ldots, \sqrt{p})$ wird jeweils $\mathbf{x}^j$ gespeichert. Im ersten Schritt berechnet jeder Prozessor lokal das Produkt $\mathbf{y}^{i,j} = \mathbf{A}_{i,j}\mathbf{x}^j$ und benötigt dazu

$$\frac{n^2}{p}$$

Multiplikationen und

$$\frac{n}{\sqrt{p}}(\frac{n}{\sqrt{p}} - 1)$$

Additionen. Es gilt $\mathbf{y}^i = \sum_{j=1}^{\sqrt{p}} \mathbf{y}^{i,j}$. Die Additionen werden so durchgeführt, daß Subvektor $\mathbf{y}^i$ in Prozessor i,i berechnet wird. Dazu werden die einzelnen Subvektoren nacheinander in Richtung des Zielprozessors weitergeleitet und zum lokalen Ergebnis addiert. Nach $\sqrt{p}$ Schritten liegt der Resultatvektor in den Prozessoren $(1,1),\ldots,(\sqrt{p},\sqrt{p})$ vor. Der Kommunikationsaufwand für diesen Schritt beläuft sich auf

$$(\sqrt{p}-1)(\beta+\frac{n}{\sqrt{p}}\gamma)\ .$$

Außerdem wurden $(\sqrt{p}-1)(n/\sqrt{p})$ Additionen durchgeführt. Im letzten Schritt werden die Teile der entsprechenden Zeile des Gitters innerhalb der Spalte des Gitters verteilt. Der Zeitaufwand entspricht dem vorherigen Aufwand zum Einsammeln der Teilergebnisse. Damit ergibt sich ein Gesamtaufwand von

$$T_p(n) = (\frac{2n^2}{p}+\frac{n(\sqrt{p}-2)}{\sqrt{p}})\alpha + 2(\sqrt{p}-1)(\beta+\frac{n}{\sqrt{p}}\gamma)$$

Der Algorithmus führt zusätzliche Rechenschritte aus, benötigt aber weniger Kommunikation. Welcher von beiden Algorithmen auf einem Gitter günstiger ist, der vorgestellte oder der Algorithmus für lineare Arrays, hängt von den konkreten Werten für α, β und γ ab. Betrachten wir nun den Speicherbedarf. Dieser liegt pro Prozessor bei $n^2/p + 2n/\sqrt{p}$ und insgesamt bei $n^2+2n\sqrt{p}$. Dies ist günstiger als beim Algorithmus für das lineare Array.

Ein einfacher Algorithmus zur Matrix-Matrix-Multiplikation

Wir nehmen an, daß die einzelnen Matrixblöcke wie beschrieben auf die Prozessoren verteilt sind und die Unterblöcke der Matrix $\mathbf{C}$ ebenfalls auf den jeweiligen Prozessoren verbleiben sollen. Der Algorithmus läuft in drei Schritten ab. Im ersten Schritt verteilt jeder Prozessor seine Submatrix $\mathbf{A}_{i,j}$ in der i-ten Zeile des Gitters. Dies entspricht einem allgemeinen Broadcast mit n^2/p Werten in einem linearen Array. In der zweiten Phase werden analog die Matrizen $\mathbf{B}_{i,j}$ in der j-ten Spalte verteilt. Auch dies entspricht einem allgemeinen Broadcast. Im dritten Schritt werden die Untermatrizen in den einzelnen Prozessoren berechnet. Unter den bereits vorgestellten Annahmen benötigt ein allgemeiner Broadcast einen Zeitaufwand von

$$(\sqrt{p}-1)(\beta+\frac{n^2}{p}\gamma)\ .$$

Die Berechnung einer Submatrix $\mathbf{C}_{i,j}$ erfordert

$$\frac{n^2(2n-1)}{p} \approx \frac{2n^3}{p}$$

Additionen und Multiplikationen. Wenn wir davon ausgehen, daß alle Schritte sequentiell ausgeführt werden, was nicht unbedingt realistisch ist aber die Analyse vereinfacht, so ergibt sich eine Laufzeit von

$$T_p(n) = \frac{2n^3}{p}\alpha + 2(\sqrt{p}-1)(\beta+\frac{n^2}{p}\gamma)$$

Die Laufzeit des Algorithmus liegt damit in $\Theta(n)$. Jeder Prozessor muß $2n^2/\sqrt{p}+n^2/p$ Real Werte speichern. Insgesamt werden damit $2n^2\sqrt{p}+n^2$ Werte gespeichert, während ein sequentieller Algorithmus nur $3n^2$ Werte speichert. Für größere n kann der Speicherplatz zum Problem werden.

Der Algorithmus kann dadurch verbessert werden, daß die Kommunikation und Berechnung überlagert werden. So können beide Broadcasts zumindest teilweise parallel ausgeführt werden. Des weiteren kann während der Kommunikationsphase schon mit der Berechnung begonnen werden. Dies spart Zeit und u. U. auch Speicherplatz, da Submatrizen, nachdem sie in die Berechnung eingeflossen sind, nicht mehr gebraucht werden.

Cannon-Algorithmus für Torus Netze

Ein oft eingesetzter paralleler Algorithmus zur Matrizenmultiplikation ist der Algorithmus von Cannon. Wie beim vorherigen Algorithmus werden die beiden Matrizen blockweise auf die Prozessoren verteilt. Der Algorithmus läuft auf einem Torus ab, d.h. der erste und letzte Prozessor in jeder Zeile/Spalte sind miteinander verbunden. Die folgende Prozedur beschreibt den Algorithmus.

procedure for processor i, j
    / Schritt 1 */*
    / überschreibe* $\mathbf{A}_{i,j}$ *durch* $\mathbf{A}_{i,(j+i)\ mod \sqrt{p}}$, *Left-Circular-Shift */*
    sende $\mathbf{A}_{i,j}$ *zu Prozessor* i, k *mit* $k = (j + \sqrt{p} - i)\ mod\ \sqrt{p}$;
    empfange Matrix $\mathbf{A}_{i,(j+i)\ mod \sqrt{p}}$ *und speichere sie als* $\mathbf{A}_{i,j}$;
    / überschreibe* $\mathbf{B}_{i,j}$ *durch* $\mathbf{B}_{(i+j)\ mod \sqrt{p},j}$, *Up-Circular-Shift */*
    sende $\mathbf{B}_{i,j}$ *zu Prozessor* k, j *mit* $k = (i - j)\ mod\ \sqrt{p}$;
    empfange Matrix $\mathbf{B}_{(i+j)\ mod \sqrt{p},j}$ *und speichere sie als* $\mathbf{B}_{i,j}$;
    berechne $\mathbf{C}_{i,j} = \mathbf{A}_{i,j} * \mathbf{B}_{i,j}$;
    / Schritt* $2 \ldots \sqrt{p}$ **/*
    for $step = 2$ *to* $\sqrt{p}$ *do*
        / Left-Circular-Shift */*
        sende $\mathbf{A}_{i,j}$ *zu Prozessor* i, k *mit* $k = (j - 1)\ mod\ \sqrt{p}$;
        empfange Matrix $\mathbf{A}_{i,(j+1)\ mod\ \sqrt{p}}$ *vom rechten Nachbarn und speichere sie als* $\mathbf{A}_{i,j}$;
        / Up-Circular-Shift */*
        sende $\mathbf{B}_{i,j}$ *zu Prozessor* k, j *mit* $k = (i - 1)\ mod\ \sqrt{p}$;
        empfange Matrix $\mathbf{B}_{(i+1)\ mod\ \sqrt{p},j}$ *vom unteren Nachbarn und speichere sie als* $\mathbf{B}_{i,j}$;
        berechne $\mathbf{C}_{i,j} = \mathbf{C}_{i,j} + \mathbf{A}_{i,j} * \mathbf{B}_{i,j}$;
    od ;
end ;

Betrachten wir den Ablauf des Algorithmus etwas genauer. Die folgenden Additionen und Subtraktionen werden alle modulo $\sqrt{p}$ ausgeführt. Die Untermatrizen werden also praktisch auf dem Ring herumgereicht, Matrizen $\mathbf{A}_{i,j}$ horizontal und nach links, Matrizen $\mathbf{B}_{i,j}$ vertikal und nach oben. Nach der initialen Verteilung der Matrizen speichert Prozessor i, j die Matrizen $\mathbf{A}_{i,(j+i) mod\ \sqrt{p}}$ und $\mathbf{B}_{(i+j) mod\ \sqrt{p},j}$ und berechnet das Produkt dieser beiden Matrizen. Im nächsten Schritt bekommt er die Matrizen $\mathbf{A}_{i,(j+i+1) mod\ \sqrt{p}}$ und $\mathbf{B}_{(i+j+1) mod\ \sqrt{p},j}$ und addiert deren Produkt zum ersten Produkt. Nach Abschluß des Algorithmus hat Prozesor i, j folgende Berechnung durchgeführt.

$$\mathbf{C}_{i,j} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{i,(j+i+k)\ mod \sqrt{p}} \mathbf{B}_{(i+j+k)\ mod \sqrt{p},j} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{i,k} \mathbf{B}_{k,j}$$

Der Ablauf des Algorithmus auf einen 3×3 Torus wird in Abb. 4.2 veranschaulicht. Der Aufwand gliedert sich in vier Teile. Wir gehen auch hier wieder davon aus, daß alle Teile sequentiell ablaufen, eine teilweise Überlagerung bringt natürlich Laufzeitvorteile. Im ersten Teil wird die initiale Verteilung der Matrizen betrachtet. Wenn das Verteilen der Matrizen in den Zeilen und den Spalten sequentiell abläuft, ist die Verteilung nach

$$2(\sqrt{p} - 1)(\beta + \frac{n^2}{p}\gamma)$$

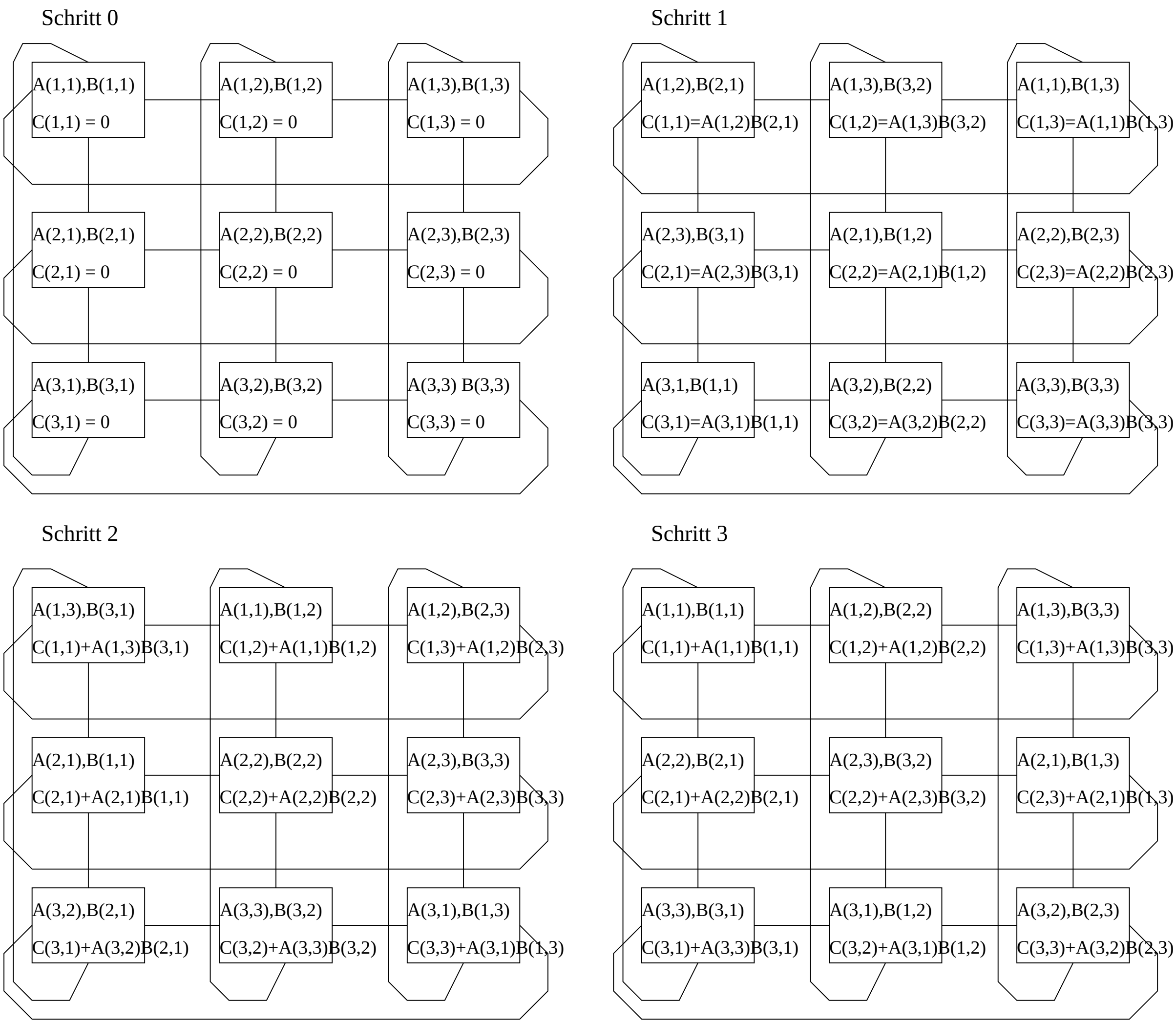


Abbildung 4.2: Ablauf des Algorithmus von Cannon auf einem 3×3 Torus.

Zeiteinheiten abgeschlossen. Der erste Berechnungsschritt erfordert

$$\frac{n^3}{p\sqrt{p}}$$

Multiplikationen und

$$\frac{n^2}{p}(\frac{n}{\sqrt{p}} - 1) \approx \frac{n^3}{p\sqrt{p}}$$

Additionen. In den folgenden Schritten werden die Untermatrizen $\mathbf{A}_{i,j}$ und $\mathbf{B}_{i,j}$ jeweils an benachbarte Knoten weitergegeben. Der erforderliche Aufwand ist

$$2(\beta + \frac{n^2}{p}\gamma) \ .$$

Ein Berechnungsschritt besteht aus

$$\frac{n^3}{p\sqrt{p}}$$

Multiplikationen und genauso vielen Additionen. Damit liegt der Gesamtaufwand bei

$$T_p(n) = \frac{2n^3}{p}\alpha + 4(\sqrt{p} - 1)(\beta + \frac{n^2}{p}\gamma)$$

Die Zahl der Operationen (Multiplikationen und Additionen) entspricht der Operationszahl im vorherigen Algorithmus, die Kommunikationszeit ist allerdings doppelt so hoch. Dies ist natürlich erst einmal ein Nachteil, auch wenn die asymptotische Komplexität beider Verfahren gleich ist. Zu beachten ist ferner, daß die Shift-Operationen in den Schritten 2 bis $\sqrt{p}$ lediglich eine Kommunikation zwischen direkt benachbarten Prozessoren erfordern. Der entscheidende Vorteil des Algorithmus von Cannon gegenüber dem vorher vorgestellten Verfahren liegt im Speicherplatzbedarf. Wenn wir davon ausgehen, daß ein Prozessor eine Untermatrix in der Kommunikationsphase zwischenspeichern muß, so sind maximal 4 Matrizen der Dimension $n/\sqrt{p}$ zu speichern. Damit muß ein Prozessor $4n^2/p$ Real Werte speichern. Insgesamt werden $4n^2$ Werte gespeichert, was eine deutliche Verbesserung gegenüber dem ersten Algorithmus darstellt.

Matrix-Matrix-Multiplikation auf höher dimensionalen Arrays

Wir betrachten kurz Matrix-Matrix-Multiplikationen auf höher dimensionalen, hier speziell drei-dimensionalen Gittern. Der zugehörige Algorithmus zeigt einen Weg, Matrizen auf Hyperwürfeln in $O(\log n)$ mit $O(n^3)$ Prozessoren zu multiplizieren.

Auf einem dreidimensionalem Array der Dimension $n^{3/4} \times n^{3/4} \times n^{3/4}$ können zwei $n \times n$ Matrizen $\mathbf{A}$ und $\mathbf{B}$ in $O(n^{3/4})$ multipliziert werden. Die Matrizen werden zuerst in $n^{1/4} \times n^{1/4}$ Untermatrizen der Ordnung $n^{3/4} \times n^{3/4}$ unterteilt. Matrix $\mathbf{A}$ wird in Untermatrizen $\mathbf{A}_{r,s}$ $(1 \leq r, s \leq n^{1/4})$ unterteilt. $\mathbf{A}_{r,s}$ beinhaltet die Zeilen $(r-1)n^{3/4} + 1$ bis $rn^{3/4}$ und die Spalten $(s-1)n^{3/4} + 1$ bis $sn^{3/4}$ der Matrix $\mathbf{A}$. Für $\mathbf{B}$ wird analog vorgegangen. Die Matrix $\mathbf{C} = \mathbf{AB}$ kann ebenfalls in Blöcke zerlegt werden und es gilt dann $\mathbf{C}_{r,s} = \sum_{t=1}^{n^{1/4}} \mathbf{A}_{r,t}\mathbf{B}_{t,s}$. Insgesamt müssen damit $n^{3/4}$ Produkte von Matrizen der Ordnung $n^{3/4} \times n^{3/4}$ gebildet werden. Da das verwendete dreidimensionale Array $n^{3/4}$ zweidimensionale Arrays der Größe $n^{3/4} \times n^{3/4}$ beinhaltet, steht für jedes Matrix-Produkt eine zweidimensionales Array zur Verfügung. Das Produkt $\mathbf{A}_{r,t}\mathbf{B}_{t,s}$ wird auf dem zweidimensionalen Array mit drittem Index $m = (r-1)n^{1/2} + (s-1)n^{1/4} + t$ gebildet. Prozessor (i, j, m) berechnet Element (i, j) des Produkts. Mit Hilfe des Matrix-Multiplikationsalgorithmus für zweidimensionale Arrays können die Matrix-Produkte in $O(n^{3/4})$ Schritten gebildet werden. Anschließend wird das i,j-te Element der Matrix $\mathbf{C}_{r,s}$ durch Summation der in den Prozessoren $(i,j,(r-1)n^{1/2}+(s-1)n^{1/4}+t)$ (für $t=1\dots n^{1/4}$) berechneten Zwischenergebnisse gebildet. Dies entspricht gerade einem allgemeinen Broadcast in einem linearen Array mit $n^{1/4}$ Prozessoren. Der Zeitaufwand liegt in $O(n^{1/4})$.

Das kurz beschriebene Vorgehen kann natürlich auf höher dimensionale Arrays erweitert werden. Auf einem k dimensionalen Array werden die Untermatrizen mittels des Algorithmus für ein $k-1$ dimensionales Array multipliziert. Auf diese Weise kann rekursiv ein Algorithmus für beliebig dimensionale Arrays definiert werden.

4.2.3 Algorithmen für Hyperwürfel

Die Einbettung von Arrays in Hyperwürfel wurde bereits gezeigt. Damit lassen sich alle für Arrays vorgestellten Algorithmen zumindest mit gleicher Laufzeit auf Hyperwürfeln realisieren. Es gibt aber zusätzlich noch spezifische Algorithmen für Hyperwürfel, die sehr effizient sind, da die reichhaltige Kommunikationsstruktur der Hyperwürfel ausgenutzt wird.

Vektor-Produkte

Zur Berechnung des Vektor-Produkts zweier n-dimensionaler Vektoren $\mathbf{x}$ und $\mathbf{y}$ auf einem Hyperwürfel mit p $(\leq n)$ Prozessoren werden die Vektorelemente auf die Prozessoren gleichmäßig verteilt. Wir nehmen zur Vereinfachung wieder $k = n/p \in \mathbb{N}$ an. Im ersten Schritt multipliziert und addiert jeder Prozessor lokal seine Vektorelemente. Für die folgenden Schritte wird ein Spannbaum im Hyperwürfel benötigt, wobei zur Vermischung aus Kommunikations- und Berechnungsschritten ein spezieller, unbalancierter Baum sich als günstig erweist. Wir nehmen dazu den Knoten mit Adresse $(0\dots0)$ als Wurzelknoten und konstruieren einen Spannbaum indem die einzelnen Adressbits konsekutiv (!) auf 1 gesetzt werden. D. h. die Prozessoren mit den Adressen $(10\dots0),\dots,(0\dots01)$ sind die Kinder des Wurzelknotens. Nachfolgende Ebenen betrachten dann nur noch Restadressen (Suffix) im Anschluß an die am weitesten rechts stehende 1 in der Adresse. Der resultierende Baum für einen 4-dimensionalen Hyperwürfel ist in Abb. 4.3 zu sehen. Der längste Weg im Baum hat eine Länge von $\log p$. Entlang dieses Spannbaums werden die Ergebnisse von den Blättern zur Wurzel hochgereicht und Zwischenergebnisse addiert. Es ist leicht nachzuvollziehen, daß es ausreicht, wenn auf einen Kommunikationsschritt ein Rechenschritt folgt, auch wenn Knoten mehrere Kinder haben. Die Tiefe der unterliegenden Teilbäume ist jeweils unterschiedlich, so daß Ergebnisse zu unterschiedlichen Zeitpunkten eintreffen.

Unter der Annahme, daß die Übertragung eines Wertes β Zeiteinheiten benötigt und eine Addition bzw. Multiplikation α Zeiteinheiten benötigt lautet der Gesamtaufwand.

$$T_p(n) = \frac{2n\alpha}{p} + (\alpha+\beta)\log p$$

Auch hier können wir wieder die optimale Zahl der Prozessoren approximieren und erhalten

$$p \approx \frac{2\alpha n}{\alpha+\beta}$$

Dies sind deutlich mehr Prozessoren als beim linearen Array. Außerdem haben wir ein Laufzeit der Ordnung $\log n$. Dies entspricht dem optimalen Wert ohne Berücksichtigung der Kommunikation.

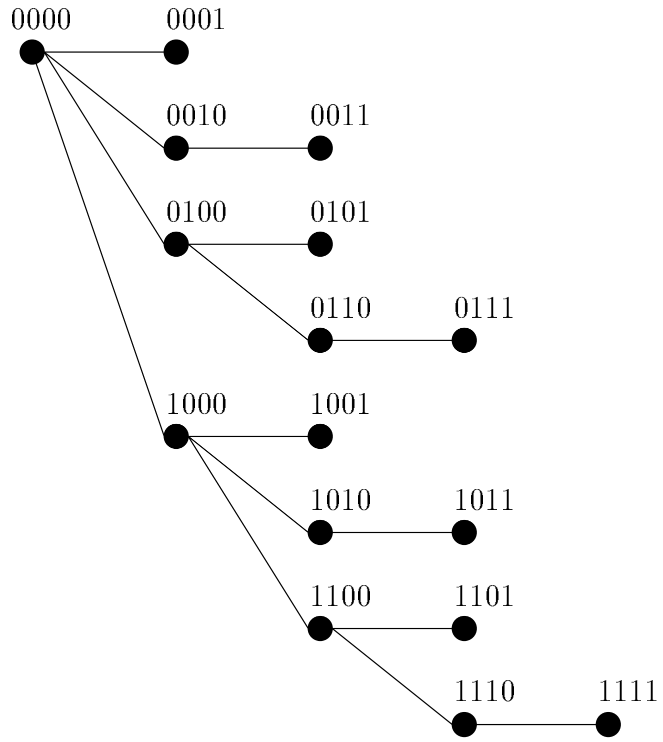


Abbildung 4.3: Spannbaum für einen 4-dimensionalen Hyperwürfel.

Vektor-Matrix-Produkte

Für ein lineares Array hatten wir einen optimalen Algorithmus zur Berechnung von $\mathbf{Ax}$ mit n Prozessoren. Eine Erhöhung der Prozessorzahl, welche theoretisch möglich ist, brachte auf dem linearen Array keine Verkürzung der Laufzeit. Wir stellen nun einen Algorithmus für Hyperwürfel vor, der mit $O(n^2)$ Prozessoren das Ergebnis in $O(\log n)$ Schritten berechnet und damit auch das theoretische Optimum erreicht.

Für den Algorithmus wird ein $\sqrt{p}\times\sqrt{p}$ Gitter in den Hyperwürfel eingebettet. Der Prozessor an Position (i,j) im Gitter speichert Submatrix $\mathbf{A}_{i,j}$ und Subvektor $\mathbf{x}_j$. Anschließend wird der Algorithmus zur Vektor-Matrix-Multiplikation auf zweidimensionalen Gittern mit einer kleinen Modifikation verwendet. Zuerst berechnet Prozessor (i,j) $\mathbf{y}^{i,j} = \mathbf{A}_{i,j}\mathbf{x}^j$ und benötigt dazu $2n^2/p - n/\sqrt{p}$ Additionen und Multiplikationen. Anschließend werden die Vektoren $\mathbf{y}^{i,j}$ zu Prozessor (i,i) übertragen und die Teilvektoren werden während der Übertragung entlang eines Spannbaums addiert. Dies entspricht einer Akkumulation in Zeile i. Da jede Zeile/Spalte des Arrays durch einen Unterwürfel des Hyperwürfels realisiert ist und jeder Unterwürfel selbst wieder ein Hyperwürfel ist, benötigt die Akkumulation $\log\sqrt{p}$ Schritte und jeder Schritt $\beta + k\gamma$ Zeiteinheiten zur Kommunikation und $n/\sqrt{p}\alpha$ Zeiteinheiten zur Berechnung. Nach Abschluß der Akkumulation beinhaltet Prozessor (i,i) den Vektor

$$\mathbf{y}^i = \sum_{j=1}^{\sqrt{p}} \mathbf{y}^{i,j} \ .$$

Im letzten Schritt werden die Ergebnisvektoren in den Spalten verteilt. Der Aufwand, dieses allgemeinen Broadcasts, besteht wieder aus $\log\sqrt{p}$ Schritten mit jeweils $\beta + k\gamma$ Zeiteinheiten, wobei $k = n/\sqrt{p}$. Bei einer vollständig sequentiellen Ausführung aller Schritte summiert sich der Gesamtaufwand zu

$$T_p(n) = (\frac{2n^2}{p} + \frac{n(\log\sqrt{p} - 1)}{\sqrt{p}})\alpha + 2\log\sqrt{p}(\beta + \frac{n}{\sqrt{p}}\gamma)$$

Wählen wir p aus $O(n^2)$, so liegt die asymptotische Komplexität in $O(\log n)$.

Matrix-Matrix-Multiplikationen

Wenn der Algorithmus von Cannon zur Matrix-Matrix-Multiplikation auf einem Hyperwürfel implementiert wird, so erfordert der erste Schritt nur eine parallele Matrixübertragung statt $\sqrt{p} - 1$ Übertragungen auf dem Array. Dies liegt daran, daß die jeweiligen Prozessoren im Hyperwürfel direkt verbunden sind und alle Übertragungen parallel über disjunkte Leitungen erfolgen können. Die restlichen Schritte des Algorithmus verändern sich nicht.

Wir betrachten nun noch einen weiteren Algorithmus zur Matrix-Multiplikation, der bis zu n^3 Prozessoren verwendet und damit eine optimale Laufzeit in $O(\log n)$ erreichen kann. Wir betten dazu ein dreidimensionales Array in einen Hyperwürfel ein. Prozessoren können dann mit dreidimensionalen Adressen angesprochen werden. Wir nehmen an, daß $p = 2^{3q}$ Prozessoren vorhanden sind und $k = n/2^q$ eine ganze Zahl ist. Die Matrizen $\mathbf{A}$ und $\mathbf{B}$ werden wieder in jeweils 2^{2q} Untermatrizen der Größe $k \times k$ unterteilt und auf die 2^{3q} Prozessoren verteilt. Auf Prozessor (i, j, k) sind die Matrizen $\mathbf{A}_{i,j}$ und $\mathbf{B}_{j,k}$ initial gespeichert. Nach Ausführung des Algorithmus befinden sich auf Prozessor (i, j, k) die Submatrizen $\mathbf{C}_{i,j}$ und $\mathbf{C}_{j,k}$ der Resultatmatrix.

Im ersten Schritt werden die Produkte $\mathbf{A}_{i,j}\mathbf{B}_{j,k}$ auf den jeweiligen Prozessoren berechnet. Dazu sind

$$\frac{2n^3}{2^{3q}} - \frac{n^2}{2^{2q}}$$

Additionen und Multiplikationen notwendig. Anschließend wird per Akkumulation über (i, j, k) $(j = 1, \ldots, q)$ die Matrix $\mathbf{C}_{i,k}$ auf Prozessor (i, i, k) berechnet. Der Kommunikationsbedarf für diesen Schritt ist

$$q(\beta + \frac{n^2}{2^{2q}}\gamma)$$

und die Rechenzeit lautet

$$q\frac{n^2}{2^{2q}}\alpha \ .$$

In der nächsten Phase werden die Submatrizen $\mathbf{C}_{i,k}$ von (i, i, k) per Broadcast an (j, i, k) $(j = 1, \ldots, q)$ verteilt und parallel dazu wird $\mathbf{C}_{i,k}$ von (i, i, k) zu Prozessor (i, k, k) gesendet. In der letzten Phase werden, ausgehend von den Prozessoren (i, j, j), die Matrizen $\mathbf{C}_{i,j}$ per Broadcast entlang (i, j, k) $(k = 1, \ldots, q)$ verteilt. Der Zeitbedarf der beiden letzten Schritte ist jeweils

$$q(\beta + \frac{n^2}{2^{2q}}\gamma)$$

Damit ergibt sich der folgende Approximation der Gesamtzeit.

$$T_p(n) \approx 3q(\beta + \frac{n^2}{2^{2q}}\gamma) + (\frac{2n^3}{2^{3q}} + (q-1)\frac{n^2}{2^{2q}})\alpha$$

Falls $p \in O(2^{3q}) = O(n^3)$, so liegen die Kommunikations- und Rechenzeit in $O(\log n)$. Der Speicherbedarf des Algorithmus ist relativ hoch. Jeder Prozessor muß 5 Matrizen der Dimension $n/2^q$ speichern. Damit werden insgesamt $5 * 2^q * n^2$ Real Werte gespeichert, im Vergleich zu $3n^2$ beim sequentiellen Algorithmus.

Problem	Zeitaufwand	Topologie
Vektor-Produkt	$O(\log n)$	Hyperwürfel mit n Prozessoren
Matrix-Vektor-Produkt	$O(n)$	Lineares Array mit n Prozessoren
Matrix-Vektor-Produkt	$O(\log n)$	Hyperwürfel mit n^2 Prozessoren
Matrix-Matrix-Produkt	$O(n)$	Zweidimensionales Array mit n^2 Prozessoren
k-te Potenz einer Matrix	$O(n \log k)$	Zweidimensionales Array mit n^2 Prozessoren
Matrix-Matrix-Produkt	$O(\log n)$	Hyperwürfel mit n^3 Prozessoren
k-te Potenz einer Matrix	$O(\log n \log k)$	Hyperwürfel mit n^3 Prozessoren

Tabelle 4.1: Zeitaufwand elementarer Algorithmen auf verschiedenen Topologien

4.2.4 Zusammenfassung

In den vorherigen Abschnitten wurden Algorithmen für die Basisoperationen auf verschiedenen Topologien vorgestellt. Es zeigte sich dabei, daß die vorher ohne Einbeziehung der Kommunikation bestimmten optimalen Laufzeiten auf entsprechenden Topologien asymptotisch erreichbar sind. In Tab. 4.1 werden die verschiedenen Algorithmen und die notwendigen Topologien noch einmal zusammenfassenden dargestellt. Nicht gesondert behandelt haben wir das Potenzieren einer Matrix. Diese Operation kann aber effizient durch mehrmaliges Quadrieren realisiert werden und basiert damit auf den vorgestellten Algorithmen zur Matrix-Matrix-Multiplikation.

4.3 Operationen für dünn besetzte Matrizen

Dünn besetzte Matrizen können natürlich mit den bereits vorgestellten Algorithmen für dicht besetzte Matrizen behandelt werden. Dies ist aber ein sehr ineffizienter Weg. Betrachten wir z. B. die Multiplikation einer dünn besetzten $n \times n$ Matrix $\mathbf{A}$ mit einem Vektor $\mathbf{x}$, so erfordert dies einen Aufwand der proportional zur Anzahl der Nichtnullelemente in $\mathbf{A}$ ist. Die Anzahl der Nichtnullelemente ist bei vielen realen Systemen in $O(n)$. Die bisher vorgestellten Algorithmen zur Vektor-Matrix-Multiplikation benötigen aber Aufwand $O(n^2)$. Ähnlich ist die Situation bei der Matrix-Matrix-Multiplikation. Der Aufwand hängt von der Zahl der Nichtnullelemente und deren Lage an und nicht von der Dimension der Matrizen.

Wir betrachten kurz Vektor Matrix und Matrix-Matrix-Multiplikation mit dünn besetzten Matrizen. Dazu beginnen wir mit Bandmatrizen, als Spezialfall dünn besetzter Matrizen und geben anschließend einen kurzen Ausblick auf unstrukturierte dünn besetzte Matrizen.

4.3.1 Bandmatrizen

Bei einer Bandmatrix befinden sich die Nichtnullelemente nur innerhalb eines Bandes um die Hauptdiagonale. Sei b_L die untere und b_U die obere Bandbreite, d.h. $\mathbf{A}_{i,j} \neq 0$ bedingt $i - b_L \leq j \leq i + b_U$. Bandmatrizen seien in einem zweidimensionalen Array gespeichert, wobei der erste Index von 1 bis n und der zweite Index von 1 bis $b_L + b_U + 1$ läuft. Arrayelement i, j enthält den Wert $\mathbf{A}_{i,j-i+b_L}$ falls $0 < j - i + b_U \leq n$ und 0 sonst. Damit werden $n(b_L + b_u + 1)$ Real Werte gespeichert.

Betrachten wir zuerst die Berechnung von Vektor-Matrix-Produkten auf linearen Arrays. Diese kann mit einer leicht modifizierten Form des bereits vorgestellten Algorithmus für dicht besetzte Matrizen durchgeführt werden. Der wesentliche Unterschied zum Algorithmus für dicht besetzte Matrizen ist, daß die Berechnungen auf einem Prozessor nur noch $2n(b_L + b_U + 1)/p$ statt n^2/p Operationen benötigen. Der Kommunikationsaufwand bleibt im wesentlichen gleich. In ähnlicher Form kann man die Multiplikation von Bandmatrizen auf einem linearen Array realisieren.

4.3.2 Unstrukturierte dünn besetzte Matrizen

Für dünn besetzte Matrizen mit unstrukturierter Besetzungsstruktur ist die Formulierung allgemeiner paralleler Algorithmen, welche nur Nichtnullelemente behandeln, schwieriger. Es gibt im wesentlichen zwei Ansätze solche Matrizen zu behandeln. Der eine Ansatz besteht darin, die Zeilen und Spalten der Matrix so zu permutieren, daß eine reguläre Besetzungsstruktur (z.B. eine Bandmatrix) entsteht. Die resultierende Matrix kann dann mit einem Algorithmus, der speziell auf die Besetzungsstruktur angepaßt ist, behandelt werden. Es gibt Algorithmen zur Minimierung der Bandbreite einer Matrix. Allerdings erweisen sich diese Algorithmen als sehr aufwendig, so daß sich ihr Einsatz nur in Ausnahmefällen lohnt. Alternativ kann versucht werden, Algorithmen für dicht besetzte Matrizen zu modifizieren um die Behandlung dünn besetzter Matrizen zu ermöglichen.

Ein allgemeinerer Ansatz besteht darin, die Zeilen einer dünn besetzten Matrix so auf die Prozessoren zu verteilen, daß eine effiziente parallele Realisierung der benötigten Operationen möglich ist. Wir wollen uns hier auf die Realisierung von Vektor-Matrix Produkten als Basisoperation beschränken, da Matrix-Matrix Produkte dünn besetzter Matrizen in der Praxis sehr selten vorkommen.

Bei dicht besetzten Matrizen ist die Anzahl der Zeilen, die ein Prozessor zugeteilt bekommt, proportional zur Rechenlast. Bei spärlich besetzten Matrizen hängen Speicherplatz und Rechenlast von der Anzahl der Nichtnullelemente in den zugeteilten Zeilen ab. Ein weiterer Unterschied zwischen dicht und spärlich besetzten Matrix ist die Kommunikationstruktur. Die parallele Realisierung von Vektor-Matrix Produkten mit dicht besetzten Matrizen erfordert Kommunikation zwischen allen Prozessoren. Dies gilt bei spärlich besetzten Matrizen nicht mehr. Ein Prozessor p benötigt nur dann Daten von einem Prozessor q, wenn Zeilen i und j existieren, so daß i auf q und j auf p gespeichert wird und $\mathbf{A}_{i,j} > 0$ ist. Existieren solche Zeilen nicht, so schickt q keine Daten an p. Durch eine geschickte Zuteilung der Zeilen zu Prozessoren kann somit der Kommunikationsaufwand reduziert werden. Bei der Verteilung der Matrixzeilen einer spärlich besetzten Matrix sind folgende Ziele zu beachten:

a) die Last sollte möglichst gleichmäßig verteilt werden (d.h. die Anzahl der gespeicherten Nichtnullelemente sollte proportional zur Prozessorgeschwindigkeit sein bzw. bei homogenen Prozessoren sollte jeder Prozessor ungefähr gleich viele Nichtnullelemente zugeteilt bekommen),

b) die Zahl der Nachrichten, die für die Durchführung der Iteration benötigt werden sollte minimiert werden,

c) die Menge der zu übertragenden Daten sollte minimiert werden,

d) die Topologie des Kommunikationsnetzes sollte ausgenutzt werden, indem möglichst benachbarte Prozessoren kommunizieren.

Die beschriebenen Ziele sind zum Teil widersprüchlich, so wird die Nachrichtenzahl und Datenmenge oft minimiert, wenn die Last sehr ungleichmäßig verteilt ist.

Insgesamt kann man das Problem als Graphproblem auffassen. Wir betrachten dazu einen bewerteten Graph. Knotengewichte beschreiben den Berechnungsaufwand der zugehörigen Funktion, wobei wir uns hier auf Vektor-Matrix Produkte beschränken. Die Funktion ist damit gegeben durch eine Matrixzeile oder eine Menge von Matrixzeilen und der Rechenaufwand ist damit proportional zur Zahl der Nichtnullelemente. Kantengewichte beschreiben den Kommunikationsaufwand. Kantengewichte beschreiben den Kommunikationsaufwand. Wenn wir vom einfachsten Fall ausgehen, daß Knoten einzelne Matrixzeilen sind, so existiert eine gerichtete Kante von Knoten i nach j, falls $\mathbf{A}_{i,j} > 0$. Das Kantengewicht ist in diesem Fall 1, da genau ein Wert übertragen werden muß. Falls die Partitionierung auf Submatrixebene stattfindet, kommen allgemeinere Gewichte zum Einsatz.

Ein Mapping ist eine Abbildung des Kommunikationsgraphen auf die Rechnertopologie, wobei die oben genannten Ziele zu beachten sind. Formal handelt es sich um ein Optimierungsproblem, wobei die Zielfunktion durch die Minimierung der Kommunikation und die Nebenbedingungen durch die gleichmäßige Lastverteilung definiert werden. Leider ist das Optimierungsproblem NP-hart und damit für größere und damit realistische Probleme nicht lösbar. Es gibt eine Reihe von Heuristiken, die eine approximativen Optimierung ermöglichen. Diese Heuristiken reichen von relativ komplexen Algorithmen auf Basis der Eigenwerte der Adjazenzmatrix des Kommunikationsgraphen bis hin zu sehr einfachen und schnellen Heuristiken. Wir werden uns hier kurz mit einfachen Algorithmen zur Partitionierung beschäftigen. Diese Algorithmen liefern zwar meist keine optimalen Ergebnisse, die gefundene Lösung ist aber in vielen Fällen ausreichend. Dies gilt insbesondere, wenn der parallele Algorithmus später auf einem Workstation Netz laufen soll. In diesem Fall ist die Kommunikationszeit und -bandbreite starken Schwankungen unterworfen, so daß eine nicht ganz optimale Lastverteilung tolerabel ist. Weiterhin sind die einfachen Algorithmen sehr effizient und damit auch für große Systeme anwendbar. Zur Vereinfachung betrachten wir im folgenden die Rechnertopologie nicht. Ziel ist es damit, den Kommunikationsgraphen so zu partitionieren, daß möglichst wenige Kanten zwischen Knoten auf unterschiedlichen Prozessoren verlaufen.

Das Vorgehen der "einfachen'' Algorithmen basiert im wesentlichen darauf, daß zuerst eine Partition nach einer Heuristik bestimmt wird, die dann anschließend durch eine lokale Greedy Strategie verbessert wird. Betrachten wir zuerst die Berechnung einer Initialpartition. Die naheliegenste Möglichkeit besteht darin, auf Basis eines Breadth-First-Search Algorithmus zu partitionieren. Ausgehend von einem (oder auch mehreren) initialen Knoten werden alle Knoten des Kommunikationsgraphen durchlaufen und erhalten eine Nummer, die dem jeweiligen Schritt entspricht, in dem der Knoten erreicht wurde. Der Algorithmus in Abb. 4.4 numeriert die Knoten entsprechend.

Nach Ablauf des Algorithmus speichert *level[k]* den Wert für Knoten k. Eine Partition kann nun gebildet werden, indem beginnend mit Ebene 1 solange Knoten einer Ebene zugewiesen werden, bis genügend Knoten für eine Partitionsgruppe zusammengekommen sind. Danach wird mit der nächsten Partitiongsgruppe begonnen. Die Einteilung erfolgt mit dem Algorithmus in Abb. 4.4.

Der Algorithmus liefert im allgemeinen keine guten Ergebnisse, wenn er in einem Schritt zur Partitionierung eines Graphen verwendet wird. Besser ist ein schrittweises Vorgehen. Der Kommunikationsgraph wird im ersten Schritt in 2 oder 4 Teilgraphen zerlegt. Anschließend wird für diese Zerlegung ein lokaler Optimierungsschritt durchgeführt, wie unten beschrieben. Dann wird der Algorithmus auf die Teilgraphen angewendet, bis genügend Teilgraphen erzeugt wurden.

Eine ähnlich einfache Vorgehensweise, um eine initiale Partitionierung zu berechnen besteht darin, den Graphen ausgehend von einigen Knoten zu expandieren. Dazu werden einige möglichst weit auseinander liegende Knoten verwendet und von diesen aus ein Breadth First Durchlauf initiiert. Knoten werden jeweils dem Initialknoten zugeordnet von dem aus sie zuerst erreicht wurden. Diese Vorgehen läßt sich auch sehr gut parallelisieren.

Betrachten wir nun noch die lokale Optimierung einer einmal gefundenen Partition. Wobei wir uns hier darauf beschränken, die grundsätzliche Idee zu erläutern, die auf eine Arbeit

```
Procedure Breadth_First_Search ;
    Eingabe Initialknoten i, Ausgabe Ebenennummer für jeden Knoten
    levset = {i} ;
    next = 1 ;
    marker[i] = 1 ;
    nlev = 1 ;
    while next < n do
        next_levset = ∅ ;
        for each j in levset do
            for each k with A_{j,k} > 0 and marker[k] = 0 do
                next_levset = next_levset ∪ k ;
                level[k] = nlev ;
                marker[k] = 1 ;
                next = next + 1 ;
            od
        od
        levset = next_levset ;
        nlev = nlev + 1 ;
    od ;
```

Abbildung 4.4: Breadth-First-Search Algorithmus

```
Procedure Einwege_Partitionierung ;
    Eingabe nlev Anzahl Ebenen, level Ebenen für die einzelnen Knoten, ip Knoten pro Prozessor
    Ausgabe proc Zuordnung Knoten zu Prozessoren
    p = 1 ;
    size = 0 ;
    for lev = 1 to nlev do
        for each j with level[j] = lev do
            proc[j] = p ;
            size = size + 1 ;
            if size ≥ ip then
                p = p + 1 ;
                size = 0 ;
            fi
        od
    od ;
```

Abbildung 4.5: Einwege_Partitionierung

von Kerninghan und Lin von 1970 zurückgeht, die in zahlreichen Nachfolgearbeiten modifiziert und verbessert wurde. Die Basis aller dieser Algorithmen ist die Definition einer Bewertungsfunktion, die die Änderung der Partitionsbewertung beim Verschieben von Kanten zwischen Partitiongruppen angibt. Zur Einführung der Gewinnfunktion benötigen wir einige zusätzliche Notationen. Sei w_{ij} das Gewicht der Kante von i nach j, wenn i und j zu verschiedenen Partitionsgruppen gehören. Für einen Knoten i mit $proc[i] = l$ sei $g^k(i)$ die Kostenfunktion, wenn i in Partitionsgruppe k verschoben wird und E ist die Menge der Kanten im Graphen (d.h., $(i,j) \in E \Leftrightarrow A_{i,j} > 0$).

$$g^k(i) = \sum_{(i,j)\in E} \begin{cases} w_{ij} & \text{if } proc[j] = k, \\ -w_{ij} & \text{if } proc[j] = l, \\ 0 & \text{sonst.} \end{cases}$$

Die Gewinnfunktion kann auf verschiedene Arten verfeinert werden (z.B. indem die Zuordnung zu Prozessoren berücksichtigt wird). Ein einfaches Vorgehen besteht nun darin Knoten i zu verschieben, wenn ein $g^k(i) > 0$ existiert. Damit werden offensichtlich die Kosten kreuzender Kanten reduziert. Falls mehrere positive $g^k(i)$ existieren, so nimmt man das k, welches den größten Wert liefert. Dieses Vorgehen beachtet allerdings nicht die Vorgabe einer möglichst gleichmäßigen Lastverteilung. Unter dieser Bedingung kann i nur von l nach k verlegt werden, wenn dadurch l nicht zu wenige und k nicht zu viele Elemente beinhaltet. Es müssen deshalb untere und obere Schranken für die Zahl der Elemente in einer Partitionsgruppe definiert werden. Die lokale Greedy Strategie kann etwas verbessert werden, indem man den Algorithmus auch weiterlaufen läßt, wenn nur negative Gewichte vorhanden sind, wobei die bisherige "beste'' Partitionierung gespeichert wird und die Änderungen mit dem kleinsten negativen Wert durchgeführt wird. Es gibt einige weitere Optimierungsschritte, die wir hier aber nicht im Detail behandeln werden. Insgesamt führt die Verwendung der einfachen lokalen Optimierung in vielen Fällen zu einer deutlichen Verbesserung der gefundenen Partition. Insbesondere kann der Algorithmus zur Verbesserung beliebiger Partitionen verwendet werden.

4.4 Literaturhinweise

Datenstrukturen für Vektoren und Matrizen werden in den meisten Büchern über wissenschaftliches Rechnen behandelt. Ein ausführliche Darstellung verschiedener dünn besetzter Matrixstrukturen kann man in [36] finden. Einige Resultate und das Modell zur Algorithmenanalyse sind [4] zu finden. Matrix - und Vektor Operationen sind in vielen Büchern über parallele Algorithmen zu finden. Als Beispiele seien hier [4, 8, 13, 14, 26] genannt. In [20] werden verschiedene Verfahren zur Matrix-Matrix-Multiplikation verglichen. Varianten des Strassen-Algorithmus und dessen Implementierung werden in [2, 14, 36] beschrieben. Die Parallelisierung dieses Algorithmus wird in [26] kurz angesprochen. Es werden in allen Quellen dicht besetzte Matrizen verwendet. Einige Resultate für Bandmatrizen sind in [13] zu finden. Arbeiten über Graphpartitionierungen unter der Web Adresse $http://www.cs.sandia.gov/CRF/papers\_chaco.html$ zu finden.

Kapitel 5

Algorithmen zur Lösung linearer Gleichungssysteme

In diesem Kapitel werden wir uns mit den nach den einfachen Matrix Vektor Operationen nächst komplexeren Problemen, der Lösung linearer Gleichungssysteme, beschäftigen. Allgemein kann man das Problem wie folgt auffassen, sei $\mathbf{A}$ eine $n \times n$ Matrix und $\mathbf{b}$ ein Vektor in $\mathbb{R}^n$, es wird ein Vektor $\mathbf{x} \in \mathbb{R}^n$ gesucht, der die linearen Gleichungen $\mathbf{Ax} = \mathbf{b}$ löst. Wir nehmen im folgenden an, daß die Lösung $\mathbf{x}$ eindeutig existiert.

Obwohl in der Realität die meisten Abhängigkeiten nichtlinear sind, ist die Analyse linearer Systeme in den Ingenieur- und Naturwissenschaften weit verbreitet. Dies liegt darin begründet, daß lineare Modelle in den meisten Fällen beherrschbar sind, während die Lösung nichtlinearer Probleme fast immer auf die Lösung linearer Gleichungssysteme zurückgeführt wird. Insofern sind Algorithmen zur effizienten Lösung linearer Gleichungssysteme von fundamentaler Bedeutung und große Teile der Rechenzeit von Höchstleistungsrechnern werden heute mit solchen Algorithmen verbraucht.

Es gibt eine Reihe von Methoden zur Lösung linearer Gleichungssysteme. Man unterscheidet diese in direkte und iterative Verfahren. Direkte Methoden finden die, bis auf numerische Ungenauigkeiten exakte Lösung in typischerweise $O(n^3)$ Schritten. Iterative Methoden konvergieren asymptotisch gegen die Lösung, finden also im allgemeinen keine exakte Lösung in endlicher Zeit, oftmals aber eine genügend genaue Lösung nach relativ kurzer Zeit. Iterative Verfahren haben meist weniger numerische Probleme als direkte Verfahren und sind insbesondere vorzuziehen, wenn die Matrix sehr groß und dünn besetzt ist, wie es bei vielen realen Anwendungen üblich ist. Typischerweise sind weniger als 1 oder sogar 0.1 Prozent der n^2 Matrixelemente ungleich 0. In einem solchen Fall ist es natürlich sinnvoll, eine Speicherstruktur für dünn besetzte Matrizen zu verwenden. Eine solche Speicherstruktur kann nur von iterativen Verfahren ausgenutzt werden, da diese die Matrix während der Gleichungslösung nicht verändern. Direkte Verfahren modifizieren die Matrix dagegen. Dies bedingt, daß an Stellen, wo eine 0 stand Nichtnullelemente auftauchen. Es entsteht "Fill In", der zusätzlichen Aufwand und Speicherplatz erfordert. Je nach Matrixstruktur, kann aus einer spärlich besetzten Matrix, mit weniger als 0.1 Prozent Nichtnullelementen, während der direkten Lösung eine dicht besetzte Matrix werden, in der fast alle Elemente ungleich 0 sind.

Neben der Lösung linearer Gleichungssysteme ist die Berechnung der inversen Matrix $\mathbf{A}^{-1}$ ein verwandtes Problem, welches auf die Lösung linearer Gleichungssysteme zurückgeführt werden kann. Z. B. indem

$$\mathbf{AA}^{-1} = \mathbf{I}$$

spaltenweise für $\mathbf{A}^{-1}$ und $\mathbf{I}$ gelöst wird.

Dieses Kapitel behandelt parallele Algorithmen zur Lösung linearer Gleichungssysteme. Einige Verfahren sind parallele Implementierungen bekannter sequentieller Algorithmen, andere sind neueren Datums und wurden speziell für eine parallele Abarbeitung entwickelt. Das Kapitel gliedert sich in 3 Abschnitte. Zuerst wird die Lösung von Gleichungssystemen mit direkten Verfahren behandelt. Der folgende Abschnitt widmet sich iterativen Verfahren. Im letzten Abschnitt wird kurz die Analyse partieller Differentialgleichungen beschrieben, die in der Praxis eine große Bedeutung hat und sich auf die Lösung speziell strukturierter linearer Gleichungssysteme zurückführen läßt.

5.1 Direkte Verfahren zur Lösung linearer Gleichungssysteme

Wir beginnen in diesem Abschnitt mit der direkten Lösung von Gleichungssystemen, die eine spezielle Struktur aufweisen. Daran anschließend betrachten wir die direkte Lösung allgemeiner Gleichungssysteme mit dicht besetzten Matrixstrukturen und behandeln abschließend direkte Verfahren für dünn besetzte Matrixstrukturen.

5.1.1 Parallele Algorithmen für Matrizen mit spezieller Struktur

Falls die Nichtnullelemente der Matrix $\mathbf{A}$ eine spezielle Struktur aufweisen, so kann das Gleichungssystem sehr effizient und einfach gelöst werden. Eine einfache Struktur haben Dreiecksmatrizen und tridiagonale Matrizen, für die in den folgenden Abschnitten spezielle Verfahren eingeführt werden.

Dreiecksmatrizen

$\mathbf{A}$ ist eine obere Dreiecksmatrix, falls $A_{i,j} = 0$ für $j < i$. $\mathbf{A}$ ist eine untere Dreiecksmatrix, falls $A_{i,j} = 0$ für $j > i$. Wir werden uns bei den folgenden Algorithmen auf untere Dreiecksmatrizen konzentrieren, eine Übertragung auf obere Dreiecksmatrizen ist aber für alle vorgestellten Algorithmen problemlos möglich.

Die Determinante einer Dreiecksmatrix ist gleich dem Produkt der Diagonalelemente. Damit ist eine hinreichende und notwendige Bedingung für eine Determinante ungleich 0, daß alle Diagonalelemente ungleich 0 sind. Dies soll im folgenden vorausgesetzt sein. Sei nun $\mathbf{A}$ eine $n \times n$ untere Dreiecksmatrix mit $A_{i,i} \neq 0$ für alle $i = 1, \ldots, n$. Wir wollen als erstes die Lösung des Gleichungssystems $\mathbf{Ax} = \mathbf{b}$ betrachten und anschließend die Invertierung von Dreiecksmatrizen untersuchen. Eine übliche sequentielle Lösungsmethode ist die sogenannte Rücksubstitution, deren parallele Variante wir nun vorstellen.

Beginnen wir mit der sequentiellen Variante der Rücksubstitution. Es gilt $x_1 = b_1/A_{1,1}$, $x_2 = (b_2 - A_{2,1}x_1)/A_{2,2}$ usw. So kann durch konsekutive Berechnung der x_i das Gesamtergebnis ermittelt werden. Für die parallele Version des Algorithmus ist es sinnvoll, x_i wir folgt darzustellen.

$$x_i = \frac{1}{A_{i,i}}(b_i - A_{i,1}x_1 - \ldots - A_{i,i-1}x_{i-1}) = \frac{1}{A_{i,i}}(b_i - \sum_{j=1}^{i-1} A_{i,j}x_j)$$

Damit kann ein paralleler Algorithmus für ein lineares Array formuliert werden. Das Vorgehen ist ähnlich zur Vektor-Matrix-Multiplikation auf linearen Arrays. Wir nehmen an, daß ein Array mit p Prozessoren zur Berechnung verwendet wird und $k = n/p$ eine ganze Zahl ist. Prozessor i berechnet dann die Element $x_{(i-1)k+1}, \ldots, x_{ik}$ und speichert initial die korrespondierenden

Elemente von $\mathbf{b}$ und die korrespondierenden Zeilen von $\mathbf{A}$. Die Prozessoren beginnen mit der Berechnung sobald die entsprechenden Komponenten von $\mathbf{x}$ zur Verfügung stehen. Prozessor 1 berechnet $\mathbf{x}^1 = (x_1, \ldots, x_k)$ und schiebt die Werte nach rechts zu Prozessor 2. Prozessor 2 gibt die Werte von $\mathbf{x}^1$ direkt weiter an Prozessor 3 und berechnet simultan $\mathbf{x}^2$ usw. Nach dem Empfang von x^j in Prozessor $i > j$ berechnet dieser

$$x_l = b_l - \sum_{m=1}^{(j-1)k} A_{l,m} x_m - \sum_{m=(j-1)k+1}^{jk} A_{l,m} x_m$$

für $l = (i-1)k+1, \ldots, ik$. Die erste Summe wurde bereits berechnet, nur die zweite erfordert die Kenntnis von $\mathbf{x}^j$. Nach Empfang von $\mathbf{x}^{i-1}$ kann Prozessor i endgültigen Werte für x_l berechnen.

$$x_l = \frac{1}{A_{l,l}}(b_l - \sum_{m=1}^{(i-1)k} A_{l,m} x_m - \sum_{m=(i-1)k+1}^{l-1} A_{l,m} x_m)$$

Auch hier muß nur die letzte Summe und die Division berechnet werden, die erste Summe liegt bereits vor. Betrachten wir kurz den Aufwand der einzelnen Schritte. Die Übertragung eines Teilvektors an einen Nachbarprozessor benötigt $\beta + k\gamma$ Zeiteinheiten, die Berechnung der Summe in Prozessor i bei Empfang von Teilvektor $\mathbf{x}^j$ $(j < i)$ benötigt $2k^2\alpha$ Zeiteinheiten und die endgültige Berechnung von $\mathbf{x}^i$ auf Prozessor i benötigt $(2k^2 + k)\alpha$ Zeiteinheiten. Da die Berechnung und Kommunikation im Algorithmus überlagert werden, hängt die Gesamtlaufzeit vom Verhältnis Berechnungszeit zu Kommunikationszeit ab. Die asymptotische Analyse zeigt, daß der Gesamtaufwand der Berechnungen auf dem letzten Prozessor in $O(kn)$ liegt und die Kommunikationszeit in $O(p)$ liegt. Damit läßt sich eine Gesamtlaufzeit in in $O(n)$ erreichen.

Der Algorithmus kann natürlich auch auf einem Hyperwürfel implementiert werden, wobei die Laufzeit von gleicher Ordnung ist (ein Hyperwürfel bettet ein lineares Array ein). Falls die Kommunikationszeiten länger als die Berechnungszeiten sind, bietet der Hyperwürfel durch seine größere Verbindungszahl die Möglichkeit einer besseren Überlagerung von Berechnung und Kommunikation, wodurch sich die Konstanten in der Laufzeitabschätzung erniedrigen, nicht aber die Ordnung der Laufzeit.

Vergleichen wir nun die Laufzeit mit der sequentiellen Rücksubstitution. Diese benötigt offensichtlich einen Aufwand in $\Theta(n^2)$, da die Zahl der Elemente von $\mathbf{A}$ in $O(n^2)$ liegt. Damit liegt der Speedup für n Prozessoren in $\Theta(n)$ und die Effizienz in $\Theta(1)$, beides sind asymptotisch optimale Werte.

Das nächste Problem ist die Berechnung der inversen Matrix $\mathbf{A}^{-1}$. Auch dazu kann die Rücksubstitution verwendet werden. Da $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ gilt kann die i-te Spalte von $\mathbf{A}^{-1}$ als Lösung des Gleichungssystems $\mathbf{A}\mathbf{x}^i = \mathbf{e}_i$ (wobei $\mathbf{e}_i$ ein Vektor mit 1 an Position i und 0 an allen anderen Positionen ist) berechnet werden. Dazu ist offensichtlich der obige Algorithmus einsetzbar. Auf einem linearen Array können wir damit in $O(n^2)$ Schritten die Inverse mit $O(n)$ Prozessoren berechnen. Durch weitere Parallelisierung können die einzelnen Spalten von $\mathbf{A}^{-1}$ parallel berechnet werden. Damit ist die inverse Matrix auf n linearen Arrays mit n Prozessoren (oder einem $n \times n$ Array) in $O(n)$ Schritten berechenbar. Beide Algorithmen sind bzgl. Speedup und Effizienz wieder optimal.

Es gibt noch andere Ansätze zur Invertierung von unteren Dreiecksmatrizen, die weniger Rechenschritte aber mehr Prozessoren benötigen. Betrachten wir zuerst den Fall, daß $A_{i,i} = 1$ für alle i gilt. Wir stellen nun $\mathbf{A}$ als $(\mathbf{I} - \mathbf{L})$ dar. $\mathbf{L}$ ist offensichtlich auch eine untere Dreiecksmatrix, deren Diagonalelemente 0 sind. Es ist einfach nachzuweisen, daß das i,j-te Element von $\mathbf{L}^k$ gleich 0 ist, falls $i - j < k$. Damit ist $\mathbf{L}^n = 0$. Für die Inverse von $\mathbf{A}$ gilt dann der folgende Satz.

Satz 5.1 *Wenn* $\mathbf{A} = \mathbf{I} - \mathbf{L}$, *wobei* $\mathbf{L}$ *eine untere Dreiecksmatrix ist, in der alle Diagonalelemente gleich 0 sind, so gilt* $\mathbf{A}^{-1} = \mathbf{I} + \mathbf{L} + \mathbf{L}^2 + \ldots + \mathbf{L}^{n-1}$.

Bew. durch Einsetzen in $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ und ausmultiplizieren. Sei $\mathbf{B}$ die rechte Seite der obigen Gleichung. Es gilt dann $\mathbf{B}(\mathbf{I} - \mathbf{L}) = \mathbf{I} - \mathbf{L}^n = \mathbf{I}$, da $\mathbf{L}^n = 0$. Damit muß $\mathbf{B} = \mathbf{A}^{-1}$ gelten. □

Der obige Satz liefert direkt einen parallelen Algorithmus zur Berechnung von $\mathbf{A}^{-1}$. Berechne parallel $\mathbf{L}^2, \ldots, \mathbf{L}^{n-1}$ und addiere die Resultate. Mit den bereits eingeführten Algorithmen kann dies in $O(\log^2 n)$ Schritten mit n^4 Prozessoren durchgeführt werden. Der resultierende Algorithmus ist zwar einfach, benutzt aber eine extreme Anzahl von Prozessoren. Etwas effizienter wird es, wenn wir den folgende Satz verwenden.

Satz 5.2 *Wenn* $\mathbf{A} = \mathbf{I} - \mathbf{L}$, *wobei* $\mathbf{L}$ *eine untere Dreiecksmatrix ist in der alle Diagonalelement gleich 0 sind, so gilt* $\mathbf{A}^{-1} = (\mathbf{I} + \mathbf{L}^{2^{\lceil \log n \rceil - 1}})(\mathbf{I} + \mathbf{L}^{2^{\lceil \log n \rceil - 2}}) \ldots (\mathbf{I} + \mathbf{L}^4)(\mathbf{I} + \mathbf{L}^2)(\mathbf{I} + \mathbf{L})$.

Bew. Wenn das obige Produkt ausmultipliziert wird, liegt die Folge $\mathbf{I} + \sum_{i=1}^{k} \mathbf{L}^{\mathbf{i}}$ mit $k = \sum_{j=0}^{\lceil \log n \rceil - 1} 2^j$ vor. Unter Ausnutzung von $\mathbf{L}^l = 0$ für $l \geq n$ erkennt man, daß diese Folge der Folge aus dem vorherigen Satz entspricht, die gleich $\mathbf{A}^{-1}$ ist. □

Durch wiederholtes Quadrieren wird $\mathbf{L}^2, \mathbf{L}^4, \ldots, \mathbf{L}^{2^{\lceil \log n \rceil - 1}}$ berechnet. Zu jeder Matrix wird die Identitätsmatrix addiert und die obigen Matrix-Produkte werden bestimmt. Die Addition mit der Identitätsmatrizen kann in einem Schritt durchgeführt werden, die anderen beiden Schritte sind durch $\log n$ Matrix-Multiplikationen zu realisieren. Mit n^3 Prozessoren wird eine Matrix-Matrix-Multiplikation in $O(\log n)$ Schritte ausgeführt. Damit benötigt der gesamte Algorithmus $O(\log^2 n)$ Schritte bei Verwendung von n^3 Prozessoren.

Wir haben bisher nur den Fall untersucht, daß die Diagonalelemente von $\mathbf{A}$ gleich 1 sind, dies soll nun verallgemeinert werden. Sei $\mathbf{D}$ eine Matrix, die in der Hauptdiagonalen die Elemente $A_{i,i}$ beinhaltet, alle anderen Elemente sind 0. Die Matrix $\mathbf{D}^{-1}\mathbf{A}$ hat offensichtlich Diagonalelemente gleich 1. Mit obigem Algorithmus kann die Inverse $(\mathbf{D}^{-1}\mathbf{A})^{-1} = \mathbf{A}^{-1}\mathbf{D}$ gebildet werden[1], durch Multiplikation von rechts mit $\mathbf{D}^{-1}$ entsteht $\mathbf{A}^{-1}$. Die Bildung von $\mathbf{D}^{-1}$ ist trivial, da nur der Kehrwert der Diagonalelemente gebildet werden muß. Die Bildung von $\mathbf{D}^{-1}$ und die zweimalige Multiplikation mit $\mathbf{D}$ bzw. $\mathbf{D}^{-1}$ sind mit n^3 Prozessoren offensichtlich in einem Schritt durchführbar und fallen bei der Laufzeit des Algorithmus nicht ins Gewicht.

Eine andere ähnliche Methode, die nach dem Prinzip des "divide and conquer" vorgeht, liefert ähnliche Leistungsdaten. Die Matrix $\mathbf{A}$ wird in 4 Blöcke zerlegt, $\mathbf{A}_1$ sei eine $\lceil n/2 \rceil \times \lceil n/2 \rceil$ Matrix.

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_1 & \mathbf{0} \\ \mathbf{A}_2 & \mathbf{A}_3 \end{pmatrix}$$

$\mathbf{A}_1$ und $\mathbf{A}_3$ sind wieder untere Dreiecksmatrizen. $\mathbf{A}^{-1}$ kann dargestellt werden als

$$\mathbf{A}^{-1} = \begin{pmatrix} \mathbf{A}_1{}^{-1} & \mathbf{0} \\ -\mathbf{A}_3{}^{-1}\mathbf{A}_2\mathbf{A}_1{}^{-1} & \mathbf{A}_3{}^{-1} \end{pmatrix}.$$

Auf Basis der obigen Darstellung ergibt sich folgender Algorithmus

a) Falls $n = 1$, dann ist die Inverse $\mathbf{A}^{-1}$ gleich dem Kehrwert des einzigen Elements.

b) Falls $n > 1$, dann zerlege $\mathbf{A}$ wie oben beschrieben und

[1] Für reguläre Matrizen $\mathbf{A}$ und $\mathbf{B}$ gilt allgemein $\mathbf{AB}^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$

1) Invertiere parallel $\mathbf{A}_1$ und $\mathbf{A}_3$ mit diesem Algorithmus.

2) Multipliziere $\mathbf{A}_3{}^{-1}$ und $\mathbf{A}_2$.

3) Multipliziere obiges Produkt mit $\mathbf{A}_1^{-1}$.

Die beiden letzten Multiplikationen benötigen $O(\log n)$ Schritte bei Verwendung von n^3 Prozessoren. Damit ist der Aufwand $T(n) = T(\lceil n/2 \rceil) + O(\log n)$ oder $T(n) = O(\log^2 n)$ mit n^3 Prozessoren.

Beim Vergleich der beiden letzten Verfahren mit der sequentiellen Rücksubstitution liegt die Effizienz in $\Theta(1/(n \log^2 n))$ und ist damit sehr schlecht. Das Verfahren benötigt zwar relativ wenige Rechenschritte, der Preis dafür ist aber eine extrem hohe Prozessorzahl. Insofern ist die nicht ganz so schnelle parallele Rücksubstitution in der Praxis meistens vorzuziehen.

Tridiagonale Matrizen

Neben Dreiecksmatrizen sind tridiagonale Matrizen günstig für eine effiziente Analyse. In einer tridiagonalen Matrix $\mathbf{A}$ sind die Elemente $A_{i,j} = 0$ falls $|i - j| > 1$, d.h. nur die Diagonal- und die beiden Nebendiagonalelemente sind ungleich 0. Tridiagonale Matrizen treten in zahlreichen realen Anwendungen auf, in denen ein Zustand (Knoten) jeweils nur mit seinen Nachbarn kommuniziert. Zur Vereinfachung der Notation bezeichnen wir die Diagonalelemente mit d_i, die Elemente der unteren Nebendiagonalen mit l_i und die Element der oberen Nebendiagonalen mit u_i. Damit hat $\mathbf{A}$ die folgende Form

$$\begin{pmatrix} d_1 & u_1 & & & & \\ l_2 & d_2 & u_2 & & 0 & \\ & l_3 & d_3 & u_3 & & \\ & & & \ddots & & \\ & 0 & & l_{n-1} & d_{n-1} & u_{n-1} \\ & & & & l_n & d_n \end{pmatrix}$$

Woraus sich die Gleichungen für ein lineares Gleichungssystem $\mathbf{A}x = b$ zu

$$\begin{aligned} d_1 x_1 + u_1 x_2 &= b_1 \\ l_i x_{i-1} + d_i x_i + u_i x_{i+1} &= b_i, \quad i = 2, \ldots, n-1 \\ l_n x_{n-1} + d_n x_n &= b_n \end{aligned} \tag{5.1}$$

ergeben. Zur Vereinfachung der Schreibweise sei $x_0 = x_{n+1} = 0$ definiert. Falls $d_i \neq 0$ können wir x_i in Abhängigkeit von x_{i-1} und x_{i+1} darstellen.

$$x_i = \frac{1}{d_i}(b_i - l_i x_{i-1} - u_i x_{i+1}) \tag{5.2}$$

Setzen wir nun den Ausdruck 5.2 in jene Gleichungen (5.1) mit geradem Index $2i$ ein, $1 \leq 2i \leq n$, und zwar jeweils für x_{2i-1} und x_{2i+1}, so erhalten wir ein lineares Gleichungssystem, in dem Abhängigkeiten nur noch zwischen Variablen mit geraden Indizes auftreten, d.h.

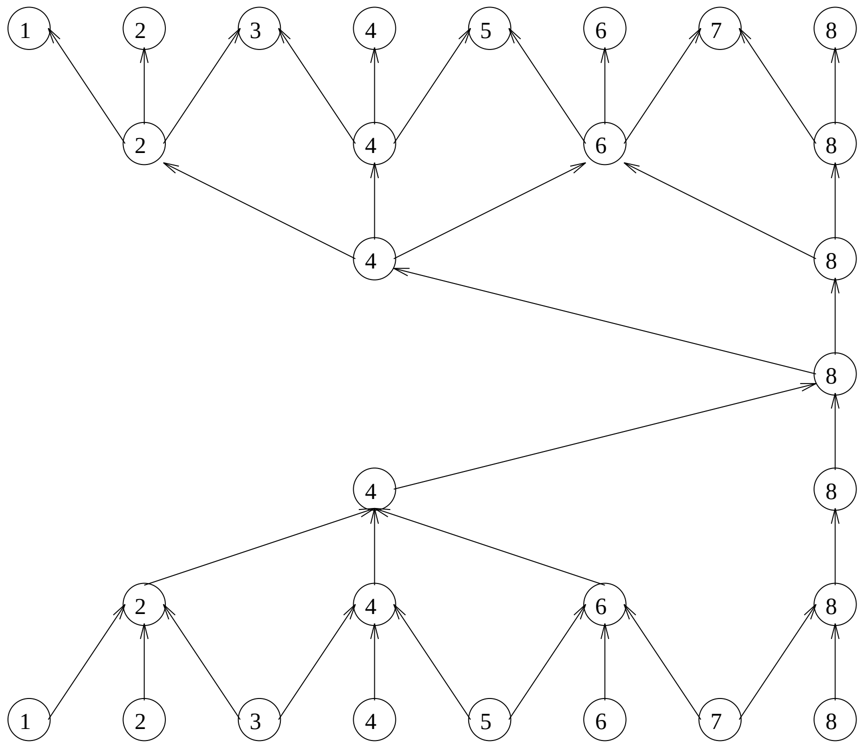


Abbildung 5.1: Reduktion und Berechnung für $n = 8$ Variablen.

$$\begin{aligned} & l^1_{2i} x_{2i-2} + d^1_{2i} x_{2i} + u^1_{2i} x_{2i+2} = b^1_{2i} \\ & \text{für} \quad 1 \leq i \leq n/2 \\ & \qquad l^1_{2i} = -\tfrac{l_{2i} l_{2i-1}}{d_{2i-1}} \\ & \qquad d^1_{2i} = -\tfrac{u_{2i-1} l_{2i}}{d_{2i-1}} + d_{2i} - \tfrac{u_{2i} l_{2i+1}}{d_{2i+1}} \\ & \qquad u^1_{2i} = -\tfrac{u_{2i} u_{2i+1}}{d_{2i+1}} \\ & \qquad b^1_{2i} = b_{2i} - \tfrac{l_{2i} b_{2i-1}}{d_{2i-1}} - \tfrac{u_{2i} b_{2i+1}}{d_{2i+1}} \end{aligned} \tag{5.3}$$

Gleichung (5.3) umfaßt nur noch die x_i mit geradem Index i und ist offensichtlich wieder ein tridiagonales Gleichungssystem. Damit sind wir wieder beim Ausgangsproblem, nämlich der Suche nach der Lösung eines Gleichungssystems mit tridiagonaler Struktur, nun aber nur noch mit $\lfloor n/2 \rfloor$ Variablen. Auf dieses System ist natürlich das obigen Vorgehen wieder anwendbar, wodurch sich die Variablenzahl abermals ungefähr halbiert. Dies wird solange wiederholt, bis nur noch eine Gleichung mit einer Variablen vorliegt, die direkt gelöst werden kann. Anschließend wird rückwärts eingesetzt, um so die Werte für die restlichen Variablen zu bestimmen. Abbildung 5.1 zeigt das Vorgehen für $n = 8$. Zuerst werden die Variablen x_1, x_3, x_5 und x_7 eliminiert, dann die Variablen x_2 und x_6 und abschließend x_4. Die verbleibende Gleichung kann direkt für x_8 gelöst werden. Unter Verwendung von x_8 wird dann x_4 bestimmt, daraus x_2 und x_6 und abschließend x_1, x_3, x_5 und x_7.

Der Algorithmus funktioniert nicht, wenn an irgend einer Stelle durch 0 dividiert werden muß (d.h. ein Diagonalelement einer Matrix 0 wird). Dies ist immer der Fall, wenn die Ausgangsmatrix eine 0 in der Diagonalen beinhaltet, kann aber auch auftreten, wenn alle Diagonalelemente der Ausgangsmatrix ungleich 0 sind. Es werden also nicht für alle lösbaren Gleichungssysteme die Lösungen berechnet. In der Praxis tritt der Fall einer Division durch 0 allerdings recht selten auf. Für viele relevante Klassen von Matrizen, wie diagonaldominante und symmetrische positiv definite Matrizen, kann man beweisen, daß eine Division durch 0 nicht vorkommen kann. Tridiagonale Matrizen, die mit obiger Methode nicht lösbar sind, können mit einem der im folgenden vorgestellten Verfahren zur Analyse allgemeiner Gleichungssysteme analysiert werden.

Betrachten wir nun die Laufzeit und den Prozessorbedarf des Algorithmus. In jedem Schritt wird die Zahl der Variablen halbiert, d.h. nach $\Theta(\log n)$ Schritten wurden alle Variablen eliminiert. In jedem Schritt müssen die Koeffizienten des reduzierten Systems berechnet werden, dazu müssen die vier Koeffizienten aus (5.3) bestimmt werden. Dies kann offensichtlich mit $O(n)$ Prozessoren in konstanter Zeit durchgeführt werden. In ähnlicher Form kann die Rücksubstitution vorgenommen werden. Damit folgt, daß der gesamte Algorithmus in $O(\log n)$ Schritten mit $O(n)$ Prozessoren ausgeführt werden kann.

Vergleichen wir nun den Aufwand mit dem Aufwand für einen sequentiellen Algorithmus, um die Effizienz zu bestimmen. Ein sequentieller Algorithmus benötigt $O(n)$ Schritte, wie man sich überlegen kann (die Anzahl der zu berechnenden Koeffizienten halbiert sich in jedem Schritt). Andererseits sind auch $\Omega(n)$ Schritte notwendig, da dies schon der Aufwand zum Einlesen der Matrix ist. Damit ist $T^*(n) = \Theta(n)$ und die Effizienz der obigen Implementierung ist $\Theta(1/\log n)$. Ein effizienterer Algorithmus, der Zeitaufwand $O(\log n)$ aber nur $O(n/\log n)$ Prozessoren benötigt, ist leicht abzuleiten, wenn man sich die Ideen zur Effizienzsteigerung bei der Skalar-Addition noch einmal vergegenwärtigt. Wir wollen diesen Ansatz hier nicht weiter vertiefen. Das beschriebene Vorgehen bezeichnet man auch als "odd/even Reduktion".

Wir wollen nun die Implementierung des Algorithmus auf unterschiedlichen Topologien etwas näher untersuchen. Beginnen wir mit dem linearen Array mit n Prozessoren und nehmen an, daß Prozessor i die i-te Zeile der Matrix $\mathbf{A}$ und das Element b_i speichert. Weiterhin soll in i auch der Wert von x_i berechnet werden. Der erste Reduktionsschritt ist mit einem Kommunikationsaufwand in $O(1)$ durchführbar, da Koeffizienten jeweils nur den Nachbarprozessoren übermittelt werden müssen. Nach $k < \log n$ bleibt ein System, in dem die Variablen x_i (mit $i = j2^k$) noch nicht reduziert wurden. Dies bedeutet, daß Prozessor $j2^k$ mit Prozessor $(j+1)2^k$ kommunizieren muß. Damit liegt die Kommunikationzeit im letzten Reduktionsschritt in $\Omega(n)$ und die Laufzeit des gesamten Algorithmus in $O(n)$, was der Laufzeit des sequentiellen Algorithmus entspricht. Dieses ineffiziente Verhalten des parallelen Algorithmus liegt darin begründet, daß die Berechnung von x_1 von Resultaten abhängt, die in Prozessoren vorliegen, deren Entfernung in $\Theta(n)$ liegt, somit muß der parallele Algorithmus eine Ausführungszeit in $\Omega(n)$ haben.

Offensichtlich ist damit die Verwendung eines n Prozessor Arrays ineffizient. Lohnt es sich, ein Array mit $p < n$ Prozessoren zu verwenden? Jeder Prozessor berechnet dann $\lfloor n/p \rfloor$ oder $\lceil n/p \rceil$ Variablen. Die ersten $m = \lfloor \log(n/p) \rfloor$ Berechnungsschritte werden lokal mit einem geringen Kommunikationsaufwand auf den einzelnen Prozessoren ausgeführt. Die Berechnungszeit für diese Schritte ist $O(n/p)$. Nach m Schritten bleibt ein tridiagonales System mit $O(p)$ Variablen. Dieses System kann auf einem einzelnen Prozessor gelöst werden, indem erst die Koeffizienten zu diesem Prozessor übertragen, dann die Lösungen berechnet und anschließend die Resultate per Broadcast zurück übertragen werden. Der dazu nötige Kommunikationsaufwand liegt in $O(p)$, der Berechnungsaufwand ebenfalls in $O(p)$. Damit ist der Gesamtaufwand $O(n/p) + O(p)$. Optimierung der Prozessorzahl p liefert $p = \Theta(\sqrt{n})$ und eine Gesamtlaufzeit in $O(\sqrt{n})$. Dies ist besser als die Laufzeit des sequentiellen Algorithmus $O(n)$, aber schlechter als die optimale parallele Laufzeit $O(\log n)$.

Betrachten wir nun die Implementierung auf einem Hyperwürfel mit n Prozessoren. Wir nehmen dazu an, daß ein lineares Array eingebettet wurde und dazu der RGC verwendet wurde, wie in Abschnitt 2.5.4 vorgestellt. Damit kann der Algorithmus wie oben beschrieben auf dem linearen Array ablaufen. Die Einbettung benutzt aber klarerweise nicht alle Verbindungen des Hyperwürfels. Aus der Konstruktion des RGCs wird ersichtlich, daß zwei Prozessoren mit Adressen $j2^k$ und $(j+1)2^k$, die im Array einen Abstand von 2^k haben, im Hyperwürfel einen Abstand von 2 haben. Es können alle Prozessorpaare der Form $(j2^k, (j+1)2^k)$ simultan in zwei Schritten miteinander kommunizieren. Gerade die Kommunikation zwischen diesen Prozessoren verursacht die langen Laufzeit des Algorithmus im linearen Array. Damit liegt die Kommunikationszeit auf einem Hyperwürfel für jeden Schritt in $O(1)$ und die Gesamtlaufzeit in $O(\log n)$, was dem optimalen Wert entspricht, der ohne Berücksichtigung der Kommunikationszeit ermittelt wurde. Es sollte noch einmal erwähnt werden, daß diese asymptotischen Betrachtungen die reale Kommunikationszeit nicht einbeziehen. Es ist immer noch möglich, daß die Kommunikationszeit sehr viel größer als die Rechenzeit ist, auch wenn beide in $O(1)$ für jeden Rechenschritt liegen. In einem solchen Fall ist es natürlich sinnvoll, weniger als n Prozessoren zu verwenden.

Die Methode der "odd/even Reduktion" ist leicht auf blocktridiagonale Matrizen übertragbar. In einer blocktridiagonalen Matrix sind die x_i und b_i k-dimensionale Vektoren und die L_i, D_i und U_i $k \times k$ Matrizen[2]. Der einzige Unterschied ist, daß die Division durch d_i nun als Multiplikation mit der inversen Matrix ${D_i}^{-1}$ interpretiert wird (d.h. das Verfahren funktioniert nur wenn ${D_i}^{-1}$ existiert). Jeder Prozessor muß damit $k \times k$ Matrizen invertieren, der Aufwand dafür liegt normalerweise in $O(k^3)$. Gleichzeitig müssen pro Schritt nur $O(k^2)$ Werte an andere Prozessoren übertragen werden. D.h. der Zeitanteil für Berechnungen steigt und der Zeitanteil für Kommunikation sinkt bei der Analyse blocktridiagonale Systeme im Vergleich zu tridiagonalen Systemen.

5.1.2 Direkte Verfahren für dicht besetzte Matrizen

In diesem Abschnitt stellen wir zwei Ansätze zur direkten Lösung des Gleichungssystems $\mathbf{Ax} = \mathbf{b}$ vor. Wir nehmen an, daß $\mathbf{A}$ eine dicht besetzte $n \times n$ Matrix ist. Zuerst wird die bekannte Gauß Elimination und deren Parallelisierung vorgestellt. Daran anschließend untersuchen wir zwei Methoden zur Matrix Faktorisierung.

Gauß Elimination

Die Gauß Elimination ist ein schon aus der Schule bekanntes Verfahren zur Lösung linearer Gleichungssysteme. Die Idee bei der Gauß Elimination ist es, sukzessive Transformationen auf $\mathbf{A}$ anzuwenden, so daß eine obere Dreiecksmatrix entsteht. Diese kann anschließend per Rücksubstitution gelöst werden. Die Transformationen bestehen in der Multiplikation von $\mathbf{A}$ mit einer Sequenz von Matrizen $\mathbf{M}^{(1)}, \ldots, \mathbf{M}^{(K)}$ von links. Die Matrizen $\mathbf{M}^{(i)}$ werden so gewählt, daß die Multiplikation mit $\mathbf{A}$ einfach auszuführen ist und das Produkt $\mathbf{M}^{(K)} \ldots \mathbf{M}^{(1)} A$ eine obere Dreiecksmatrix ist.

Falls die einzelnen $\mathbf{M}^{(i)}$ invertierbar sind, so entspricht $\mathbf{Ax} = \mathbf{b}$ dem Gleichungssystem

$$\mathbf{M}^{(k)} \ldots \mathbf{M}^{(1)} \mathbf{Ax} = \mathbf{M}^{(k)} \ldots \mathbf{M}^{(1)} \mathbf{b} \tag{5.4}$$

Da $\mathbf{M}^{(K)} \ldots \mathbf{M}^{(1)} \mathbf{A}$ eine obere Dreiecksmatrix ist, kann die obige Gleichung in $O(n)$ Schritten mit n Prozessoren durch Rücksubstitution gelöst werden. Soll die Inverse $\mathbf{A}^{-1}$ gebildet werden, so kann dies kann dies auf ähnliche Weise mit der folgenden Gleichung erfolgen.

$$\mathbf{A}^{-1} = (\mathbf{M}^{(K)} \ldots \mathbf{M}^{(1)} \mathbf{A})^{-1} (\mathbf{M}^{(K)} \ldots \mathbf{M}^{(1)}) \tag{5.5}$$

[2]Die einzelnen Blöcke können natürlich auch eine Dimension k_i haben, die vom Index i abhängt.

Die Gültigkeit von Gleichung 5.5 läßt sich durch linksseitige Multiplikation mit $\mathbf{M}^{(K)} \ldots \mathbf{M}^{(1)}\mathbf{A}$ unmittelbar nachvollziehen. Die Dreiecksmatrix $\mathbf{M}^{(K)} \ldots \mathbf{M}^{(1)}\mathbf{A}$ kann in $O(n)$ Schritten mit n^2 Prozessoren invertiert werden. Da die Inversion leicht aus der Lösung $\mathbf{Ax} = \mathbf{b}$ abzuleiten ist, betrachten wir im folgenden nur dieses Problem.

Bei der Gauß Elimination wird jede Variable x_i als Funktion der Variablen $x_{i+1}, \ldots, x_n$ dargestellt, nach $n-1$ Schritten bleibt nur noch eine Gleichung mit x_n über. Diese kann einfach gelöst und die restlichen Variablen per Rücksubstitution gewonnen werden. Sei nun $\mathbf{C}^{(0)} = \mathbf{A}$ und $\mathbf{C}^{(i)} = \mathbf{M}^{(i)}\mathbf{C}^{(i-1)} = \mathbf{M}^{(i)} \ldots \mathbf{M}^{(1)}\mathbf{A}$. Nehmen wir nun an $\mathbf{C}^{(i-1)}$ sei bekannt und erfülle folgende Bedingung: $C_{k,j}^{(i-1)} = 0$ für $k < \min(i,j)$. Damit hat $\mathbf{C}^{(i-1)}$ die folgende Struktur.

$$\mathbf{C}^{(i-1)} = \begin{pmatrix} C_{1,1}^{(i-1)} & \cdots & \cdots & \cdots & \cdots & C_{1,n}^{(i-1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & \ddots & & & \\ 0 & \cdots & 0 & C_{i,i}^{(i-1)} & \cdots & C_{i,n}^{(i-1)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & C_{n,i}^{(i-1)} & \cdots & C_{n,n}^{(i-1)} \end{pmatrix}$$

Nehmen wir an, daß $C_{i,i}^{(i-1)} \neq 0$. Sei ferner $\mathbf{M}^{(i)} = \mathbf{I} - \mathbf{N}^{(i)}$, wobei $\mathbf{N}^{(i)}$ eine Matrix ist, in der nur die Elemente $N_{i+1,i}^{(i)}, N_{i+2,i}^{(i)}, \ldots, N_{n,i}^{(i)}$ ungleich 0 sind. Die Nichtnullelemente von $\mathbf{N}^{(i)}$ sind gegeben als

$$N_{j,i}^{(i)} = \frac{C_{j,i}^{(i-1)}}{C_{i,i}^{(i-1)}} \qquad \text{für } j > i. \tag{5.6}$$

Bleibt zu zeigen, daß $\mathbf{C}^{i} = (\mathbf{I} - \mathbf{N}^{(i)})\mathbf{C}^{(i-1)}$ die geforderte Struktur aufweist. Da die ersten $i-1$ Spalten von $\mathbf{N}^{(i)}$ gleich 0 sind, sind die ersten $i-1$ Spalten von $\mathbf{C}^{(i)}$ identisch zu den entsprechenden Spalten in $\mathbf{C}^{(i-1)}$. Für die Elemente $C_{j,i}^{(i)}$ mit $j > i$ gilt

$$C_{j,i}^{(i)} = C_{j,i}^{(i-1)} - \frac{C_{j,i}^{(i-1)}}{C_{i,i}^{(i-1)}} C_{i,i}^{(i-1)} = 0.$$

Womit $\mathbf{C}^{(i)}$ die geforderte Struktur aufweist. Der Vollständigkeit halber betrachten wir noch die restlichen Elemente $C_{j,k}^{(i)}$ mit $j,k > i$.

$$C_{j,k}^{(i)} = C_{j,k}^{(i-1)} - \frac{C_{j,i}^{(i-1)}}{C_{i,i}^{(i-1)}} C_{i,k}^{(i-1)}$$

Das obige Vorgehen beschreibt die einfache Gauß Elimination. Die Matrizen $\mathbf{M}^{(i)}$ sind invertierbar, wie man sich leicht überlegen kann. Das Verfahren funktioniert nicht, wenn ein Wert $C_{i,i}^{(i)} = 0$ vorkommt. Numerische Probleme treten auf, wenn der Wert von $C_{i,i}^{(i)}$ sehr klein wird und dadurch die Matrix $\mathbf{M}^{(i)}$ sehr große Werte beinhaltet. Es gibt einige Klassen von Matrizen, wie diagonaldominante und symmetrische Matrizen, bei denen bekannt ist, daß die Diagonalelemente niemals 0 werden. Für den allgemeinen Fall wird mit der zeilenweisen Pivotisierung gearbeitet, d.h. es werden Zeilen ausgetauscht, um so möglichst große Diagonalelemente zu erreichen. Vor der Berechnung von $\mathbf{C}^{(i)}$ wird ein Zeilenindex j^* wie folgt bestimmt.

$$|C_{j^*,i}^{(i-1)}| = \max_{j \geq i} |C_{j,i}^{(i-1)}| \tag{5.7}$$

Falls $j^* \neq i$ werden die Zeilen i und j^* ausgetauscht. Dies geschieht durch Linksmultiplikation von $\mathbf{C}^{(i-1)}$ und auch der rechten Seite der Gleichung mit einer (Permutations-) Matrix $\mathbf{P}^{ij^*}$ mit folgender Struktur (nicht dargestellte Elemente sind gleich 0).

$$\mathbf{P}^{ij^*} = \begin{array}{cc}
\left(\begin{array}{ccccccccccc}
1 &&&&&&&&&& \\
& \ddots &&&&&&&&& \\
&& 1 &&&&&&&& \\
&&& 0 &&&& 1 &&& \\
&&&& 1 &&&&&& \\
&&&&& \ddots &&&&& \\
&&&&&& 1 &&&& \\
&&& 1 &&&& 0 &&& \\
&&&&&&&& 1 && \\
&&&&&&&&& \ddots & \\
&&&&&&&&&& 1
\end{array}\right) &
\begin{array}{c} \\ \\ \\ i \\ \\ \\ \\ j^* \\ \\ \\ \\ \end{array} \\
\begin{array}{ccccccccccc} & & & i & & & & j^* & & & \end{array} &
\end{array}$$

Die Pivotisierung braucht natürlich nicht unbedingt in jedem Schritt ausgeführt zu werden, es reicht oftmals aus, nur Zeilen zu vertauschen, wenn die Diagonalelemente zu klein oder 0 werden. Auch Pivotisierung schützt nicht zwangsläufig vor numerischen Problemen, zeigt sich in der Praxis aber als relativ robust. Es kann bei der Verwendung von Pivotisierung außerdem davon ausgegangen werden, daß für alle nicht singulären Matrizen eine Lösung gefunden wird (d.h. man findet immer eine Pivotzeile, so daß das Diagonalelement ungleich 0 wird). Neben der Matrix muß auch der rechtsseitige Vektor durch Multiplikation mit $\mathbf{M}^{(k)} \ldots \mathbf{M}^{(1)}$ transformiert werden. Das Vorgehen ist vollkommen analog zur Berechnung der Matrixelemente $C_{i,k}^{(i)}$ $(k \geq i)$. Natürlich müssen die Elemente von $\mathbf{b}$ auch permutiert werden, wenn ein Pivotschritt eine Permutation der Matrixzeilen erfordert. Zur Vereinfachung der Schreibweise kann $\mathbf{Ab}$ als $n \times (n+1)$ Matrix interpretiert werden und die beschriebenen Operationen können auf diese Matrix angewendet werden.

Beginnen wir mit der parallelen Implementierung der Gauß Elimination ohne Pivotisierung und vernachlässigen die Kommunikationszeiten. Jedes Element von $\mathbf{C}^{(i)}$ kann mit einer Multiplikation, einer Division und einer Subtraktion berechnet werden. Mit n^2 Prozessoren sind dazu drei Schritte simultan für alle Elemente der Matrix $\mathbf{C}^{(i)}$ notwendig. Damit beträgt die Gesamtlaufzeit $3n$ mit n^2 Prozessoren und liegt in $O(n^2)$ bei Verwendung von n Prozessoren. Vergleichen wir dies mit der sequentiellen Implementierung, deren Laufzeit in $\Theta(n^3)$ liegt, so zeigt sich, daß die Effizienz bei der Verwendung von n oder n^2 Prozessoren in $\Theta(1)$ liegt.

Betrachten wir nun die Implementierung auf speziellen Topologien und beziehen die Kommunikationszeiten in die Analyse mit ein. Gauß Elimination ohne Pivotisierung kann auf einem $n \times n$ (bzw. $n \times (n+1)$) Gitter mit einer Laufzeit in $O(n)$ implementiert werden. Jeder Prozessor bekommt einen Eintrag der Matrix zugeteilt. Um wirklich eine Laufzeit in $O(n)$ zu erreichen, müssen Berechnungen und Kommunikation geeignet überlagert werden, d.h. die Berechnung der i-ten Stufe beginnt bevor Stufe $i-1$ abgeschlossen ist. Prozessor (i,i) verteilt nach der Berechnung $C_{i,i}^{(i-1)}$ den Wert nach "unten" (an die Prozessoren (j,i) mit $j > i$). Prozessor (j,i) reicht den Wert $C_{i,i}^{(i-1)}$ weiter an Prozessor $(j+1,i)$, berechnet $C_{j,i}^{(i-1)}/C_{i,i}^{(i-1)}$ und reicht diesen Wert direkt an Prozessor $(j,i+1)$ weiter. Gleichzeitig werden die Werte von $C_{i,k}^{(i-1)}$ in der k-ten Spalte nach "unten" gereicht. Durch diese Überlappung von Kommunikation und Berechnung wird erreicht, daß der Kommunikationsaufwand in der selben Größenordnung wie der Berechnungsaufwand liegt. Auch an dieser Stelle sollte wieder erwähnt werden, daß der tatsächliche

Prozessorzahl	Topologie	ohne Pivotschritt	mit Pivotschritt
n^2	Hyperwürfel	$O(n)$	$O(n \log n)$
n^2	$n \times n$ Array	$O(n)$	$\Omega(n^{4/3})$
$n^2/\log n$	Hyperwürfel	$O(n \log n)$	$O(n \log n)$
n	lineares Array	$O(n^2)$	$O(n^2)$

Tabelle 5.1: Zeitaufwand der Gauß Elimination

Aufwand nicht allein von der Größenordnung, sondern auch von den Konstanten abhängt. Bei längeren Kommunikationszeiten ist deshalb die Verwendung von weniger Prozessoren sinnvoll. Nach Transformation der Matrix in eine obere Dreiecksmatrix muß natürlich noch die Rücksubstitution durchgeführt werden, die ebenfalls einen Zeitaufwand in $O(n)$ benötigt. Der Zeitaufwand der Rücksubstitution fällt bei der parallelen Implementierung stärker ins Gewicht als bei der sequentiellen. Deshalb sollte auf eine effiziente Implementierung geachtet werden.

Auf einem Hyperwürfel mit n^2 Prozessoren kann die Gauß Elimination natürlich auch in $O(n)$ Schritten ausgeführt werden, da ein $n \times n$ Array eingebettet werden kann. Auf einem linearen Array mit n Prozessoren kann der Algorithmus in $O(n^2)$ ablaufen, da ein $n \times n$ Array mit einer Reduktion der Geschwindigkeit, die in $O(n)$ liegt, auf einem linearen Array mit n Prozessoren simuliert werden kann.

Betrachten wir nun den zusätzlichen Aufwand der Pivotisierung zuerst wieder unabhängig von der Topologie. Ein Pivotschritt erfordert den Vergleich von $O(n)$ Zahlen, zur Bestimmung des Maximums. Unter Weglassen der Kommunikationszeiten und der Annahme, daß ein Prozessor zwei Zahlen in einem Schritt vergleichen kann, werden $\Omega(n)$ Vergleiche zur Bestimmung des Maximums benötigt. Der Aufwand der Maximumsbestimmung liegt damit in $\Omega(\log n)$ egal wie viele Prozessoren verwendet werden, jedenfalls solange wir vom zugrunde gelegten Fixed-connection Modell ausgehen. Daraus folgt natürlich auch, daß die Gesamtlaufzeit in $\Omega(n \log n)$ und die Effizienz in $\Theta(1/\log n)$ liegt, wenn n^2 Prozessoren verwendet werden. Mit n Prozessoren benötigt jeder Schritt einen Aufwand in $O(n)$, womit die Gesamtlaufzeit in $O(n^2)$ und die Effizienz in $\Theta(1)$ liegt.

Die Realisierung der Pivotisierung auf einer festen Topologie ist schwieriger. Es existiert kein $O(n \log n)$ Algorithmus für ein $n \times n$ Gitter. Andererseits ist eine solche Implementierung auf einem Hyperwürfel mit n^2 (und sogar mit $O(n^2/\log n)$) Prozessoren möglich. Auf einem linearen Array mit n Prozessoren ist eine Implementierung mit Laufzeit in $O(n^2)$ möglich, wenn jeder Prozessor eine Spalte der Matrix speichert, da dann der Austausch von Zeilen ohne Datenübertragungen erfolgen kann. Die Ergebnisse dieses Abschnitts sind noch einmal in Tabelle 5.1 zusammengefaßt.

Matrix Faktorisierung

Die vorgestellte Gauß Elimination modifiziert simultan die Koeffizientenmatrix und den Lösungsvektor. Eine Alternative besteht darin, die Modifikationen zuerst einmal nur auf der Matrix $\mathbf{A}$ durchzuführen und erst bei der Lösung des Systems auch den Vektor $\mathbf{b}$ zu modifizieren. Dieses Vorgehen hat den Vorteil, daß die relativ aufwendige Faktorisierung der Matrix $\mathbf{A}$ nur einmal durchgeführt werden muß und die Lösung anschließend für verschiedene Vektoren $\mathbf{b}$ bestimmt werden kann. Ein solcher Fall tritt bei der Bestimmung der inversen Matrix über die Lösung linearer Gleichungssysteme auf.

Wir beginnen mit der LU-Faktorisierung, die im wesentlichen der Gauß Elimination entspricht. Sei $\mathbf{A} = \mathbf{LU}$, wobei $\mathbf{L}$ eine untere und $\mathbf{U}$ eine obere Dreiecksmatrix ist. Angenommen wir hätten eine solche Zerlegung gefunden, so läßt sich die Lösung von $\mathbf{Ax} = \mathbf{b}$ wie folgt berechnen.

$$\mathbf{Ly} = \mathbf{b} \text{ und } \mathbf{Ux} = \mathbf{y} \tag{5.8}$$

Da $\mathbf{L}$ und $\mathbf{U}$ Dreiecksmatrizen sind, erfordert die obige Gleichung den zweimaligen Einsatz der Rücksubstitution. Bleibt damit noch die Generierung der Matrizen $\mathbf{L}$ und $\mathbf{U}$ zu zeigen. Ähnlich zum Vorgehen im letzten Abschnitt definieren wir eine Matrix $\mathbf{D}^{(0)} = \mathbf{A}$. Ausgehend von dieser Matrix werden nun die Spalten $\mathbf{L}$ und Zeilen von $\mathbf{U}$ erzeugt. Sei $\mathbf{l}^{(k)}$ die k-te Spalte von $\mathbf{L}$ und $\mathbf{u}^{(k)}$ die k-te Zeile von $\mathbf{U}$. Es soll gelten

$$\mathbf{A} = \sum_{i=1}^{k} \mathbf{l}^{(i)}\mathbf{u}^{(i)} + \mathbf{D}^{(k)} \text{ für alle } k = 1, \ldots, n \tag{5.9}$$

Die Elemente von $\mathbf{l}^{(k)}$, $\mathbf{u}^{(k)}$ und $\mathbf{D}^{(k)}$ werden wie in den folgenden Gleichungen gezeigt erzeugt[3].

$$\mathbf{l}^{(k)} = (0, \ldots, 0, 1, \frac{D^{(k-1)}_{k+1,k}}{D^{(k-1)}_{k,k}}, \ldots, \frac{D^{(k-1)}_{n,k}}{D^{(k-1)}_{k,k}})^T \tag{5.10}$$

$$\mathbf{u}^{(k)} = (0, \ldots, 0, D^{(k-1)}_{k,k}, D^{(k-1)}_{k,k+1}, \ldots, D^{(k-1)}_{k,n}) \tag{5.11}$$

$$\mathbf{D}^{(k)} = \begin{pmatrix} 0 & 0 & \cdots & 0 & \cdots & \cdots & 0 \\ 0 & 0 & & \vdots & & & 0 \\ \vdots & & \ddots & \vdots & & & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & \cdots & 0 \\ \vdots & & & \vdots & D^{(k)}_{k+1,k+1} & \cdots & D^{(k)}_{k+1,n} \\ \vdots & & & \vdots & \vdots & & \vdots \\ 0 & \cdots & \cdots & 0 & D^{(k)}_{n,k+1} & \vdots & D^{(k)}_{n,n} \end{pmatrix} \tag{5.12}$$

$$\text{mit } D^{(k)}_{i,j} = \begin{cases} D^{(k-1)}_{i,j} - D^{(k-1)}_{i,k} D^{(k-1)}_{k,j} / D^{(k-1)}_{k,k} & \text{falls } i, j > k \\ 0 & \text{sonst} \end{cases}$$

Die Parallelisierung der LU-Faktorisierung ist sehr ähnlich zur normalen Gauß Elimination, Matrix U entspricht der oberen Dreiecksmatrix aus dem Gauß Verfahren, Matrix L speichert die hierzu erforderlichen Transformationen. Die Elemente der Matrix $\mathbf{D}^{(i)}$ können wieder in 3 Schritten mit n^2 Prozessoren ohne Berücksichtigung der Kommunikationszeit bestimmt werden. Damit liegt der Gesamtaufwand bei $3n$ Schritten. Auf einem $n \times n$ Array kann der Algorithmus mit Zeitaufwand in $O(n)$ implementiert werden, wobei das Vorgehen analog zur Gauß Elimination ist. Auch bei der LU-Faktorisierung ist Pivotisierung notwendig und erfordert die Vertauschung von Matrixzeilen während der Generierung der Matrizen $\mathbf{L}$ und $\mathbf{U}$.

Neben der LU-Faktorisierung gibt es noch andere Arten der Faktorisierung. Wir betrachten hier einen Ansatz, den man mit QR-Faktorisierung bezeichnet. Der Vorteil dieses Verfahrens ist eine größere numerische Stabilität. Die Größenordnung der Rundungsfehler verstärkt sich nicht über die Schritte der Berechnung, wie es bei der Gauß Elimination oder LU-Faktorisierung der Fall sein kann. Die Idee des Verfahrens ist wieder, die Matrix $\mathbf{A}$ als Produkt zweier Matrizen $\mathbf{QR}$ darzustellen. $\mathbf{R}$ ist eine obere Dreiecksmatrix und $\mathbf{Q}$ ist eine orthogonale Matrix, d.h. $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. Die Lösung von $\mathbf{Ax} = \mathbf{b}$ kann dann dargestellt werden als

[3]Wegen $\mathbf{l}^{(k)}(k) = 1$ handelt es sich um eine sogenannte Doolittle Zerlegung.

$$\mathbf{R}\mathbf{x} = \mathbf{Q}^T\mathbf{b} \tag{5.13}$$

Da $\mathbf{R}$ eine obere Dreiecksmatrix ist, kann zur Lösung die Rücksubstitution eingesetzt werden. Es gibt unterschiedliche Ansätze QR-Faktorisierungen zu berechnen, wir werden hier die sogenannte Givens Rotation einsetzen. Dies ist die Methode, die sich am besten parallelisieren läßt. Das Vorgehen ist vom Prinzip ähnlich zur Gauß Elimination. Wir definieren wieder Matrizen $\mathbf{M}^{(i)}$, die aber nun orthogonal sein sollen $((\mathbf{M}^{(i)})^T\mathbf{M}^{(i)} = \mathbf{I})$. Sei $\mathbf{C}$ eine Matrix, in der die Elemente $\mathbf{C}_{i,1}, \ldots, \mathbf{C}_{i,k-1}$ und $\mathbf{C}_{j,1}, \ldots, \mathbf{C}_{j,k-1}$ Null sind und $\mathbf{C}_{j,k}$ ungleich Null ist. Sei ferner $\mathbf{M}$ eine Matrix, in der die Elemente $\mathbf{M}_{k,k}$ für $k \neq j, i$ gleich 1 sind, die Elemente $\mathbf{M}_{i,j}$ und $\mathbf{M}_{j,i}$ gleich s sind, $\mathbf{M}_{i,i}$ gleich c, $\mathbf{M}_{j,j}$ gleich $-c$ ist und alle anderen Elemente 0 sind. $\mathbf{M}$ hat folgende Struktur (nicht dargestellte Elemente sind gleich 0).

$$\mathbf{M} = \begin{array}{cc}
\begin{pmatrix}
1 & & & & & & & & & & \\
& \ddots & & & & & & & & & \\
& & 1 & & & & & & & & \\
& & & c & & & & s & & & \\
& & & & 1 & & & & & & \\
& & & & & \ddots & & & & & \\
& & & & & & 1 & & & & \\
& & & s & & & & -c & & & \\
& & & & & & & & 1 & & \\
& & & & & & & & & \ddots & \\
& & & & & & & & & & 1
\end{pmatrix} &
\begin{matrix} \\ \\ \\ i \\ \\ \\ \\ j \\ \\ \\ \\ \end{matrix} \\
\begin{matrix} & & & i & & & & j & & & \end{matrix} &
\end{array}$$

Die Werte von c und s errechnen sich aus den folgenden Gleichungen.

$$c = \frac{C_{i,k}}{\sqrt{C_{i,k}^2 + C_{j,k}^2}} \tag{5.14}$$

$$s = \frac{C_{j,k}}{\sqrt{C_{i,k}^2 + C_{j,k}^2}} \tag{5.15}$$

Die Matrix $\mathbf{M}$ wird eine Givens Rotation genannt und ist orthogonal. Betrachten wir nun die Matrix $\mathbf{MC}$, es gilt

- Alle Zeilen bis auf die i-te und j-te sind identisch zu $\mathbf{C}$.
- Die Elemente i, l und j, l sind für alle $l < k$ gleich 0.
- Element j, k ist 0.

Mit Hilfe der Givens Rotation können nun nacheinander die Elemente unterhalb der Hauptdiagonalen zu 0 gesetzt werden. Nehmen wir an, daß nach einigen Schritten des Algorithmus eine Matrix C erreicht wurde, in der wir Indizes i, j und k $(k < j)$ finden, für die gilt $C_{i,l} = C_{j,l} = 0$ $(\forall l < k)$ und $C_{j,k} \neq 0$. M wird dann wie oben beschrieben gewählt, wodurch das Element j, k zu Null gesetzt wird. Außer i und j bleiben alle Zeilen unverändert, ferner bleiben die ersten $k - 1$ Einträge in den Zeilen i und j gleich 0. Da in jedem Schritt ein neues Nullelement hinzukommt, muß der ganze Prozeß nach einer endlichen Anzahl von Schritten stoppen. In diesem

Fall existieren keine Indizes i, j, k, die obige Bedingung erfüllen und die resultierende Matrix ist eine obere Dreiecksmatrix. Weiterhin bildet das Produkt der Matrizen $\mathbf{M}^{(i)}$ die orthogonale Matrix $\mathbf{Q}$.

Betrachten wir nun die parallele Implementierung der Givens Rotation. Jede Rotation beeinflußt genau zwei Zeilen der Matrix und läßt alle übrigen unverändert. Damit können theoretisch $\lfloor n/2 \rfloor$ Givens Rotationen parallel und unabhängig voneinander durchgeführt werden. Um diesem Wert zumindest nahe zu kommen, ist ein systematisches Vorgehen notwendig, welches beschreibt welche Zeilen wann behandelt werden. Dieses Vorgehen wird durch einen Schedule beschrieben, der aus zwei Funktionen $T(j,k)$ und $S(j,k)$ für $j < k$ besteht. $T(j,k)$ gibt an, in welchem Schritt Element j,k zu Null gesetzt wird. $S(j,k)$ beschreibt, welche zusätzliche Zeile beim Null setzen von j,k mit in die Givens Rotation einbezogen wird. Jeder Schedule muß die folgenden Bedingungen erfüllen:

- Gleichzeitige Rotationen operieren auf verschiedenen Zeilen, d.h. falls $T(j,k) = T(j',k')$ und $(j,k) \neq (j',k')$, dann müssen die Mengen $\{j, S(j,k)\}$ und $\{j', S(j',k')\}$ disjunkt sein.
- Falls $T(j,k) = t$ und $S(j,k) = i$, dann ist $T(j,l) < t$ und $T(i,l) < t$ für alle $l < k$.

Ein bestimmter Schedule kann wie folgt definiert werden

$$T(j,k) = n - j + 2k - 1 \text{ und } S(j,k) = j - 1 \tag{5.16}$$

Für $n = 8$ werden die Elemente unterhalb der Hauptdiagonalen in der im folgenden dargestellten Reihenfolge zu Null gesetzt.

$$\begin{pmatrix} * & & & & & & & \\ 7 & * & & & & & & \\ 6 & 8 & * & & & & & \\ 5 & 7 & 9 & * & & & & \\ 4 & 6 & 8 & 10 & * & & & \\ 3 & 5 & 7 & 9 & 11 & * & & \\ 2 & 4 & 6 & 8 & 10 & 12 & * & \\ 1 & 3 & 5 & 7 & 9 & 11 & 13 & * \end{pmatrix}$$

Analysieren wir nun den Aufwand des in (5.16) angegebenen Vorgehens. $T(j,k)$ erreicht eine Maximum für $j = n$ und $k = n-1$, der Wert beträgt $2n-3$. Jede einzelne Rotation kann mit n Prozessoren in $O(1)$ Schritten ohne Berücksichtigung der Kommunikationszeit durchgeführt werden. Da bis zu $n/2$ Rotationen parallel ausgeführt werden, benötigen wir n^2 Prozessoren, um jeden Schritt in $O(1)$ und damit die gesamte Rotation in $O(n)$ Schritten durchzuführen. Mit n Prozessoren liegt der Aufwand in $O(n^2)$. Damit liegt die Effizienz in $\Theta(1)$ im Vergleich zum sequentiellen Benchmark mit $T^*(n) = \Theta(n^3)$.

Der asymptotische Aufwand der Givens Rotation entspricht damit dem asymptotischen Aufwand der Gauß Elimination. Detailliertere Analysen zeigen, daß die Gauß Elimination ungefähr um den Faktor drei schneller bei gleicher Prozessorzahl ist. Dieser Nachteil wird dadurch ausgeglichen, daß keine Pivotschritte notwendig sind und die QR Faktorisierung im Gegensatz zur Gauß Elimination numerisch stabil ist (d.h. Rundungsfehler pflanzen sich nicht fort).

Die Implementierung der Givens Rotation auf spezifischen Topologien wollen wir nur sehr kurz betrachten. Es ist möglich den Algorithmus auf einem $n \times n$ Array mit Laufzeit $O(n)$ zu implementieren. Damit ist auch eine $O(n)$ Implementierung auf einem Hyperwürfel mit n^2 Prozessoren und eine $O(n^2)$ Implementierung auf einem linearen Array mit n Prozessoren möglich. Die Details der Implementierung sollen nicht weiter betrachtet werden und sind recht komplex.

5.1.3 Direkte Verfahren für dünn besetzte Matrizen

Wir können zwei Klassen von spärlich besetzten Matrizen unterscheiden. Nämlich solche mit einer spezifischen Besetzungsstruktur und vollkommen unstrukturiert besetzte Matrizen. Im ersten Fall liegen sehr oft Bandmatrizen vor. Betrachtet man die vorgestellten Faktorisierungsverfahren und auch die Gauß Elimination, so stellt man fest, daß Elemente außerhalb des "Bandes" während der gesamten Rechnung gleich 0 bleiben. Dies bedeutet, daß die entsprechenden Operationen gar nicht erst ausgeführt werden müssen und Fill In nur innerhalb des Bandes entstehen kann. Damit wird auch klar, daß die vorgestellten speziellen Speicherstrukturen für Bandmatrizen, die alle Elemente innerhalb des Bandes abspeichern, für eine direkte numerische Analyse ausreichen. Die vorgestellten Algorithmen können leicht modifiziert werden, so daß Bandmatrizen behandelbar sind.

Problematischer ist die Analyse von System mit unstrukturierter Besetzungsstruktur. Es können auch für diese Matrizen die bereits vorgestellten Algorithmen verwendet werden. Der mögliche Preis ist aber Fill in, so daß die anfangs dünn besetzte Matrix vollkommen dicht werden kann. Dies sorgt dafür, daß der Speicherplatz zunimmt und verhindert oftmals zusätzlich eine effiziente Parallelisierung. Ein Ausweg ist manchmal die Umordnung von Zeilen oder Spalten, um die Bandbreite der Matrix zu verringern. Klarerweise kann man die Gleichungen oder auch Variablen in einem linearen Gleichungssystem beliebig umordnen, ohne die Lösung zu verändern. Leider sind die bekannten Algorithmen zur Verkleinerung der Bandbreite aufwendig und schlecht parallelisierbar.

An dieser Stelle soll ein speziell für dünn besetzte Gleichungssysteme entwickelter direkter paralleler Algorithmus vorgestellt werden. Wir beschränken uns auf die Lösung von Systemen mit symmetrischer und positiv definiter Matrix unter Verwendung der Cholesky Faktorisierung. Im Prinzip lassen sich die folgenden Ansätze auch auf allgemeine Gleichungssysteme und die LU- oder QR-Faktorisierung anwenden. Der Vorteil der Cholesky Faktorisierung und der Beschränkung der Matrix Klasse ist der Wegfall des Pivotschritts. Bei der Cholesky Faktorisierung wird $\mathbf{A}$ als $\mathbf{LL}^T$ dargestellt, wobei $\mathbf{L}$ eine unter Dreiecksmatrix ist. Aus dieser Darstellung kann durch Rücksubstitution der Lösungvektor $\mathbf{x}$ gewonnen werden. Wir betrachten zuerst die sequentielle Variante der Cholesky Faktorisierung, wobei die Matrix $\mathbf{L}$ durch Überschreiben der Elemente in $\mathbf{A}$ erzeugt wird. Der Algorithmus arbeitet spaltenweise. In ähnlicher Weise kann eine zeilenweise Funktion beschreiben werden, die $\mathbf{L}^T$ berechnet.

procedure cholesky_faktorisierung $(\mathbf{A})$
    for $j = 1$ *to* n *do*
        for $k = 1$ *to* $j-1$ *do*
            for $i = j$ *to* n *do*   `cmod(j,k)`
                $A_{i,j} = A_{i,j} - A_{i,k}A_{j,k}$;
        $A_{j,j} = \sqrt{A_{j,j}}$;   `cdiv(j)`
        for $k = j+1$ *to* n *do*
            $A_{k,j} = A_{k,j}/A_{j,j}$;
end ;

Der Algorithmus besteht aus drei Schritten. Im ersten Schritt wird jede Spalte durch ein Vielfaches der vorhergehenden Spalten modifiziert. Im zweiten Schritt wird das neue Diagonalelement berechnet und im dritten Schritt wird die aktuelle Spalte mit dem neue Diagonalelement skaliert. Zur Vereinfachung der Schreibweise soll Schritt 1 im folgenden als $cmod(j,k)$ bezeichnet werden und die Schritte 2 und 3 werden unter der Bezeichnung $cdiv(j)$ zusammengefaßt. Durch Umordnen der *for* Schleifen lassen sich verschiedene Versionen der Cholesky Faktorisierung erzeugen. Üblich ist die oben gezeigte spaltenweise Variante und auch eine Variante, die über Submatrizen läuft. Beide Varianten werden für dünn besetzte Matrizen später vorgestellt. Wir werden im folgenden die Elemente von $\mathbf{A}$, die in die Faktorisierung einfließen, also die Elemente der unteren Dreiecksmatrix, mit $L_{i,j}$ statt $A_{i,j}$ bezeichnen. Je nach Fortschritt des Algorithmus beschreibt $L_{i,j}$ das ursprüngliche Element $A_{i,j}$ oder den Inhalt der faktorisierten Matrix.

Ist die Matrix $\mathbf{A}$ dünn besetzt, so treten zwei Effekte auf. In $cdiv(j)$ brauchen nur Elemente mit $A_{k,j} \neq 0$ behandelt werden und falls $A_{j,k} = 0$, dann hängt Spalte j in der Faktorisierung nicht von Spalte k ab. Dies bedeutet auch, daß eine dicht besetzte erste Spalte in der Matrix Fill In in allen nachfolgenden Spalten erzeugen kann. Eine Vereinfachung, die durch die Beschränkung auf symmetrische positiv definite Matrizen entsteht, ist die Möglichkeit, Fill In unabhängig von den numerischen Werten zu bestimmen und damit die Möglichkeit, eine optimale Ordnung der Spalten zu wählen. Dies hat noch einen zweiten positiven Effekt. Durch die Kenntnis der späteren Besetzungsstruktur, kann direkt eine Datenstruktur aufgebaut werden, die Platz für alle Nichtnullelemente bereitstellt. Das aufwendige Umspeichern von Matrixelementen und Allokieren/Freigeben von Speicherplatz entfällt.

Für die Beschreibung des parallelen Algorithmus wollen wir einige zusätzliche Notationen einführen. Sei $\mathbf{L}_{i*}$ die i-te Zeile und $\mathbf{L}_{*j}$ die j-te Spalte der Matrix $\mathbf{L}$. Die Nichtnullelemente einer Zeile oder Spalte werden in den folgenden Mengen gesammelt.

$$Nz(\mathbf{L}_{i*}) = \{k < i | L_{i,k} \neq 0\} \text{ und } Nz(\mathbf{L}_{*j}) = \{k > j | L_{k,j} \neq 0\}$$

Mit dieser Schreibweise kann die Cholesky Faktorisierung über Spalten und Submatrizen wie folgt dargestellt werden.

```
Spalten_Cholesky
    for j = 1 to n do
        for k ∈ Nz(L_j*) do
            cmod(j,k)
        cdiv(j) ;
end ;
```

```
Submatrix_Cholesky
    for k = 1 to n do
        cdiv(k) ;
        for j ∈ Nz(L_*k) do
            cmod(j,k) ;
end ;
```

Das grundsätzliche Vorgehen beider Varianten wird in Abb. 5.2 gezeigt. Wird über die Spalten gegangen, dann wird in einem Schritt eine Spalte bzgl. aller vorherigen Spalten modifiziert. Im zweiten Fall wird eine Menge von Spalten bzgl. einer einzelnen Spalte modifiziert. In beiden Fällen wird von links nach rechts vorgegangen.

Es gibt bei der Cholesky Faktorisierung, wie bei fast allen anderen Algorithmen, verschiedene Ebenen auf denen parallelisiert werden kann. Wir nehmen an, daß die Zahl der Prozessoren nicht größer als die Dimension der Matrix ist und die Parallelisierung auf Ebene der Spaltenmodifikation erfolgt. Ein Prozessor speichert also eine oder mehrere Spalten von $\mathbf{L}$ und modifiziert diese gemäß des Faktorisierungsalgorithmus. Bei Ausnutzung der Besetzungsstruktur ist es natürlich anzustreben, daß Spalten, die auf unterschiedlichen Prozessoren bearbeitet werden, möglichst unabhängig sind. Die Unabhängigkeit von Spalten wollen wir nun untersuchen und definieren dazu eine Funktion

$$parent(j) = \begin{cases} min\{i \in Nz(\mathbf{L}_{*j})\} & \text{falls } Nz(\mathbf{L}_{*j}) \neq \emptyset \\ \\ j & \text{sonst} \end{cases}$$

Offensichtlich gilt $j \leq parent(j)$. Betrachtet man das Vorgehen bei der Cholesky Faktorisierung, so stellt man fast, daß $i = parent(j)$ bedingt, daß Spalte j zur Modifikation von Spalte i

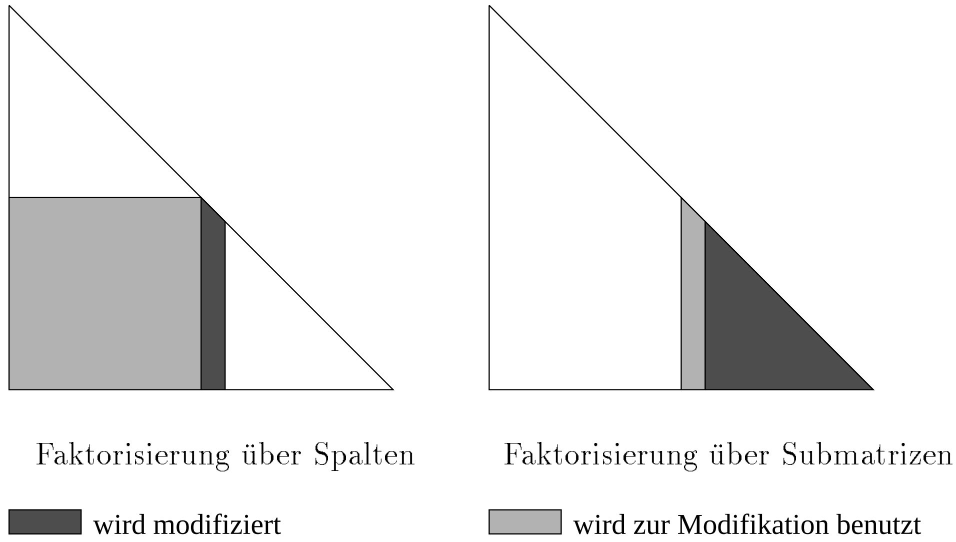


Abbildung 5.2: Vorgehen bei der Cholesky Faktorisierung.

verwendet wird. Dies bedeutet auch, daß die i-te Spalte erst nach der j-ten Spalte vollständig berechnet werden kann. Mit Hilfe der Funktion *parent* kann ein Baum $T(\mathbf{A})$ definiert werden, dessen Knoten von 1 bis n numeriert werden und dessen gerichtete Kanten durch die Funktion *parent* definiert werden. D.h. es existiert eine Kante von i nach j wenn $i = parent(j)$. Der resultierende Graph hat nur Knoten mit Eingangsgrad ≤ 1. Für irreduzible Matrizen kann man leicht zeigen, daß der entstandene Graph wirklich ein einziger Baum mit Wurzel n ist, ohne diese Restriktion kann i.a. ein Wald entstehen.

$T(\mathbf{L})$ beschreibt genau die Abhängigkeiten, die bei der Cholesky Faktorisierung auftreten. Unterbäume können vollkommen unabhängig voneinander bearbeitet werden, da die zugehörigen Spalten sich nicht gegenseitig beeinflußen. Damit gibt die Tiefe von $T(\mathbf{A})$ eine Abschätzung für die Laufzeit bei voller Ausnutzung der Parallelität an. Die Breite des Baumes beschreibt die maximal erreichbare Parallelität. Klarerweise hängt $T(\mathbf{A})$ von der Anordnung der Elemente in der Matrix ab. Abb. 5.3 zeigt vier verschiedene Anordnungen einer einfachen Beispielmatrix $\mathbf{A}$, die daraus resultierenden faktorisierten Matrizen $\mathbf{L}$ und die Eliminationsbäume $T(\mathbf{A})$.

Die Ausgangsmatrix ist eine einfache tridiagonale Matrix. Die daraus resultierende faktorisierte Matrix entsteht ohne zusätzliches Fill In, was natürlich optimal ist. Der zugehörige Eliminationsbaum ist eine einfache Kette. Damit muß die Faktorisierung vollständig sequentiell durchgeführt werden. In der zweiten Anordnung wurden die Variablen mit einem Algorithmus umgeordnet, der auf einer Auswahl bzgl. des minimalen Knotenausgangsgrades beruht. Die Idee ist es, immer den oder einen Knoten mit der geringsten Zahl von Nachbarn zu wählen. Wird Knoten i gewählt, so wird jede hereinkommende Kante zu jedem Nachfolgeknoten verlängert. D.h. für $\mathbf{A}(j,i) \neq 0$ und $\mathbf{A}(i,k) \neq 0$ wird bei Auswahl von i eine Kante von j nach k eingeführt. Auf diese Weise werden Zustände konsekutiv numeriert. Der Algorithmus beschreibt eine Heuristik zur Minimierung von Fill In. Ist der Knoten mit der minimalen Zahl von Nachbarn nicht eindeutig bestimmt, so wird ein Knoten unter den möglichen Kandidaten ausgewählt. Diese Auswahl bestimmt natürlich die Struktur der resultierenden Matrix. Im Beispiel wurde eine bzgl. der Tiefe des Baumes $T(\mathbf{A})$ optimale Auswahl getroffen. Der resultierende Baum hat nur noch eine Tiefe von 4 und kann mit zwei Prozessoren bearbeitet werden. Jeder Prozessor berechnet einen Teilbaum und ein Prozessor behandelt zusätzlich Spalte 7, wozu er die Ergebnisse des anderen Prozessors benötigt. Die Matrixfaktorisierung erzeugt auch für diese Variablenanordnung kein Fill In. Die dritte Anordnung resultiert aus einem weiteren heuristischem Algorithmus, der zur Minimierung des Fill In entwickelt wurde. Die Idee ist es Zustände

Ordnung 1

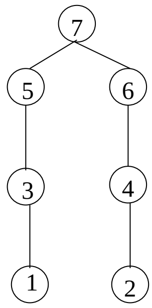
$$A = \begin{bmatrix} X & X & & & & & \\ X & X & X & & & & \\ & X & X & X & & & \\ & & X & X & X & & \\ & & & X & X & X & \\ & & & & X & X & X \\ & & & & & X & X \end{bmatrix} \qquad L = \begin{bmatrix} X & & & & & & \\ X & X & & & & & \\ & X & X & & & & \\ & & X & X & & & \\ & & & X & X & & \\ & & & & X & X & \\ & & & & & X & X \end{bmatrix}$$

T(A)

Ordnung 2

$$A = \begin{bmatrix} X & & X & & & & \\ & X & & X & & & \\ X & & X & & X & & \\ & X & & X & & X & \\ & & X & & X & & X \\ & & & X & & X & X \\ & & & & X & X & X \end{bmatrix} \qquad L = \begin{bmatrix} X & & & & & & \\ & X & & & & & \\ X & & X & & & & \\ & X & & X & & & \\ & & X & & X & & \\ & & & X & & X & \\ & & & & X & X & X \end{bmatrix}$$

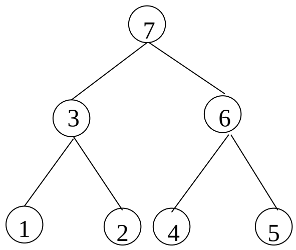
T(A)



Ordnung 3

$$A = \begin{bmatrix} X & & X & & & & \\ & X & X & & & & X \\ X & X & X & & & & \\ & & & X & & X & \\ & & & & X & X & X \\ & & & X & X & X & \\ & X & & & X & & X \end{bmatrix} \qquad L = \begin{bmatrix} X & & & & & & \\ & X & & & & & \\ X & X & X & & & & \\ & & & X & & & \\ & & & & X & & \\ & & & X & X & X & \\ & X & O & & X & O & X \end{bmatrix}$$

T(A)



Ordnung 4

$$A = \begin{bmatrix} X & & & & X & & \\ & X & & & & X & \\ & & X & & X & & X \\ & & & X & & X & X \\ X & & X & & X & & \\ & X & & X & & X & \\ & & X & X & & & X \end{bmatrix} \qquad L = \begin{bmatrix} X & & & & & & \\ & X & & & & & \\ & & X & & & & \\ & & & X & & & \\ X & & X & & X & & \\ & X & & X & & X & \\ & & X & X & O & O & X \end{bmatrix}$$

T(A)

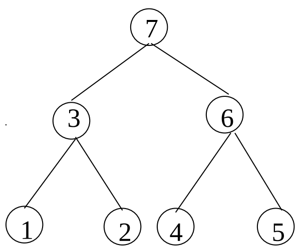


Abbildung 5.3: Faktorisierte Matrizen und Eliminationsbäume für eine Beispielmatrix und verschiedene Anordnungen.

zu finden, so daß die Matrix in irreduzible Teilmatrizen zerfällt, wenn der Zustand und alle zugehörigen Kanten entfernt werden. Mit den resultierenden irreduziblen Untermatrizen wird anschließend genauso verfahren. In unserem Beispiel ist es relativ einfach solche Zustände zu finden, für allgemeine Matrizen müssen sie aber nicht existieren. Der aus der Variablenanordnung resultierende Eliminationsbaum hat nur eine Tiefe von 3. Bei der Faktorisierung entsteht aber ein Fill In von 2 neuen Elementen. Letzteres zeigt, daß ein Eliminationsbaum minimaler Tiefe nicht gleichzeitig für eine Minimierung des Fill In sorgt. Bei großen Systemen bedingen niedrige Eliminationsbäume aber tendenziell weniger Fill In als hohe. Die letzte Variablenordnung ist aus einer gerade-ungerade Anordnung entstanden. Das Vorgehen funktioniert wie folgt: Zuerst werden alle Variablen mit geradem Index in aufsteigender Reihenfolge numeriert, dann alle Variablen mit ungeradem Index in aufsteigender Reihenfolge numeriert. Diese Idee wird auf die so entstanden Teilsequenzen angewendet bis jede Sequenz nur noch aus einem Element besteht. Auch aus dieser Variablenordnung entsteht ein Eliminationsbaum der Tiefe 3 und ein Fill In von 2 Elementen. Die beiden letzteren Eliminationsbäume beschreiben eine Struktur, die mit 4 Prozessoren in 3 Schritten analysiert werden kann.

Die Konstruktion von Eliminationsbäumen und auch die Heuristiken zur Variablenumordnung können parallelisiert werden und erlauben eine strukturelle Analyse zur Optimierung des parallelen Ablauf in Abhängigkeit von der Besetzungsstruktur der Matrix und der zur Verfügung stehenden Rechnerarchitektur.

Wir wollen jetzt die Parallelisierung des Cholesky Algorithmus betrachten, ausgehend von einem gegebenen Eliminationsbaum. Die maximal sinnvoll einsetzbare Anzahl von Prozessoren ist durch die Zahl der Blätter im Eliminationsbaum gegeben. Der Eliminationsbaum wird nun ebenenweise in Teilbäume zerlegt und jeder Teilbaum einem Teilsystem des Multiprozessors zugewiesen. Unter einem Teilsystem eines Multiprozessors versteht man einen Unterwürfel eines Hyperwürfels, ein Untergitter bei einem Gitter oder ein Unterarray eines linearen Arrays. Die einzelnen Teilbäume werden in der Unterstruktur genauso behandelt wie der gesamte Baum in der vollständigen Topologie. Bei der Prozessorzuteilung ist darauf zu achten, die Kommunikationswege möglichst kurz zu halten. Dies ist unter Umständen schwierig, wenn komplexere Eliminationsbäume auf linearen Arrays oder Gittern behandelt werden sollen.

Auf jedem einzelnen Prozessor läuft die Cholesky Faktorisierung für eine Teilmenge der Spalten ab. Sei dazu $map(Nz(\mathbf{L}_{*j}))$ die Menge der Prozessoren, die Spalten aus $Nz(\mathbf{L}_{*j})$ bearbeiten. Weiterhin sei $mycols$ die Menge der Spalten, die auf dem lokalen Prozessor bearbeitet wird. Betrachten wir zuerst die Cholesky Faktorisierung über Submatrizen.

Submatrix_Cholesky_auf_Prozessor_p ;
    for all $j \in mycols$ *do*
        if j *keine weiteren Spalten benötigt then*
            $cdiv(j)$;
            sende $\mathbf{L}_{*j}$ *an alle* $p \in map(Nz(\mathbf{L}_{*j}))$;
            $mycols = mycols \setminus \{j\}$;
        fi
    od
    while $mycols \neq \emptyset$ *do*
        receive $\mathbf{L}_{*k}$;
        for $j \in mycols \cap Nz(\mathbf{L}_{*k})$ *do*
            $cmod(j,k)$;
            if j *keine weiteren Spalten benötigt then*
                $cdiv(j)$;
                sende $\mathbf{L}_{*j}$ *an alle* $p \in map(Nz(\mathbf{L}_{*j}))$;

$mycols = mycols \setminus \{j\}$;
fi
od
end ;

Im Algorithmus wird beim Versenden von Spalte j an den Prozessor, der Spalte $k \in Nz(\mathbf{L}_{*j})$ beinhaltet, der Fall $k \in mycols$ und $k \notin mycols$ nicht unterschieden. Im ersten Fall ist es natürlich nicht nötig, die Daten zu verschicken, da sie lokal benötigt werden. Wir nehmen an, um die Fallunterscheidung in der Beschreibung des Algorithmus zu vermeiden, daß dies automatisch festgestellt wird und die Daten in den Empfangspuffer des lokalen Prozessors geschrieben werden. Im zweiten Schritt braucht dann nicht unterschieden werden, ob eine empfangene Spalte von einem anderen Prozessor geschickt wurde oder lokal gespeichert wurde.

Der Algorithmus berechnet asynchron lokale Ergebnisse und versendet Spalten, die vollständig berechnet wurden, an andere Prozessoren. Ein Nachteil des Verfahrens ist, daß relativ hohe Kommunikationskosten anfallen können, da viele Spalten verschickt werden. Der Kommunikationsaufwand kann verringert werden, wenn lokale Berechnungen erst zu Ende geführt, bevor Ergebnisse verschickt werden. Dies führt zu einer Variante der spaltenweisen Faktorisierung. Diese wird mit Hilfe eines update Vektors $\mathbf{u}$ wie folgt realisiert.

Spalten_Cholesky_auf_Prozessor_p ;
for $j = 1$ *to* n *do*
if $j \in mycols$ *or* $mycols \cap Nz(\mathbf{L}_{j*}) \neq \emptyset$ *then*
$\mathbf{u} = \mathbf{0}$;
for $k \in mycols \cap Nz(\mathbf{L}_{j*})$ *do*
$\mathbf{u} = \mathbf{u} + L_{j,k}\mathbf{L}_{*k}$;
if $j \in mycols$ *then*
$\mathbf{L}_{*j} = \mathbf{L}_{*j} - \mathbf{u}$;
while weitere Spalten für j *benötigt werden do*
empfange $\mathbf{u}_j$ *für Spalte* j ;
$\mathbf{L}_{*j} = \mathbf{L}_{*j} - \mathbf{u}_j$;
od ;
$cdiv(j)$;
else
sende $\mathbf{u}$ *als* $\mathbf{u}_j$ *an Prozessor* $p \in map(j)$;
fi
fi
od
end ;

Im Unterschied zum ersten Algorithmus werden nun möglichst viele Berechnungen lokal auf den Prozessoren ausgeführt. Die erste *for* Schleife kann ablaufen, ohne daß Information von anderen Prozessoren benötigt wird. Erst in der zweiten Phase werden die Informationen anderer Prozessoren benötigt und eingearbeitet. Damit ist auch eine gute Überlagerung von Berechnung und Kommunikation gegeben. Diese wird noch verbessert, wenn in der ersten Schleife zuerst die Vektoren $\mathbf{u}$ für andere Prozessoren berechnet werden und erst anschließend die Werte der lokalen Spalten berechnet werden. Auf diese Weise kann während der Berechnung der lokalen Spalten die Information zu anderen Prozessoren übertragen werden.

Das Beispiel der Cholesky Faktorisierung zeigt, daß dünn besetzte Matrizen mit allgemeiner Besetzungsstruktur sich durchaus für eine effiziente parallele Implementierung verschiedener Algorithmen eignen und durch Ausnutzung der Besetzungsstruktur eine effiziente Parallelisierung möglich ist. Der vorgestellte Ansatz läßt sich im Prinzip auch auf allgemeinere Matrizen und die LU- bzw. QR-Faktorisierung übertragen. So ist die symbolische Faktorisierung, d.h. die Bestimmung des Eliminationsbaums, auch für allgemeine Matrizen möglich und gibt eine "Obergrenze" möglicher Abhängigkeiten an. Nicht alle diese Abhängigkeiten müssen tatsächlich existieren, da Elemente während der Berechnung zu Null werden können. Problematisch ist in einem solchen Fall die Verwendung von Pivotschritten, die in allgemeinen Matrizen notwendig ist. In einem Pivotschritt werden Zeilen oder Spalten der Matrix vertauscht, dadurch bedingt ändert sich die Struktur und damit auch der Eliminationsbaum, so daß die symbolische a priori Faktorisierung wertlos wird. Insofern besteht bei der Parallelisierung direkter Lösungsverfahren für allgemeine dünn besetzte Matrizen noch Bedarf an effizienten Verfahren. Mir sind zumindest kaum brauchbare Implementierungen bekannt. Das hier vorgestellte Vorgehen, welches sich an einzelnen Spalten orientiert, kann auch für Gruppen von Spalten definiert werden, um so die Granularität zu erhöhen.

5.2 Iterative Verfahren

Reale Probleme, die die Lösung linearer Gleichungssysteme erfordern, haben oftmals eine Dimension, die im Bereich von mehreren Hunderttausend oder sogar Millionen Variablen liegt. Gleichzeitig sind die Übergangsmatrizen spärlich besetzt, so daß oftmals deutlich weniger als 1% der Matrixelemente ungleich 0 sind. Für diese Problemklasse sind die vorgestellten direkten Verfahren nicht geeignet, da durch das Entstehen von "Fill In" der Speicherplatz auch massiv paralleler Rechner nicht ausreicht, um mehrere Millionen oder Milliarden Elemente einer Matrix zu speichern, die während der Faktorisierung einer ursprünglich spärlich besetzte Matrix mit Dimension $1.0e+5$ oder $1.0e+6$ entstehen können. Außerdem ist in solchen Fällen natürlich nicht daran zu denken, Prozessorzahlen im Bereich n^3 oder gar n^4 zur Lösung einzusetzen. Vielmehr wird die Prozessorzahl, auch bei den nächsten Generationen von Parallelrechnern, deutlich unter der Zahl der Variablen liegen. Für die genannte Problemklasse sind iterative Verfahren besser als direkte Verfahren geeignet. Iterative Verfahren berechnen ausgehend von einem Startvektor iterativ Approximationen des Lösungsvektors. Der entscheidende Vorteil ist, daß die Matrix $\mathbf{A}$ nicht oder nur geringfügig für die Iteration verändert wird. Damit bleibt eine spärlich besetzte Matrix auch spärlich besetzt. Gleichzeitig wird mit iterativen Verfahren im allgemeinen nicht die exakte Lösung, sondern nur eine "genügend genaue" Approximation berechnet. A priori ist nicht bekannt, wann diese erreicht ist, d.h. im Gegensatz zu direkten Verfahren ist die Zahl der Schritte bis zum Erreichen der (approximativen) Lösung nicht bestimmbar, sondern hängt vom Problem und der geforderten bzw. geschätzten Genauigkeit ab. Wir werden deshalb bei der Aufwandsanalyse iterativer Verfahren den Aufwand für eine Iteration ermitteln und die sequentielle mit der parallelen Implementierung vergleichen. Da die Zahl der Iterationen in sequentiellen und parallelen Implementierungen identisch ist (zumindest bei synchronen Verfahren), ist dieser Vergleich durchaus aussagekräftig. Der Nachteil, daß iterative Verfahren nur approximative, direkte Verfahren aber exakte Lösungen berechnen, ist in vielen Fällen mehr theoretischer Natur. So sind iterative Verfahren in der Praxis unempfindlicher gegenüber numerischen Problemen. Direkte Verfahren, insbesondere die oft verwendete Gauß Elimination, liefern bei schlecht konditionierten Problemen dagegen sehr ungenaue Resultate bedingt durch Rundungsfehler.

Es gibt zahlreiche unterschiedliche iterative Verfahren, von denen nur eine kleine Auswahl hier vorgestellt werden kann. Im ersten Abschnitt werden einige Basisoperationen eingeführt, die in vielen iterativen Verfahren zu finden sind. Der nächste Abschnitt ist klassischen und wohlbekannten Methoden wie dem Jacobi- oder Gauß-Seidel Verfahren und deren Relaxationsvarianten gewidmet. Im darauf folgenden Abschnitt untersuchen wir eine generelles Problem iterativer Verfahren, nämlich die Frage nach der Konvergenz eines Verfahrens und der Abschätzung der Genauigkeit der erreichten Approximation. Danach wird das Verfahren des konjugierten Gradienten als ein Beispiel eines moderneren, nicht-stationären Verfahrens vorgestellt.

5.2.1 Basisoperationen für iterative Verfahren

Fast alle iterative Methoden basieren auf den folgenden Operationen:

- Vektor-Skalar-Produkte
- Additionen von Vektoren
- Produkte von Vektoren
- Vektor-Matrix-Produkte

Vektor-Skalar-Produkte und Vektor-Additionen sind einfach zu realisieren und auch einfach zu parallelisieren. Die Parallelisierung von Vektor-Produkten und Vektor-Matrix-Produkten wurde bereits im Kapitel über numerische Basisoperationen mit und ohne Berücksichtigung der Kommunikationszeit behandelt. Wir können auf die dort vorgestellten Ergebnisse zurückgreifen, da auch sichergestellt wurde, daß nach Berechnung eines Vektor- oder Vektor-Matrix-Produkts die Resultate auf den Prozessoren verfügbar sind, wo sie für die nachfolgende Iteration benötigt werden. Insofern wird die folgende Parallelisierung iterativer Methoden teilweise daraus bestehen, die bereits vorgestellten numerischen Basisalgorithmen auf geeignete Art und Weise zu kombinieren.

5.2.2 Klassische iterative Verfahren

Wir beginnen mit der Vorstellung sogenannter klassischer iterativer Verfahren. Diese Verfahren sind als sequentielle Algorithmen schon lange bekannt und werden in der Praxis breit eingesetzt. Sie werden auch als stationäre Methoden bezeichnet, da die Iterationsmatrix während der Iteration konstant bleibt. In Abschnitt 5.2.3 setzen wir uns mit der Konvergenzanalyse der Verfahren auseinander. Hier betrachten wir nur das prinzipielle Vorgehen unter der Annahme, daß das Verfahren gegen die gesuchte Lösung konvergiert. Wir beginnen mit der Vorstellung der einzelnen Verfahren und betrachten erst danach die Parallelisierung.

Es liegt das schon bekannte Problem $\mathbf{Ax} = \mathbf{b}$ vor. $\mathbf{A}$ ist eine reelle, invertierbare $n \times n$ Matrix, $\mathbf{x}$ und $\mathbf{b}$ sind n-dimensionale Vektoren, wobei $\mathbf{b}$ gegeben und $\mathbf{x}$ gesucht ist. Wir nehmen $A_{i,i} \neq 0$ an, dies kann immer durch Umordnung der Matrix erreicht werden.

Wir definieren die folgende Zerlegung der Matrix $\mathbf{A} = \mathbf{L}+\mathbf{D}+\mathbf{U}$. $\mathbf{D}$ ist eine Diagonalmatrix, die in der Diagonalen die Diagonalelement $A_{i,i}$ beinhaltet. $\mathbf{L}$ umfaßt alle Elemente von $\mathbf{A}$, die unterhalb der Hauptdiagonalen liegen, d.h. $A_{j,i}$ mit $j > i$. $\mathbf{U}$ besteht aus allen Elemente von $\mathbf{A}$, die oberhalb der Hauptdiagonalen liegen, d.h. $A_{j,i}$ mit $j < i$. Unter der Voraussetzung, daß alle Diagonalelemente von $\mathbf{A}$ ungleich 0 sind, ist $\mathbf{D}$ invertierbar.

Jacobi Verfahren

$\mathbf{Ax} = \mathbf{b}$ ergibt für ein einzelnes x_i die folgende Gleichung.

$$x_i = -\frac{1}{A_{i,i}}[\sum_{j \neq i} A_{i,j} x_j - b_i] \tag{5.17}$$

Wären alle Komponenten x_j mit $j \neq i$ bekannt, so könnten wir die Lösung direkt berechnen. Sind nur Approximationen für x_j $j \neq i$ bekannt, so kann mit der obigen Gleichung eine Approximation für x_i berechnet werden. Dieses Vorgehen führt direkt zum Jacobi Verfahren. Ausgehend von einem Startvektor $\mathbf{x}^{(0)}$ berechnen wir $\mathbf{x}^{(t+1)}$ nach folgendem Iterationsschema.

$$x_i^{(t+1)} = -\frac{1}{A_{i,i}}[\sum_{j \neq i} A_{i,j} x_j^{(t)} - b_i] \tag{5.18}$$

Wenn das Verfahren konvergiert, so erzeugt es eine Sequenz von Vektoren $\mathbf{x}^{(t)}$, die gegen die gesuchte Lösung $\mathbf{x}$ konvergieren.

Gleichung (5.18) wird in der Literatur häufig unter zuhilfenahme des obigen Matrixsplittings folgendermaßen notiert:

$$\mathbf{x}^{(t+1)} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(t)} + \mathbf{D}^{-1}\mathbf{b} \tag{5.19}$$

Eine Erweiterung des Jacobi Verfahrens ist das JOR Verfahren (Jacobi Overrelaxation), bei dem der neue Iterationsvektor aus einer gewichteten Summe des alten Vektors und des nach (5.18) berechnete Vektor entsteht. Es liegt folgende Iterationsvorschrift vor.

$$x_i^{(t+1)} = (1-\gamma)x_i^{(t)} - \frac{\gamma}{A_{i,i}}[\sum_{j \neq i} A_{i,j} x_j^{(t)} - b_i] \tag{5.20}$$

JOR läßt sich auch als

$$\mathbf{x}^{(t+1)} = [(1-\gamma)\mathbf{I} - \gamma\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})]\mathbf{x}^{(t)} + \gamma\mathbf{D}^{-1}\mathbf{b} \tag{5.21}$$

aufschreiben. Die neue Approximation entsteht also durch Extra- oder Interpolation aus der alten Lösung und dem mit dem Jacobi Verfahren berechneten Wert. $\gamma = 1$ liefert die normale Jacobi Iteration, $0 < \gamma < 1$ liefert eine gedämpfte Iteration und $1 < \gamma < 2$ eine "Überrelaxation". Durch eine geschickte Wahl von γ, die allerdings nicht einfach ist, kann die Konvergenzgeschwindigkeit des Verfahrens deutlich erhöht werden.

Jacobi und JOR folgen einem allgemeinen Iterationsschema $\mathbf{x}^{(t+1)} = \mathbf{Mx}^{(t)} + \mathbf{Gb}$ wobei für Jacobi $\mathbf{M} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ und $\mathbf{G} = \mathbf{D}^{-1}$, während für JOR $\mathbf{M} = [(1-\gamma)\mathbf{I} - \gamma\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})]$ und $\mathbf{G} = \gamma\mathbf{D}^{-1}$.

Betrachten wir nun die Parallelisierbarkeit der Verfahren. Der wesentliche Berechnungsaufwand besteht in der Vektor-Matrix-Multiplikation, diese wurde schon ausführlich für die verschiedenen Architekturen analysiert. Jacobi und JOR Verfahren können direkt mittels der vorgestellten Vektor-Matrix-Multiplikationsalgorithmen auf den Topologien implementiert werden. Stehen n Prozessoren zur Verfügung, so berechnet Prozessor i $x_i(t)$ für jede Iteration t und speichert dazu die i-te Zeile der Matrix $\mathbf{A}$. Der Aufwand für eine Iteration liegt damit in $\Theta(n)$. Daran anschließend müssen die berechneten Werte an alle anderen Prozessoren verteilt werden. Dies entspricht einem allgemeinen Broadcast mit Aufwand in $O(n)$ auf einem linearen Array und $O(n/\log n)$ auf einem Hyperwürfel. Die Verwendung weiterer Prozessoren ist nur in Hyperwürfeln sinnvoll. In einem Hyperwürfel mit $p = n^2$ Prozessoren speichert eine Prozessor ein Element der Matrix $\mathbf{A}$. Die Iteration entspricht der Vektor-Matrix-Multiplikation auf Hyperwürfeln in $O(\log n)$, wie in Abschnitt 2.5.4 vorgestellt. In einer Topologie mit n^2 Prozessoren entspricht die Verteilung der berechneten Werte nicht mehr einem allgemeinen Broadcast, da x_j nur noch von Prozessoren benötigt wird, die Elemente $A_{i,j}$ speichern. In realistischen Fällen ist aber davon auszugehen, daß die Zahl der Gleichungen/Variablen oft deutlich größer als die Prozessorzahl ist. Damit ist eine Implementierung auf einem Multiprozessor mit n^2 Prozessoren utopisch und wird es wohl auch in absehbarer Zukunft bleiben. Wir haben hier den Fall der zeilenweisen Speicherung der Matrix $\mathbf{A}$ untersucht, eine spaltenweise Speicherung ist ebenfalls möglich (wie bereits bei der Vektor-Matrix-Multiplikation auf linearen Arrays beschrieben), unterscheidet sich vom Aufwand her nicht, solange dicht besetzte Matrizen verwendet werden.

Das bisher beschriebene Vorgehen basiert auf einer dichten Matrix, d.h. alle Elemente werden gespeichert und ein Großteil der Elemente ist ungleich 0. Ist die Matrix spärlich besetzt (die meisten Elemente sind 0), so werden nur die Nichtnullelemente gespeichert. Dies bedeutet gleichzeitig, daß Prozessor i zur Berechnung von $x_i^{(t)}$ nur wenige $x_j^{(t-1)}$ benötigt. Die Abhängigkeiten zwischen den einzelnen Variablen sind im schon vorgestellten Abhängigkeitsgraphen dargestellt. Gelingt es den Abhängigkeitsgraphen des Gleichungssystems in die Topologie der Rechnerarchitektur einzubetten, so ist eine sehr effiziente Implementierung möglich, da nur benachbarte Prozessoren kommunizieren müssen. In Spezialfällen, mit sehr regelmäßigen Abhängigkeiten, ist eine solche Einbettung einfach zu finden. Für den allgemeinen Fall ist die Einbettung dagegen schwierig.

Gauß-Seidel Verfahren

Betrachten wir den rein sequentiellen Ablauf des Jacobi Verfahrens, so sind zum Zeitpunkt der Berechnung von $x_i^{(t+1)}$ die Komponenten $x_j^{(t+1)}$ für $j < i$ bereits berechnet und verfügbar. Eine naheliegende Überlegung ist es, diese Komponenten direkt in die Iteration einzubeziehen. Dies führt zum Gauß-Seidel Verfahren, welches manchmal auch als Relaxation bezeichnet wird. Die Iterationsvorschrift ist in der folgenden Gleichung angegeben.

$$x_i^{(t+1)} = -\frac{1}{A_{i,i}}[\sum_{j<i} A_{i,j}x_j^{(t+1)} + \sum_{j>i} A_{i,j}x_j^{(t)} - b_i] \tag{5.22}$$

bzw in Matrixnotation

$$\mathbf{x}^{(t+1)} = -\mathbf{D}^{-1}[\mathbf{L}\mathbf{x}^{(t+1)} + \mathbf{U}\mathbf{x}^{(t)} - \mathbf{b}].$$

Für ein Gleichungssystem der Ordnung n gibt es im Prinzip $n!$ verschiedene Gauß-Seidel Algorithmen, je nachdem in welcher Reihenfolge die Variablen angeordnet bzw. neu berechnet werden. Interessanterweise hängt die Konvergenzgeschwindigkeit der Verfahren von der Variablenanordnung ab. Es gibt aber kaum einsetzbare Verfahren, die bestimmen welche Variablenanordnung gewählt werden soll, um die Konvergenzgeschwindigkeit zu maximieren. Ein heuristischer Ansatz besteht darin, zwischen der Reihenfolge $1, 2, \ldots, n$ und $n, n-1, \ldots, 1$ zu wechseln und zwar immer dann, wenn man meint, eine sich verlangsamende Konvergenz zu beobachten (siehe auch Abschnitt 5.2.3). Inwieweit dieses Vorgehen etwas bringt, ist schwer zu beurteilen. Auch beim Gauß-Seidel Verfahren gilt wieder, wenn das Verfahren konvergiert, so konvergiert es gegen die gesuchte Lösung.

Ähnlich wie das Jacobi Verfahren kann auch das Gauß-Seidel Verfahren mit Inter- oder Extrapolation zur Berechnung des Lösungsvektors verbunden werden. Dies führt zu folgender Relaxationsvariante des Gauß-Seidel Verfahrens, die als SOR (successive overrelaxation) bezeichnet wird.

$$x_i^{(t+1)} = (1-\gamma)x_i^{(t)} - \frac{\gamma}{A_{i,i}}[\sum_{j<i} A_{i,j}x_j^{(t+1)} + \sum_{j>i} A_{i,j}x_j^{(t)} - b_i] \tag{5.23}$$

Die zugehörige Matrixnotation

$$\mathbf{x}^{(t+1)} = (1-\gamma)\mathbf{x}^{(t)} - \gamma\mathbf{D}^{-1}[\mathbf{L}\mathbf{x}^{(t+1)} + \mathbf{U}\mathbf{x}^{(t)} - \mathbf{b}].$$

läßt sich wie bereits Jacobi und JOR in das allgemeine Iterationsschema $\mathbf{x}^{(t+1)} = \mathbf{M}\mathbf{x}^{(t)} + \mathbf{G}\mathbf{b}$ umformen:

$$\mathbf{x}^{(t+1)} = (\mathbf{I} + \gamma\mathbf{D}^{-1}\mathbf{L})^{-1}[(1-\gamma)\mathbf{I} - \gamma\mathbf{D}^{-1}\mathbf{U}]\mathbf{x}^{(t)} + \gamma(\mathbf{I} + \gamma\mathbf{D}^{-1}\mathbf{L})^{-1}\mathbf{D}^{-1}\mathbf{b} \tag{5.24}$$

Obwohl damit die bisherigen Iterationsverfahren formal recht ähnlich sind, ist die Parallelisierung des Gauß-Seidel oder SOR Verfahrens wesentlich komplizierter als die Parallelisierung des Jacobi oder JOR Verfahrens. Dies ist insbesondere ärgerlich, da die Konvergenzgeschwindigkeit dieser Verfahren oftmals deutlich schneller als die Konvergenzgeschwindigkeit des Jacobi oder JOR Verfahrens ist.

Bei einer dicht besetzten Matrix sind die genannten Verfahren im Prinzip vollkommen sequentiell, $x_i^{(t)}$ kann erst nach allen $x_j^{(t)}$ $(j < i)$ berechnet werden. Es kann allerdings ausgenutzt werden, daß mit der Berechnung schon begonnen werden kann, bevor alle Werte $x_j^{(t)}$ $(j < i)$ zugreifbar sind. Diese Überlegung führt direkt zu einem Algorithmus auf einem linearen Array mit n Prozessoren, der kurz vorgestellt werden soll. Prozessor i speichert die i-te Zeile von $\mathbf{A}$, $\mathbf{b}_i$ und den Vektor $\mathbf{x}^{(t)}$. Jeder Prozessor beginnt mit der Berechnung von $\sum_{j>i} A_{i,j}x_j^{(t)} - b_i$. Der erste Prozessor kann $x_1^{(t+1)}$ berechnen und verbreitet den Wert per allgemeinem Broadcast an alle anderen Prozessoren. Bei Empfang eines neuen Wertes wird dieser direkt zur Berechnung verwendet. So wird verfahren bis alle Elemente von $\mathbf{x}^{(t+1)}$ berechnet sind. Durch diesen Trick läßt sich eine gewisse Parallelität erreichen, aber unter realen Gegebenheiten ist diese eher unbefriedigend. Nachdem der erste Prozessor $x_1^{(t+1)}$ berechnet hat, hat der letzte Prozessor noch keinen Rechenschritt ausgeführt. Dies bedeutet, daß Prozessoren deutlich mehr als 50% ihrer Zeit mit Warten verbringen. Selbst wenn mehrere Variablen auf einem Prozessor berechnet werden und Kommunikation und Berechnung weiter überlagert werden, ist es schwierig des Gauß-Seidel Verfahren für dicht besetzte Matrizen effizient auf Systemen mit vielen Prozessoren zu realisieren.

Ist die Matrix dünn besetzt, so kann durch Färbung des Abhängigkeitsgraphen eine optimale Parallelität erreicht werden. Das Problem der Färbung von Abhängigkeitsgraphen und die darauf basierende Parallelisierung wurden bereits vorgestellt. Auch wenn das theoretische Problem einer optimalen Färbung komplex ist, sind viele dünn besetzte Matrizen sehr regulär strukturiert, so daß die Färbung recht einfach ist. Beispiele für regulär strukturierte Matrizen sind Gitterstrukturen oder (block)tridiagonale Matrizen.

Bisher haben wir nur die Zeit für eine Iteration untersucht. Die Laufzeit eines iterativen Verfahrens resultiert aber aus mehreren Iterationen, deren Anzahl von der Konvergenz abhängt. Es stellen sich nun zwei Fragen, nämlich wieviele Iterationen benötigt man und wann soll man das Verfahren abbrechen. Ersteres ist problemabhängig und a priori nicht zu sagen. Abbruchkriterien müssen zwangsläufig aus der Schätzung des noch vorhandenen Fehlers resultieren. Wir werden uns damit im Rahmen der Konvergenzanalyse im nächsten Abschnitt beschäftigen.

5.2.3 Konvergenz klassischer iterativer Verfahren

Beginnen wir das Kapitel über die Konvergenzanalyse mit einem sehr einfachen Beispiel und betrachten dazu die beiden folgenden Gleichungssysteme.

$$\mathbf{A}\mathbf{x} = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad \mathbf{B}\mathbf{x} = \begin{pmatrix} -1 & 2 \\ 2 & -1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

	Jacobi $\mathbf{Ax}$		Jacobi $\mathbf{Bx}$		GS $\mathbf{Ax}$		GS $\mathbf{Bx}$	
t	$x_1(t)$	$x_2(t)$	$x_1(t)$	$x_2(t)$	$x_1(t)$	$x_2(t)$	$x_1(t)$	$x_2(t)$
0	1.000e+0	1.000e+0	1.00e+0	1.00e+0	1.000e+0	1.000e+0	1.00e+0	1.00e+0
1	5.000e-1	5.000e-1	2.00e+0	2.00e+0	5.000e-1	2.500e-1	2.00e+0	4.00e+0
2	2.500e-1	2.500e-1	4.00e+0	4.00e+0	1.250e-1	6.250e-2	8.00e+0	1.60e+1
3	1.250e-1	1.250e-1	8.00e+0	8.00e+0	3.125e-2	1.563e-2	3.20e+1	6.40e+1
4	6.250e-2	6.250e-2	1.60e+1	1.60e+1	7.813e-3	3.906e-3	1.28e+2	2.56e+2
5	3.125e-2	3.125e-2	3.20e+1	3.20e+1	1.953e-3	9.766e-4	5.12e+2	1.02e+3

Tabelle 5.2: Konvergenz des Jacobi und Gauß-Seidel Verfahrens

In beiden Fällen ist offensichtlich $x^* = (0,0)^T$ eine Lösung. Sei $x(0) = (1,1)^T$ der initiale Vektor, die Werte der Iterationsvektoren $x(1),\ldots,x(5)$ für das Jacobi- und das Gauß-Seidel Verfahren sind sind in Tabelle 5.2 angegeben. Überraschenderweise konvergieren beide Verfahren bei Anwendung auf das System $\mathbf{Ax}$ gegen die Lösung x^*, während beide Verfahren divergieren, wenn sie auf $\mathbf{Bx}$ angewendet werden. Dieses einfache Beispiel zeigt, daß nicht für jedes lösbare Gleichungssystem mit den vorgestellten Verfahren auch die Lösung gefunden wird. Wir wollen in diesem Abschnitt untersuchen, in welchen Fällen die "klassischen" iterativen Verfahren konvergieren.

Die Iterationsgleichungen für Jacobi, JOR, Gauß-Seidel und SOR folgen dem allgemeinen Schema

$$\begin{array}{llrlrl}
 & \mathbf{x}^{(t+1)} = & \mathbf{M} & \mathbf{x}^{(t)}+ & \mathbf{G} & \mathbf{b} \\
\text{Jacobi} & \mathbf{x}^{(t+1)} = & -\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U}) & \mathbf{x}^{(t)}+ & \mathbf{D}^{-1} & \mathbf{b} \\
\text{JOR} & \mathbf{x}^{(t+1)} = & [(1-\gamma)\mathbf{I}-\gamma\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})] & \mathbf{x}^{(t)}+ & \gamma\mathbf{D}^{-1} & \mathbf{b} \\
\text{Gauß-Seidel} & \mathbf{x}^{(t+1)} = & (\mathbf{I}+\mathbf{D}^{-1}\mathbf{L})^{-1}[-\mathbf{D}^{-1}\mathbf{U}] & \mathbf{x}^{(t)}+ & (\mathbf{I}+\mathbf{D}^{-1}\mathbf{L})^{-1}\mathbf{D}^{-1} & \mathbf{b} \\
\text{SOR} & \mathbf{x}^{(t+1)} = & (\mathbf{I}+\gamma\mathbf{D}^{-1}\mathbf{L})^{-1}[(1-\gamma)\mathbf{I}-\gamma\mathbf{D}^{-1}\mathbf{U}] & \mathbf{x}^{(t)}+ & \gamma(\mathbf{I}+\gamma\mathbf{D}^{-1}\mathbf{L})^{-1}\mathbf{D}^{-1} & \mathbf{b}
\end{array}$$

$\mathbf{M}$ ist die Iterationsmatrix. Wir nehmen an, daß $(\mathbf{I}-\mathbf{M})$ invertierbar sei und damit die eindeutige Lösung $\mathbf{x}^* = \mathbf{Mx}^* + \mathbf{Gb}$ existiert. Bei allen bisher beschriebenen Iterationsverfahren folgt aus der Invertierbarkeit von $\mathbf{A}$ die Invertierbarkeit von $(\mathbf{I}-\mathbf{M})$. Sei

$$\mathbf{y}(t) = \mathbf{x}(t) - \mathbf{x}^* = \mathbf{My}(t-1) = \mathbf{M}^{t-1}\mathbf{y}(0)$$

Offensichtlich konvergiert $\mathbf{x}(t)$, wenn $\mathbf{y}(t)$ gegen $\mathbf{0}$ konvergiert. Dies gilt aber gerade, wenn $\mathbf{M}^t$ gegen $\mathbf{0}$ konvergiert. Die Potenzen einer Matrix konvergieren gegen die $\mathbf{0}$ Matrix, wenn der Betrag aller Eigenwerte kleiner 1 ist, d.h. der Spektralradius $\rho(\mathbf{M}) < 1$. Diese relativ informelle Herleitung ist gleichzeitig der Beweis für den folgenden Satz.

Satz 5.3 *Wenn* $(\mathbf{I}-\mathbf{M})$ *invertierbar und* $\mathbf{x}^*$ *der eindeutige Lösungsvektor der Gleichung* $\mathbf{x}^* = \mathbf{Mx}^* + \mathbf{Gb}$ *ist, so konvergiert die Sequenz* $\mathbf{x}(t+1) = \mathbf{Mx}(t) + \mathbf{Gb}$ *für beliebige Startvektoren* $\mathbf{x}(0)$ *gegen* $\mathbf{x}^*$*, wenn* $\rho(\mathbf{M}) < 1$.

Die Konvergenz hängt damit nur von der Matrix $\mathbf{M}$, nicht aber vom Vektor $\mathbf{b}$ ab. Wir nehmen deshalb im folgenden $\mathbf{b} = \mathbf{0}$ an. Die Konvergenzrate eines Verfahrens hängt davon ab, wie schnell $\mathbf{M}^t$ gegen $\mathbf{0}$ konvergiert und ist hauptsächlich von der Größe $\rho(\mathbf{M})$ beeinflußt. Es ist also anzustreben, Iterationsmatrizen mit einem möglichst kleinen Spektralradius zu erzeugen. Der Spektralradius der Iterationsmatrix dient auch zum Vergleich der Konvergenzgeschwindigkeit der unterschiedlichen Verfahren.

Der obige Satz ist eine sehr allgemeine Aussage über die Konvergenz eines Verfahrens. Das Problem ist, daß in der Praxis die Eigenwerte von $\mathbf{M}$ nicht bekannt sind und ihre Bestimmung genauso aufwendig wie die Lösung des ursprünglichen Gleichungssystems ist. Deshalb werden andere Ansätze benötigt, um die Konvergenz eines Verfahrens zu zeigen und die Konvergenzgeschwindigkeit zu bestimmen oder zumindest zu schätzen. Die zentrale Idee der dazu verwendeten Ansätze ist es, mit Normen zu arbeiten, um zu zeigen, daß die Distanz zwischen dem Iterationsvektor und dem Vektor $\mathbf{x}^*$ abnimmt. Aus diesen Untersuchungen werden Klassen von Matrizen ermittelt werden, für die die Konvergenz a priori gezeigt werden kann, ohne Bestimmung der Eigenwerte.

Die gewichtete Maximumsnorm $\| \dots \|_\infty^w$ einer $n \times n$ Matrix $\mathbf{A}$ ist definiert als

$$\|\mathbf{A}\|_\infty^{\mathbf{w}} = \max_i \frac{1}{w_i} \sum_{j=1}^{n} |A_{i,j}| w_j \qquad \text{mit } \mathbf{w} > \mathbf{0} \tag{5.25}$$

$\mathbf{w}$ ist ein n-dimensionaler Vektor, falls $\mathbf{w}$ der Einsvektor ist, entfällt der Index $\mathbf{w}$ bei der Norm. Bedeutsam sind Matrixnormen u.a. weil gilt $\rho(\mathbf{M}) \leq \|\mathbf{M}\|_\infty^{\mathbf{w}}$. Eine einfache Methode zum Nachweis von $\rho(\mathbf{M}) < 1$ ist der Nachweis von $\|\mathbf{M}\|_\infty^{\mathbf{w}} < 1$. Problematisch ist dabei unter Umständen die Bestimmung eines adäquaten Vektors $\mathbf{w}$. Außerdem gibt es Matrizen $\mathbf{M}$ mit $\rho(\mathbf{M}) < 1$ aber $\|\mathbf{M}\|_\infty^{\mathbf{w}} \geq 1$ für alle $\mathbf{w} > 0$. Für eine sehr wichtige Klasse von Matrizen, nämlich die nichtnegativen Matrizen, sind die Bedingungen $\rho(\mathbf{M}) < 1$ und $\|\mathbf{M}\|_\infty^{\mathbf{w}} < 1$ äquivalent. Eine Matrix $\mathbf{M}$ ist nichtnegativ ($\mathbf{M} \geq \mathbf{0}$) wenn sie keine negativen Elemente beinhaltet.

Wir werden nun noch ein weiteres Konvergenzresultat kennenlernen, ohne dieses ausführlich herzuleiten. Eine vollständige Herleitung würde den Rahmen der Vorlesung sprengen. Das wesentliche Ziel ist es, aus der Struktur der Matrix ohne aufwendige Analysen die Konvergenz eines Verfahrens abzuleiten.

Definition 5.1 *Eine quadratische Matrix* $\mathbf{A}$ *heißt diagonaldominant, wenn* $\sum_{i \neq j} |A_{i,j}| < |A_{i,i}|$ *für alle* i.

Betrachten wir nun die Iterationsmatrix $\mathbf{M}$ des Jacobi Verfahrens für das System $\mathbf{Ax} = \mathbf{b}$, wenn $\mathbf{A}$ diagonaldominant ist. Es gilt dann $|M_{i,j}| = |A_{i,j}|/|A_{i,i}|$ für $j \neq i$ und $M_{i,i} = 0$. Aus der Diagonaldominanz folgt $\sum_{j=1}^{n} |M_{i,j}| < 1$ für alle i. Dies ist äquivalent zu $\|\mathbf{M}\|_\infty < 1$ und damit $\rho(\mathbf{M}) < 1$. Mit dieser Herleitung haben wir den folgenden Satz bewiesen.

Satz 5.4 *Falls* $\mathbf{A}$ *eine diagonaldominante Matrix ist, so konvergiert das Jacobi Verfahren bei der Lösung des Gleichungssystems* $\mathbf{Ax} = \mathbf{b}$.

Wir wollen nun noch das folgende System $\mathbf{Ax} = \mathbf{b}$ mit $A_{i,j} \leq 0$ falls $i \neq j$ und $A_{i,i} > 0$ kurz untersuchen. Sei $\mathbf{M}^J$ die korrespondierende Iterationsmatrix des Jacobi Verfahrens, mit $M_{i,i}^J = 0$ und $M_{i,j}^J = -A_{i,j}/A_{i,i}$ für $i \neq j$, d.h. $\mathbf{M}^J \geq \mathbf{0}$. Sei $\mathbf{M}^{GS}$ die Iterationsmatrix des Gauß-Seidel Verfahrens, es gilt $\mathbf{M}^{GS} \geq \mathbf{0}$. Der folgende Satz wird ohne Beweis vorgestellt.

Satz 5.5 *Sei* $\mathbf{A}$ *eine quadratische Matrix wie oben beschrieben, es gilt:*

a) Falls $\rho(\mathbf{M}^J) < 1$, *dann ist* $\rho(\mathbf{M}^{GS}) \leq \rho(\mathbf{M}^J)$.

b) Falls $\rho(\mathbf{M}^J) \geq 1$, *dann ist* $\rho(\mathbf{M}^{GS}) \geq \rho(\mathbf{M}^J)$.

Dies bedeutet, daß falls das Jacobi Verfahren für die beschriebene Klasse von Matrizen konvergiert, so konvergiert auch das Gauß-Seidel Verfahren und zwar für alle Variablenanordnungen. Diese Beobachtung zeigt, daß die Klasse von positiven Iterationsmatrizen eine gewisse

Robustheit gegenüber der Berechnung der Variablenwerte aufweist und damit insbesondere für asynchrone Algorithmen interessant ist.

Zum Abbruch der Iteration reicht es im allgemeinen aus, eine genügend genaue Approximation der Lösung zu ermitteln, d.h. die Iteration kann nach t Iterationsschritten abgebrochen werden, falls $\|\mathbf{x}^* - \mathbf{x}(t)\| < \epsilon$ für einen genügend kleinen Wert ϵ. Leider soll nun aber gerade $\mathbf{x}^*$ oder eine Approximation dafür berechnet werden und steht deshalb nicht als Kriterium für die Güte der erreichten Approximation zur Verfügung. Deshalb behilft man sich meist mit einer anderen Abschätzung, nämlich $\|\mathbf{Ax}(t) - \mathbf{b}\| < \epsilon$. Im allgemeinen wird, falls die Bedingung erfüllt ist, $\mathbf{x}(t)$ nahe bei $\mathbf{x}^*$ liegen, auch wenn dies nicht garantiert ist. Betrachten wir aber nun die Bestimmung der obigen Norm und wählen die Maximumsnorm $\|\cdots\|_\infty$, d.h. wir müssen $\max_i |[\mathbf{Ax}]_i - \mathbf{b}_i|$ berechnen. Der Reduktionsfakor der Iteration (d.h. der Wert mit der sich der Iterationsvektor der Lösung annähert) kann, unter der Annahme geometrischer Konvergenz, während der Iteration durch $\rho = (\|x(t+1) - x(t)\| / \|x(t) - x(t-1)\|$ geschätzt werden. Damit ist auch eine, oftmals sehr grobe, Schätzung des Konvergenzverhaltens möglich.

Falls die Matrix $\mathbf{A}$ dicht besetzt ist, bestimmt bei einer parallelen Implementierung jeder Prozessor nach jeder Iteration das Maximum für die Variablen, die er berechnet. Diese Werte können nun entlang eines Spannbaums im Netz weitergeleitet werden, wobei jeweils das Maximum des eigenen Wertes und des Wertes der Kinderknoten an den Elterknoten weitergeleitet wird. Der Wurzelknoten beinhaltet schließlich das Maximum aller Werte und kann, falls dieses kleiner als das gewählte ϵ ist, die Iteration abbrechen. Die benötigte Kommunikation entspricht einer einfachen Akkumulation und ist vernachlässigbar im Vergleich zum allgemeinen Broadcast, der für den Austausch der Berechnungen nach jedem Iterationsschritt notwendig ist.

Anders sieht die Situation aus, wenn $\mathbf{A}$ spärlich besetzt ist und die Funktionen/Variablen so auf Prozessoren verteilt wurden, daß nur benachbarte Prozessoren kommunizieren müssen (z.B. bei den vorgestellten Gitterstrukturen). In diesem Fall ist der Kommunikationszeitaufwand nach jeder Iteration proportional zur Anzahl der Variablen pro Prozessor, der Kommunikationszeitaufwand zur Bestimmung der Norm aber proportional zur maximalen Weglänge in der Topologie. Damit kann die Bestimmung der Norm die Geschwindigkeit des Algorithmus deutlich verlangsamen. Ein Ausweg ist die Bestimmung nicht nach jeder Iteration, sondern nur alle 10 oder 20 Iterationsschritte. Alternativ können Iterationen und Normbestimmung auch überlagert werden. Es werden die Maxima von allen Prozessoren während der Iteration an einen Prozessor übertragen, dieser beendet bei unterschreiten von ϵ die Iteration. In diesem Fall beruht die Beendigung der Iteration aber auf veralteten Informationen, d.h. der Wert von $\mathbf{x}(t)$, der zum Abbruch der Iteration führt, ist inzwischen nicht mehr verfügbar. Die einzelnen Prozessoren haben schon $\mathbf{x}(t')$ $(t' > t)$ berechnet. In den meisten Fällen ist dieser Unterschied unkritisch, da t und t' sich nur geringfügig unterscheiden und das Konvergenzverhalten relativ gleichmäßig ist. Man kann sich aber durchaus Fälle vorstellen, in denen $\|\mathbf{Ax}(t) - \mathbf{b}\| < \epsilon$ und $\|\mathbf{Ax}(t') - \mathbf{b}\| >> \epsilon$ für $t' > t$ gilt.

5.2.4 Vorkonditionierung und Blockiterationsverfahren

Bei der Vorkonditionierung wird das System $\mathbf{Ax} = \mathbf{b}$ in ein äquivalentes System $\mathbf{MAx} = \mathbf{Mb}$ mit identischer Lösung, aber besseren "Eigenschaften" transformiert. Bessere Eigenschaften bedeutet in diesem Zusammenhang, daß die iterativen Verfahren schneller eine Lösung finden, bzw. die Lösung nach einer festen Zeit (d.h. festen Zahl von Iterationen) genauer als bei Verwendung des ursprünglichen Systems ist. In den meisten Fällen kann dieses "bessere" Verhalten nicht garantiert werden, tritt aber meist ein. Eine solche Wahl ist z.B. $\mathbf{M} \approx \mathbf{A}^{-1}$ weil dadurch $\mathbf{MAx} \approx \mathbf{I}x \approx \mathbf{Mb}$ eine einfache Bestimmung von x verspricht. Natürlich darf die Bestimmung von $\mathbf{M}$ nicht sehr aufwendig sein.

Es gibt eine Reihe von Verfahren zur Vorkonditionierung, die hier nicht im einzelnen behandelt werden sollen. Beispiele für solche Verfahren sind die Umordnung von Zeilen in der Matrix oder die teilweise LU-Faktorisierung, die den Algorithmus zur LU-Faktorisierung nicht vollständig durchführt, sondern nur einige Werte berechnet. Die Parallelisierung von Verfahren zur Vorkonditionierung ist oft das größte Problem bei der Parallelisierung iterativer Verfahren. Deshalb werden bei parallelen iterativen Verfahren gar keine oder sehr einfache Vorkonditionierungen eingesetzt.

5.2.5 Nichtstationäre Verfahren

Nichtstationäre Verfahren zur Lösung linearer Gleichungssysteme erlangten in den letzten Jahren eine sehr große Popularität, so daß zahlreiche unterschiedliche Varianten publiziert wurden. Im Rahmen der Vorlesung kann nur ein kleiner Ausschnitt der Verfahren vorgestellt werden. Die grundlegenden Ideen lassen sich aber aus diesen Verfahren herleiten. Das Kapitel beschäftigt sich in stärkerem Maße mit der Einführung nichtstationärer Verfahren als mit deren Parallelisierung. Dies liegt darin begründet, daß die Parallelisierung sich zu großen Teilen aus bereits behandelten Konzepten, wie der Parallelisierung von Vektor-Matrix-Produkten herleiten läßt. Es reicht deshalb aus, nach der Vorstellung der Verfahren, deren Parallelisierung kurz abzuhandeln.

Im folgendem Kapitel führen wir zunächst eine etwas andere Sichtweise bei der Lösung linearer Gleichungssysteme ein. Unter bestimmten Bedingungen läßt sich die Lösung eines linearen Gleichungssystems nämlich als Minimierungsproblem auffassen. Diese Sicht ist die Basis vieler nichtstationärer Verfahren und beschreibt das Verfahren des konjugierten Gradienten quasi automatisch. In einem separaten Abschnitt werden wir das Verfahren des konjugierten Gradienten noch einmal allgemeiner betrachten und insbesondere die Parallelisierbarkeit untersuchen. Daran anschließend werden die Verfahren MINRES und GMRES vorgestellt. GMRES ist ein allgemein anwendbares Verfahren, welches gewisse Optimalitätskriterien erfüllt, bei der praktischen Realisierung aber sehr schnell Speicherplatz und Laufzeitprobleme aufweist. Deshalb werden in einem abschließenden Abschnitt andere nichtstationäre Verfahren kurz vorgestellt.

Die Lösung linearer Gleichungssysteme als Minimierungsproblem

Zur Beschreibung der Lösung linearer Gleichungssysteme als Minimierungsproblem nehmen wir an, daß die Matrix $\mathbf{A}$ symmetrisch und positiv definit ist. Die zentrale Idee besteht darin, daß das Minimum/Maximum einer Funktion eine Nullstelle der 1. Ableitung ist und man somit eine geeignete Funktion wählen kann, die in $\mathbf{Ax} - \mathbf{b} = 0$ ein Minimum annimmt. Dies ist der Fall für die quadratische Kostenfunktion

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b} \tag{5.26}$$

Damit können wir ein Minimierungsproblem ohne Nebenbedingungen lösen, um die Lösung des Gleichungssystems zu erhalten. Da $\mathbf{A}$ positiv definit ist, kann man zeigen, daß die Kostenfunktion konvex ist und ein eindeutiges Minimum, nämlich $\mathbf{x}^*$ besitzt. Ausgehend von einem initialen Vektor $\mathbf{x}^{(0)}$ werden in den folgenden Iterationsschritten die $\mathbf{x}^{(t)}$ berechnet, bis der erreichte Iterationsvektor nahe genug bei der Lösung $\mathbf{x}^*$ liegt. In einem Iterationsschritt wird der Vektor wie in der folgenden Gleichung gezeigt verändert.

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \alpha(t)\mathbf{p}^{(t)} \tag{5.27}$$

Der Vektor $\mathbf{p}^{(t)} \in \mathbb{R}^n$ beschreibt die Richtung der Änderung für jede Vektorkomponente und $\alpha(t)$ ist ein Skalar, der die Schrittweite angibt und so gewählt wird, daß

$$f(\mathbf{x}^{(t)} + \alpha(t)\mathbf{p}^{(t)}) = \min_{\alpha \in \mathrm{R}}(f(\mathbf{x}^{(t)} + \alpha\mathbf{p}^{(t)})). \tag{5.28}$$

Wir definieren den Residuenvektor

$$\mathbf{r}^{(t)} = \mathbf{A}\mathbf{x}^{(t)} - \mathbf{b} \tag{5.29}$$

Falls $\mathbf{r}^{(t)} = \mathbf{0}$ gilt, wurde die Lösung offensichtlich gefunden. Eine notwendige Bedingung für das Minimum bzgl. α in (5.28) ist

$$\frac{df(\mathbf{x} + \alpha\mathbf{p})}{d\alpha} = \alpha\mathbf{p}^T\mathbf{A}\mathbf{p} + \mathbf{p}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = 0 \tag{5.30}$$

In der obigen Gleichung wurde der Laufindex t weggelassen, um die Übersichtlichkeit zu erhöhen. Da die zweite Ableitung

$$\frac{d^2 f(\mathbf{x} + \alpha\mathbf{p})}{d\alpha^2} = \mathbf{p}^T\mathbf{A}\mathbf{p} > 0$$

für positiv definite $\mathbf{A}$ immer größer Null ist, liegt eine Minimum vor. Nullsetzen der ersten Ableitung liefert

$$\alpha_{min} = \frac{-\mathbf{p}^T\mathbf{r}}{\mathbf{p}^T\mathbf{A}\mathbf{p}} \tag{5.31}$$

Die Wahl von α_{min} garantiert die größtmögliche Verkleinerung der quadratischen Funktion f in Richtung $\mathbf{p}$. Falls $\mathbf{p}^T\mathbf{r} = 0$ gilt, d.h. Richtungs- und Residuenvektor orthogonal sind, so stagniert das Verfahren. Eine solche Wahl des Richtungsvektors ist deshalb zu vermeiden. Durch die Wahl unterschiedlicher Richtungsvektoren lassen sich unterschiedliche Verfahren definieren. Wir betrachten im folgenden einige Beispiele.

Wenn wir die Vektoren $\mathbf{p}^{(t)}$ zyklisch als Einheitsvektoren wählen, so gilt $\mathbf{p}^{(1)} = \mathbf{e}^1, \mathbf{p}^{(2)} = \mathbf{e}^2, \ldots, \mathbf{p}^{(n)} = \mathbf{e}^n, \mathbf{p}^{(n+1)} = \mathbf{e}^1, \ldots$, wobei $\mathbf{e}^i$ ein n-dimensionaler Vektor mit 1 in Position i und 0 an den übrigen Positionen ist. Bei dieser Wahl ergibt sich folgendes Iterationsschema.

$$\alpha(m+1) = -\frac{(\mathbf{e}^i)^T\mathbf{r}^{(m)}}{(\mathbf{e}^i)^T\mathbf{A}\mathbf{e}^i} = \frac{r_i^{(m)}}{\mathbf{A}_{i,i}} \text{ und } \mathbf{x}^{(m+1)} = \mathbf{x}^{(m)} - \frac{r_i^{(m)}}{\mathbf{A}_{i,i}}\mathbf{e}^i$$

mit $i = (m \; mod \; n) + 1$. Es wird nur die i-te Komponente des Vektors modifiziert und die Änderung entspricht der Iterationsvorschrift des Gauß-Seidel Verfahrens für die i-te Komponente. Damit ergeben n Minimierungsschritte einem Iterationsschritt des Gauß-Seidel Verfahrens.

Ein sehr einfacher Ansatz wäre, den Vektor der Richtungsänderungen $\mathbf{p}^{(t)}$ auf $-\mathbf{r}^{(t)}$ zu setzen. Diesen Ansatz bezeichnet man als Methode des steilsten Abstiegs. Dieses Verfahren zeigt in der Praxis leider eine sehr schlechte Konvergenz und wird deshalb kaum eingesetzt. Besser ist die Methode des konjugierten Gradienten (CG-Methode) für deren Beschreibung einige Begriffsdefinitionen notwendig sind.

Definition 5.2 *Zwei Vektoren* $\mathbf{p}, \mathbf{q} \in \mathbb{R}^n$ *sind bzgl. einer symmetrischen positiv definiten Matrix* $\mathbf{A}$ *konjugiert, falls*

$$\mathbf{q}^T\mathbf{A}\mathbf{p} = \mathbf{p}^T\mathbf{A}\mathbf{q} = 0 \; .$$

Linear unabhängige Vektoren $\mathbf{p}(1), \ldots, \mathbf{p}(n)$ *mit der Eigenschaft*

$$(\mathbf{p}(i))^T\mathbf{A}\mathbf{p}(j) = 0 \textit{ für alle } i \neq j \; ,$$

bezeichnet man als konjugierte Basis des $\mathbb{R}^n$ *bzgl.* $\mathbf{A}$.

Den folgenden Satz wollen wir ohne Beweis einführen.

Satz 5.6 *Falls die Richtungsvektoren zur Minimierung eine konjungierte Basis des $\mathbb{R}^n$ bilden, so berechnet ein Minimierungsverfahren mit exakter Arithmetik nach maximal n Schritten die exakte Lösung.*

Das CG Verfahren wählt Richtungsvektoren gerade so, daß nach maximal n Schritten eine konjugierte Basis entsteht.

CG-Verfahren
Initialisierung:
    wähle initiale Belegung von $\mathbf{x}^{(0)}$
    $\mathbf{r}^{(0)} = b - \mathbf{A}\mathbf{x}^{(0)}$
    $\mathbf{p}^{(0)} = 0; \beta_0 = 0$;
Iteration:
    for $t = 1, 2, \ldots$ *do*
        $\mathbf{p}^{(t)} = \mathbf{r}^{(t-1)} + \beta_{t-1}\mathbf{p}^{(t-1)}$
        $\alpha_t = (\mathbf{p}^{(t)})^T r^{(t-1)} / (\mathbf{p}^{(t)})^T \mathbf{A}\mathbf{p}^{(t)}$
        $\mathbf{r}^{(t)} = \mathbf{r}^{(t-1)} - \alpha_t \mathbf{A}\mathbf{p}^{(t)}$
        $\beta_t = (\mathbf{r}^{(t)})^T \mathbf{r}^{(t)} / (\mathbf{r}^{(t-1)})^T \mathbf{r}^{(t-1)}$
        $\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} + \alpha_t \mathbf{p}^{(t)}$
    od
end ;

Der folgende Satz faßt die wesentlichen Eigenschaften des CG-Verfahrens zusammen

Satz 5.7 *Sei* $\mathbf{A}$ *eine symmetrische, positiv definite, (reellwertige)* $n \times n$ *Matrix und* $\mathbf{b} \in \mathbb{R}^n$, *seien ferner Vektoren* $\mathbf{x}^{(i)}$, $\mathbf{r}^{(i)}$, $\mathbf{p}^{(i)}$ *wie im i-ten Schritt des CG-Verfahrens definiert, dann ist für jede initiale Belegung von* $\mathbf{x}^{(0)}$ *sichergestellt, daß ein* $l \leq n$ *mit* $\mathbf{p}^{(l)} = 0$ *existiert. Für Werte* $i \leq l$ *gilt:*

a) $\mathbf{A}\mathbf{x}^{(l)} = \mathbf{b}$,

b) $(\mathbf{p}^{(i)})^T \mathbf{r}^{(j)} = 0$ *für* $0 \leq i < j \leq l$

c) $(\mathbf{p}^{(i)})^T \mathbf{r}^{(i)} = (\mathbf{r}^{(i)})^T \mathbf{r}^{(i)}$ *für* $0 \leq i \leq l$

d) $(\mathbf{p}^{(i)})^T \mathbf{A}\mathbf{p}^{(j)} = 0$ *für* $0 \leq i < j \leq l$,
$(\mathbf{p}^{(i)})^T \mathbf{A}\mathbf{p}^{(i)} > 0$ *für* $0 \leq i \leq l$

e) $(\mathbf{r}^{(i)})^T \mathbf{r}^{(j)}) = 0$ *für* $0 \leq i < j \leq l$,
$(\mathbf{r}^{(i)})^T \mathbf{r}^{(i)} > 0$ *für* $0 \leq i < l$

f) $\mathbf{r}^{(i)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(i)}$ *für* $0 \leq i \leq l$

Das Verfahren findet die Lösung des Gleichungssystems nach maximal n Iterationen, für große n werden oftmals aber sehr viel weniger Iterationen benötigt. Dies ist ein wesentlicher Unterschied zu den bisherigen iterativen Verfahren, die i.a. keine exakte Lösung erreichen.

Zum Abschluß dieses Abschnitts untersuchen wir die parallele Implementierung des Verfahrens. Betrachten wir dazu Iterationsschritt t. $\mathbf{x}^{(t-1)}$, $\mathbf{p}^{(t-1)}$, $\mathbf{r}^{(t-1)}$, α_{t-1} und β_{t-1} seien bereits berechnet. Wir müssen nun die folgenden Operationen durchführen, wobei Vektor-Vektor-Additionen und Vektor-Skalar-Multiplikationen vernachlässigt werden.

a) α_t durch eine Vektor-Matrix-Multiplikation und 2 Vektor-Vektor-Multiplikationen berechnen.

b) $\mathbf{r}^{(t)}$ durch eine Vektor-Matrix-Multiplikation berechnen, die jedoch mit der Vektor-Matrix-Multiplikation zur Berechnung von α_t identisch ist.

c) β_t durch 2 Vektor-Vektor-Multiplikationen berechnen.

Beachtenswert ist, daß für die beiden Vektor-Matrix-Produkte die selbe Matrix $\mathbf{A}$ verwendet wird. Stehen n Prozessoren zur Verfügung, so kann der i-te Prozessor die Komponenten $x_i^{(t)}$, $p_i^{(t)}$ und $r_i^{(t)}$ berechnen. Da $\mathbf{A}$ eine symmetrische Matrix ist, macht es keinen Unterschied, ob ein Prozessor eine Spalte oder eine Zeile von $\mathbf{A}$ speichert. Wir nehmen hier an, daß $\mathbf{A}$ zeilenweise auf den Prozessoren gespeichert wird. Im ersten Schritt wird $p_i^{(t)}$ berechnet und an alle Prozessoren in $O(n)$ verteilt. Im zweiten Schritt wird $\mathbf{Ap}^{(t)}$ zeilenweise berechnet, der Aufwand liegt in $O(n)$. Die Vektor-Produkte werden gebildet, indem Prozessor i die i-te Komponente berechnet, also z.B. im zweiten Schritt $p_i^{(t)} r_i^{(t-1)}$ und $p_i^{(t)} \sum_j \mathbf{A}_{i,j} p_j^{(t)}$ und die einzelnen Werte entlang eines Spannbaums im Netz addiert und das Resultat anschließend an alle Prozessoren verteilt. Die Kommunikation entspricht gerade einer einfachen Akkumulation und einem einfachem Broadcast, der Zeitaufwand ist proportional zur maximalen Weglänge der verwendeten Topologie. Im dritten Schritt wird $r_i^{(t)}$ lokal aktualisiert, um dann im vierten Schritt wiederum in einer Akkumulation entlang eines Spannbaums verwendet zu werden. Im letzten Schritt reicht es lokal $x_i^{(t)}$ zu berechnen.

Insgesamt liegt der Aufwand damit in $O(n)$ auf einem linearen Array und in $O(\log n)$ auf einem Hyperwürfel. Der Gesamtaufwand mit n Prozessoren liegt damit in $O(n)$ für alle Topologien. Auf Grund der schnelleren Vektor-Produkte und der schnelleren Verteilung der Vektorelemente ist ein Hyperwürfel einem Array vorzuziehen.

MINRES und GMRES

Weitere nichtstationäre Iterationsverfahren

Falls $\mathbf{A}$ invertierbar, nicht aber symmetrisch ist, so kann das Verfahren auf $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$ mit identischen Lösungsvektor angewendet werden.

Die entsprechende Erweiterung zu CG wird in der Literatur als CGNR (Conjugate Gradient for Normal Equations) bezeichnet. Die Funktionsweise ist analog, da jedoch die Kondition von $\mathbf{A}^T\mathbf{A}$ schlechter als von $\mathbf{A}$ ist wird, wird CGNR wenig angewendet. Als Alternative zu CGNR sind weitere Varianten bekannt, hierzu zählt Biconjugate Gradient (BCG), bei dem zwei Residualvektoren $\mathbf{r}$ and $\tilde{\mathbf{r}}$ verwendet werden, die die Biorthogonalitätsbedingung $\tilde{\mathbf{r}}^T\mathbf{r} = 0$ falls $\tilde{\mathbf{r}}$ und $\mathbf{r}$ aus unterschiedlichen Iterationsschritten stammen. Für die zugehörigen Richtungsvektoren $\mathbf{p}$ und $\tilde{\mathbf{p}}$ muß gelten, daß $\tilde{\mathbf{p}}^T\mathbf{Ap} = 0$ falls $\mathbf{p}$ und $\tilde{\mathbf{p}}$ aus unterschiedlichen Iterationsschritten stammen. BCG ist verwandt mit GMRES, weil die Approximationen der Lösungsvektors im gleichen Krylov Teilraum berechnet werden.

Eine Erweiterung zu BCG nennt sich CGS (Conjugate Gradient Squared) bei dem der Rechenaufwand gegenüber BCG verringert und eine bessere Konvergenzgeschwindigkeit erreicht wird.

5.2.6 Analyse von Markov Ketten als Anwendungsbeispiel

5.3 Schnelle iterative Matrix Invertierung

Die Invertierung von Matrizen haben wir bisher auf die Lösung linearer Gleichungssysteme zurückgeführt. In diesem Abschnitt behandeln wir ein iteratives Verfahren zur Berechnung der inversen Matrix, welches extrem schnell konvergiert und deutlich effizienter als die direkte Matrix Invertierung ist. Das Verfahren basiert auf der Newton Iteration und ist numerisch stabil.

Gesucht ist die Inverse $\mathbf{A}^{-1}$ der Matrix $\mathbf{A}$. Sei $\mathbf{B}[k]$ eine Approximation für $\mathbf{A}^{-1}$. Als Matrix der Residuen definieren wir

$$\mathbf{R}(\mathbf{B}[k]) = \mathbf{I} - \mathbf{B}[k]\mathbf{A} \tag{5.32}$$

Die Residuen beschreiben wie "weit" $\mathbf{B}[k]$ von $\mathbf{A}^{-1}$ entfernt ist. Aus der Residuen Matrix und $\mathbf{B}[k]$ wird eine neue Approximation der Inversen nach der folgenden Formel berechnet.

$$\mathbf{B}[k+1] = (\mathbf{I} + \mathbf{R}(\mathbf{B}[k]))\mathbf{B}[k] = 2\mathbf{B}[k] - \mathbf{B}[k]\mathbf{A}\mathbf{B}[k] \tag{5.33}$$

Betrachten wir nun die Residuen Matrix $\mathbf{R}(\mathbf{B}[k+1])$. Es gilt

$$\begin{aligned}\mathbf{R}(\mathbf{B}[k+1]) &= \mathbf{I} - \mathbf{B}[k+1]\mathbf{A} = \mathbf{I} - (\mathbf{I} + \mathbf{R}(\mathbf{B}[k]))\mathbf{B}[k]\mathbf{A} \\ &= (\mathbf{I} - \mathbf{B}[k]\mathbf{A}) - \mathbf{R}(\mathbf{B}[k])\mathbf{B}[k]\mathbf{A} = \mathbf{R}(\mathbf{B}[k])(I - \mathbf{B}[k]\mathbf{A}) = (\mathbf{R}(\mathbf{B}[k]))^2\end{aligned}$$

Damit gilt $\mathbf{R}(\mathbf{B}[k]) = (\mathbf{R}(\mathbf{B}[0]))^{2^k}$. Falls es nun gelingt, eine Matrix $\mathbf{B}[0]$ zu finden, für die gilt $\|\mathbf{R}(\mathbf{B}[0])\|_2 < 1$, so konvergieren die Residuen Matrizen sehr schnell gegen Null und die Matrix $\mathbf{B}[k]$ sehr schnell gegen $\mathbf{A}^{-1}$. Durch den Exponenten 2^k ist die Konvergenzrate deutlich schneller als bei den klassischen iterativen Verfahren, deren Konvergenz meist geometrisch (Exponent k) ist. Bleibt noch das Problem, Matrix $\mathbf{B}[0]$ so zu wählen, daß die initiale Residuen Matrix die geforderte Bedingung erfüllt (siehe auch die Abschätzungen für Normen im Anhang). Der folgende Satz gibt eine Matrix an, die einfach zu bestimmen ist und die geforderte Bedingung erfüllt.

Satz 5.8 *Sei* $\mathbf{B}[0] = \mathbf{A}^T/tr(\mathbf{A}^T\mathbf{A})$, *dann gilt* $\|\mathbf{I} - \mathbf{B}[0]\mathbf{A}\|_2 \leq 1 - 1/(n\kappa^2(\mathbf{A}))$. *Wobei* $\kappa(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2 = \sqrt{\lambda_n/\lambda_1}$ *die Kondition von* $\mathbf{A}$ *ist,* λ_n *ist der größte und* λ_1 *der kleinste Eigenwert von* $\mathbf{A}^T\mathbf{A}$.

$tr(\mathbf{A}^T\mathbf{A})$ bezeichnet die Spur (Trace) der Matrix $\mathbf{A}^T\mathbf{A}$, die Spur ist als Summe der Diagonaleinträge definiert ($tr(\mathbf{C}) = \sum_i \mathbf{C}_{ii}$).
Bew:
Seien $\lambda_1 \leq \ldots \leq \lambda_n$ die Eigenwerte von $\mathbf{A}^T\mathbf{A}$. Es gilt $tr(\mathbf{A}^T\mathbf{A}) = \lambda_1 + \ldots + \lambda_n$. Der i-te Eigenwert ρ_i von $\mathbf{I} - \mathbf{B}[0]\mathbf{A} = \mathbf{I} - \mathbf{A}^T\mathbf{A}/tr(\mathbf{A}^T\mathbf{A})$ entspricht $1 - \lambda_i/(\lambda_1 + \ldots + \lambda_n)$ und ist damit offensichtlich immer größer 0. Weiterhin gilt

$$\rho_i \leq 1 - \frac{\lambda_i}{n\lambda_n} \leq 1 - \frac{\lambda_1}{n\lambda_n} = 1 - \frac{1}{n\kappa^2(\mathbf{A})}$$

für alle i. Daraus folgt $\rho(\mathbf{I} - \mathbf{B}[0]\mathbf{A}) = \max_i |\rho_i| \leq 1 - 1/(n\kappa^2(\mathbf{A}))$. Da $\mathbf{I} - \mathbf{B}[0]\mathbf{A} = \mathbf{I} - \mathbf{A}^T\mathbf{A}/tr(\mathbf{A}^T\mathbf{A})$ symmetrisch ist gilt $\|\mathbf{I} - \mathbf{B}[0]\mathbf{A}\|_2 = \rho(I - \mathbf{B}[0]\mathbf{A})$. □

Wenn $\mathbf{A}$ nicht extrem schlecht konditioniert ist, d.h. $\kappa(\mathbf{A})$ nicht sehr groß ist, so reichen relativ wenige Iterationen aus, um $\mathbf{A}^{-1}$ mittels $\mathbf{B}[k]$ zu approximieren. Die folgende Abschätzungen wollen wir ohne Beweis einführen. Falls $\kappa(A) \leq n^d$ für eine Konstante d und c eine positive ganze Zahl ist, so gelten die folgenden Fehlerschranken nach $k \geq (c+2d+1)\log n$ Iterationen.

$$\|\mathbf{I} - \mathbf{B}[k]\mathbf{A}\|_2 \leq 2^{-n^c} \qquad \|\mathbf{B}[k] - \mathbf{A}^{-1}\|_2 \leq 2^{-n^c}\frac{\kappa(\mathbf{A})}{\|\mathbf{A}\|_2} \tag{5.34}$$

Für jede Iteration müssen 2 Matrix-Matrix-Multiplikationen durchgeführt werden, diese können in $O(\log n)$ Schritten auf einem Hyperwürfel mit $O(n^3)$ Prozessoren ausgeführt werden. Da $O(\log n)$ Iterationen zu berechnen sind, liegt der Gesamtaufwand in $O(\log^2 n)$. Die Berechnung von $tr(\mathbf{A}^T\mathbf{A})$ fällt nicht ins Gewicht, da sie nur einmal ausgeführt werden muß und auch $O(\log n)$ Schritte mit $O(n^3)$ Prozessoren benötigt.

Stehen weniger Prozessoren zur Verfügung, so kann der Algorithmus z.B. in $O(n^2 \log n)$ Schritten auf $O(n)$ Prozessoren ablaufen. Dies ist langsamer als die Laufzeit der vorgestellten direkten Verfahren zur Lösung linearer Gleichungssysteme, die $O(n^2)$ Schritte auf $O(n)$ Prozessoren benötigen. Allerdings lassen sich die üblichen direkten Verfahren nicht auf $O(n^3)$ Prozessoren sinnvoll implementieren.

Der Algorithmus ist natürlich auch zur Lösung linearer Gleichungssysteme $\mathbf{Ax} = \mathbf{b}$ einsetzbar. Es wird zuerst die Inverse $\mathbf{A}^{-1}$ berechnet und anschließend mittels einfacher Vektor-Matrix-Multiplikation $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Dieses Vorgehen scheint auf den ersten Blick sehr interessant zu sein, da die Konvergenz deutlich schneller als bei klassischen iterativen Verfahren ist. Der entscheidende Nachteil ist aber beim Speicherplatzbedarf zu sehen. Iterative Verfahren werden primär für große dünn besetzte Matrizen eingesetzt. Die Inverse einer dünn besetzten Matrix ist aber im allgemeinen dicht besetzt, dadurch bedingt wächst der Speicherplatzbedarf durch die Berechnung von $\mathbf{B}[k]$ an und realistische Matrixdimensionen sind nicht handhabbar. Dazu kommt, daß $O(n^3)$ Prozessoren für Problemgrößen, bei denen sich der Einsatz des Algorithmus lohnen würde, da das Standardverfahren zu langsam ist, nicht zur Verfügung stehen.

5.4 Lösung partieller Differentialgleichungen

Viele natürliche und technische Prozesse lassen sich durch Differentialgleichungen beschreiben. Eine Differentialgleichung beschreibt die Änderung eines Prozeßzustandes in Abhängigkeit räumlichen und/oder zeitlichen Änderungen. Eine Differentialgleichung für eine Funktion mit mehreren Variablen nennt man eine partielle Differentialgleichung. Die Lösung partieller Differentialgleichungen ist ein praktisch wichtiges und theoretisch breit untersuchtes Gebiet, für das eine Vielzahl von numerischen Methoden existiert. Partielle Differentialgleichungen führen oftmals zu sehr großen, aber wohlstrukturierten linearen Gleichungssystemen, die den Einsatz von Parallelrechnern notwendig und auch lohnenswert machen. Wir werden hier nur einen kurzen Blick in das breite Gebiet der partiellen Differentialgleichungen und deren numerischer Analyse werfen. Im folgenden Abschnitt wird anhand eines einfachen physikalischen Problems eine partielle Differentialgleichung und deren Diskretisierung betrachtet. Daran anschließend werden die resultierenden Gleichungssysteme bzgl. ihrer Struktur und der Anwendung der vorgestellten Lösungsverfahren untersucht. Im letzten Abschnitt betrachten wir schließlich eine Klasse von Lösungsverfahren, die speziell für die Analyse partieller Differentialgleichungen entwickelt wurden.

5.4.1 Die Poisson Gleichung

Als Motivation betrachten wir die Bestimmung der Wärmeverteilung auf einer quadratischen Platte. Wir nehmen zur Vereinfachung an, daß die Kantenlänge der Platte 1 sei. Weiterhin sei

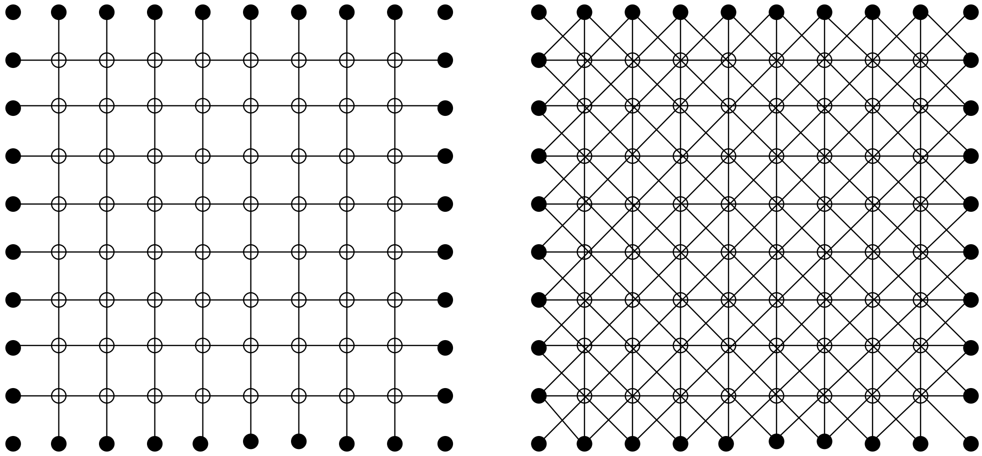


Abbildung 5.4: Gitterstruktur der Fünf- und Neunpunktdiskretisierung.

die Temperatur an den Rändern der Platte bekannt. Berechnet werden soll die Temperatur im Inneren. Die Temperatur an einem Punkt im Inneren der Platte berechnet sich als Durchschnitt der Temperatur der Punkte in der Nachbarschaft und einer zusätzlichen Konstanten zur Darstellung der Wärmezufuhr bzw. Wärmeabgabe an diesem Punkt. Punkte auf der Platte werden durch Koordinaten $x, y \in [0,1]$ beschrieben. Die Wärmezufuhr/-abgabe sei durch eine bekannte Funktion $g(x,y) \to \mathbb{R}$ dargestellt. Die Temperatur am Punkt (x,y) ist durch die Funktion $f(x,y) : [0,1]^2 \to \mathbb{R}$ beschrieben. Da die Randwerte als bekannt vorausgesetzt wurden, sind alle Werte $f(0,y)$, $f(1,y)$, $f(x,0)$ und $f(x,1)$ bekannt. Die übrigen Werte sind durch die folgende Gleichung beschrieben.

$$\frac{\partial^2 f}{\partial x^2}(x,y) + \frac{\partial^2 f}{\partial y^2}(x,y) = g(x,y), \qquad (x,y) \in [0,1]^2 \tag{5.35}$$

Diese Differentialgleichung bezeichnet man als Poisson Gleichung und falls $g(x,y) = 0$ als Laplace Gleichung. Da für die Lösung der Gleichung keine geschlossene Form bekannt ist, müssen die Werte numerisch berechnet werden. Berechnet werden soll die stationäre Lösung, d.h. das System befindet sich in einem Gleichgewichtszustand, der sich mit der Zeit nicht mehr ändert. Eine übliche Methode zur Lösung von Differentialgleichungen ist die Methode der Diskretisierung. Das Verhalten der kontinuierlichen Funktion wird dazu durch das Verhalten an einigen diskreten Punkten approximiert. Auf unser Problem übertragen bedeutet dies, daß die Platte durch ein Gitter ersetzt wird und die Temperaturverteilung auf den Knotenpunkten des Gitters berechnet wird. Das zugehörige Modell für die Poisson Gleichung ist auf der linken Seite von Abb. 5.4 dargestellt und wird auch als Fünfpunktdiskretisierung bezeichnet. Die schwarzen Punkte sind Randpunkte, deren Temperaturen bekannt sind. Die Werte für die weißen Punkte sollen berechnet werden. Die Temperatur auf einem Gitterpunkt ist abhängig von der Temperatur seiner vier Nachbarpunkte. Zur Diskretisierung verwenden wir die folgenden Approximationen der Ableitungen.

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2}(x,y) &\approx \frac{f(x+h,y) - 2f(x,y) + f(x-h,y)}{h^2} \\ \frac{\partial^2 f}{\partial y^2}(x,y) &\approx \frac{f(x,y+h) - 2f(x,y) + f(x,y-h)}{h^2} \end{aligned} \tag{5.36}$$

Indem wir $h = 1/n$ wählen unterteilen wir das Intervall $[0,1]$ in n äquidistante Teile und definieren

$$f_{i,j} = f(\tfrac{i}{n}, \tfrac{j}{n}) \quad 0 \leq i, j \leq n$$

$$g_{i,j} = g(\tfrac{i}{n}, \tfrac{j}{n}) \quad 0 < i, j < n$$

Damit ergibt sich die folgende Gleichung zur Beschreibung von $f_{i,j}$ durch Einsetzen in (5.35).

$$f_{i,j} = \frac{f_{i+1,j} + f_{i-1,j} + f_{i,j-1} + f_{i,j+1}}{4} - \frac{g_{i,j}}{4n^2} \tag{5.37}$$

Alle Werte $f_{i,j}$ mit i oder j gleich 0 oder n sind als Grenzpunkte bekannt. Zusammengefaßt liegt nun ein lineares Gleichungssystem mit $(n-1)^2$ Gleichungen und Unbekannten zur approximativen Lösung der Poisson Gleichung vor. Ein größeres n führt zu einer besseren Approximation, erhöht aber auch den Aufwand der Lösung.

Enthält eine Differentialgleichung gemischte Ableitungen der Form $\frac{\partial^2 f}{\partial x \partial y}$, so reicht die Fünfpunktdiskretisierung zur Beschreibung nicht mehr aus. In diesem Fall hängt der Wert $f_{x,y}$ auch von den Werten $f_{i\pm1,j\pm1}$ ab. Man erhält dann eine Neunpunktdiskretisierung, deren Gitterstruktur auf der rechten Seite von Abb. 5.4 gezeigt wird.

Da partielle Differentialgleichungen physikalische und technische Systeme und deren Verhalten in Raum und Zeit beschreiben, lassen sie sich im allgemeinen als zwei- oder drei dimensionale Strukturen auffassen. Eine Diskretisierung führt aber nicht immer zu so regulären Gittern wie wie sie hier kennengelernt haben. Oftmals ist es sinnvoll, in Bereichen, in denen ein gleichmäßiges Verhalten zu erwarten ist, eine grobe Diskretisierung zu wählen und in Bereichen wo ein weniger gleichmäßiges Verhalten erwartet wird oder sehr genaue Resultate benötigt werden, sehr fein zu diskretisieren. Auf diese Weise entstehen irreguläre Gitter. Außerdem werden bei komplexeren Problemen, wie Aerodynamikberechnungen oder Berechnungen zur Wettervorhersage, die Werte einzelner Punkte durch mehrere oft auch nichtlineare Gleichungen beschrieben. Wir wollen dieses Thema aber nicht weiter vertiefen, sondern von einer regelmäßigen, aus einer Fünfpunktdiskretisierung resultierenden Gitterstruktur ausgehen und deren Analyse betrachten.

5.4.2 Lineare Gleichungssysteme aus diskretisierten Differentialgleichungen

Gleichung (5.37) beschreibt offensichtlich ein lineares Gleichungssystem, welches zur Bestimmung der stationären Temperaturverteilung gelöst werden muß. Um eine genügend genaue Lösung zu erhalten, wird die Zahl der Variablen meist recht groß sein. Gleichzeitig hängt der Wert einer Variablen nur von vier anderen Variablenwerten ab, wodurch die Koeffizientenmatrix dünn besetzt ist. Diese beiden Punkte sprechen gegen eine direkte Lösung des Systems, vielmehr ist die Verwendung iterativer Verfahren angebracht. Für das JOR Verfahren würde sich z.B. folgende Iteration ergeben.

$$f_{i,j}(t+1) = (1-\gamma) f_{i,j}(t) + \frac{\gamma}{4}[f_{i+1,j}(t) + f_{i-1,j}(t) + f_{i,j-1}(t) + f_{i,j+1}(t)] - \frac{g_{i,j}}{4n^2} \tag{5.38}$$

Damit ist die Parallelisierung auf einem $(n-1) \times (n-1)$ Array von Prozessoren offensichtlich, Prozessor i, j berechnet $f_{i,j}(t)$ und reicht das Ergebnis an seine Nachbarn weiter. Die Iterationsvorschrift folgt direkt aus der obigen Gleichung. Da der Abhängigkeitsgraph eine Gitterstruktur aufweist, läßt sich das Gauß-Seidel oder SOR Verfahren genauso parallel wie das Jacobi Verfahren implementieren, wenn jeder Prozessor zwei Gleichungen berechnet. Die zugehörige Färbung des Abhängigkeitsgraphen mit zwei Farben ist auf der linken Seite von

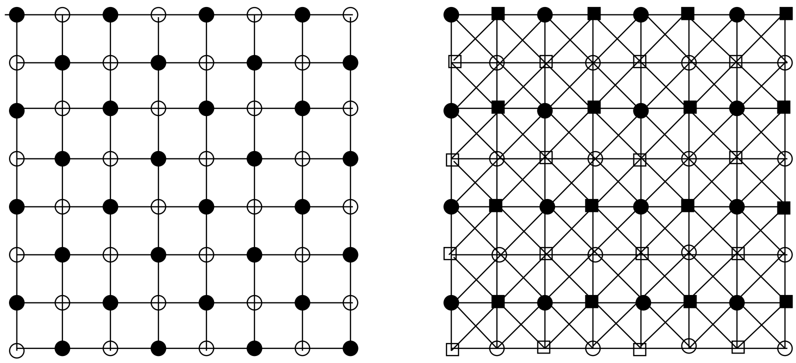


Abbildung 5.5: Färbung der Fünf- und Neunpunktgitter.

Abb. 5.5 zu sehen. Die Randpunkte sind in der Abbildung weggelassen. Sei zur Vereinfachung $(n-1)/2$ eine ganze Zahl, dann kann das SOR Verfahren auf einem $(n-1)/2 \times (n-1)$ Gitter parallelisiert werden. Prozessor (i,j) berechnet die Variablen $(2i-1,j)$ und $(2i,j)$. Falls j eine gerade Zahl ist, beginnt er mit der Berechnung von $(2i-1,j)$ und berechnet anschließend $(2i,j)$. Für ungerade j wird die umgekehrte Reihenfolge gewählt. Da die Anzahl der Variablen normalerweise ein Vielfaches der Anzahl der verfügbaren oder sinnvoll einsetzbaren Prozessoren sein wird, berechnet jeder Prozessor eine Menge von Variablen, die auf einem Untergitter liegen. Durch die Abhängigkeitsstruktur brauchen nur die Randpunkte des Untergitters zwischen den Prozessoren ausgetauscht zu werden. Auch die Gitterstruktur einer Neunpunktdikretisierung erlaubt eine parallele Ausführung der SOR Iteration, wenn jeweils vier Variablen auf einem Prozessor berechnet werden. Die zugehörige Färbung der Gitterpunkte ist auf der rechten Seite von Abb. 5.5 zu sehen. Durch die Einbettung eines zweidimensionalen Arrays in einen Hyperwürfel können die Algorithmen genauso effizient auf Hyperwürfeln implementiert werden. Bei Gitterstrukturen mit drei Dimensionen besteht die Möglichkeit des Einsatzes von dreidimensionalen Gitter oder Hyperwürfel als Multiprozessortopologie. Meist reichen aber auch zweidimensionale Gitter aus, da die Anzahl der Punkte in zwei Dimensionen schon größer als die Zahl der Prozessoren ist. So berechnet ein Prozessor Werte für die Gitterpunkte (i,j,k) für einige i,j und alle k.

Man kann die aus der Fünfpunktdiskretisierung resultierenden Gleichungen natürlich in der Form $\mathbf{Ax} = \mathbf{b}$ aufschreiben und mit einem der vorgestellten Verfahren lösen. Für die Darstellung der Matrix sollte eine dünn besetzte Matrixstruktur verwendet werden. Wir wollen nun die aus (5.37) resultierende Matrix $\mathbf{A}$ etwas näher untersuchen. Zur Darstellung der Variablen als Vektor muß zuerst die zweidimensionale Beschreibung, realisiert durch die Koordinaten (i,j), in eine eindimensionale Adressierung transformieren. Definiere dazu ganzzahlige Werte k mit $0 \leq k < (n-1)^2$. Jedem k wird nun genau ein Gitterpunkt (i_k, j_k) zugeteilt. Ein übliches Vorgehen ist es, i_k als $k \; div \; (n-1)+1$ und j_k als $k \; mod \; (n-1)+1$ zu wählen.

Offensichtlich ist $A_{k,k} = 1$, weiterhin ist $A_{k,l} = -\frac{1}{4}$ für $|i_k - i_l| + |j_k - j_l| = 1$ und 0 sonst. Damit ergibt sich, daß $A_{k,l} = A_{l,k}$, d.h. die Matrix ist symmetrisch. $\mathbf{A}$ beinhaltet Nichtnullelemente in der Hauptdiagonalen, den beiden Nebendiagonalen und den beiden Diagonalen, die $(n-1)$ Elemente oberhalb bzw. unterhalb der Hauptdiagonalen liegen. Eine solche Matrix nennt man auch pentadiagonal. Hierbei ist zu beachten, daß wegen der Randpunkte, für die f konstante Werte liefert und die sich in b statt in A niederschlagen, auch die Diagonalen mit Nichtnulleinträgen an einigen Stellen durchaus Nulleinträge aufweisen, wie das folgende Beispiel für $n = 3$ mit den Einträgen für $\mathbf{A}_{2,1}$ und $\mathbf{A}_{1,2}$ illustriert.

	A				b
k	0	1	2	3	
(i_k, j_k)	(1,1)	(1,2)	(2,1)	(2,2)	
0	1	$-\frac{1}{4}$	$-\frac{1}{4}$	0	$-\frac{g_{1,1}}{4n^2} - \frac{1}{4}f_{0,1} - \frac{1}{4}f_{1,0}$
1	$-\frac{1}{4}$	1	0	$-\frac{1}{4}$	$-\frac{g_{1,2}}{4n^2} - \frac{1}{4}f_{0,2} - \frac{1}{4}f_{1,3}$
2	$-\frac{1}{4}$	0	1	$-\frac{1}{4}$	$-\frac{g_{2,1}}{4n^2} - \frac{1}{4}f_{2,0} - \frac{1}{4}f_{3,1}$
3	0	$-\frac{1}{4}$	$-\frac{1}{4}$	1	$-\frac{g_{2,2}}{4n^2} - \frac{1}{4}f_{3,0} - \frac{1}{4}f_{2,3}$

Wir wollen nun noch zeigen, daß **A** auch positiv definit ist. Sei dazu $\mathcal{N}(i,j)$ die Menge der Nachbarpunkte von (i,j), d.h. $(i-1,j),(i+1,j),(i,j-1)$ und $(i,j+1)$. Sei ferner $\mathcal{N}_1(i,j)$ die Menge der Nachbarpunkte von (i,j), die keine Randpunkte sind und $\mathcal{N}_0(i,j)$ die Menge der Nachbarpunkte, die Randpunkt sind. Letztere sind in **A** nicht enthalten. Die Multiplikation $f^T\mathbf{A}f$ liefert (wobei wir zur Vereinfachung f zweidimensional adressieren).

$$f^T(\mathbf{A}f) = \sum_{(i,j)} f_{i,j}\left(f_{i,j} - \frac{1}{4}\sum_{(k,l)\in\mathcal{N}_1(i,j)} f_{k,l}\right) = \frac{1}{4}\sum_{(i,j)}\left(4f_{i,j}^2 + \sum_{(k,l)\in\mathcal{N}_1(i,j)} -f_{i,j}f_{k,l}\right)$$

Offensichtlich beruht die Nachbarschaftsbeziehung auf Gegenseitigkeit, d.h. falls für ein Paar (i,j) und (k,l) gilt $(k,l)\in\mathcal{N}_1(i,j)$ so gilt auch $(i,j)\in\mathcal{N}_1(k,l)$. Da 4 Paare $(i,j),(k,l)\in\mathcal{N}(i,j)$ existieren, lassen sich die 4 Terme $f_{i,j}^2$ auf diese Paare verteilen, so daß die Summation über ein Paar benachbarter Punkte die obige Gleichung auf folgende Weise darstellt:

$$\frac{1}{4}\left(\sum_{(i,j),(k,l)\in\mathcal{N}_1(i,j)} f_{i,j}^2 + f_{k,l}^2 - 2f_{i,j}f_{k,l} + \sum_{(i,j),(k,l)\in\mathcal{N}_0(i,j)} f_{i,j}^2\right)$$

Jeder Term $f_{i,j}^2 + f_{k,l}^2 - 2f_{i,j}f_{k,l} = (f_{i,j} - f_{k,l})^2$ ist offensichtlich nicht negativ. Falls $f^TAf = 0$, so muß jeder Term der obigen Summe 0 sein, damit muß auch jedes $f_{i,j}$ gleich 0 sein und die Matrix ist positiv definit und symmetrisch.

Die Poisson Gleichung ist damit auch mit dem Verfahren des konjugierten Gradienten lösbar. Man kann auch zeigen, daß sowohl das Gauß-Seidel als auch das Jacobi Verfahren konvergieren. Da die Nichtdiagonalelemente kleiner gleich 0 und die Diagonalelemente positiv sind, konvergiert das Gauß-Seidel Verfahren nach Satz 5.5 nicht langsamer als das Jacobi Verfahren.

5.4.3 Mehrgitterverfahren

Durch die Gitterstruktur diskretisierter Differentialgleichungen benötigen iterative Verfahren wie das JOR oder SOR Verfahren oft sehr viele Iterationsschritte, um eine genügend genaue Lösung zu ermitteln. Dies kann man sich durch die folgende simplifizierte Betrachtung des Iterationsprozesses klarmachen. Bei der Iteration werden die Fehler gedämpft. Durch die Gitterstruktur werden zuerst und nach wenigen Iterationen Fehler zwischen benachbarten oder nahe beieinander liegenden Punkten gedämpft. Zum Abgleich zwischen physikalisch weit auseinander liegenden Punkten sind sehr viele Iterationen notwendig. So benötigt das JOR Verfahren bei einer Fünfpunktdiskretisierung $|i-k|+|j-l|$ Iterationen damit überhaupt eine Auswirkung des Wertes in Punkt (i,j) auf Punkt (k,l) sichtbar wird. Beim SOR Verfahren zumindest in Vorwärtsrichtung die Auswirkung schneller sichtbar, in Rückwärtsrichtung bleibt aber auch hier das Problem einer sehr langsamen Sichtbarkeit von Auswirkungen auf weit entfernte Punkte. Das SOR Verfahren kann verbessert werden indem die Richtung der Iteration von Zeit zu Zeit geändert wird, wodurch für manche Probleme die Konvergenzgeschwindigkeit deutlich erhöht wird. Als Alternative bieten sich für Gitterstrukturen sogenannte Mehrgitterverfahren an.

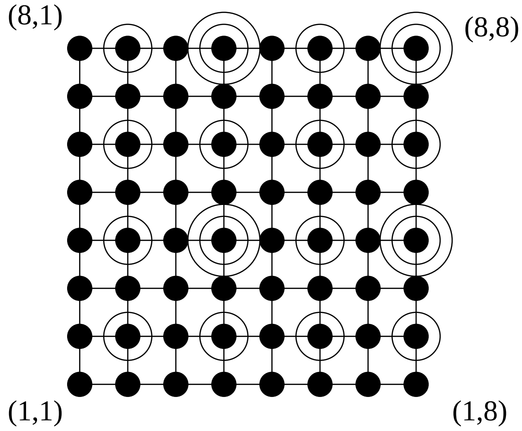


Abbildung 5.6: Mehrgitterstruktur.

Die zentrale Idee der Mehrgitterverfahren ist es, verschieden grobe Ebenen der Diskretisierung zu verwenden (verschiedene Gitter). Das feinste Gitter wird benötigt, um ein genügend genaues Resultat zu bestimmen. Die Resultate für gröbere Gitter sind zwar ungenauer, aber auch sehr viel schneller berechenbar. Globale Effekte, die zwischen weit voneinander entfernten Gitterpunkten auftreten, werden durch gröbere Gitter schneller erfaßt als durch feine Gitter. Feinere Gitter können dagegen lokale Effekte, die in gröberen Gittern nicht oder nur stark verfälscht auftreten, analysieren. Wir wollen hier zweidimensionale Probleme untersuchen, auch wenn die Verfahren im Prinzip auf beliebig viele Dimensionen anwendbar sind. Sei $G_0 = \{(i,j) | 1 \leq i,j \leq n\}$ das feinste Gitter und $n = 2^K$. Gröbere Gitter $G_1, \ldots, G_D$ $(D < K)$ werden wie folgt definiert

$$G_d = \{(i,j) \in G_0 | i,j \text{ sind Vielfache von } 2^d\} \qquad d = 1, \ldots, D$$

Die Lage der einzelnen Gitterpunkte ist in Abbildung 5.6 für ein 8×8 Gitter dargestellt. Die Punkte beschreiben Gitterpunkte im feinsten Gitter, kleine Kreise Punkte im mittleren Gitter und große Kreise Punkte im gröbsten Gitter. Sei $x^d_{i,j}$ der Wert von $x_{i,j}$ im Gitter d. Es gibt drei Ebenen, auf denen Abhängigkeiten zwischen den Variablen bestimmt werden müssen, dies erfordert die folgenden Berechnungen.

- Iterationen in einem festen Gitter d, der Vektor $\mathbf{x}^d(t)$ wird aus $\mathbf{x}^d(t-1)$ berechnet, wobei durch die Diskretisierung ein linearer oder auch nichtlinearer Zusammenhang zwischen benachbarten Gitterpunkten existiert. Zur Iteration wird ein klassisches Iterationsverfahren verwendet. Im linearen Fall wird meist das SOR benutzt.

- Übertragung der Ergebnisse auf ein feineres Gitter (Interpolation) (von d nach $d-1$). $x^{d-1}_{i,j}$ wird nach einer lokalen Berechnungsvorschrift aus den Werten $x^d_{k,l}$ gewonnen. Eine mögliche Berechnungsvorschrift die Gewichtung der $x^d_{k,l}$ in Abhängigkeit vom Abstand (i,j) von (k,l).

- Übertragung der Ergebnisse auf ein gröberes Gitter (Projektion) (von $d-1$ nach d). Die Werte $x^d_{i,j}$ werden aus den $x^{d-1}_{k,l}$ gewonnen. Die einfachste Form der Übertragung ist $x^d_{i,j} = x^{d-1}_{i,j}$, komplexere Formen benutzen wieder die gewichtete Summe über die Nachbargitterpunkte.

Mehrgitterverfahren führen Iterationen für die Gleichungssysteme der verschiedenen Ebenen aus und übertragen die Ergebnisse auf die feineren und gröberen Gitter. Wir wollen hier nicht auf die Einzelheiten der Verfahren eingehen, sondern stattdessen die parallele Implementierung etwas näher untersuchen. Es gibt zwei grundsätzliche Möglichkeiten der Parallelisierung. Im ersten Fall werden nur Iterationen in einem Gitter zu einem Zeitpunkt durchgeführt. Im zweiten Fall werden Iterationen in den verschiedenen Gittern parallel ausgeführt. Praktisch wird hauptsächlich der erste Ansatz eingesetzt.

Beginnen wir mit dem ersten Vorgehen. Eine naheliegende Implementierung wäre die direkte Abbildung der Gitterpunkte auf die Prozessortopologie und damit die Verwendung eines zweidimensionalen Arrays. Nehmen wir an, daß ein $n \times n$ Prozessorarray verfügbar ist. Prozessor i, j berechnet $x_{i,j}^0$ und muß während der Iterationen im Gitter G_0 nur mit den direkt verbundenen Nachbarprozessoren kommunizieren. Wenn wir nun aber Iterationen auf einem gröberen Gitter G_d untersuchen, so zeigt sich,daß benachbarte Gitterpunkte nicht mehr auf benachbarten Prozessoren berechnet werden und der Kommunikationsaufwand überproportional ansteigt und dafür sorgt, daß Iterationen in den gröberen Gittern sehr ineffizient werden. Damit ist ein zweidimensionales Prozessorarray für Mehrgitterverfahren nur geeignet, wenn relativ wenige Prozessoren zur Verfügung stehen und damit die Iterationen auf den gröberen Gittern Kommunikation zwischen "fast benachbarten" Prozessoren erfordern, für die feineren Gittern werden dann entsprechend viele Funktionen auf einem Prozessor berechnet. Diese Aussage mag auf den ersten Blick überraschen, da zweidimensionale Arrays die optimale Topologie für diskretisierte zweidimensionale Probleme und klassische iterative Verfahren waren.

Untersuchen wir nun die Implementierung auf einem Hyperwürfel, indem wir ein $n \times n$ Prozessorarray in einen Hyperwürfel mit n^2 Prozessoren einbetten und dazu den RGC (siehe Abschnitt 2.5.4) verwenden. Die folgenden Eigenschaften resultieren aus der Einbettung.

- Die Gitterpunkte der j-ten Spalte $C_j = \{(i, j) \in G_0 | 1 \leq i \leq n\}$ $(1 \leq j \leq n)$ werden auf einen kleineren Hyperwürfel mit n Prozessoren abgebildet. Das selbe gilt für die Prozessoren der i-ten Zeile $R_i = \{(i, j) \in G_0 | 1 \leq j \leq n\}$ $(1 \leq i \leq n)$. Unterwürfel, auf die beliebige Zeilen R_i und Spalten C_j abgebildet werden, haben keine gemeinsamen Leitungen. Die Einbettung jeder Zeile und Spalte ist durch die Einbettung eines linearen Arrays unter Verwendung des RGCs beschrieben.

- Wird ein lineares Array unter Verwendung des RGCs in einen Hyperwürfel eingebettet, so entspricht die physikalische Distanz zwischen Prozessor i und $i + 2^d$ 1 falls $d = 0$ und 2 für alle anderen d. Die Kommunikation zwischen allen Prozessoren i und $i + 2^d$ kann simultan in zwei Schritten für alle Prozessoren i durchgeführt werden (d.h. die einzelnen Verbindungen benutzen keine gemeinsamen Leitungen in einem Schritt).

Damit sind Mehrgitterverfahren sehr effizient auf Hyperwürfeln implementierbar, die Kommunikation für eine Iteration in einem Gitter G_d geht über maximal einen Zwischenknoten und ist für alle Prozessoren simultan ausführbar. Auch hier gilt wieder, daß in der Praxis die Zahl der Gitterpunkte in den feineren Gittern die Anzahl real verfügbarer Prozessoren um Größenordnungen übersteigt und somit mehrere Funktionen der feineren Gitter auf einem Prozessor berechnet werden.

Betrachten wir nun die zweite Implementierungsmöglichkeit, nämlich die parallele Ausführung der Iterationen für alle Gitter. Die Struktur der Abhängigkeiten entspricht in der zweidimensionalen Ebene einem Gitter und im dreidimensionalem Raum der Verbindung zwischen den Gittern. Eine Architektur, die diese Struktur als Untergraphen beinhaltet, ist ein Pyramiden Netz (siehe Abb. 5.7). Pyramiden Netze lassen sich in Hyperwürfel einbetten, so daß der Hyperwürfel auch geeignet ist für eine parallele Implementierung der Gitteriterationen.

Die hier betrachteten Gitter waren alle von vollständig regelmäßiger Struktur und ließen sich deshalb sehr gut auf regelmäßige Rechnertopologien abbilden. Sehr viel schwieriger wird

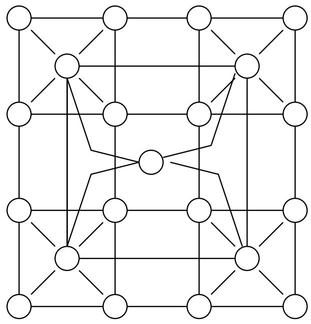


Abbildung 5.7: Pyramiden Netz.

die Abbildung, wenn unregelmäßige Gitterstrukturen, die in der Praxis durchaus vorkommen, auftreten, das zu lösende Problem entspricht dann wieder der Einbettung des Abhängigkeitsgraphen in den Topologiegraphen.

5.5 Literaturhinweise

Ein großer Teil der vorgestellten Algorithmen ist in [4] zu finden. Parallele numerische Algorithmen zur Analyse von Gleichungssystemen sind auch Inhalt der Lehrbücher [8, 14]. Mehr der mathematischen Seite sind die Lehrbücher [12, 3, 15, 22, 36] gewidmet. Einen Überblick über die Lösung linearer Gleichungssysteme mit spärlich besetzter Koeffizientenmatrix findet man in [23], Ergebnisse für dicht besetzte Matrizen bietet [16]. Die parallele Cholesky Faktorisierung für dünn besetzte Matrizen wird in [8, 14, 39] behandelt. Parallele Mehrgitterverfahren und die Analyse von Gitterstrukturen auf Multiprozessoren werden in [10] behandelt. Das Beispiel für die partielle Differentialgleichung wurde aus [33] entnommen und leicht modifiziert. Der Vergleich der Leistungsfähigkeit verschiedener Parallelrechnerarchitekturen zur direkten Lösung linearer Gleichungssystem ist [9] entnommen. Die parallele Implementierung des Verfahrens des konjugierten Gradienten wird in [21] ausführlich analysiert. Der asynchrone Algorithmus zur Analyse von Markov Ketten ist in [28] zu finden.

Kapitel 6

Asynchrone Algorithmen und verteilte Berechnungen

Bei den bisher betrachteten parallelen Verfahren können die einzelnen Prozesse nicht beliebig fortschreiten, sondern müssen nach einigen Rechenschritten typischerweise auf neue Informationen warten, die von anderen Prozessen geliefert werden. Diese Synchronisationspunkte hemmen den Rechenfortschritt, erleichtern dafür jedoch die Korrektheit und Terminierung eines parallelen Programms sicherzustellen. Die Grundidee asynchroner Algorithmen ist es, einzelne Prozesse nicht aufeinander warten zu lassen, sondern die Berechnung anhand der lokal verfügbaren Information fortschreiten zu lassen. Es ist klar, daß ein solches Vorgehen nicht für beliebige Problemstellungen zum Ziel führt, aber ein prinzipiell reizvoller und bei iterativen Verfahren auch naheliegender Ansatz ist. In diesem Kapitel werden wir zunächst anhand eines Beispiels für eine Fixpunktberechnung einige Betrachtungen zur Konvergenz eines asynchronen Verfahrens anstellen und anschließend asynchrone Ansätze zur Lösung von linearen Gleichungssystemen betrachten.

6.0.1 Ein einfaches Beispiel

Wir betrachten ein sehr einfaches Beispiel zur Analyse des Verhaltens eines synchronen und eines asynchronen Algorithmus für ein Fixpunktproblem. Gesucht wird ein Vektor $\mathbf{x} = (x_1, x_2)$, der die Gleichung $\mathbf{x} = \mathbf{xA}$ mit $\mathbf{A} = \begin{pmatrix} a & b \\ b & a \end{pmatrix}$ erfüllt. Zur Lösung verwenden wir die Iteration

$$\begin{aligned} x_1 &= ax_1 + bx_2 \\ x_2 &= bx_1 + ax_2 \end{aligned} \tag{6.1}$$

ausgehend von Initialwerten $x_i = x_i(0)$. Die Variablen x_1 und x_2 werden auf den Prozessoren $P1$ und $P2$ berechnet. Bei der Analyse der Algorithmen zur Lösung des Problems werden wir stark vereinfachte Annahmen treffen. Die Berechnung einer Funktion benötige konstant eine Zeiteinheit und die Übertragung einer Nachricht $D \geq 1$ Zeiteinheiten. Im synchronen Algorithmus haben wir unter diesen Annahmen folgenden Ablauf der Iteration

$$\begin{aligned} x_1(t+1) &= ax_1(t-D) + bx_2(t-D) \\ x_2(t+1) &= bx_1(t-D) + ax_2(t-D) \\ &\text{mit } x_i(t) = x_i(0) \text{ falls } -D \leq t < 0,\ i = 1,2 \end{aligned} \tag{6.2}$$

Im asynchronen Algorithmus führt jeder Prozessor soviele Iterationen wie möglich durch und wir erhalten

$$
\begin{aligned}
x_1(t+1) &= ax_1(t) + bx_2(t-D) \\
x_2(t+1) &= bx_1(t-D) + ax_2(t)
\end{aligned}
\tag{6.3}
$$

$$\text{mit } x_i(t) = x_i(0) \text{ falls } -D \leq t < 0,\ i = 1,2$$

Wir wollen nun die Konvergenz der Verfahren betrachten und nehmen an, daß $|a| + |b| < 1$. In diesem Fall konvergiert das synchrone Verfahren gegen den Lösungsvektor $(0,0)$ wie man sich leicht überlegen kann. Auch die Konvergenz des asynchronen Verfahrens ist gesichert. Sei nun $\max(|x_1(0)|, |x_2(0)|) \leq r$, dann ist offensichtlich $\max(|x_1(1)|, |x_2(1)|) \leq r(|a| + |b|)$, $\max(|x_1(D+1)|, |x_2(D+1)|) \leq r(|a| + |b|)^2$ usw. Diese Sequenz konvergiert unter der obigen Bedingung gegen den Nullvektor. Wir wollen nun die Konvergenzgeschwindigkeit untersuchen. Sei dazu $\rho > 0$ so definiert, daß die folgende Ungleichung erfüllt ist

$$|a|\rho^{-D} + |b|\rho^{-D} \leq \rho \ \text{ bzw. } \ (|a| + |b|)^{1/(D+1)} \leq \rho. \tag{6.4}$$

Dann gilt

$$|x_i(t)| \leq C\rho^t \ t = 0, 1 \ldots \ \text{ und } C = \max(|x_1(0)|, |x_2(0)|) \tag{6.5}$$

Den Beweis für die Aussage kann man per Induktion führen. Offensichtlich gilt die Aussage für $t \leq 0$. Wir nehmen nun an, daß sie für ein t' gilt und bekommen die folgenden Ungleichungsketten

$$x_1(t'+1) \leq |a||x_1(t'-D)| + |b||x_2(t'-D)| \leq (|a| + |b|)C\rho^{t'-D} \leq C\rho^{t'+1} \text{ undÊ}$$

$$x_2(t'+1) \leq |b||x_1(t'-D)| + |a||x_2(t'-D)| \leq (|a| + |b|)C\rho^{t'-D} \leq C\rho^{t'+1} \ .$$

Damit gilt die Ungleichung auch für alle $t > 0$. Wir wählen nun das kleinste erlaubte ρ als

$$\rho_s = (|a| + |b|)^{1/(D+1)} \tag{6.6}$$

Man kann sich überlegen, daß für jedes $\rho < \rho_s$ initiale Vektoren existieren, so daß (6.5) nicht mehr erfüllt ist. Die Konvergenz des synchronen Verfahrens ist damit geometrisch mit Parameter ρ_s.

Falls ρ der Ungleichung

$$|a| + |b|\rho^{-D} \leq \rho \tag{6.7}$$

genügt, ist Ungleichung (6.5) für das asynchrone Verfahren erfüllt. Auch dies ist wieder über Induktion zu zeigen. Offensichtlich gilt die Aussage für $t \leq 0$. Sei die Ungleichung für ein t' erfüllt dann gilt

$$x_1(t'+1) \leq |a||x_1(t')| + |b||x_2(t'-D)| \leq (|a| + |b|\rho^{-D})C\rho^{t'} \leq C\rho^{t'+1}$$

und in analoger Weise eine ähnliche Ungleichung für $x_2(t'+1)$. Der minimale Wert für ρ_a ist die eindeutige Lösung der Gleichung

$$|a| + |b|\rho_a^{-D} = \rho_a. \tag{6.8}$$

Für alle erlaubten Werte von D, a und b ist $\rho_a \leq \rho_s$. Damit ist die Konvergenz des asynchronen Verfahren zumindest genauso schnell wie die des synchronen Verfahrens und für viele Parametersätze sogar wesentlich schneller. Die Analyse und der Test vieler asynchroner Verfahren zeigt, daß asynchrone Verfahren, falls sie konvergieren, schneller konvergieren als vergleichbare synchrone Verfahren. Andererseits werden von asynchronen Verfahren mehr Nachrichten als von synchronen Verfahren übertragen. Dies liegt einfach darin begründet, daß asynchrone Verfahren mehr Iterationen durchführen und am Ende jeder Iteration i.a. eine Nachricht versenden. Zusätzliche Nachrichtenübertragungen können auch zu einer höheren Belastung der Verbindungen und damit längeren Übertragungszeiten führen. Wir werden diesen Aspekt allerdings nicht in die Analyse einbeziehen.

Die Konvergenzanalyse asynchroner Algorithmen ist meist viel schwieriger als die Konvergenzanalyse synchroner Algorithmen, es gibt aber eine Reihe relevanter Probleme für die auch die Konvergenz asynchroner Verfahren nachgewiesen wurde.

6.0.2 Asynchrone Verfahren

Die Grundlagen asynchrone Algorithmen wurden bereits erläutert und es hatte sich gezeigt, daß für Multiprozessorsysteme mit verteiltem Speicher asynchrone Algorithmen gegenüber synchronen Algorithmen Vorteile haben. Dies liegt insbesondere daran, daß aufwendige Synchronisationen vermieden werden und die Geschwindigkeit nicht durch den jeweils langsamsten Prozessor bestimmt wird. Dadurch sind mit asynchronen Algorithmen deutlich mehr Iterationen durchführbar als mit synchronen, auch wenn der Algorithmus in homogene Prozesse zerlegbar ist und die Architektur aus homogenen Prozessoren besteht. Diesen Vorteilen asynchroner Algorithmen stehen aber auch gravierende Nachteile gegenüber. Insbesondere sind die Bedingungen für die Korrektheit asynchroner Algorithmen restriktiver. Während für synchrone parallele Algorithmen eine partielle Ordnung der einzelnen Schritte durch die Synchronisationspunkte gegeben ist und viele synchrone parallele Algorithmen direkt aus sequentiellen Algorithmen abgeleitet wurden, ist dies bei asynchronen Algorithmen nicht der Fall. Insofern ist auch die verfügbare Literatur über asynchrone numerische Algorithmen recht eingeschränkt und kann nicht auf die lange Historie sequentieller numerischer Algorithmen zurückgreifen. Wir werden uns in diesem Abschnitt mit einigen wenigen asynchronen Ansätzen zur Lösung linearer Gleichungssysteme beschäftigen.

Asynchrone Algorithmen zur Lösung linearer Gleichungssysteme sind klarerweise iterative Algorithmen. Beim einem synchronen iterativen Algorithmus wurde eine Komponente des Lösungsvektors wie folgt berechnet.

$$x_i(t+1) = f_i(x_1(t), \ldots, x_n(t))$$

Bei einer asynchronen Iteration werden Werte aus unterschiedlichen, vorhergehenden Rechenschritten verwendet. Um den jeweiligen Berechnungszeitpunkt (das "Alter") eines Wertes $x_j(t)$ zu erfassen, wird die Notation folgendermaßen erweitert.

$$x_i(t+1) = f_i(x_1(\tau_1^i), \ldots, x_{i-1}(\tau_{i-1}^i), x_i(t), x_{i-1}(\tau_{i+1}^i), \ldots, x_{i-1}(\tau_n^i))$$

Der Wert τ_j^i beschreibt den Zeitpunkt, zu dem der Wert von x_j im Prozessor der x_i berechnet gehört. Offensichtlich gilt $0 \leq \tau_i^j \leq t$. $t - \tau_i^j$ ist das Alter von x_j im Prozessor der x_i berechnet und umfaßt die Kommunikationszeit zwischen beiden Prozessoren. Sei T^i die Menge der Zeitpunkte, zu denen die Variable i einen neuen Wert zugewiesen bekommt. Bei den folgenden Überlegungen nehmen wir an, daß jede Komponente in jedem Prozessor unendlich oft einen Wert bekommt, wenn wir den Algorithmus unendlich lange laufen lassen (d.h. die Kardinalität von T^i ist unendlich) . Formal bedeutet dies $\lim_{t\to\infty} \tau_j^i(t) = \infty$ für alle j. Oder etwas anders ausgedrückt, es existiert ein t_2, so daß gilt

$$\tau_j^i(t) \geq t_1, \quad \forall i, j, t_1 \text{ und } t \geq t_2 \tag{6.9}$$

Die Differenz $t - \tau_j^i(t)$ kann natürlich für wachsendes t trotzdem gegen unendlich gehen. Betrachten wir nun wieder unser lineares Gleichungssystem $\mathbf{x}^* = \mathbf{A}\mathbf{x}^* + \mathbf{b}$ und die asynchrone Iterationsvorschrift

$$x_i(t+1) = \sum_{j=1}^{n} A_{i,j} x_j(\tau_j^i(t)) + b_i \quad t \in T^i \tag{6.10}$$

Für $t \notin T^i$ gilt $x_i(t+1) = x_i(t)$. Die Iteration konvergiert unter den obigen Bedingungen, falls $\rho(|\mathbf{A}|) < 1$. $\rho(|\mathbf{A}|)$ ist der Spektralradius der Matrix $|\mathbf{A}|$, in der jedes Element von $\mathbf{A}$ durch den Absolutwert ersetzt wird. Es ist zu beachten, daß die Bedingung $\rho(\mathbf{A}) < 1$ nicht ausreicht, wenn $\rho(|\mathbf{A}|) \geq 1$ ist. In diesem Fall hängt die Konvergenz oder Divergenz von der gesamten Matrix $\mathbf{A}$ und nicht allein vom Spektralradius ab.

Wir werden nun die Voraussetzungen wieder etwas einschränken, indem wir nicht mehr ein vollständig asynchrones Verhalten, sondern nur ein partiell asynchrones Verhalten untersuchen. Dazu existiere eine Konstante β , so daß

- jeder Prozessor seine Funktionsberechnungen mindesten einmal in einem Intervall der Länge β ausführt.
- die Information die von einem Prozessor zur Berechnung verwendet wird maximal β Zeiteinheiten alt ist.

Formal können partiell asynchrone Algorithmen genauso wie asynchrone Algorithmen beschrieben werden. Es gilt nun aber $t - \beta \leq \tau_j^i(t) \leq t$ für alle i, j. Die Verwendung von t bei der Beschreibung der Algorithmen bedeutet natürlich nicht, daß jeder Prozessor diese Zeit vorliegen hat (was in verteilten Systemen auch schwierig oder unmöglich ist). t ist vielmehr die absolute Zeit eines externen Beobachters. In vielen realen Systemen kann eine maximale Zeitschranke β garantiert bzw. durch Synchronisation herbeigeführt werden.

Als Beispiel für einen komplett und einen partiell asynchronen Algorithmus betrachten die Berechnung der stationären Verteilung für eine Markov Kette. Sei $\mathbf{P}$ eine irreduzible und aperiodische stochastische Matrix (siehe Anhang). Gesucht ist $\pi^* = \pi^*\mathbf{P}$. Beginnen wir mit einem komplett asynchronen Algorithmus. $\mathbf{P}$ wird wie folgt dargestellt

$$\mathbf{P} = \begin{pmatrix} p_{11} & \mathbf{a} \\ \mathbf{b} & \tilde{\mathbf{P}} \end{pmatrix}$$

$\mathbf{a}$ und $\mathbf{b}$ sind $n-1$-dimensionale Zeilen/Spaltenvektoren, p_{11} ist ein Skalar und $\tilde{\mathbf{P}}$ eine $n-1 \times n-1$ Matrix. Die Iterationsvorschrift $\pi(t+1) = \pi(t)P$ kann nun modifiziert werden zu

$$\tilde{\pi}(t+1) = \tilde{\pi}(t)\tilde{\mathbf{P}} + \pi_1(0)\mathbf{a} \tag{6.11}$$

Wobei $\tilde{\pi}$ ein $n-1$-dimensionaler Zeilenvektor ist und $\tilde{\pi}_i(0) = \pi_{i-1}(0)$ für $i = 2, \ldots, n$. Für den initialen Vektor $\pi(0)$ sei nur vorausgesetzt, daß $\pi_1(0) > 0$. Unter den obigen Voraussetzungen für $\mathbf{P}$ ist leicht zu zeigen, daß $\lim_{k\to\infty} \tilde{\mathbf{P}}^k = \mathbf{0}$. Damit ist $\rho(\tilde{\mathbf{P}}) < 1$ und da $\tilde{\mathbf{P}}$ nur nichtnegative Elemente enthält, gilt offensichtlich auch $\rho(|\tilde{\mathbf{P}}|) = \rho(\tilde{\mathbf{P}}) < 1$, womit die Konvergenz der vollständig asynchronen Iteration gesichert ist. Der Vektor $\tilde{\pi}(t)$ konvergiert für $t \to \infty$ gegen $\tilde{\pi}^*$, die stationäre Lösung in Abhängigkeit vom gewählten $\pi_1(0)$. π^* erhält man durch Normierung, d.h. $\pi_i^* = \tilde{\pi}_i^*/(\sum_{j=1}^{n-1} \tilde{\pi}_j^* + \pi_1(0))$ für $i > 1$ und $\pi_1^* = \pi_1(0)/(\sum_{j=1}^{n-1} \tilde{\pi}_j^* + \pi_1(0))$. Die Wahl von $\pi_1(0)$ ist bedeutsam für die Konvergenzgeschwindigkeit des Algorithmus, eine ungünstige Wahl kann unter Umständen eine relativ langsame Konvergenz bedeuten und zu numerischen Problemen führen.

Wird die Matrix $\mathbf{P}$ direkt in der Iteration verwendet, so kann die Konvergenz der asynchronen Iteration, im Gegensatz zu einer synchronen Iteration, nicht garantiert werden, da $\rho(\mathbf{P}) = \rho(|\mathbf{P}|) = 1$ gilt. Überraschenderweise kann man trotzdem einen partiell asynchronen Algorithmus finden und dessen Konvergenz beweisen. Es sind dazu einige nicht weiter einschränkende Annahmen notwendig, die zuerst vorgestellt werden. Es muß ein $i^* \in \{1, \ldots, n\}$ existieren, so daß

$$
\begin{aligned}
&P_{i^* i^*} > 0, \\
&\tau_{i^*}^{i^*}(t) = t, \qquad\qquad (6.12) \\
&\pi_{i^*}(0) > 0.
\end{aligned}
$$

Die zweite Annahme ist immer erfüllt in unserem Modell, da der Prozessor natürlich den aktuellen Wert einer Variablen die er lokal berechnet kennt. Berechnet mehr als ein Prozessor eine Variable, so muß $\tau_i^i(t)$ nicht zwangsläufig gleich t sein. Für den Algorithmus muß also vorausgesetzt werden, daß zumindest die Variable i^* auf einem Prozessor berechnet wird. Die dritte Annahme besagt, daß der Startvektor so gewählt werden muß, daß die Komponente i^* positiv ist. Da der Startvektor beliebig vorgegeben werden kann, ist auch diese Annahme nicht einschränkend. Im allgemeinen wird der Startvektor als Gleichverteilung gewählt, sofern nicht weitergehende Informationen vorliegen. Bleibt die erste Annahme, die aussagt, daß mindestens ein Diagonalelement größer 0 in der Matrix existieren muß. Dadurch bedingt ist die Matrix aperiodisch, andererseits muß eine aperiodische Matrix aber nicht zwangsläufig ein positives Diagonalelement beinhalten. Dies ist eine wirkliche Einschränkung, da offensichtlich Matrizen existieren, die grundsätzlich analysierbar und lösbar sind, aber mit dem Algorithmus nicht behandelt werden können. Es gibt aber eine sehr einfache Matrixtransformation, die dafür sorgt, daß die Diagonalelemente größer 0 werden und die stationäre Verteilung unverändert bleibt. Man transformiert dafür $\mathbf{P}$ in $\bar{\mathbf{P}} = \alpha\mathbf{P} + (1-\alpha)\mathbf{I}$ mit $0 < \alpha < 1$. $\bar{\mathbf{P}}$ ist eine stochastische Matrix und es gilt $\pi^*\bar{\mathbf{P}} = \pi^*(\alpha\mathbf{P} + (1-\alpha)\mathbf{I}) = \pi^*$, womit π^* auch die stationäre Lösung für den modifizierten Prozeß ist. Damit kann jede Matrix so transformiert werden, daß obige Bedingungen erfüllt sind. Zusätzlich benötigen wir noch die Bedingung für partiell asynchrone Algorithmen, nämlich $t - \tau_i^j(t) \leq \beta < \infty$ für ein beliebiges, aber endliches β.

Die Iterationsvorschrift für die einzelnen Komponenten lautet

$$
\pi_i(t+1) = \begin{cases} \sum\limits_{j=1}^{n} \pi_j(\tau_j^i(t)) P_{j,i} & \text{falls } t \in T^i \\ \pi_i(t) & \text{falls } t \notin T^i \end{cases} \qquad (6.13)
$$

sie unterscheidet nach Zeitpunkten $t \in T^i$, zu denen $\pi_i(t+1)$ aktualisiert werden muß, und allen anderen, an denen der Wert lediglich bestehen bleibt. Wir wollen hier nicht im einzelnen die Konvergenz des Verfahrens nachweisen. Es sei nur erwähnt, daß die Methode unter den vorgestellten Voraussetzungen gegen $c\pi^*$ konvergiert, wobei c eine positive endliche Konstante ist. Die endgültige Lösung wird durch Normierung des erreichten Lösungsvektors gewonnen. Wird in einer Implementierung die Iteration gestoppt, wenn der maximale Absolutwert der Residuen kleiner ϵ ist, d.h. wenn

$$
\frac{\max_i(\sum_{j=1}^{n} \pi_j(t) P_{j,i})}{\sum_{j=1}^{n} \pi_j(t)} < \epsilon
$$

gilt, so müssen im Algorithmus Synchronisationspunkte eingefügt werden, zu denen die Prozessoren stoppen und die Residuen Berechnung ausgeführt wird. Dies macht natürlich die Vorteile des asynchronen Ablaufs zunichte, auch wenn größere Abstände zwischen den Synchronisationspunkten gewählt werden. Als Alternative bietet sich an, die Iteration abzubrechen, wenn sich die Werte der einzelnen Vektorkomponenten von einem zum nächsten Iterationsschritt nur noch um einen kleinen Wert ϵ ändern. Dieses Kriterium kann lokal von den Prozessoren getestet werden, ist aber i.a. weniger genau als die Residuenberechnung.

Wird der beschriebene Algorithmus auf einem Einprozessorsystem implementiert, so entspricht er dem Gauß-Seidel Verfahren. In der Originalarbeit werden eine Reihe von Beispielläufen auf einer simulierten Architektur mit geteiltem Speicher dokumentiert und ein Speed Up von $20 - 30\%$ Prozent gegenüber einem synchronen Algorithmus angegeben. Die Resultate sind allerdings nur bedingt aussagekräftig, da eine Matrix der Größe 128×128 analysiert wird, was weit unterhalb realistischer und interessanter Größenordnungen liegt.

Kapitel 7

Algorithmen zur Lösung nichtlinearer Gleichungssysteme

Nichtlineare Gleichungssysteme ist als Begriff zur Charakterisierung einer Problemklasse dann hilfreich, wenn man sich nicht damit befassen möchte, weil hierdurch eine Reihe recht unterschiedlicher Probleme mit unterschiedlichen zugehörigen Lösungsverfahren zusammengefaßt werden. Der interessierte Leser sei an dieser Stelle auf [4], Kapitel 3, S. 180-290 verwiesen.

Kapitel 8

Optimierungsprobleme

Neben der Lösung linearer Gleichungssysteme sind Optimierungsprobleme eine wichtige Problemklasse im technischen Bereich und auch in ökonomischen und ökologischen Modellen treten Optimierungsprobleme auf. Im Rahmen der Vorlesung können wir nur einen sehr kleinen Bereich des großen Gebiets der Optimierung betrachten. Es werden einige ausgewählte Probleme und effiziente parallele Algorithmen zur Analyse dieser Probleme vorgestellt.

Eine erst Form der Optimierung ohne Nebendingungen haben wir bereits kennengelernt als das Verfahren des konjugierten Gradienten zur Lösung linearer Gleichungssysteme eingeführt wurde. Das Verfahren des konjugierten Gradienten bestimmt das Minimum der Funktion

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A}\mathbf{x} - \mathbf{x}^T \mathbf{b} \ .$$

Die erste Ableitung dieser Funktion ist gerade $\mathbf{A}\mathbf{x}-\mathbf{b}$. Nullsetzen der Ableitung liefert das Optimum. Das Verfahren des konjugierten Gradienten und Abarten davon sind deshalb auch Lösung von Optimierungsproblemen ohne Nebenbedingungen geeignet. Überraschen mag auf den ersten Blick, daß zwar lineare Optimierungsprobleme im folgenden behandelt werden, die übliche Lösungsmethode aus dem sequentiellen Bereich, der Simplexalgorithmus, aber nicht erwähnt wird. Dies hat einen sehr einfachen Grund. So ist der Simplexalgorithmus zwar die allgemeinste und oftmals beste Methode zur Lösung linearer Optimierungsprobleme im sequentiellen Bereich. Leider gilt dies aber nicht für die parallele Implementierung. Es gibt, zumindest meines Wissens nach, keine parallele Variante des Simplexalgorithmus, die wirklich signifikant schneller als die sequentielle Variante ist. Dies liegt im Pivotschritt des Algorithmus begründet, der sich bisher allen Parallelisierungsversuchen widersetzt.

Beginnen werden wir mit dem Problem der kürzesten Wege. Für diesen Problemkreis gibt es einige sehr gut parallelisierbare Algorithmen. Es wird sich zeigen, daß die meisten vorgestellten Lösungsansätze sehr ähnlich zum Jacobi- bzw. Gauß-Seidel Verfahren sind. Zum Abschluß des Kapitels wird kurz das allgemeinere Problem der optimalen Klammerung bei der Matrizen-Multiplikation behandelt.

8.1 Kürzeste Wege und dynamische Programmierung

Das Problem des kürzesten Weges ist ein altbekanntes und bedeutsames kombinatorisches Optimierungsproblem. Ziel ist es, in einem gerichteten Graphen einen möglichst kurzen Weg von jedem Knoten i zum Ausgangsknoten 1 zu finden. Als Verallgemeinerung kann man auch den Fall betrachten, daß kürzeste Wege zwischen allen Knoten im Graphen gesucht sind.

Die Suche nach dem kürzesten Weg tritt in zahlreichen Anwendungen auf. Als Beispiele seien hier nur Wegewahlprobleme in Rechnernetzen, Vekehrsflußlenkungen, optimale Kontrollprobleme oder die heuristische Suche genannt. Viele der genannten Probleme erfordern die Suche des kürzesten Pfades zwischen einer sehr großen Anzahl von Knoten und sind deshalb von einer solchen Komplexität, daß Hochleistungsrechner und damit insbesondere auch parallele Architekturen zur Analyse eingesetzt werden müssen. Wie auch die Lösung linearer Gleichungssysteme tritt das Problem der Suche nach dem kürzesten Weg in vielen komplexeren Optimierungsproblemen als ein Schritt der Optimierung auf. Die zugehörige Routine wird sehr oft mit variierenden Parametern aufgerufen, so daß eine schnelle Ausführung sehr wichtig für die Laufzeit des Gesamtalgorithmus ist.

Das Problem des kürzesten Weges ist nur eine spezielle Formulierung einer größeren Klasse von Optimierungsproblemen, die man unter dem Namen "dynamische Programmierung" zusammenfaßt. Ziel der dynamischen Programmierung ist es, eine Menge sequentieller Entscheidungen zu treffen, die jeweils spezifische Kosten verursachen und zukünftige Entscheidungen (bzw. deren Kosten) beeinflußen. Am Ende der Entscheidungskette soll ein vorgegebenes Ziel mit möglichst geringen Kosten erreicht werden. Die Problemstellungen der dynamischen Programmierung umfassen dabei auch Probleme, bei denen Kosten für Entscheidungen nicht exakt bekannt sind, sondern durch eine Wahrscheinlichkeitsverteilung gegeben sind, die auch von vergangen Entscheidungen beeinflußt werden kann. Letzteren Fall werden wir aus Zeitgründen hier nicht untersuchen und deshalb im Prinzip beim Problem des kürzesten Weges mit deterministisch vorgegeben Weglängen (bzw. Kosten) bleiben. Diese Problemstellung werden wir später etwas weiter ausdehnen.

Beginnen wir mit einer etwas formaleren Problemdarstellung. Wir gehen von einem gerichteten Graphen mit n Knoten, die von 1 bis n numeriert werden, aus. Sei $\mathcal{A}(i)$ die Menge der Knoten j, für die eine gerichtete Kante (i, j) existiert. Knoten 1 betrachten wir als den Quellknoten, zu dem der kürzeste Weg gefunden werden soll. Sei $\mathcal{A}(1)$ leer, d.h. von Knoten 1 gehen keine Kanten aus. Für jede Kante (i, j) sei $a_{i,j}$ die Länge des Weges von i nach j über die Kante (i, j). Die Länge eines Weges $(i, i_1, i_2, \ldots, i_k, 1)$ ist dann gegeben durch $a_{i,i_1} + a_{i_1,i_2} + \ldots + a_{i_k,1}$. Das Problem ist es, den kürzesten Weg zwischen i und 1 zu finden. Der kürzeste Weg ist dadurch charakterisiert, daß die obige Summe minimiert wird. Sinnvollerweise nehmen wir an, daß zwischen jedem Knoten $i = 2, \ldots, n$ und Knoten 1 ein Pfad existiert. Weiterhin nehmen wir an, daß jeder Zyklus im Graphen eine positive Länge hat. Dies ist selbstverständlich, wenn die $a_{i,j}$ als Weglängen interpretiert werden, stellen sie aber allgemein Kosten dar, so kann man sich auch negative Werte vorstellen, solange die vorherige Einschränkung bestehen bleibt. Wir werden im folgenden sehen, daß der kürzeste Weg die eindeutige Lösung des Systems

$$\begin{aligned} x_i^* &= \min_{j \in \mathcal{A}(i)} (a_{i,j} + x_j^*) \quad i = 2, \ldots, n \\ x_1^* &= 0 \end{aligned} \tag{8.1}$$

ist. Obige Formulierung führt direkt zum Bellman-Ford Algorithmus, den wir im nächsten Abschnitt betrachten. Daran anschließend werden alternative Algorithmen vorgestellt und analysiert. In Abschnitt 8.1.3 beschäftigen wir uns mit den asynchronen Varianten der Algorithmen. Den Abschluß bildet eine etwas allgemeinere Formulierung des Problems und deren Analyse.

8.1.1 Bellman-Ford Algorithmus

Der Bellman-Ford Algorithmus läßt sich direkt aus (8.1) ableiten und führt zum folgenden Iterationsschema

$$\begin{aligned} x_i^k &= \min_{j \in \mathcal{A}(i)} (a_{i,j} + x_j^{k-1}) \quad i = 2, \ldots, n \\ x_1^k &= 0 \end{aligned} \tag{8.2}$$

Es stellt sich nun die Frage, wie die initialen Werte von x_i^0 gesetzt werden sollen. Eine in der Literatur oft verwendete Initialisierung ist $x_1^0 = 0$ und $x_i^0 = \infty$ für $i = 2, \ldots, n$. Dies ist die beste Initialisierung, wenn keine weiteren Infomationen vorhanden sind. Der Algorithmus terminiert nach k Iterationen, wenn $x_i^k = x_i^{k-1}$ für alle $i = 1, \ldots n$. Der folgende Satz zeigt, daß der Algorithmus konvergiert und wieviele Iterationen dazu maximal notwendig sind.

Satz 8.1 *Sei G ein gerichteter Graph mit n Knoten und $a_{i,j} < \infty$ das Kantengewicht der Kante (i, j). Es existiere von jedem Knoten i ein Weg zum Knoten 1, alle Zyklen im Graph haben ein positives Kantengewicht und Knoten 1 besitze keine Ausgangskanten. Es gilt dann:*

a) Es existiert ausgehend von jedem Knoten $i \neq 1$ ein kürzester Weg zum Knoten 1. Jeder kürzeste Weg beinhaltet maximal $n - 1$ Kanten.

b) Für jede initiale Belegung der Variablen x_i^0 mit $x_1^0 = 0$ terminiert der Bellman-Ford Algorithmus nach einer endlichen Anzahl von k Schritten und $x_i^k = x_i^*$ für $i = 1, \ldots, n$.

c) Falls $x_i^0 \geq x_i^*$ für alle $i \neq 1$, so terminiert der Bellman-Ford Algorithmus nach maximal $m^* + 1$ Iterationen, wobei

$$m^* = \max_{i=2,\ldots,n} m_i \leq n - 1$$

und m_i ist die minimale Anzahl von Kanten in einem kürzesten Weg von i nach 1. Die Lösung x_i^* liegt nach maximal m^* Schritten vor.

d) Die Längen der kürzesten Wege x_i^* sind die eindeutige Lösung von (8.1).

Bew.

a) Auf Grund der Annahmen existiert ein Pfad zwischen jedem Knoten und Knoten 1. Falls ein solcher Pfad mehr als $n - 1$ Kanten umfaßt, so muß er einen Zyklus beinhalten. Da jeder Zyklus laut Annahme ein positives Kantengewicht hat, wird durch Löschung des Zyklus die Länge des Weges verkürzt. Damit kann ein kürzester Weg keine Zyklen beinhalten und damit auch nur maximal $n - 1$ Kanten umfassen.

b) Sei $w_{i,j}^k$ die Länge des kürzesten Weges zwischen i und j ($i \neq j$), der genau k Kanten enthält. $w_{i,j}^k = \infty$ falls kein Weg mit k Kanten zwischen i und j existiert. Weiterhin sei $w_{i,1}^k$ gleich der minimalen Weglänge eines Pfades von i nach 1, der k oder weniger Kanten enthält. $w_{i,1}^k = \infty$ wenn kein solcher Pfad existiert. Es gilt $x_i^k = \min_{j=1,\ldots,n}(w_{i,j}^k + x_j^0)$, wie man per Induktion zeigen kann. Da $w_{i,j}^k \to \infty$ für $k \to \infty$ und $w_{i,1}^k = x_i^* < \infty$ für $k > n - 1$, muß $x_i^k = x_i^*$ für ein genügend großes k gelten.

c) Aus dem vorherigen Schritt folgt, daß für die initiale Belegung $x_i^0 = \infty$ für $i \neq 1$ x_i^k die minimale Weglänge zwischen i und 1 unter Benutzung von Pfaden der Länge $\leq k$ enthält. Damit gilt auch $x_i^k = x_i^*$ für $k \geq m^*$ (m^* ist die maximale Länge eines kürzesten Weges).

Falls $x_i^0 \geq x_i^*$ so gilt

$$x_i^* = \min_{j \in \mathcal{A}(i)}(a_{i,j} + x_j^*) \leq \min_{j \in \mathcal{A}(i)}(a_{i,j} + x_j^0) = x_i^1.$$

Wenden wir diese Ungleichungskette mehrfach an, so gilt

$$x_i^* \leq x_i^k.$$

Sei $\tilde{x}_i^k$ der k-te Iterationswert des Bellman-Ford Algorithmus, der mit $x_i^0 = \infty$ $(i \neq 1)$ startet. Mit einer Ungleichungskette ähnlich der obigen kann man zeigen, daß

$$x_i^* \leq x_i^k \leq \tilde{x}_i^k \text{ für alle } i, k \text{ falls } x_i^0 \geq x_i^*.$$

Da wir wissen, daß $\tilde{x}_i^k = x_i^*$ für $k \geq m^*$ gilt auch $x_i^k = x_i^*$ für $k \geq m^*$.

d) Wenn wir den Bellman-Ford Algorithmus mit der Lösung von (8.1) starten, so terminiert er nach der ersten Iteration und die Lösung muß den kürzesten Wegen nach obigem Beweisschritt entsprechen.

□

Die Anzahl der benötigten Iterationen hängt natürlich von der Wahl der x_i^0 ab. Nach obigen Satz benötigen wir maximal $m^*+1 \leq n$ Schritte, wenn wir $x_i^0 = \infty$ für $i = 2, \ldots, n$ wählen. Andere initiale Belegungen können deutlich weniger, aber auch deutlich mehr als m^*+1 Iterationen bedingen. In der Praxis tritt oft die Situation auf, daß der Algorithmus mehrmals nacheinander auf einen Graph angewendet wird, dessen Kantengewichte sich nur leicht geändert haben (z.B. bei der Routenwahl in Rechnernetzen, wenn die aktuelle Belastung der Pfade als Gewicht gewertet und von Zeit zu Zeit aktualisiert wird). In einem solchen Fall ist es naheliegend, statt der obigen initialen Belegung, die vorherige Lösung als initiale Belegung zu verwenden. Dies führt meist zu deutlich weniger Iterationen, was aber nicht zwangsläufig so sein muß. Man kann einfache Beispiele finden, in denen sich nur ein Kantengewicht minimal ändert und der Algorithmus, ausgehend von der vorherigen Lösung, deutlich mehr als $m^* + 1$ Iterationen bis zur Terminierung benötigt.

Betrachten wir nun die Aufwandsanalyse für den Bellman-Ford Algorithmus. Sei $|\mathcal{A}|$ die Anzahl der Kanten im Graphen, dann benötigt eine Iteration einen Aufwand in $O(|\mathcal{A}|)$. Für obige initiale Belegung der x_i^0 benötigt der Algorithmus einen Aufwand in $O(m^*|\mathcal{A}|)$.

Wir wollen nun zwei Arten der parallelen Implementierung untersuchen. Im ersten Fall sei der zu untersuchende Graph in die Topologie eingebettet. Jeder Prozessor berechnet ein x_i und ist direkt mit den Prozessoren $j \in \mathcal{A}(i)$ verbunden. In jedem Iterationsschritt berechnet Prozessor i

$$x_i^k = min_{j \in \mathcal{A}(i)}(a_{i,j} + x_j^{k-1})$$

und verteilt das Ergebnis an seine Nachbarn j mit $i \in \mathcal{A}(j)$. Der Aufwand für einen Iterationsschritt ist $O(r)$ mit $r = \max_{i \neq 1} |\mathcal{A}(i)|$, der maximalen Anzahl von Ausgangskanten eines Knoten im Graphen. Wenn ein Prozessor selbst ein Parallelrechner ist, so kann obiges Minimum in $O(\log r)$ Schritten berechnet werden. Während der Iteration kann ein lokales Synchronisationsschema verwendet werden, da ein Prozessor alle Nachbarn von denen er Nachrichten erwarten darf kennt. Um die lokale Synchronisation zu ermöglichen, muß ein Prozessor auch die Werte von x_i versenden, wenn keine Änderung in der aktuellen Iteration stattgefunden hat.

Die zweite Implementierung wird auf einer regelmäßigen Topologie vorgenommen, ohne den zu untersuchenden Graphen einzubetten. Es ist zuerst anzumerken, daß ein Iterationsschritt des Bellman-Ford Algorithmus sehr ähnlich zu einem Vektor-Matrix-Produkt ist. Beim Vektor-Matrix-Produkt wird Komponente i des Lösungsvektors als $\sum_{j=1}^{n} A_{i,j} x_j$ berechnet, der Bellman-Ford Algorithmus führt die Berechnung $\min_{j \in \mathcal{A}(i)}(a_{i,j} + x_j)$ aus. Ersetzung der Multiplikation durch die Addition und der Addition durch die Minimumbildung liefert die Vektor-Matrix-Produktbildung den Iterationsschritt des Bellman-Ford Algorithmus auf einem voll vermaschten Graphen. Das Vektor-Matrix-Produkt auf Basis spärlich besetzter Matrizen ist damit im Prinzip identisch zum Bellman-Ford Algorithmus.

Beginnen wir mit der Implementierung auf einem System mit $p \leq n$ Prozessoren. Sei zur Vereinfachung $k = n/p$ eine ganze Zahl. Prozessor i berechnet nun die Werte $(i-1)k+1$ bis ik. Der Aufwand für einen Iterationsschritt liegt in $O(kr)$ mit $r = \max_{i \neq 1} |\mathcal{A}(i)|$. Anschließend müssen die k berechneten Werte an alle anderen Prozessoren verteilt werden, was einem allgemeinen Broadcast mit Nachrichten die jeweils k Werte enthalten entspricht. Wenn wir annehmen, daß eine solche Nachricht die k-fache Zeit einer Nachricht mit einem Wert benötigt (was nicht ganz realistisch ist), so benötigt die Kommunikation auf einem linearen Array einen Aufwand in $O(kp) = O(n)$. Auf einem Hyperwürfel liegt der Kommunikationsaufwand in $O(n/\log p)$. Der Gesamtaufwand auf einem linearen Array ist in $O(\max(n, kr))$ und auf einem Hyperwürfel $O(\max(n/\log p, kr))$. Es ist beachtenswert, daß mit steigendem r auch die Berechnungszeit, nicht aber die Kommunikationszeit steigt. Auf einem Hyperwürfel mit n^2 Prozessoren können wir einen Iterationsschritt des Algorithmus in $O(\log n)$ implementieren, wie man sich leicht auf Basis der Vektor-Matrix-Multiplikation überlegen kann.

Bisher haben wir nur die Problemstellung betrachtet, daß der kürzeste Weg zu einem festen Ziel gesucht wird. Eine logische Erweiterung ist die Suche nach dem kürzesten Weg zwischen zwei beliebigen Knoten im Graphen. Unter Verwendung des Bellman-Ford Algorithmus muß man n sequentielle oder parallele Läufe durchführen, um das Problem zu lösen. Dabei wird jeder Knoten als Zielknoten definiert indem die ausgehenden Kanten gelöscht werden. Im nächsten Abschnitt werden wir zwei Algorithmen vorstellen, die speziell auf dieses allgemeinere Problem zugeschnitten sind.

Die bisherigen Laufzeitanalysen haben nur den Aufwand einer Iteration untersucht, im Gegensatz zu den meisten iterativen Verfahren für lineare Gleichungssysteme kann für den Bellman-Ford Algorithmus mit $m^* + 1$ eine obere Schranke für die Iterationszahl angegeben werden. Dieser Wert ist oftmals höher als die tatsächliche Iterationszahl, aber auch nicht vollkommen unrealistisch, da m^* für die meisten Graphen wesentlich kleiner als die Knotenzahl n ist.

8.1.2 Andere Algorithmen

In diesem Abschnitt betrachten wir zwei Algorithmen, nämlich den Floyd Warshall und den Doubling Algorithmus, die speziell für das Problem der kürzesten Wege zwischen allen Knoten in einem Graphen entwickelt wurden. Diese Algorithmen sind schneller als der Bellman-Ford Algorithmus für dieses allgemeinere Problem. Beide Algorithmen laufen allerdings nicht schneller, wenn sie den kürzesten Weg zu einem Zielknoten berechnen sollen. Im Gegensatz zum Bellman-Ford Algorithmus, der um den Faktor n schneller wird wenn nur kürzeste Weg zu einem Zielknoten berechnet werden sollen, oder besser ausgedrückt, der Bellman-Ford Algorithmus wird um den Faktor n langsamer bei der Berechnung aller Paare kürzester Wege, da er einmal gewonnene Informationen nicht weiter verwendet. Weiterhin werden die Laufzeiten der beiden folgenden Algorithmen nicht durch die Kantenzahl im Graphen beeinflußt.

Der Floyd-Warshall Algorithmus startet mit der initialen Belegung

$$x^0_{i,j} = \begin{cases} a_{i,j} & \text{falls } j \in \mathcal{A}(i) \\ \infty & \text{sonst} \end{cases} \qquad \text{für alle } 1 \leq i, j \leq n \tag{8.3}$$

und führt die Iterationen $k = 1, \ldots, n$ für alle Knoten i und j wie folgt aus.

$$x^k_{i,j} = \begin{cases} \min(x^{k-1}_{i,j}, x^{k-1}_{i,k} + x^{k-1}_{k,j}) & \text{falls } j \neq i \\ \infty & \text{sonst} \end{cases} \tag{8.4}$$

Per Induktion kann man zeigen, daß die obige Iteration nach k Schritten die minimale Distanz zwischen den Knoten i und j berechnet hat, wobei als Zwischenknoten nur die Knoten $1, \ldots, k$ benutzt werden. Der entsprechende Wert ist in $x_{i,j}^k$ gespeichert. Damit liegt der sequentielle Aufwand in $O(n^3)$ im Vergleich zu $O(nm^*|\mathcal{A}|)$ beim Bellman-Ford Algorithmus. Der Floyd-Warshall Algorithmus hat eine geringere sequentielle Komplexität, wenn der Graph viele Kanten enthält.

Der Floyd-Warshall Algorithmus kann in $O(n)$ auf einem $n \times n$ Array implementiert werden, wobei Kommunikation und Berechnungen entsprechend zu überlagern sind. Durch die Simulation eines $n \times n$ Arrays auf einem linearen Array mit n Prozessoren, ist eine Implementierung in $O(n^2)$ auf dem linearen Array möglich. Klarerweise ist dann auch eine Implementierung in $O(n^3/p)$ auf einem linearen Array mit $p \leq n$ Prozessoren möglich.

Das Iterationsschema des Doubling Algorithmus lautet

$$\begin{array}{l} x_{i,j}^k = \begin{cases} \min_m (x_{i,m}^{k-1} + x_{m,j}^{k-1}) & \text{für } i \neq j \\ 0 & \text{für } i = j \end{cases} \\ \text{für } k = 1, 2, \ldots \lceil \log(n-1) \rceil \end{array} \tag{8.5}$$

Die initiale Belegung der Variablen wird wie folgt gewählt.

$$x_{i,j}^0 = \begin{cases} a_{i,j} & \text{für } j \in \mathcal{A}(i) \\ 0 & \text{für } i = j \\ \infty & \text{sonst} \end{cases} \tag{8.6}$$

Per Induktion kann gezeigt werden, daß $x_{i,j}^k$ die kürzeste Distanz zwischen i und j beinhaltet, wenn Pfade einer Länge mit maximal 2^{k-1} Kanten benutzt werden. Damit ist nach $\log m^*$ Iterationen das Minimum erreicht (m^* ist die maximale Kantenzahl eines kürzesten Weges). Der Aufwand einer sequentiellen Implementierung liegt in $O(n^3 \log m^*)$.

Betrachten wir nun die parallele Implementierung. Zuerst ist festzuhalten, daß ein Iterationsschritt sehr ähnlich zur Multiplikation zweier $n \times n$ Matrizen ist. Die Multiplikation von Elementen wird durch die Addition und die Summation durch die Minimumbildung ersetzt. Deshalb können wir auf die parallele Implementierung der Matrix-Matrix-Multiplikation zurückgreifen. Ein Iterationsschritt kann in $O(n)$ auf einem $n \times n$ Array implementiert werden. Der gesamte Algorithmus benötigt einen Aufwand in $O(n \log m^*)$ auf einem $n \times n$ Array. Auf einem Hyperwürfel mit n^3 Prozessoren ist eine Implementierung in $O((\log n)(\log m^*))$ möglich. Beide Algorithmen sind direkt aus den entsprechenden Matrix-Multiplikationsalgorithmen ableitbar und brauchen deshalb nicht vorgestellt zu werden. Die Implementierung auf dem Hyperwürfel ist gleichzeitig die schnellste bekannte Lösung für das kürzeste Wege Problem für alle möglichen Knotenpaare.

Vergleicht man die drei vorgestellten Algorithmen, so stellt man fest, daß der Bellman-Ford Algorithmus am besten geeignet ist, wenn kürzeste Wege zu einem Ziel ermittelt werden sollen. Sind die kürzesten Wege zwischen allen Knotenpaaren zu ermitteln, so hängt der "beste" Algorithmus von der verwendeten Topologie ab. Auf einem linearen Array ist der Bellman-Ford Algorithmus am besten geeignet, wenn der Graph sehr wenige Kanten enthält, andernfalls ist der Floyd-Warshall Algorithmus vorzuziehen. Auf einem $n \times n$ Array ist der Floyd-Warshall Algorithmus am schnellsten. Auf einem Hyperwürfel mit n^3 Prozessoren benötigt der Doubling Algorithmus die geringste Zeit zur Lösung. Auch hier gilt aber, daß bei praktischen Problemen die Zahl der Knoten im Graph die Zahl der verfügbaren Prozessoren übertrifft. Außerdem ist durch die großen Konstanten in den Kommunikationszeiten eine Realisierung mit zu vielen Prozessoren im allgemeinen ineffizient. Insofern ist für praktische Anwendungen der Doubling Algorithmus dem Floyd-Warshall Algorithmus unterlegen.

8.1.3 Asynchrone Algorithmen

In diesem Abschnitt wird die asynchrone Variante des Bellman-Ford Algorithmus vorgestellt und mit der synchronen verglichen. Das Problem sei die Suche nach dem kürzesten Weg zu einem Zielknoten. Ein Iterationsschritt des Bellman-Ford Algorithmus hat im asynchronen Algorithmen Modell folgende Darstellung

$$x_i(t+1) = \begin{cases} \min_{j\in\mathcal{A}(i)}(a_{i,j} + x_j(\tau_j^i(t)) & \text{falls } i = 2,\ldots,n \text{ und } t \in T^i \\ 0 & \text{falls } i = 1 \\ x_i(t) & \text{falls } t \notin T^i \end{cases} \tag{8.7}$$

Der Algorithmus terminiert nach endlicher Zeit mit der exakten Lösung für alle initialen Belegungen mit $x_1(0) = 0$. Wir wollen wieder von der initialen Belegung $x_i(0) = \infty$ für $i \neq 1$ ausgehen und die synchrone und asynchrone Variante vergleichen. Wir betrachten dazu die Implementierung auf einer Topologie, die den zu untersuchenden Graphen einbettet. Wir nehmen an, daß jeder Prozessor genau eine Variable berechnet, die realistische Reduktion der Prozessorzahl ist, wie bei anderen parallelen Algorithmen, kein Problem. Jeder Prozessor berechnet eine Variable x_i und zwischen Prozessor i und j existiert eine direkte Verbindung falls $j \in \mathcal{A}(i)$. Wird ein neuer Wert für x_j berechnet, so wird dieser direkt an alle Prozessoren i mit $i \in \mathcal{A}(j)$ weitergeleitet. Prozessor i beginnt mit der Berechnung wobei angenommen wird, daß der Wert x_j für benachbarte Prozessoren ∞ ist, solange keine andere Information vorhanden ist. Jedesmal wenn neue Werte von Nachbarprozessoren empfangen wurden, wird der Wert der eigenen Variablen aktualisiert. Wir nehmen in der folgenden Analyse an, daß die Übertragung eines Wertes von i nach j konstant $t_{i,j}$ Zeiteinheiten erfordert. Nachrichten überholen sich nicht und gehen auch nicht verloren. Weiterhin sei die Zeit für einen Berechnungsschritt vernachlässigbar. Wenn Prozessor i zum Zeitpunkt t einen neuen Wert für x_j empfängt, so berechnet er sofort den neuen Wert von x_i, den wir mit $x_i(t)$ bezeichnen. Auf Grund der initialen Variablenbelegung und der Iterationsvorschrift des Bellman-Ford Algorithmus bilden die $x_i(t)$ eine monoton nicht steigende Folge, die gegen x_i^* konvergiert.

Zur Analyse des Algorithmus wollen wir eine etwas andere Betrachtungsweise wählen. Wir betrachten dazu eine Menge von Token, die zum Zeitpunkt $t = 0$ in Knoten 1 generiert werden und an alle Nachbarknoten verteilt wird. Ein Token benötigt zur Übertragung über eine Leitung genauso lange wie eine normale Nachricht und beinhaltet jeweils die Pfadlänge, die bisher zurückgelegt wurde. Trifft ein Token zum Zeitpunkt t im Knoten i ein und beinhaltet eine Pfadlänge, die kürzer als die bisher gespeicherte ist (initial wird ∞ gespeichert), so wird der Wert in x_i gespeichert und das Token weitergeleitet. Ist der Wert des ankommenden Tokens größer gleich dem aktuellen Wert von x_i, so wird das Token vernichtet. Treffen zwei Token zum gleichen Zeitpunkt mit gleichen Werten ein, so wird eins vernichtet. Die Verbreitung der Token im Graph entspricht genau den berechneten kürzesten Wegen des asynchronen Bellman-Ford Algorithmus. $x_i(t)$ speichert die minimale Pfadlänge über alle Pfade, deren Token bis zum Zeitpunkt t Knoten i erreicht haben. Sei die Länge eines Pfades p, der die Knoten $i_1, \ldots, i_{k+1}$ umfaßt, definiert als

$$L_p = \sum_{m=1}^{k} a_{i_m, i_{m+1}}. \tag{8.8}$$

Die Kommunikationszeit für Pfad p ist durch

$$t_p = \sum_{m=1}^{k} t_{i_m, i_{m+1}} \tag{8.9}$$

gegeben. Sei für jedes $i \neq 1$ $P_i(t)$ die Menge aller Pfade p, die in i enden (und 1 starten) und für die $t_p \leq t$ gilt. Der ermittelte kürzeste Weg zum Zeitpunkt t ist damit

$$x_i(t) = \begin{cases} \infty & \text{falls } P_i(t) \text{ leer ist} \\ min_{p \in P_i(t)} L_p & \text{sonst} \end{cases} \tag{8.10}$$

Betrachten wir nun die synchrone Version des Algorithmus unter gleichen Voraussetzungen. Prozessor i führt die $k+1$ Neuberechnung von x_i aus, wenn er alle Werte x_j ($j \in \mathcal{A}(i)$) aus der k-ten Neuberechnung empfangen hat (lokale Synchronisation). Sei P_i^k die Menge aller Pfade zwischen i und 1, die k oder weniger Kanten benutzen. Weiterhin sei

$$k_i(t) = \max(k | t_p \leq t, \forall p \in P_i^k). \tag{8.11}$$

$k_i(t)$ beschreibt die Anzahl der synchronen Bellman-Ford Iterationen, die zum Zeitpunkt t im Knoten i ausgeführt worden sind. Für den aktuellen Wert der synchronen Iteration zum Zeitpunkt t gilt.

$$x_i^S(t) = \begin{cases} \infty & \text{falls } P_i^{k_i(t)} \text{ leer ist} \\ \min_{p \in P_i^{k_i(t)}} L_p & \text{sonst} \end{cases} \tag{8.12}$$

Offensichtlich ist aber $P_i^{k_i(t)} \subseteq P_i(t)$ für alle i und t. Daraus folgt direkt

$$x_i(t) \leq x_i^S(t) \text{ für alle } t \tag{8.13}$$

Da die $x_i(t)$ eine monoton nicht steigende gegen x_i^* konvergierende Folge bilden, konvergiert der asynchrone Algorithmus nicht langsamer als der synchrone. Leider führt der asynchrone Algorithmus mehr und in Extremfällen deutlich mehr Nachrichtenübertragungen als der synchrone durch.

8.1.4 Einige Erweiterungen

Wir haben bisher nur einen sehr kleinen Teil der unter dem Oberbegriff dynamische Programmierung zusammengefaßten Optimierungsprobleme untersucht. Es wurde bereits erwähnt, daß mit Wahrscheinlichkeiten belastete Kosten ebenfalls darunter fallen. Eine andere Erweiterung betrachtet komplexere Kostenfunktionen, die nicht nur von einem Term (beim Problem des kürzesten Weges von der Weglänge) abhängen, sondern durch weitere rekursive Terme beeinflußt werden. Solche Probleme nennt man auch polyadisch, im Gegensatz zu monadisch. Weiterhin gibt es die Unterscheidung in serielle und nicht-serielle Probleme. Ein serielles Problem kann in Ebenen von Subproblemen so zerlegt werden, daß die Lösung der Subprobleme einer Ebene aus der Lösung der Subprobleme der angrenzenden Ebenen berechnet werden kann. In einem nicht-seriellen Problem ist eine solche Einteilung in Ebenen nicht möglich. Das Problem der kürzesten Wege ist ein monadisches serielles Problem. Diese Art von Problemen kann mit Algorithmen gelöst werden, die auf die Vektor-Matrix-Multiplikation zurückzuführen und einfach parallelisierbar sind. In diesem Abschnitt werden wir uns kurz mit der komplexeren Problemstellung der polyadischen nicht-seriellen Probleme beschäftigen.

Das Vorgehen soll an einem einfachen Beispiel erläutert werden, welches für eine ganze Klasse von isomorphen oder ähnlichen Problemen steht. Es geht um die optimale Klammersetzung. Ziel ist es, das Produkt von n Matrizen $\mathbf{A}[1], \mathbf{A}[2], \ldots, \mathbf{A}[n]$ zu berechnen. $\mathbf{A}[i]$ ist eine Matrix mit r_{i-1} Zeilen und r_i Spalten. Die Reihenfolge in der die Matrizen multipliziert werden hat natürlich Einfluß auf die Anzahl der Operationen, die durchgeführt werden müssen. Wir wollen dies kurz an einem Beispiel erläutern. Multipliziert werden sollen die folgenden sechs Matrizen.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \\ A_{4,1} & A_{4,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,1} & B_{2,2} & B_{2,3} \end{pmatrix} \begin{pmatrix} C_{1,1} \\ C_{2,1} \\ C_{3,1} \end{pmatrix} \begin{pmatrix} D_{1,1} & D_{1,2} \end{pmatrix} \begin{pmatrix} E_{1,1} & E_{1,2} \\ E_{2,1} & E_{2,2} \end{pmatrix} \begin{pmatrix} F_{1,1} & F_{1,2} & F_{1,3} \\ F_{2,1} & F_{2,2} & F_{2,3} \end{pmatrix}$$

Wenn wir von links nach rechts die Multiplikation durchführen, so werden dazu insgesamt 84 Skalar-Multiplikationen und 44 Skalar-Additionen benötigt. Wird dagegen von rechts nach links multipliziert, so sind nur 69 Skalar-Multiplikationen und 33 Skalar-Additionen notwendig. Dieses einfache Beispiel zeigt, daß durch eine geschickte Auswahl der Reihenfolge der Rechnenaufwand deutlich reduziert werden kann. Die möglichen Einsparungen werden noch deutlicher, wenn die Dimension der einzelnen Matrizen größer wird. Falls die Matrizen **B**, **C** und **F** jeweils Dimension 300 statt 3 besitzen, so würde eine Multiplikation von links nach rechts 6024 Skalar-Multiplikationen erfordern, von rechts nach links wären dagegen 274200 Skalar-Multiplikationen notwendig.

Das geschilderte Problem der optimale Klammerung ist offensichtlich deutlich komplexer als die Bestimmung kürzester Wege in einem Graphen, da Entscheidungen nun nicht mehr lokal getroffen werden können. Ein vollständiges Ausprobieren aller Klammerungen ist utopisch für größere Beispiele, da ungefähr $4^n/n\sqrt{\pi n}$ Möglichkeiten existieren, das Produkt von n Matrizen zu klammern. Ein serieller Algorithmus der dynamischen Programmierung benötigt dagegen nur einen Aufwand in $O(n^3)$. Wir werden zuerst den Algorithmus und dann dessen Parallelisierung vorstellen. Dabei beschränken wir uns auf die Minimierung der Zahl der Skalar-Multiplikationen. Sei dazu $c(i,j)$ die minimale Anzahl von Skalar-Multiplikationen, die zur Multiplikation von $\mathbf{A}[i], \mathbf{A}[i+1], \ldots, \mathbf{A}[j]$ notwendig sind. Ein Algorithmus der den optimalen Wert ermittelt benutzt die folgende rekursive Struktur.

$$c(i,j) = \begin{cases} min_{i \le k \le j}(c(i,k) + c(k+1,j) + r_{i-1}r_k r_j & \text{für } 1 \le i < j \le n \\ 0 & \text{sonst} \end{cases} \tag{8.14}$$

Die Gesamtlösung besteht in der Berechnung von $c(1,n)$. Zur Implementierung des Algorithmus werden die Werte $c(i,j)$ rekursiv berechnet und gespeichert. Gleichzeitig wird die Klammerung gespeichert, die zu einem optimalen Wert $c(i,j)$ gehört. Unter dem Wert von $c(i,j)$ wollen wir also im folgenden die minimale Zahl von Operationen und die zugehörige optimale Klammerung zur Berechnung von $\prod_{k=i}^{j} \mathbf{A}[i]$ verstehen. Den Ablauf des Algorithmus kann man sich am besten graphisch veranschaulichen, es wird die in Abbildung 8.1 gezeigte zweidimensionale Tabelle entlang der Diagonalen aufgefüllt. Dazu wird folgender sequentieller Algorithmus benutzt.

```
for i = 1 to n do
    for j = i + 1 to n do
        c(i, j) = ∞ ;
for i = 1 to n do
    c(i, i) = 0 ;
for j = 1 to n do
    for i = 1 to n − j do
        for k = i + 1 to i + j do
            t = c(i, k − 1) + c(k, i + j) + r_i r_k r_{i+j+1} ;
            if (t < c(i, i + j))
                c(i, i + j) = t ;
                b(i, i + j) = k ;
```

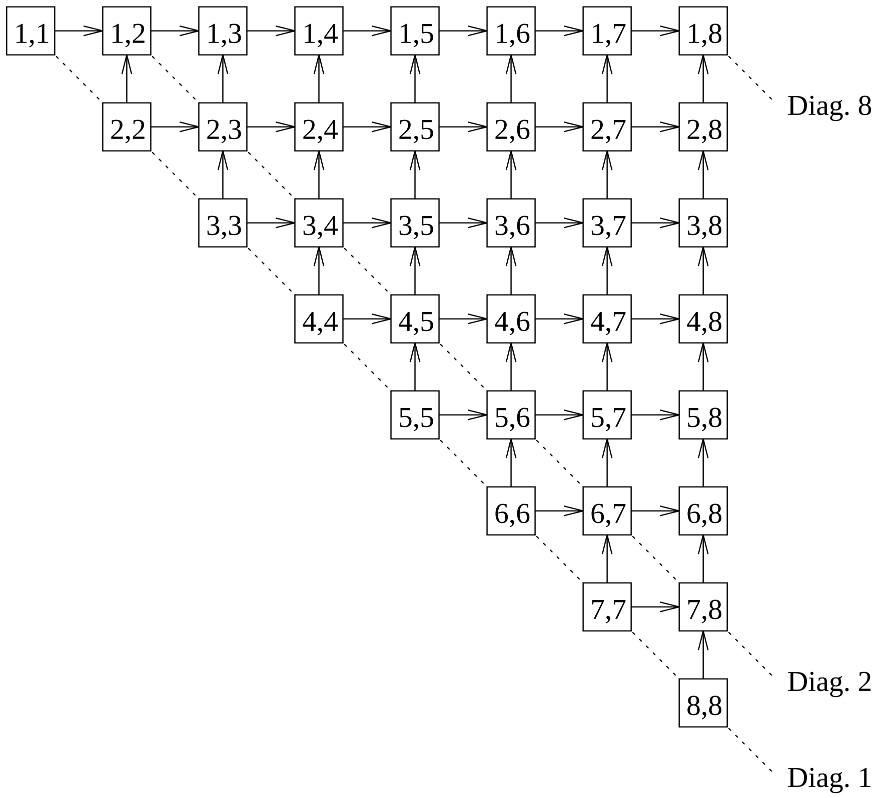


Abbildung 8.1: Ablauf des Algorithmus zu Ermittlung der optimalen Klammerung.

od ;

An Position $c(i,j)$ steht nach Ablauf des Algorithmus die minimale Anzahl von Skalar-Multiplikationen zur Berechnung von $\prod_{k=i}^{j} \mathbf{A}[k]$ und $b(i,j)$ beinhaltet das letzte Produkt, daß gebildet wird. Also $\prod_{k=i}^{j} \mathbf{A}[k] = (\prod_{k=i}^{b(i,j)-1} \mathbf{A}[k])(\prod_{l=b(i,j)}^{j} \mathbf{A}[l])$. Die Werte im Array b beschreiben die optimale Klammerung. Für das Beispiel entsteht die in Abb. 8.2 gezeigte Struktur. Die optimale Klammerung ist $((\mathbf{A}(\mathbf{BC})((\mathbf{DE})\mathbf{F}))$ und benötigt nur 36 Skalar-Multiplikation und 13 Skalar-Additionen.

Analysieren wir nun den Aufwand des Algorithmus. Die Berechnung von $c(i,j)$ erfordert die Bestimmung von $(j-i)$ möglichen Klammerungen und die Bildung des Minimums über diese. Damit liegt die sequentielle Komplexität bei $\sum_{i=1}^{n-1}(n-i)i = \frac{n^3}{6} - \frac{n}{6} \in \Theta(n^3)$.

Der obige Algorithmus kann sehr einfach parallelisiert werden. Betrachten wir zuerst die Parallelisierung auf einem linearem Array mit $p \leq n$ Prozessoren. Auch der parallele Algorithmus berechnet die Werte entlang der Diagonalen. Befinden sich q Elemente in der aktuellen Diagonalen, so bekommt ein Prozessor $\lceil q/p \rceil$ oder $\lfloor q/p \rfloor$ Elemente zugeteilt und berechnet die Wert $c(i,j)$ für diese Elemente. Die Ergebnisse werden anschließend an alle anderen Prozessoren verteilt, was einem allgemeinen Broadcast entspricht. Da zum Zeitpunkt der Berechnung eines einzelnen Wertes $c(i,j)$ die Werte aus vorherigen Diagonalen alle bekannt sind, kann die Berechnung ohne weitere Kommunikation erfolgen. Der Berechnungsaufwand liegt damit in $O(n^3/p)$, der Kommunikationsaufwand in $O(n^2)$, wenn wir davon ausgehen, daß die Übertragung von mit $O(n/p)$ Nachrichten zwischen p Prozessoren einen Aufwand in $O(n)$ benötigt.

Ein schnellerer Algorithmus kann für zweidimensionales Array mit $\frac{n(n+1)}{2}$ Prozessoren angegeben werden. Die Prozessoren seien so angeordnet wie in Abbildung 8.1 gezeigt. Prozessoren sind mit zwei Indizes i und j $(1 \leq i \leq j < n)$ numeriert. Prozessor (i,j) berechnet das Element $c(i,j)$. Die grundlegende Idee des Algorithmus ist es, daß mit der Berechnung in der Diagonalen t schon begonnen werden kann, bevor alle Elemente aus Diagonalen $s < t$ bekannt

	B	C	D	E	F
A	24 **(A)(B)**	14 **(A)(BC)**	22 **(ABC)(D)**	26 **(ABC)(DE)**	36 **(ABC)(DEF)**
B		6 **(B)(C)**	10 **(BC)(D)**	14 **(BC)(DE)**	22 **(BC)(DEF)**
C			6 **(C)(D)**	10 **(C)(DE)**	19 **(C)(DEF)**
D				4 **(D)(E)**	10 **(DE)(F)**
E					12 **(E)(F)**

Abbildung 8.2: Optimale Klammerung und zugehörige Anzahl von Skalar-Multiplikationen für das Beispiel.

sind. Wenn ein Prozessor den Wert $c(i,j)$ berechnet hat, so überträgt er es direkt nach rechts und oben. Prozessoren, die Elemente empfangen, können diese direkt für aktuelle Berechnungen verwenden. Der Aufwand des Algorithmus ist $2n$. Nach n Schritten erreichen die Werte $c(1,1)$ und $c(n,n)$ Prozessor $(1,n)$, der das letzte Element berechnet. Anschließend treffen in jedem Schritt 2 neue Elemente von links bzw. unten ein. Nach n Schritten ist die Berechnung abgeschlossen. Dieses Vorgehen setzt natürlich voraus, daß Berechnung und Kommunikation ungefähr gleich lange dauern, was in der Praxis bei der gewählten feinen Granularität nicht realistisch ist. Vergleichen wir den parallelen Algorithmus mit dem sequentiellen Algorithmus, so ergibt sich eine Effizienz von ungefähr 1/6. Die verwendete Prozessorstruktur ist allerdings keine übliche Verbindungsstruktur. Zwar läßt sich das Gitter in ein $n \times n$ Array einbetten, dies würde aber die Effizienz auf ungefähr 1/12 verschlechtern, wenn wir davon ausgehen, daß nicht benötigte Prozessoren untätig bleiben. Wir wollen deshalb noch kurz die Abbildung auf ein zweidimensionales Array betrachten und dabei insbesondere auch berücksichtigen, daß die tatsächlich verfügbare Prozessorzahl deutlich kleiner als n^2 ist. Damit erhöht sich die Granularität des Algorithmus, wodurch der Berechnungsaufwand pro Prozessor wächst, der relative Kommunikationsaufwand aber geringer wird. Nehmen wir dazu an, daß ein $\sqrt{p} \times \sqrt{p}$ Array von Prozessoren zur Vefügung steht. Zur Vereinfachung der Beschreibung sei n ein Vielfaches von $\sqrt{p}$.

Die einfachste Art der Abbildung besteht darin, $k = n/\sqrt{p}$ Zeilen und Spalten des ursprünglichen Arrays auf die Prozessoren abzubilden. Prozessor i,j berechnet die Elemente der Zeilen $(i-1)k+1,\ldots,ik$ und der Spalten $(j-1)k+1,\ldots,jk$. Abbildung 8.3 zeigt die Einbettung einer 8×8 Struktur auf ein 2×2 Array. Die Abbildung wird auch als Schachbrettabbildung bezeichnet. Ein Prozessor berechnet bis zu n^2/p Elemente. Am Beispiel wird schon die ungleiche Verteilung der Last auf die Prozessoren deutlich. Während Prozessor $(2,1)$ überhaupt keine Elemente berechnen muß, werden Prozessor $(1,2)$ 16 Elemente zugewiesen, die darüberhinaus noch die längste Rechenzeit erfordern. Im $\sqrt{p} \times \sqrt{p}$ Gitter wird Prozessor $(1,\sqrt{p})$ die längste Rechenzeit benötigen. So werden zur Bestimmung von $c(i,j)$ $j-i$ Rechenschritte durchgeführt. D.h. alle Elemente in einer Diagonalen benötigen die selbe Rechenzeit. Wenn ein Rechenschritt α Zeiteinheiten benötigt, so benötigt die Berechnung der Elemente in der i-ten Diagonalen $i\alpha$

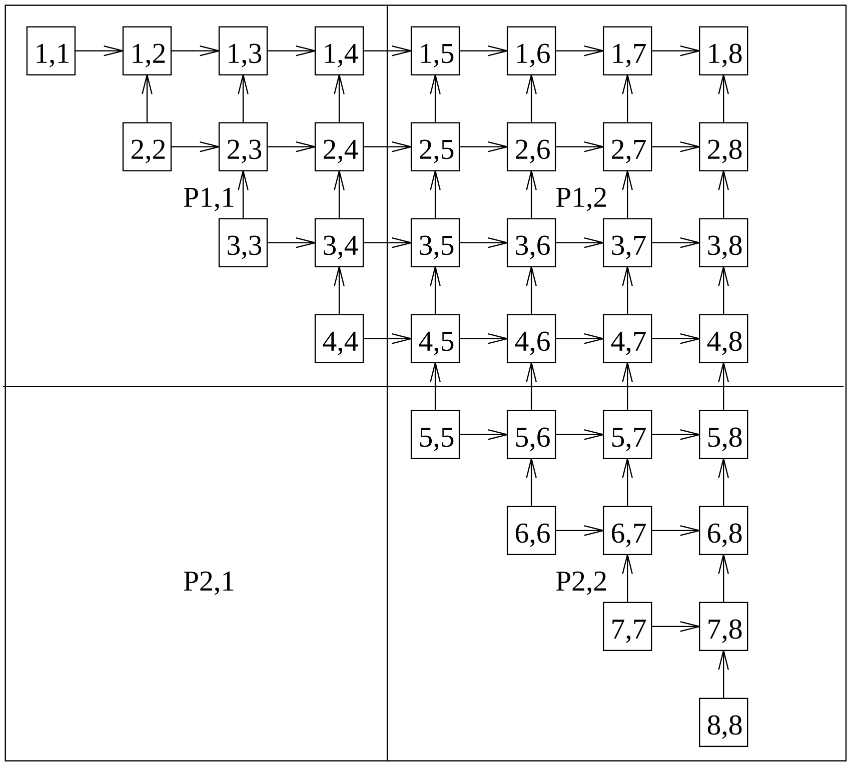


Abbildung 8.3: Schachbrettanordnung.

Zeiteinheiten. Damit benötigt Prozessor $(i, \sqrt{p})$

$$\frac{n^2}{p}(n - \frac{n}{\sqrt{p}})\alpha$$

Zeiteinheiten, um seine lokalen Berechnungen durchzuführen. Dies ist allerdings eine untere Schranke, da die Berechnung erst beginnen kann, wenn die Ergebnisse der Nachbarprozessoren verfügbar sind. Betrachten wir nun den längsten Pfad auf dem Ergebnisse zu Prozessor $(1, \sqrt{p})$ gelangen. Dieser Pfad umfaßt $(\sqrt{p} - 1)n/\sqrt{p} - 1$ Elemente, die von 1 bis $(\sqrt{p} - 1)n/\sqrt{p} - 1$. Erst wenn diese Schritte abgearbeitet wurde kann Prozessor seine Berechnungen beenden. Man kann zeigen, ohne daß wir es hier tun wollen, daß die Effizienz für größere n und p ohne Berücksichtigung der Kommunikationszeit bei ungefähr $1/12$ liegt.

Eine einfache Modifikation führt zur sogenannten modifizierten Schachbrettabbildung, die in Abbildung 8.4 gezeigt wird. Die Zuteilung der Elemente zu den Prozessoren ist recht einfach, nachdem man sich das grundsätzliche Vorgehen verdeutlicht. Zuerst werden die einzelnen Elemente in drei Bereiche L_1, L_2 und L_3 unterteilt, wie in Abbildung 8.4 gezeigt. Anschließende wird L_1 entlang der Verbindung $L_1|L_3$ auf L_3 und L_2 entlang der Verbindung $L_2|L_3$ auf L_3 gefaltet. Das so entstandene $n/2 \times n/2$ Gitter wird direkt auf das $\sqrt{p} \times \sqrt{p}$ Gitter der Prozessoren abgebildet. Wie auch bei der Schachbrettabbildung bleibt die Kommunikationsstruktur erhalten, d.h. ein Kommunikationsschritt zwischen benachbarten Prozessoren im ursprünglichen Algorithmus erfordert keine Kommunikation oder Kommunikation zwischen benachbarten Prozessoren. Dies kann man leicht sehen, wenn man die Faltung nachvollzieht. Die resultierende Abbildung verteilt die Last deutlich fairer über die Prozessoren, da jeder Prozessor eine ungefähr gleiche Zahl von Elementen zugeteilt bekommt. Die Zahl der Elemente auf einem Prozessor entspricht $n^2/2p + n/2\sqrt{p}$ auf den Prozessoren $i, \sqrt{p} + 1 - i$ für $1 \leq i \leq \sqrt{p}$ und n^2/p auf den restlichen Prozessoren. Wie schon bei der Schachbrettanordnung benötigt Prozessor $(1, \sqrt{p})$ die längste Rechenzeit. Auch bei der modifizierten Schachbrettanordnung

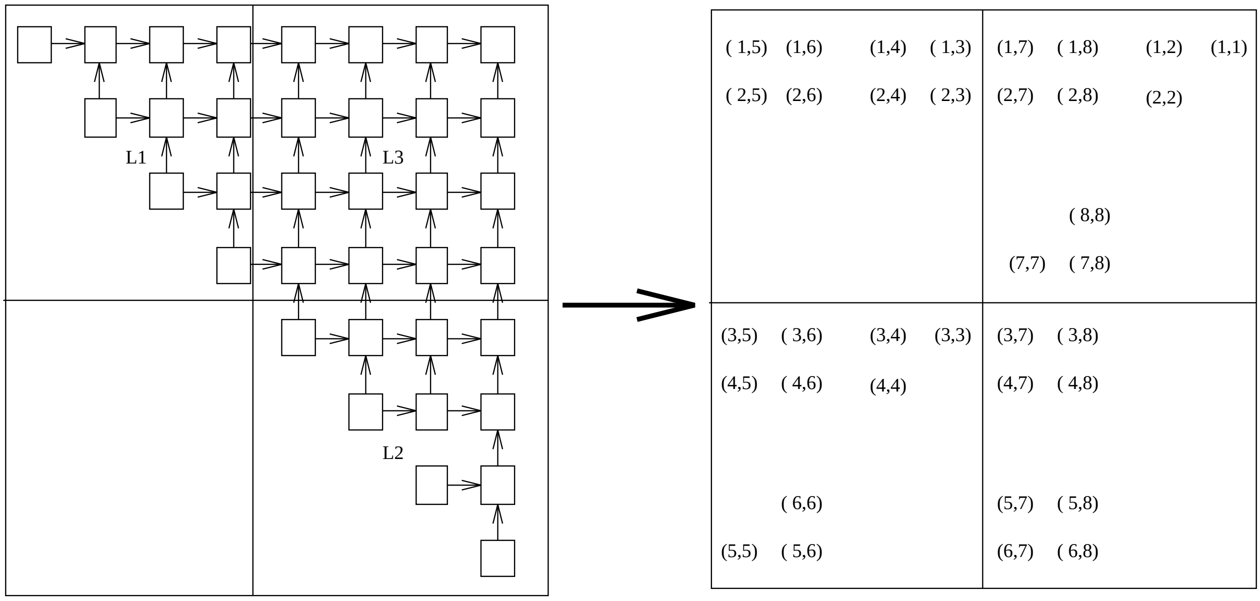


Abbildung 8.4: Modifizierte Schachbrettanordnung.

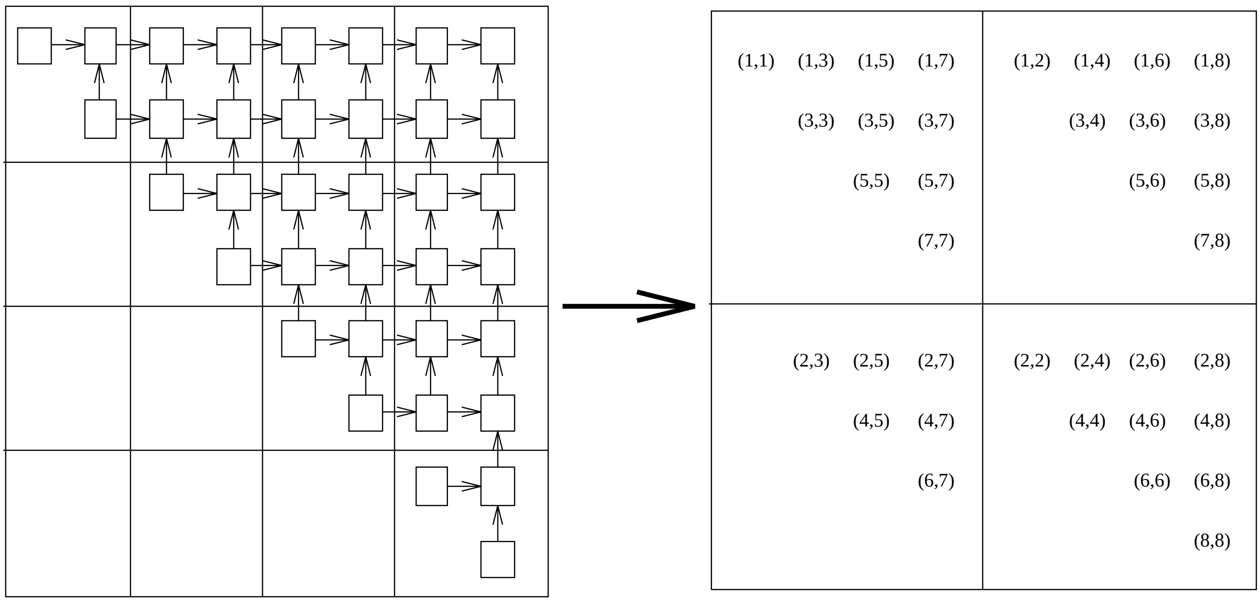


Abbildung 8.5: “Shuffle” Anordnung.

beinhaltet Prozessor $(1, \sqrt{p})$ die rechenintensivsten Elemente. Eine detaillierte Analyse ohne Berücksichtigung der Kommunikationszeiten ergibt, daß für große p maximal eine Effizienz von $1/3$ erreichbar ist.

Die beiden bisher vorgestellten Abbildungen liefern keine optimale Effizienz, egal welche Problemgröße wir auch wählen. Man kann aber eine Abbildung finden, die eine asymptotische Effizienz von 1 erlaubt. Die Abbildung ist, wie schon die modifizierte Schachbrettabbildung, am besten über die Faltung von Prozessoren zu erklären und wird in Abbildung 8.5 dargestellt. Wir unterteilen die Anordnung in $n/\sqrt{p}$ zusammengesetzte Spalten (Zeilen), die jeweils $\sqrt{p}$ Zeilen (Spalten) der ursprünglichen Anordnung umfassen. Anschließend werden die zusammengesetzten Spalten gefaltet, zuerst 1 auf 2 entlang der gemeinsamen Grenze, dann $1, 2$ auf 3 usw. Schließlich erhält man eine Struktur, mit nur noch einer kombinierten bzw. $n/\sqrt{p}$ ursprünglichen Spalten. Dieses Vorgehen wird in analoger Weise für die zusammengesetzten Zeilen durchgeführt. Das Ergebnis ist ein $\sqrt{p} \times \sqrt{p}$ Array, wobei einem Prozessor $\frac{n}{\sqrt{p}}(\frac{n}{\sqrt{p}} - 1)$ oder $\frac{n}{\sqrt{p}}(\frac{n}{\sqrt{p}} + 1)$ Elemente zugewiesen werden. Die Kommunikationsstruktur bleibt auch bei dieser Abbildung erhalten, für entsprechend große n und p liegt die Effizienz des Algorithmus bei 1.

8.2 Literaturhinweise

Dieses Kapitel beruht im wesentlichen auf Kapitel 4 von [4]. Der asynchrone Algorithmus für das Problem des kürzesten Weges wird in Kapitel 6.4 von [4] vorgestellt. Der Algorithmus zur optimalen Klammerung und dessen Abbildung auf zweidimensionale Arrays ist in [25] zu finden. Das Beispiel, welches zur Veranschaulichung unterschiedlicher Klammerungen verwendet wurde, ist [34] entnommen, dort ist auch der serielle Algorithmus zur Lösung dieses Problems beschrieben.

Anhang A

Lineare Algebra und Matrizen

Wir werden in diesem Anhang einige elementare Definitionen, Sätze und Notationen aus dem Bereich der linearen Algebra einführen. Auf Details und Beweise wird verzichtet, da diese in allen Büchern über lineare Algebra zu finden sind und auch aus der Grundvorlesung bekannt sein sollten.

A.1 Matrizen

Im Rahmen der Vorlesung haben wir uns nur mit reellwertigen Matrizen beschäftigt. Wir definieren $\mathbb{R}^{n,m}$ als die Menge der $n \times m$ Matrizen. Die Menge der Vektoren mit n Elementen werden als $\mathbb{R}^n = \mathbb{R}^{1,n}$ bezeichnet. Einzelne Elemente aus Matrizen oder Vektoren werden durch Indizes adressiert. D. h. $A_{i,j}$ bezeichnet das Element (i, j) in Matrix $\mathbf{A}$. Wir definieren für zwei Matrizen $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,m}$: $\mathbf{A} = \mathbf{B}$ falls $A_{i,j} = B_{i,j}$, $\mathbf{A} \geq \mathbf{B}$ falls $A_{i,j} \geq B_{i,j}$ und $\mathbf{A} > \mathbf{B}$ falls $\mathbf{A} \geq \mathbf{B}$ und $\mathbf{A} \neq \mathbf{B}$ für alle $1 \leq i \leq n$ und $1 \leq j \leq m$. $\mathbf{A}$ ist eine nichtnegative Matrix, falls $\mathbf{A} \geq \mathbf{0}$. Die folgende Definition führt einige spezielle Matrixtypen ein. $\mathbf{A} \in \mathbb{R}^{n,m}$ ist eine quadratische Matrix falls $n = m$.

Definition A.1 *Sei $\mathbf{A}$ eine quadratische $n \times n$ Matrix,*

a) $\mathbf{A}$ *ist eine stochastische Matrix, falls* $\mathbf{A}$ *eine nicht negative Matrix ist und die Summe der Elemente aller Zeilen gleich* 1 *ist.*

b) $\mathbf{A}$ *ist symmetrisch, falls* $\mathbf{A} = \mathbf{A}^T$.

c) $\mathbf{A}$ *ist eine Diagonalmatrix, falls* $A_{i,j} = 0$ *für alle* $i \neq j$.

d) $\mathbf{A}$ *ist eine obere (untere) Dreiecksmatrix, falls* $A_{i,j} = 0$ *für* $i < j$ $(i > j)$.

e) $\mathbf{A}$ *ist tridiagonal, falls* $A_{i,j} = 0$ *für* $|i - j| > 1$.

f) $\mathbf{A}$ *ist eine Bandmatrix, falls* $A_{i,j} = 0$ *für* $|i - j| \leq b < n$.

Definition A.2 *Die Determinante einer quadratischen Matrix* $\mathbf{A}$ *ist definiert als*

$$det(\mathbf{A}) = \sum_{i=1}^{n} (-1)^{j+1} A_{1,j} det(\mathbf{A}[1, j]) \ ,$$

wobei $\mathbf{A}[1, j]$ *eine Matrix ist, die aus* $\mathbf{A}$ *durch Streichen der ersten Zeile und* j*-ten Spalte entsteht und* $det(a) = a$ *für* $a \in \mathbb{R}$.

Definition A.3 *Eine quadratische Matrix* $\mathbf{A}$ *ist singulär, wenn ihre Determinante 0 ist. Andernfalls ist sie regulär oder invertierbar.*

Satz A.1 *Sei* $\mathbf{A}$ *eine* $n \times n$ *Matrix, dann sind folgende Aussagen äquivalent.*

a) $\mathbf{A}$ *ist regulär.*

b) $\mathbf{A}^T$ *ist regulär.*

c) Für jeden Vektor $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \neq \mathbf{0}$ *ist* $\mathbf{xA} \neq \mathbf{0}$.

d) Für jedes $\mathbf{y} \in \mathbb{R}^n$ *existiert ein eindeutiges* $\mathbf{x} \in \mathbb{R}^n$ *mit* $\mathbf{xA} = \mathbf{y}$.

e) Es existiert eine eindeutige Matrix $\mathbf{B}$ *mit* $\mathbf{AB} = \mathbf{BA} = \mathbf{I}$ ($\mathbf{B} = \mathbf{A}^{-1}$).

f) Die Spalten von $\mathbf{A}$ *sind linear unabhängig.*

Definition A.4 *Das charakteristische Polynom* ϕ *einer* $n \times n$ *Matrix ist definiert als* $\phi(\lambda) = det(\lambda\mathbf{I} - \mathbf{A})$. *Die* n*, möglicherweise mehrfach vorkommenden, Lösungen von* $\phi(\lambda) = 0$ *heißen Eigenwerte von* $\mathbf{A}$. *Eine Vektor* $\mathbf{x}$ *mit* $\mathbf{Ax} = \lambda\mathbf{x}$ *heißt Eigenvektor von* $\mathbf{A}$ *zum Eigenwert* λ.

Satz A.2 *Eine* $n \times n$ *Matrix* $\mathbf{A}$ *ist dann und nur dann singulär, wenn ein Eigenwert gleich 0 ist.*

Definition A.5 *Der Spektralradius* $\rho(\mathbf{A})$ *einer quadratischen Matrix* $\mathbf{A}$ *ist definiert als der betragsmäßig größte Eigenwert der Matrix* $\mathbf{A}$.

Definition A.6 *Eine reelle* $n \times n$ *Matrix* $\mathbf{A}$ *ist positiv definit, falls* $\mathbf{x}^T\mathbf{Ax} > 0$ *für alle* $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} \neq \mathbf{0}$.
$\mathbf{A}$ *ist nicht negativ definit, falls* $\mathbf{x}^T\mathbf{Ax} \geq 0$ *für alle* $\mathbf{x} \in \mathbb{R}^n$.

Satz A.3 *Für jede* $n \times n$ *Matrix* $\mathbf{A}$ *ist* $\mathbf{A}^T\mathbf{A}$ *symmetrisch und nicht negativ definit. Das Produkt ist positiv definit genau dann wenn* $\mathbf{A}$ *regulär ist.*

A.2 Normen

Definition A.7 *Eine Norm auf dem* $\mathbb{R}^n$ *ist eine Abbildung, die jedem* $\mathbf{x} \in \mathbb{R}^n$ *eine reelle Zahl* $\|\mathbf{x}\|$ *zuweist und für die folgende Bedingungen gelten*

a) $\|\mathbf{x}\| \geq 0$ *für alle* $\mathbf{x} \in \mathbb{R}^n$.

b) $\|c\mathbf{x}\| = |c|\|\mathbf{x}\|$ *für alle* $\mathbf{x} \in \mathbb{R}^n$ *und* $c \in \mathbb{R}$.

c) $\|\mathbf{x}\| = 0$ *dann und nur dann wenn* $\mathbf{x} = \mathbf{0}$.

d) $\|\mathbf{x} + \mathbf{y}\| = \|\mathbf{x}\| + \|\mathbf{y}\|$ *für alle* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.

Wir unterscheiden folgende Normen

a) Euklidsche Norm $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$.

b) Maximum Norm $\|\mathbf{x}\|_\infty = \max_i |x_i|$.

c) Gewichtete Maximum Norm $\|\mathbf{x}\|_\infty^{\mathbf{w}} = \max_i |x_i/w_i|$ für ein $\mathbf{w} > \mathbf{0}$.

d) l_1-Norm $\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$.

Satz A.4 *Für die einzelnen Normen gelten folgende Ungleichungen*

a) $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty$.

b) $\|\mathbf{x}\|_1 \leq \sqrt{n}\|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_1$.

Satz A.5 *Für zwei beliebige Normen* $\|\ldots\|$ *und* $\|\ldots\|'$ *aus dem* $\mathbb{R}^n$ *gibt es eine Konstante* $c \in \mathbb{R}$, *so daß* $\|\mathbf{x}\| \leq c\|\mathbf{x}\|'$ *für alle* $\mathbf{x} \in \mathbb{R}^n$.

Wir können in ähnlicher Form Matrixnormen definieren, wobei uns hier induzierte Matrixnormen interessieren. Eine induzierte Matrixnorm wird wie folgt konstruiert. Ausgehend von einer Vektornorm $\|\ldots\|$ wird eine induzierte Matrixnorm definiert als

$$\|\mathbf{A}\| = \max_{\mathbf{x}\in\mathbb{R}^n,\ \|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| \tag{A.1}$$

Satz A.6 *Sei* $\mathbf{A} \in \mathbb{R}^{n\times n}$, *dann ist*

a) $\|\mathbf{A}\|_\infty^{\mathbf{w}} = max_i 1/w_i \sum_{j=1}^{n} |A_{i,j}|w_j$, $\|\mathbf{A}\|_\infty = \|\mathbf{A}\|_\infty^{\mathbf{e}}$.

b) $\|\mathbf{A}\|_1 = \max_j \sum_{i=1}^{n} |A_{i,j}|$.

c) $\|\mathbf{A}\|_1 = \|\mathbf{A}^T\|_\infty$.

Satz A.7 *Seien* $\mathbf{A}, \mathbf{B}$ *zwei* $n \times n$ *Matrizen, es gilt*

a) $\|\mathbf{A}\| = \max_{\mathbf{x}\neq\mathbf{0}} \|\mathbf{A}\mathbf{x}\|/\|\mathbf{x}\|$

b) $\|\mathbf{A}\mathbf{x}\| \leq \|\mathbf{A}\|\|\mathbf{x}\|$ *für alle* $\mathbf{x} \in \mathbb{R}^n$

c) $\|\mathbf{A}\mathbf{B}\| \leq \|\mathbf{A}\|\|\mathbf{B}\|$

d) $\|\mathbf{A}\|_\infty \leq n^{1/2}\|\mathbf{A}\|_2$

e) $\|\mathbf{A}\|_1 \leq n^{1/2}\|\mathbf{A}\|_2$

Satz A.8 *Für jede induzierte Norm* $\|\ldots\|$ *und jede* $n \times n$ *Matrix* $\mathbf{A}$ *gilt*

$$\lim_{k\to\infty} \|\mathbf{A}^k\|^{1/k} = \rho(\mathbf{A}) \leq \|\mathbf{A}\|.$$

Satz A.9 *Sei* $\mathbf{A}$ *eine* $n \times n$ *Matrix. Es gilt* $\lim_{k\to\infty} \mathbf{A}^k = 0 \Leftrightarrow \rho(\mathbf{A}) < 1$.
Falls $\mathbf{A}$ *symmetrisch ist, so gilt* $\|\mathbf{A}\|_2 = \rho(\mathbf{A})$.

A.3 Konvergenz

Definition A.8 *Eine Sequenz* $\{x^k | k = 1, 2\ldots\}$ *aus* $\mathbb{R}$ *konvergiert gegen eine reelle Zahl* x, *falls für alle* $\epsilon > 0$ *ein* $k_0 \in \mathbb{N}$ *existiert, so daß* $|x^k - x| < \epsilon$ *für alle* $k > k_0$.
Die Konvergenz ist geometrisch, falls Konstanten $c \geq 0$ *und* $\rho \in [0,1)$ *existieren, so daß* $|x^k - x| < c\rho^k$.

Konvergenz eines Vektor $\mathbf{x} \in \mathbb{R}^n$ wird über die Konvergenz der einzelnen Komponenten definiert. Gleiches gilt für geometrische Konvergenz von Vektoren.

Literaturverzeichnis

[1] G. S. Almasi and A. Gottlieb. *Highly parallel computing.* Benjamin/Cummings, 1989.

[2] D. Bailey. Extra high speed matrix multiplication on the Cray 2. *SIAM Journal on Scientific and Statistical Computing*, 9:603–607, 1988.

[3] A. Berman and R. J. Plemmons. *Nonnegative matrices in the mathematical sciences.* Academic Press, 1979.

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation, numerical methods.* Prentice Hall, 1989.

[5] F. Bodin, P. Beckmann, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: basic ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[6] D. Y. Cheng. *Tools for portable high-performance computing*, pages 865–896. in A. Y. H. Zomaya. *Parallel and Distributed Computing Handbook.* McGraw-Hill, 1996.

[7] G. R. Desrochers. *Principles of parallel and multi-processing.* McGraw-Hill, 1987.

[8] diverse Autoren. *Computational Science Education Project.* CSEP, 1993.

[9] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, Oak Ridge National Laboratory, Mathematical Sciences Section, 1996.

[10] C. C. Douglas. Parallel multilevel and multigrid methods. Technical report, Yale University, Department of Computer Science, 1992.

[11] G. A. Geist et al. *PVM: Parallel Virtual Machine: A user's guide an dtutorial for networked parallel computing.* MIT Press, 1991.

[12] R. Barrett et al. *Templates for the solution of linear systems - buildung blocks for iterative methods.* Siam Press, 1993.

[13] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!* Morgan Kaufmann, 1994.

[14] T. L. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall, 1992.

[15] Golub G, H and C. F. van Loan. *Matrix computations.* Hopkins University Press, 1989.

[16] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32:54–135, 1990.

[17] E. Gelenbe. *Multiprocessor Performance.* Wiley, 1989.

[18] W. Giloi. *Rechnerarchitektur.* Springer, 1993.

[19] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms. *IEEE Parallel & Distributed Technology*, 3:12–21, 1993.

[20] A. Gupta and V. Kumar. Scalability of parallel algorithms for matrix multiplication. Technical report, University of Minnesota, April 1994.

[21] A. Gupta, V. Kumar, and A. Sameh. Performance and scalability of conjugate gradient methods on parallel computers. Technical report, University of Minnesota, November 1992.

[22] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* Teubner Studienbücher, 1991.

[23] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[24] J. L. Hennessy and D. A. P. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufman, 1990.

[25] G. Karypis and V. Kumar. Efficient parallel formulations for some dynamic programming algorithms. Technical report, University of Minnesota, October 1992.

[26] S. Lakshmivarahn and S. K. Dhall. *Analysis and design of parallel algorithms.* Mc Graw Hill, 1990.

[27] F. T. Leighton. *Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[28] B. Lubachevsky and D. Mitra. A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix with unit spectral radius. *Journal of the ACM*, 33:130–150, 1986.

[29] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[30] R. H. Perrott. *Parallel languages*, pages 843–863. in A. Y. H. Zomaya. *Parallel and Distributed Computing Handbook.* McGraw-Hill, 1996.

[31] M. J. Quinn. *Algorithmenbau und Parallelcomputer.* Mc Graw Hill, 1987.

[32] K. R. Reischuk. *Einführung in die Komplexitätstheorie.* Teubner, 1990.

[33] U. Rüde. Technik des Wissenschaftlichen Rechnens. Technical report, TU München, June 1995.

[34] R. Sedgewick. *Algorithmen in C.* Addison Weseley, 1990.

[35] H. S. Stone. *High performance computer architectures.* Addison-Wesley, 1990.

[36] C. Überhuber. *Computernumerik 1 & 2.* Springer, 1995.

[37] I. Wegener. *Effiziente Algorithmen für grundlegende Funktionen.* Teubner, 1989.

[38] S. A. Williams. *Programming models for parallel systems.* Wiley, 1990.

[39] E. Zmijewski and J. R. Gilbert. A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor. *Parallel Computing*, 7:199–210, 1988.

[40] A. Y. H. Zomaya. *Parallel and Distributed Computing Handbook.* McGraw-Hill, 1990.