Vorlesungsskript

Paralleles Höchstleistungsrechnen

Eine anwendungsorientierte Einführung

Version 1.2, 2. Oktober 2008

Peter Bastian

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg, Im Neuenheimer Feld 368
D-69120 Heidelberg, Germany

Inhaltsverzeichnis

Vorwort

In dieser Vorlesung geht es darum, wie man mehrere Computer benutzen kann um die Berechnung eines Problemes zu beschleunigen. Warum sollte man dies tun? Es gibt dafür mindestens drei Gründe. Zunächst die absolute Rechenleistung. Viele Problemstellungen haben unstillbaren Hunger nach Rechenleistung. Sequentielle Rechner erreichen irgendwann prinzipielle physikalische Grenzen (Atomgröße, Lichtgeschwindigkeit), darüberhinaus ist keine Leistungssteigerung mehr möglich. Heutige Rechner haben vor allem das Problem, dass die Daten nicht schnell genug aus dem Speicher gelesen werden können um mit dem Prozessor Schritt zu halten, auch hier sind Parallelrechner eine Lösung. Die leistungsfähigsten heute verfügbaren Rechner sind Parallelrechner[1]. Waren Parallelrechner früher eine Sache für Großforschungseinrichtungen und Universitäten, so hat sich dies mit der Verfügbarkeit von billiger und schneller Ethernet Hardware und dem Betriebssystem LINUX drastisch geändert. Heute kann sich jede winzige Firma einen Parallelrechner leisten. PCs mit zwei Prozessoren sind Standard. PC-basierte Parallelrechner bieten ein besseres Preis–/Leisungsverhältnis als Workstations, vorausgesetzt man hat entsprechend parallelisierte Software. Als letzter Grund soll genannt werden, dass sich viele Anwendungsprobleme in natürlicher Weise parallel formulieren lassen. Man denke etwa an einen Web–Browser wie Netscape. Das Herunterladen einer (oder mehrerer) Dateien und die Darstellung von Webseiten ist (quasi) gleichzeitig möglich. An diesem Beispiel sehen wir auch, dass die parallele Ausführung von Aktionen nicht unbedingt einen Parallelrechner erfordert. Auch ein einzelner Prozessor kann die gleichzeitige Ausführung von Programmen simulieren. Die dabei auftretenden Probleme hinsichtlich der Programmierung sind prinzipiell die selben wie bei echt gleichzeitiger Ausführung. Da heute viele große Anwendungsprogramme Parallelität in dieser Form (multithreading) benutzen ist diese Vorlesung für jeden Programmierer von Interesse.

Parallelverarbeitung wird auf verschiedenen Ebenen in einem Computersystem verwendet. Zunächst ist da die wortparallele Verarbeitung von Zahlen (anstatt einzelne Bits nacheinander zu verarbeiten), übliche Wortbreiten sind 32 oder 64 Bit. Weiterhin können mehrere Operationen (z. B. Addition oder Multiplikation) gleichzeitig (superskalar) oder überlappend (pipelining) verarbeitet werden. Eine weitere Möglichkeit ist die echt gleichzeitige Abarbeitung verschiedener Programme auf separaten Prozessoren. Diese letzte (und allgemeinste) Form der Parallelverarbeitung ist es die uns interessieren wird. Sie bietet potentiell die Möglichkeit die Rechengeschwindigkeit um den Faktor Tausend oder mehr zu steigern. Wir betrachten somit ein paralleles Programm als eine Kollektion sequentieller Programme, die miteinander interagieren um das gestellte Problem gemeinsam zu lösen.

Natürlich eignet sich nicht jedes Problem für die parallele Verarbeitung. Manche Rechenvorschriften (Algorithmen) müssen streng sequentiell ausgeführt werden, andere wiederum lassen Parallelverarbeitung zu, bzw. können entsprechend modifiziert werden. Meistens gibt es in Abhängigkeit der Problemgröße eine Zahl von Prozessoren die sinnvoll eingesetzt werden kann. Von einem skalierbaren Verfahren spricht man dann, wenn für steigende Problemgröße immer mehr Prozessoren sinnvoll eingesetzt werden können.

Wir werden die Problematik des parallelen Rechnens auf drei Ebenen betrachten:

[1] Siehe `http://www.top500.org/`.

I. Rechnerarchitektur

II. Programmiermodelle

III. Algorithmen

Im ersten Teil beschäftigen wir uns mit der Hardware von Parallelrechnern. Im zweiten Teil geht es darum wie die einzelnen Komponenten eines parallelen Programmes miteinander interagieren können und wie eine parallele Ausführung initiiert werden kann. Schließlich geht es im dritten Teil um die Formulierung paralleler Algorithmen für einige ausgewählte Anwendungsprobleme numerischer und nichtnumerischer Art.

Der überwiegende Teil des Textes zielt auf eine prinzipielle Behandlung der Problematik des parallelen Rechnens ab. Von einer bestimmten Rechnerarchitektur bzw. Programmiersprache wird, soweit möglich, abstrahiert. Ergänzt wird dieses Vorgehen durch Einschübe über praxisrelevante Ansätze und Standards. Hier werden Dinge wie LINUX Cluster, Posix threads, MPI, etc., behandelt. Weiterhin wird versucht dem Leser Übungsaufgaben zur freiwilligen Bearbeitung zur Verfügung zu stellen. Die im Text gegebenen Programmbeispiele werden in einer imperativen Programmiersprache mit explizitem Parallelismus formuliert. Wir setzen daher die Kenntnis einer imperativen Programmiersprache (vorzugsweise C, C++, Java, Pascal, Modula-2) voraus.

Version 0.0

Erste Teile des Skriptes wurden von M. Altieri, S. Buijssen, M. Schraud und S. Nauber im Wintersemester 1998 freiwillig (!) getippt. Hierfür möchte ich Ihnen danken.

Version 1.0, März 2000

Diese erste komplette Version des Skriptes wurde von Dirk Oliver Theis im Wintersemester 1999/2000 getippt. Ohne seinen großen Einsatz würde der Text (und die Bilder!) nicht in dieser Form vorliegen und es sei Ihm an dieser Stelle recht herzlich gedankt. Gerne hätte ich noch einige (praktische) Dinge behandelt, wie etwa OpenMP und HPF, sowie remote procedure call und CORBA in Teil II und dynamische Lastverteilung sowie parallele depth first search in Teil III. Überhaupt sollte die Vorlesung durch mehr praktische Übungen am Rechner ergänzt werden (wir haben dies nur einmal, bei der LU-Zerlegung, getan). Somit bleiben noch einige Erweiterungen für die „nächste Iteration“ übrig.

Version 1.1, Oktober 2001

Viele der noch ausstehenden Dinge haben wir im Wintersemester 2000/2001 behandelt. Herrn Roland Schulte danke ich recht herzlich für die Erweiterung des Skriptes. Erfolgreich bearbeitete Übungsaufgaben werden in aufbereiteter Form als „working examples“ als Anhänge angefügt.

Version 1.2, Oktober 2003

Dank Herrn Hendrik Ballhausens sehr genauer Durchsicht des Skriptes ist das Skript nun deutlich fehlerfreier! Danke!

Heidelberg, 2. Oktober 2008 Peter Bastian

1 Ein erster Rundgang

Dieses Kapitel wird anhand eines einfachen Beispiels in die Problematik des parallelen Rechnens einführen sowie eine Notation für parallele Programme vorstellen. Diese Notation soll möglichst einfach sein und von komplizierten praktischen Details absehen. Weiterhin vereinigt sie verschiedene Programmiermodelle.

1.1 Ein einfaches Problem

Wir betrachten die Berechnung des Skalarproduktes zweier Vektoren $x, y \in \mathbb{R}^N$ gegeben durch

$$s = x \cdot y = \sum_{i=0}^{N-1} x_i y_i.$$

Welche Schritte können bei dieser Berechnung gleichzeitig ausgeführt werden?

1. Berechnung der Summanden $x_i y_i$ kann für alle i gleichzeitig erfolgen.
2. Da die Anzahl der Indizes N größer als die Anzahl der Prozessoren P sein darf weisen wir jedem Prozessor eine Teilmenge der Indizes $I_p \subseteq \{0, \ldots, N-1\}$ zu. Jeder Prozessor berechnet dann die Teilsumme $s_p = \sum_{i \in I_p} x_i y_i$.
3. Bei der Berechnung der Gesamtsumme aus den Prozessorteilsummen können wir die Assoziativität nutzen und schreiben z.B. für $P = 8$:

$$s = \underbrace{\underbrace{\underbrace{s_0 + s_1}_{s_{01}} + \underbrace{s_2 + s_3}_{s_{23}}}_{s_{0123}} + \underbrace{\underbrace{s_4 + s_5}_{s_{45}} + \underbrace{s_6 + s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

d.h. die Summe kann in $3 = \operatorname{ld} 8$ aufeinanderfolgenden, jeweils parallelen Schritten berechnet werden.

1.2 Kommunizierende Prozesse

Ein *sequentielles Programm* besteht aus einer Folge von Anweisungen. Der Prozessor bearbeitet diese Anweisungen der Reihe nach ab.

Definition 1.1 Ein *sequentieller Prozess* ist die Abstraktion eines sequentiellen Programmes *in Ausführung*. Ein Prozess hat einen klar definierten *Zustand*. Dies ist die Belegung aller Variablen des Programmes und der nächste auszuführende Befehl (Befehlszähler).

Damit kommen wir zu folgender

Definition 1.2 Unter einer *parallelen Berechnung* stellen wir uns eine Menge von interagierenden sequentiellen Prozessen vor.

Die Ausführung dieser Prozessmenge kann dabei auf einem oder mehreren Prozessoren erfolgen. Stehen weniger Prozessoren wie Prozesse zur Verfügung so schaltet ein Prozessor zyklisch zwischen der Ausführung verschiedener Prozesse um (*engl.* multiprocessing).

Dieses Konzept ist zum einen sehr allgemein und zum anderen auch eine natürliche Erweiterung unserer bisherigen Vorstellung vom Ablauf einer Berechnung. Eine parallele Berechnung wird von einem *parallelen Programm* beschrieben. Parallele Programme werden nach folgendem Muster aufgebaut:

Programm 1.3 (Muster eines parallelen Programmes)

```
parallel <Programmname>
{
    // Der Rest dieser Zeile ist ein Kommentar
    // Sektion mit globalen Variablen, die von allen Prozessen
    // gelesen und geschrieben werden können.
    process <Prozessname-1> [<Kopienparameter>]
    {
        // lokale Variablen, die nur von Prozess <Prozessname-1>
        // gelesen und geschrieben werden können
        // Anwendungen in C-ähnlicher Syntax. Mathematische
        // Formeln oder Prosa zur Vereinfachung erlaubt.
    }
    ...
    process <Prozessname-n> [<Kopienparameter>]
    {
        ...
    }
}
```

In dieser Notation wird jeder Prozess der parallelen Berechnung durch das Schlüsselwort **process** eingeleitet und hat einen eigenen Namen. Im äussersten Block können globale, für alle Prozesse sichtbare Variablen deklariert werden. Variablen werden wie in C deklariert:

double x, $y[P]$;

Initialisieren von Feldern geschieht mittels

int $n[P] = \{1[P]\}$;

Hier wird den P Elementen des Feldes der Wert 1 zugewiesen.

Innerhalb jedes **process**{}–Blockes sind die Variablen nur für den jeweiligen Prozess sichtbar (lokale Variablen).

Die Ausführung eines parallelen Programmes stellen wir uns so vor: Zu Beginn der parallelen Berechnung werden alle globalen Variablen initialisiert und dann die Prozesse zur Ausführung

gebracht. Das parallele Programm ist beendet wenn alle Prozesse beendet sind. Offensichtlich ist in dieser Notation die Prozessmenge statisch, d.h. zur Laufzeit können keine neuen Prozesse gestartet werden. Diese Notation werden wir im folgenden um weitere Elemente anreichern.

Der von uns verwendete (einfache) Prozessbegriff bedarf noch einer weiteren Anmerkung. Üblicherweise (z. B. in dem Betriebssystem UNIX) unterscheidet man zwischen Prozessen (*engl.* processes) und leichtgewichtigen Prozessen (*engl.* threads). Die Idee dabei ist, dass die Umschaltung zwischen zwei leichtgewichtigen Prozessen viel weniger Zeit kostet als zwischen zwei normalen Prozessen. Ein weiterer Unterschied ist, dass Prozesse üblicherweise getrennte Adressräume haben, also keine gemeinsamen Variable besitzen können. Bei unserer oben eingeführten Notation handelt es sich daher eigentlich um leichtgewichtige Prozesse. Wir verwenden trotzdem das Schlüsselwort **process**.

Ein Beispiel mit zwei Prozessen

Als erstes Beispiel betrachten wir die parallele Berechnung des Skalarproduktes zweier Vektoren mit zwei Prozessen:

Programm 1.4 (Skalarprodukt mit zwei Prozessen)

```
parallel two-process-scalar-product
{
    const int N=8;          // Problemgröße
    double x[N], y[N];      // Vektoren
    double s=0;             // Resultat
    process Π1
    {
        int i;
        double ss=0;
        for (i = 0; i < N/2; i++)
            ss += x[i]*y[i];
        s=s+ss;             // Gefahr!
    }
    process Π2
    {
        int i;
        double ss=0;
        for (i = N/2; i < N; i++)
            ss += x[i]*y[i];
        s=s+ss;             // Gefahr!
    }
}
```

Sowohl die beiden Eingabevektoren x und y als auch das Ergebnis $s = x \cdot y$ sind globale Variablen. Die Größe der Vektoren N ist als **const** Variable deklariert. Dies entspricht einer einfachen Ersetzung von N durch die Konstante 8 im gesamten Programmtext. N belegt also keinen Speicher und kann nicht verändert werden.

In Programm 1.4 wurde die Eingabe der Vektoren x, y vernachlässigt. Wir nehmen an, daß x und y bereits zu Beginn die gewünschten Werte enthalten. Genauer gesagt sollen alle globalen Variablen initialisiert sein bevor die Prozesse gestartet werden.

Während auf x und y nur lesend zugegriffen wird, wird die Variable s von beiden Prozessen geschrieben. Was passiert, wenn beide Prozesse *gleichzeitig* die Anweisung $s = s + ss$ ausführen wollen?

1.3 Kritischer Abschnitt

Der Compiler übersetzt die Anweisung $s = s + ss$ der Hochsprache in eine *Folge* von Maschinenbefehlen, etwa:

	Prozess Π_1		Prozess Π_2
1	lade s in R1	3	lade s in R1
	lade ss in R2		lade ss in R2
	add R1 und R2 Ergebnis in R3		add R1 und R2 Ergebnis in R3
2	speichere R3 nach s	4	speichere R3 nach s

Hierbei haben wir die Speicherzugriffe auf s mit den Nummern 1–4 versehen. Im Rahmen der Berechnung werden die Anweisungen jedes Prozesses nacheinander abgearbeitet (dies muss später nicht mehr so sein, siehe Abschnitt über Konsistenzmodelle), die Ausführungsreihenfolge von Anweisungen *verschiedener* Prozesse relativ zueinander ist jedoch nicht festgelegt. In Bezug auf die Speicherzugriffe 1–4 ergeben sich sechs verschiedene mögliche Ausführungsreihenfolgen die in Abb. 1.1 dargestellt sind

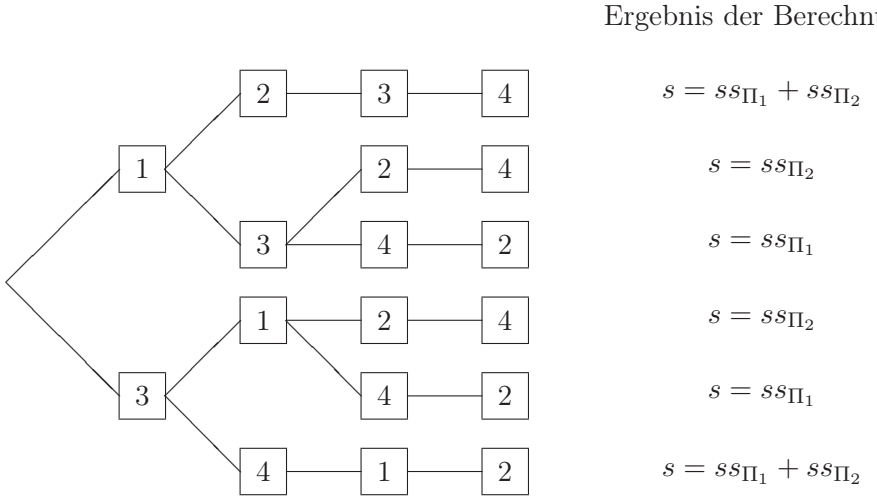


Abbildung 1.1: Mögliche Ausführungsreihenfolgen

Nur die Reihenfolgen 1-2-3-4 oder 3-4-1-2 führen zum korrekten Ergebnis. Offensichtlich muss jeder Prozess seine Speicherzugriffe auf s in Folge ausführen können ohne dabei vom anderen Prozess unterbrochen zu werden.

Die Anweisungsfolgen, die ohne Unterbrechung abgearbeitet werden müssen nennt man einen *kritischen Abschnitt* (*engl.* critical region) eines parallelen Programms. Die Bearbeitung kritischer Abschnitte erfolgt unter *wechselseitigem Ausschluss* (*engl.* mutual exclusion). Dies ist eine Form der *Synchronisation* von Prozessen.

Wichtig ist zu beachten, dass das Problem des wechselseitigen Ausschluss (in diesem Fall) auf Hochsprachenebene nicht bemerkt wird. Für den Moment wollen wir einen kritischen Abschnitt durch eckige Klammern kennzeichnen: $[s = s + ss]$. Diese Notation bedeutet, dass alle Zugriffe auf Variablen, die innerhalb der eckigen Klammern geschrieben werden unter wechselseitigem Ausschluss erfolgen. Im Konfliktfall wählt das Symbol [aus welcher Prozess dabei als erster zum Zug kommt.

Die effiziente Umsetzung eines kritischen Abschnitts erfordert spezielle Hardwareinstruktionen. Es gibt auch allgemeinere Formulierungen kritischer Abschnitte, die wir später behandeln werden (z.B. bis zu $k > 1$ Prozesse dürfen sich gleichzeitig in einem kritischen Abschnitt befinden).

Zur „Schwierigkeit“ der parallelen Programmierung tragen vor allem die vielen möglichen unterschiedlichen Ausführungsreihenfolgen (siehe die Übung 1.1) der Instruktionen paralleler Programme bei.Viele Forscher haben sich darum bemüht, formale Methoden zum Nachweis der Korrektheit von parallelen Programmen zu entwickeln.

1.4 Parametrisieren von Prozessen

In Programm 1.4 verwenden beide Prozesse nahezu identischen Code. Falls wir noch mehr oder gar eine variable Zahl von Prozessen benötigen, kann man dies mit einem „parametrisierten Programm“ realisieren:

Programm 1.5 (Skalarprodukt mit P Prozessoren)

```
parallel many-process-scalar-product
{
    const int N;              // Problemgröße
    const int P;              // Anzahl Prozesse
    double x[N], y[N];        // Vektoren
    double s = 0;             // Resultat
    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i; double ss = 0;
        for (i = N ∗ p/P; i < N ∗ (p + 1)/P; i++)
            ss += x[i]*y[i];
        [s = s + ss];         // Hier warten dann doch wieder alle
    }
}
```

Nach dem Prozessnamen wird die Menge der Prozessparameter angegben. Für jeden möglichen Wert der Variablen p wird ein Prozess gestartet. Die Zahl von Prozessen P ist variabel (aber zur Übersetzungszeit bekannt). Es werden somit die Prozesse Π_0 bis Π_{P-1} gestartet. Prozess Π_k hat eine *lokale* Variable p mit dem Wert k. In Abhängigkeit des Wertes der Variablen p werden die Prozesse unterschiedliche Anweisungen ausführen.

Man kann sich dieses Programm als eine abkürzende Schreibweise für ein Programm mit P Prozessen vorstellen. Parametrisierte Programme werden sehr häufig verwendet, insbesondere in numerischen Anwendungen bei denen verschiedene Prozesse nur auf unterschiedlichen Daten arbeiten aber im Prinzip die selben Instruktionen ausführen. Im englischen wird dafür der Begriff *single program multiple data (SPMD)* verwendet.

1.5 Parallelisieren der Summe

Der kritische Abschnitt $[s = s + ss]$ in Programm 1.5 führt zwar zu einem korrekten Rechenergebnis, stellt jedoch ein potentielles Effizienzproblem dar: Kommen alle Prozesse gleichzeitig an ihrem kritischen Abschnitt an, was ja zu erwarten ist, so werden diese sequentiell abgearbeitet. Wie im Abschnitt 1.1 angedeutet kann die Summe von P Zahlen in $\operatorname{ld} P$ Schritten parallel berechnet werden.

Um dies in einem Programm zu formulieren betrachten wir die Prozessnummern in Binärdarstellung. Im ersten Schritt bilden alle Prozesse, deren letztes Bit 0 ist ein Zwischenergebnis. Im zweiten Schritt berechnen die Prozessoren, deren letzte beiden Bits 0 sind, ein Zwischenergebnis usw. Dies verdeutlicht Abb. 1.2 für $2^3 = 8$ Prozessoren.

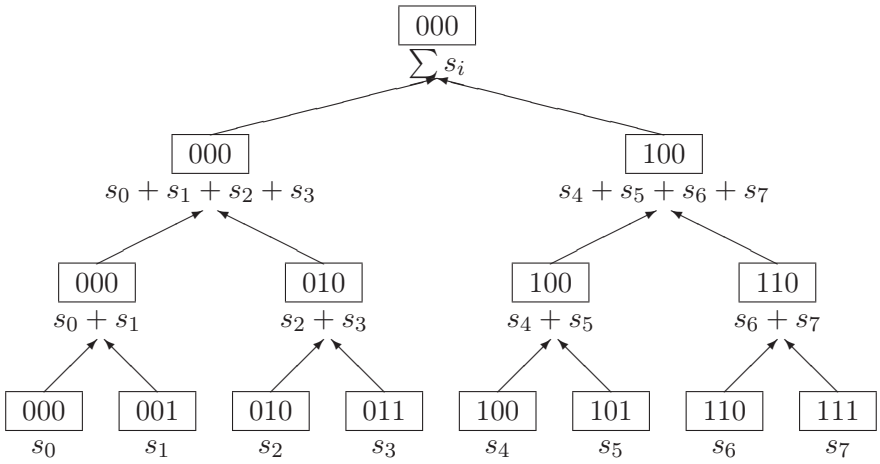


Abbildung 1.2: Parallele Summation

Sei $d = \operatorname{ld} P$. In Schritt $i \in \{1, \ldots, d\}$ addiert Prozessor $b_{d-1}b_{d-2}\ldots b_i \underbrace{0\ldots0}_{i\text{-mal}}$ seinen Wert mit dem des Prozesses $b_{d-1}b_{d-2}\ldots b_i 1 \underbrace{0\ldots0}_{i-1\text{-mal}}$. Die Zwischenergebnisse werden in einem globalen Feld $s[P]$ gespeichert.

Bevor ein Prozess auf das Zwischenergebnis eines anderen Prozesses zugreifen kann, muß er sicher sein, daß dieser auch schon fertig gerechnet hat! Um dies anzuzeigen wird ein Feld $\mathit{flag}[P]$ deklariert und beim Programmstart mit 0–Werten initialisiert. Die Komponente $\mathit{flag}[p]$ nimmt den Wert 1 an, wenn das Zwischenergebnis von Prozess p bereitsteht. Man beachte, daß ein Prozess, sobald er sein Zwischenergebnis einem anderen zur Verfügung stellt, nicht mehr an der Rechnung teilnimmt. Hier das Programm, wobei wir uns auf den Fall $P = 2^d, d \geq 0$ beschränken:

Programm 1.6 (Parallele Summation)

```
parallel parallel-sum-scalar-product
{
    const int d = 4;
    const int N = 100;          // Problemgröße
    const int P = 2^d;          // Anzahl Prozesse
    double x[N], y[N];          // Vektoren
    double s[P] = {0[P]};       // Resultat
```

```
    int flag[P] = {0[P]};          // Prozess p ist fertig

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, r, m, k;

        for (i = N ∗ p/P; i < N ∗ (p + 1)/P; i++)
            s[p]+ = x[i] ∗ y[i];

        for (i = 0; i < d; i++)
        {
            r = p & [∼ (Σ_{k=0}^{i} 2^k)];      // lösche letzten i + 1 bits
            m = r | 2^i;                        // setze Bit i
            if (p == m) flag[m]=1;
            if (p == r)
            {
                while (!flag[m]);               // Bedingungssynchronisation
                s[p] = s[p] + s[m];
            }
        }
    }
}
```

Die zweite **for**–Schleife berechnet die Summe. In jedem Durchlauf werden in p die letzten $i + 1$ Bits gelöscht und der Variablen r zugewiesen (die Operatoren &, $\sim$ und | führen eine bitweise *und*, *nicht* bzw. *oder* Verknüpfung durch. Diese Prozesse müssen in diesem Schritt je eine Addition durchführen. Der andere Operand ist jeweils das Zwischenergebnis von Prozess m. Im Programm 1.6 wartet ein Prozess mit $p == r$ in einer **while**–Schleife darauf, daß das Zwischenergebnis des Partnerprozesses gültig wird. Man nennt dies *aktives Warten* (*engl.* busy-wait). Je nach Hardware kann dies sehr ineffizient sein, da in jedem Durchlauf der (leeren!) Schleife ein Zugriff auf den globalen Speicher notwendig ist.

Durch den busy-wait auf die Variable *flag*$[m]$ werden die zwei beteiligten Prozesse synchronisiert. Man nennt diese Form der Synchronisation eine *Bedigungssynchronisation* (*engl.* condition synchronization), da der eine Prozess erst weiterarbeiten darf, wenn eine bestimmte Bedingung erfüllt ist.

1.6 Lokalisieren

In vielen parallelen Rechnerarchitekturen ist nur der Zugriff auf lokale Daten schnell realisierbar. Zugriff auf globale Variablen ist entweder sehr viel langsamer oder überhaupt nicht möglich. In diesem Fall möchten wir neben den Operationen auch die Daten, d.h. die zwei Vektoren x und y auf die Prozesse aufteilen.

Im nun folgenden Programm werden die Vektoren x und y in jeweils P Teile zerlegt und als lokale Daten gespeichert:

Programm 1.7 (Skalarprodukt mit lokalen Daten)
parallel local-data-scalar-product

```
{
    const int P, N;
    double s = 0;

    process Π [ int p ∈ {0, ..., P − 1}]
    {
        double x[N/P + 1], y[N/P + 1];
                                    // Lokaler Ausschnitt der Vektoren
        int i;
        double ss=0;

        for (i = 0,i < (p + 1) ∗ N/P − p ∗ N/P;i++) ss = ss + x[i] ∗ y[i];
        [s = s + ss; ]
    }
}
```

Wieder haben wir das Belegen der Vektoren x, y vernachlässigt. Weiterhin nehmen wir an, dass N durch P ohne Rest teilbar ist (sonst hätten nicht alle Prozesse dieselbe Anzahl Komponenten der Vektoren). Man beachte, daß nun der Indexbereich in jedem Prozess bei 0 beginnt. Der Zusammenhang zwischen lokalen und globalen Indizes ist hier noch recht einfach:

$$i_{\text{lokal}} = i_{\text{global}} - p * N/P$$

Offensichtlich wird jedoch in Programm 1.7 die Summe noch in einer globalen Variable gebildet. Um dies zu ändern benötigen wir ein weiteres Konzept.

1.7 Nachrichtenaustausch

Nun betrachten wir den Fall, daß keine globalen Variablen außer der Problemgröße und der Prozessorzahl erlaubt sind. Globale Konstanten sind keine eigentlichen Variablen, da sie beim übersetzen durch den entsprechenden Zahlenwert ersetzt werden.

Wir benötigen somit einen anderen Mechanismus, der eine Interaktion der Prozesse erlaubt. Wir wollen annehmen, daß sich die Prozesse gegenseitig *Nachrichten* senden können. Dazu stehen zwei neue Befehle **send** und **receive** mit folgender Syntax zur Verfügung:

send(<Process>,<Variable>)

und

receive(<Process>,<Variable>).

Der **send**–Befehl sendet den Inhalt einer Variablen an einen anderen Prozess und wartet bis dieser die Nachricht in Empfang genommen hat. Der **receive**–Befehl wartet auf eine Nachricht von einem anderen Prozess und speichert diese in der genannten Variablen ab. Da die Operationen solange warten bis die Nachricht tatsächlich gesendet/empfangen wurde werden sie auch *blockierend* genannt. Wir werden später noch Operationen zum Nachrichtenaustausch mit anderer Semantik kennenlernen.

Nun können wir das Programm mit vollständig lokalen Daten und Nachrichtenaustausch formulieren, wobei wir uns wieder auf $P = 2^d, d \geq 0$ und N durch P teilbar beschränken.

Programm 1.8 (Skalarprodukt mit Nachrichtenaustausch)

```
parallel message-passing-scalar-product
{
```

```
const int d, P= 2^d, N;        // Konstanten!

process Π [int p ∈ {0,...,P − 1}]
{
    double x[N/P], y[N/P];// Lokaler Ausschnitt der Vektoren
    int i, r, m;
    double s = 0, ss;

    for (i = 0,i < (p + 1) ∗ N/P − p ∗ N/P;i++) s = s + x[i] ∗ y[i];
    for (i = 0,i < d,i++)      // d Schritte
    {
        r = p & [∼ (Σ_{k=0}^{i} 2^k)];
        m = r | 2^i;
        if (p == m)
            send(Π_r,s);
        if (p == r)
        {
            receive(Π_m,ss);
            s = s + ss;
        }
    }
}
```

Man beachte, daß die Prozesse über den Nachrichtenaustausch implizit synchronisiert werden, das *flag*–Feld ist nicht mehr notwendig. Am Ende des Programmes enthält die Variable s des Prozesses 0 das Endergebnis.

Die Dekomposition der Datenmenge und die Lokalisierung der Indexbereiche macht die Programmierung für Rechner mit Nachrichtenaustausch in der Regel schwieriger als für solche mit gemeinsamem Speicher. Andererseits kann das Prinzip des Nachrichtenaustausches auch für Rechner mit sehr vielen Prozessoren realisiert werden, wohingegen Rechner mit gemeinsamen Speicher meist aus deutlich weniger als hundert Prozessoren bestehen.

1.8 Leistungsanalyse

Wir wollen nun der Frage nachgehen, wie leistungsfähig ein paralleler Algorithmus ist. Grundlage der Analyse der Leistungsfähigkeit ist der Vergleich von sequentieller und paralleler Version eines Algorithmus. Betrachten wir das Skalarprodukt als Beispiel. Die Laufzeit der sequentiellen Version beträgt in Abhängigkeit der Vektorlänge

$$T_s(N) = 2Nt_a,$$

wobei t_a die Zeit für eine arithmetische Operation ist.

Die Laufzeit der parallelen Variante in Programm 1.8 ist

$$T_p(N,P) = \underbrace{2\frac{N}{P}t_a}_{\text{lokales Skalarprodukt}} + \underbrace{\operatorname{ld} P(t_m + t_a)}_{\text{parallele Summe}},$$

wobei t_m die Zeit ist, die für das Versenden einer Fließkommazahl notwendig ist.

Als *Speedup* S bezeichnet man das Verhältnis von sequentieller zu paralleler Laufzeit:

$$S(N,P) = \frac{T_s(N)}{T_p(N,P)} = \frac{2Nt_a}{2\frac{N}{P}t_a + \operatorname{ld} P(t_m + t_a)}$$

$$= \frac{P}{1 + \frac{P}{N}\operatorname{ld} P\frac{t_m+t_a}{2t_a}}$$

Da $\frac{P}{N}\operatorname{ld} P\frac{t_m+t_a}{2t_a} \geq 0$, gilt $S(N,P) \leq P$.
Als *Effizienz* bezeichnet man

$$E(N,P) = \frac{S(N,P)}{P} = \frac{1}{1 + \frac{P}{N}\operatorname{ld} P\frac{t_m+t_a}{2t_a}}.$$

Wegen $S(N,P) \leq P$ gilt $E \leq 1$.

Asymptotisch lassen sich folgende Aussagen machen, die sehr typisch für viele parallele Algorithmen sind:

1. festes N: $\lim_{P\to\infty} E(N,P) = 0$

   In der Praxis ist natürlich im obigen Beispiel $P \leq N$ notwendig. Aber es gilt auch folgende Monotonie:

   $$E(N,P+1) \leq E(N,P)$$

2. festes P, wachsendes N: $\lim_{N\to\infty} E(N,P) = 1$ Für welches Verhältnis $\frac{P}{N}$ „akzeptable“ Effizienzwerte erreicht werden, regelt der Faktor $\frac{t_m+t_a}{t_a}$, das Verhältnis von Kommunikations- zu Rechenzeit.

3. *Skalierbarkeit* für ein gleichzeitiges Anwachsen von N und P in der Form $N = kP$:

   $$E(kP,P) = \frac{1}{1 + \operatorname{ld} P\frac{t_m+t_a}{2t_a k}}$$

   Die Effizienz ist bei hinreichend großem k sehr gut und fällt nur langsam mit steigendem P (und Problemgröße N) ab. Man bezeichnet einen Algorithmus mit diesem Verhalten als „*gut skalierbar*“.

1.9 Ausblick

Nach diesem kleinen Rundgang sollen nun die angesprochenen Dinge vertieft werden. Dazu ist zunächst ein Grundverständnis der parallelen Rechnerarchitektur notwendig, auf denen parallele Programme ablaufen sollen. Insbesondere kommen dabei die Unterschiede (aber auch Gemeinsamkeiten!) der Systeme mit gemeinsamem Speicher und Nachrichtenaustausch zur Sprache.

In einem Abschnitt über Parallele Programmierung werden wir verschiedene Formen der Synchronisation und der globalen Kommunikation vertiefen (ein Beispiel war die parallele Summenbildung).

Das damit erlangte Grundwissen ermöglicht uns die Parallelisierung diverser Algorithmen numerischer und nichtnumerischer Art.

1.10 Übungen

Übung 1.1 Zeigen Sie, dass es bei n Prozessen die je m Anweisungen ausführen genau $(mn)!/(m!)^n$ verschiedene Ausführungsreihenfolgen gibt.

Übung 1.2 Wie könnte man in Programm 1.4 den kritischen Abschnitt vermeiden? Als Hinweis sei genannt, dass man eine Bedingungssynchronisation verwenden kann. Wann ist dies eine sinnvolle Alternative? Geht das immer?

Übung 1.3 Überlegen Sie wie Programm 1.6 auf den Fall $N/P \notin \mathbb{N}$ erweitert werden kann.

Übung 1.4 Was passiert in Programm 1.6 wenn man mehrmals hintereinander eine Summe berechnen möchte? Erweitern Sie das Programm so, dass man beliebig oft eine Summe bilden kann.

Übung 1.5 Überlegen Sie wann aktives Warten eine effiziente Implementierung der Bedingungssynchronisation ist und wann nicht.

Übung 1.6 Erweitern Sie Programm 1.8 so, dass am Ende alle Prozesse das Ergebnis kennen (diese Operation bezeichnet man als *Broadcast*).

Teil I

Rechnerarchitektur

2 Sequentielle Rechnerarchitekturen

In diesem Kapitel behandeln wir den grundlegenden Aufbau sequentieller Rechner und einiger gebräuchlicher Beschleunigungstechniken. Sequentielle Rechner sind dadurch charakterisiert, dass sie nur einen Instruktions– und Datenstrom verarbeiten.

2.1 Der von–Neumann–Rechner

John von Neumann war ein berühmter Mathematiker am Institute for Advanced Study der Princeton University (USA). Im Jahr 1944 hörte er vom (damals geheimen) ENIAC–Projekt, dem ersten elektronischen Computer, und interessierte sich dafür ihn zu verbessern. In dem Bericht „First Draft of a Report on the EDVAC“ aus dem Jahr 1945 wurden diese Verbesserungen zusammengefasst (Einige Leute denken, dass die Anerkennung dafür nicht nur von Neumann gebührt, siehe die Diskussion in (Hennessy und Patterson 1996)). Eine wesentliche Neuerung die von Neumann einführte war die des gespeicherten Programmes, d.h. der Speicher des Computers enthält sowohl Daten als auch das Programm. Den grundlegenden Aufbau des von–Neumann–Rechners zeigt Abb. 2.1.

Der Speicher M (*engl.* memory) enthält sowohl Daten als auch Instruktionen (Befehle). Die Instruktionseinheit IU (*engl.* instruction unit) liest den nächsten Befehl aus dem Speicher (Ort wird durch den Programmzähler angegeben), dekodiert ihn und steuert das Rechenwerk ALU (*engl.* arithmetic logic unit) entsprechend an. Die Operanden für das Rechenwerk werden üblicherweise aus einem Satz von Registern gelesen, können aber auch aus dem Speicher gelesen werden. Ergebnisse werden in ein Register geschrieben. Instruktionseinheit und Rechenwerk werden als Prozessor (*engl.* CPU) bezeichnet).

Die einzelnen Phasen des Befehlszyklus werden in einem festen Zeittakt abgearbeitet. Die Geschwindigkeit mit der ein solcher Rechner Programme auszuführen vermag ist wird zum einen durch die Taktrate bestimmt und zum anderen durch die Rate mit der Daten und Befehle zwischen Prozessor und Speicher transportiert werden können.

Transaktionen zwischen Prozessor und Speicher werden über einen Bus durchgeführt. Ein Bus besteht aus einem Bündel elektrischer Leitungen über die die Signale laufen. Der Bus dient dazu den Prozessor mit verschiedenen Teilen des Computers zu verbinden, z.B. dem Speicher oder Ein–/Ausgabegeräten. In einer Vermittlungsphase (*engl.* bus arbitration) wird festgelegt welche am Bus angeschlossene Einheiten als nächstes kommunizieren. Der Datenaustausch auf dem Bus wird durch ein Busprotokoll geregelt. Üblicherweise gibt es viele verschiedene Busse in einem Computersystem, innerhalb der CPU und auch außerhalb.

2.2 Pipelining

Das Pipelining, zu deutsch etwa „Fließbandtechnik“, ist eine häufig verwendete Beschleunigungstechnik die sowohl auf Hardware– als auch auf Softwareebene eingesetzt wird. Wir werden sie zunächst abstrakt beschreiben und dann einige konkrete Anwendungsbeispiele geben.

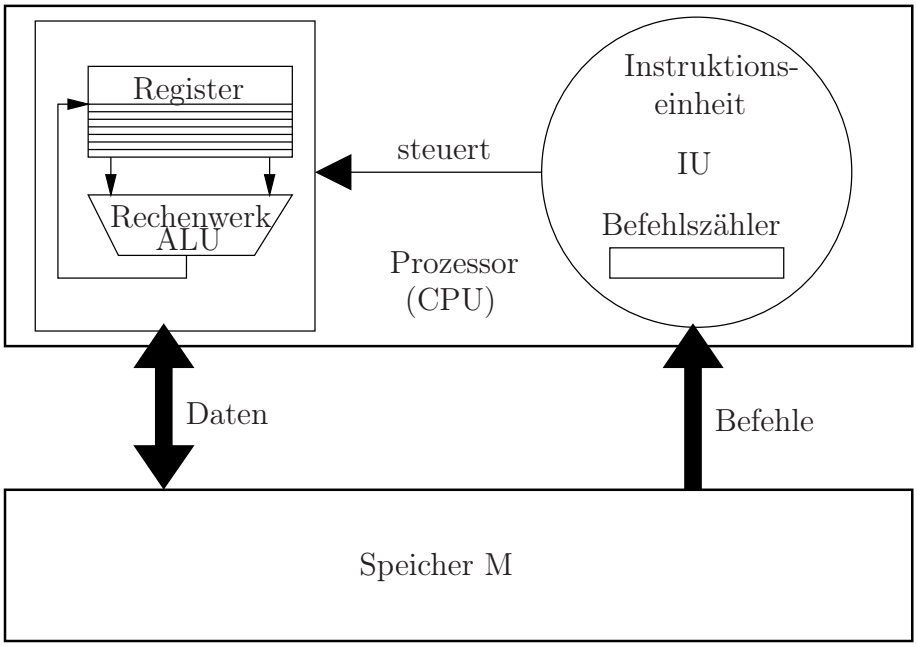


Abbildung 2.1: Schematischer Aufbau des von–Neumann–Rechners.

Unter Pipelining versteht man die gleichzeitige, überlappende Verarbeitung mehrerer Operationen. Voraussetzungen für die Anwendbarkeit von Pipelining sind:

- Eine Operation $OP(x)$ muss auf viele Operanden $x_1, x_2, \ldots$ in Folge angewandt werden.
- Die Operation kann in $m > 1$ Teiloperationen (oder auch Stufen) zerlegt werden, die in (möglichst) gleicher Zeit bearbeitet werden können.
- Ein Operand x_i darf nur mit Einschränkungen das Ergebnis einer früheren Operation sein (siehe unten).

Abb. 2.2 zeigt das Prinzip des Pipelining für den Fall von $m = 4$ Teiloperationen (TOP). Im

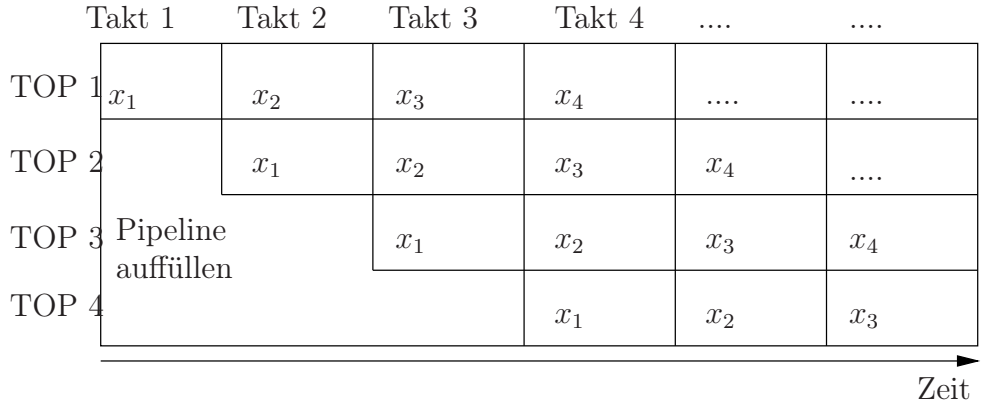


Abbildung 2.2: Illustration des Pipelining.

ersten Takt wird die erste Teiloperation auf den ersten Operanden x_1 angewandt, dann im zweiten Takt die zweite Teiloperation auf den ersten Operanden und zeitgleich die erste Teiloperation auf den zweiten Operanden und so weiter. Nach m Takten ist die Bearbeitung des ersten Operanden beendet, dann wird in jedem Takt eine weitere Berechnung beendet. Pipelining erhöht also den Durchsatz (Operationen/Zeit), nicht jedoch die Bearbeitungszeit für einen einzelnen Operanden. Die Bearbeitungszeit für einen einzelnen Operanden kann sogar etwas höher sein als ohne Pipelining (ungleicher Zeitbedarf für Einzeloperationen, Synchronisation zwischen den Stufen).

Der Zeitbedarf für die Verarbeitung von N Operanden beträgt

$$T_P(N) = (m + N - 1)\frac{T_{OP}}{m}, \tag{2.1}$$

die Beschleunigung somit

$$S(N) = \frac{T_S(N)}{T_P(N)} = \frac{N * T_{OP}}{(m + N - 1)\frac{T_{OP}}{m}} = m\frac{N}{m + N - 1}. \tag{2.2}$$

Für $N \to \infty$ geht die Beschleunigung gegen m.

Probleme gibt es dann, wenn die Bearbeitung eines Operanden von einem früheren Ergebnis abhängt. So darf der Operand x_i nur das Ergebnis einer früheren Operation $OP(x_{i-k})$ sein, falls $k \geq m$ gilt.

Wir betrachten nun verschiedene Anwendungen des Pipelining–Prinzips.

Instruktionspipelining

Hierunter versteht man die überlappende Verarbeitung mehrerer Befehlszyklen. Die dabei üblichen „Teiloperationen“bzw. Stufen sind:

- Instruktion holen (*fetch*),
- Instruktion decodieren (*decode*),
- Instruktion ausführen (*execute*)
- Ergebnis zurückschreiben (*write back*),

und die Operanden sind die Maschinenbefehle.

Wesentliche Voraussetzung (siehe oben!) zur Ausnutzung des Instruktionspipelining ist ein entsprechend homogener Befehlssatz bei dem für alle zu bearbeitenden Befehle die einzelnen Stufen jeweils in der gleichen Zeitspanne ausgeführt werden können. Agressives Ausnutzen von Instruktionspipelining ist das wesentliche Merkmal der sog. RISC–Prozessoren (reduced instruction set computer). Datenabhängigkeiten treten beim Instruktionspipelining in der Form von bedingten Sprungbefehlen auf. Viele heute übliche Prozessoren (Intel Pentium, PowerPC) enthalten eine Einheit die das Sprungziel von bedingten Sprungbefehlen vorhersagen sollen (*engl.* branch prediction unit).

Effektives Ausnutzen von Instruktionspipelining erfordert Optimierungen des Instruktionsstromes durch den Compiler. Eine häufig verwendete Optimierungstechnik ist das Abrollen von Schleifen (*engl.* loop unrolling). Hierzu betrachten wir folgende Schleife:

```
for(i = 0; i < n; i++)
    x[i] = a * x[i];
```

Hier gibt es nur wenige Befehle innerhalb der Schleife. Dies bedeutet, dass der Code zur Implementierung der Schleife (Zähler dekrementieren, bedingter Sprungbefehl) etwa soviel Befehle beansprucht wie der eigentliche Nutzcode, die Skalierung eines Vektors. Ausserdem hat der Compiler wenig Möglichkeit durch Umordnen der Maschinenbefehle die Ausnutzung der Pipeline zu optimieren.

Durch das Abrollen der Schleife vermeidet man beides. Dazu wird der Schleifenkörper vervielfältigt und entsprechend angepasst, so dass die Schleife weniger oft durchlaufen wird:

```
for(i = 0; i < n; i+=4)
{
    x[i] = a * x[i];
    x[i + 1] = a * x[i + 1];
    x[i + 2] = a * x[i + 2];
    x[i + 3] = a * x[i + 3];
}
```

Hierbei haben wir angenommen, dass n ein Vielfaches von 4 ist. Jeder gute Compiler wird diese Transformation automatisch durchführen (er macht das üblicherweise besser als Menschen).

Arithmetisches Pipelining

Anwendung des Pipelining–Prinzips auf arithmetische Operationen wie Addition und Multiplikation. Mögliche Teiloperationen sind dabei etwa:

- Herstellen gleicher Exponenten bei beiden Argumenten,
- Addition der Mantisse,
- Normieren der Mantisse und Abgleich des Exponenten.

Besonders vorteilhaft bei der Verarbeitung langer Kolonnen von Zahlen (Vektoren) mit gleichen Befehlen, wie z. B. bei Skalarprodukt oder Matrix mal Vektor. Rechner mit arithmetischem Pipelining werden auch Vektorrechner genannt.

Verschränkter Speicher

(*engl.* interleaved memory) Insbesondere die oben erwähnten Vektorrechner benötigen eine sehr hohe Datentransferrate vom Hauptspeicher in den Prozessor und zurück. Dies kann man dadurch erreichen, daß der Hauptspeicher aus mehreren voneinander unabhängigen Modulen aufgebaut wird, die eine überlappende Bearbeitung von Zugriffen erlauben. Die Adressierung erfolgt derart, daß konsekutive Speicheradressen verschiedenen Modulen zugeordnet werden. Dies erreicht man einfach dadurch, dass die niedrigsten Bits der Speicheradresse zur Auswahl des Speichermoduls verwendet werden (*engl.* low order interleaving). Verschränkter Speicher benötigt ein Busprotoll, das mehrere laufende Transaktionen zulässt.

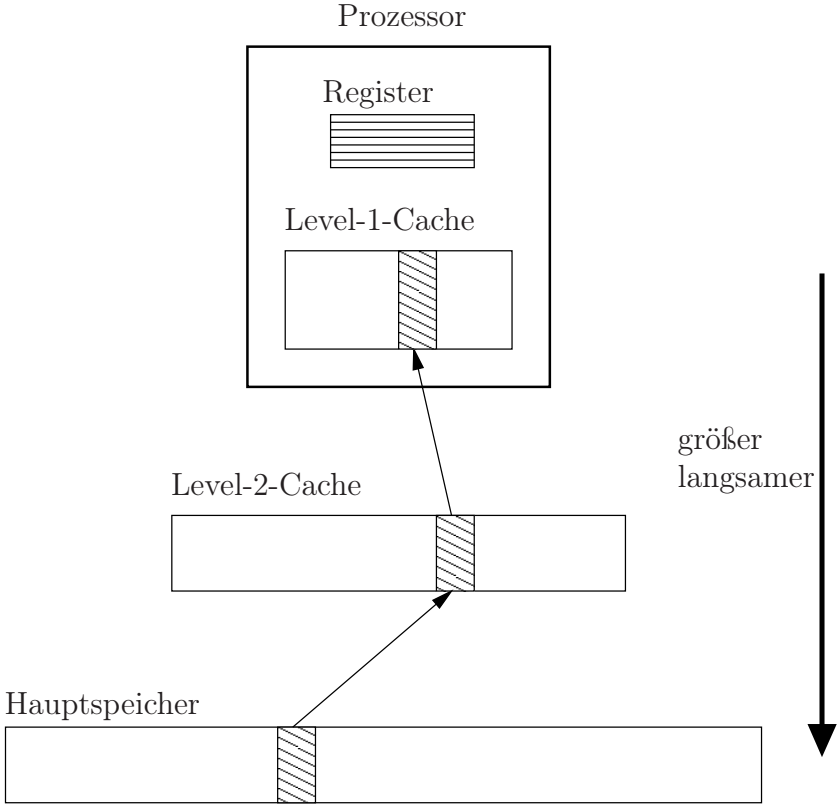


Abbildung 2.3: Beispiel einer Speicherhierarchie.

2.3 Mehrfache Funktionseinheiten

Es ist möglich mehrere Rechenwerke (z.B. mehrere Additions–, Multiplikations– und Integereinheiten) vorzusehen und somit mehr als einen Befehl pro Takt zu bearbeiten. So eine Architektur bezeichnet man als „superskalar“ (*engl.* superscalar). Typischerweise wird superskalare Bearbeitung mit Instruktionspipelining verknüpft.

Jeder Befehl benötigt gewisse Resourcen für seine Ausführung, wie etwa die Verfügbarkeit von Operanden und die Verfügbarkeit einer Verarbeitungseinheit. Nun kann es vorkommen, dass ein Befehl darauf warten muss bis die benötigten Resourcen verfügbar sind, ein nachfolgender Befehl aber schon zur Ausführung bereit wäre. Erlaubt eine Architektur die dynamische Zuordnung von Resourcen zu Befehlen und deren Ausführung sobald alle benötigten Resourcen zur Verfügung stehen spricht man von „außer–der–Reihe–Ausführung“ (*engl.* out–of–order execution).

Mit dieser Technik kann man nur einen relativ kleinen Parallitätsgrad von 3 bis 5 für allgemeine Aufgaben erreichen. Für eine detaillierte Beschreibung sei auf (Hennessy und Patterson 1996) verwiesen.

2.4 Caches

Moderne Prozessoren sind sehr viel schneller als der Hauptspeicher. Typische Prozessoren haben Taktfrequenzen von 500 MHz und können bis zu zwei Befehle pro Takt ausführen (z.B. Pentium II). Normaler DRAM Speicher hingegen arbeitet mit einer Taktfrequenz von 100 MHz und in

7 Takten können 4 Worte (à 64 Bit) gelesen werden (100MHz SDRAM). Somit kann nicht in jedem Takt des Prozessors ein Operand aus dem Hauptspeicher gelesen werden. Im Prinzip ist das Problem kein technisches sondern ein ökonomisches. Man kann sehr wohl Speicher bauen, der so schnell wie der Prozessor arbeitet nur ist der Preis pro Byte sehr viel höher als bei langsamerem Speicher.

Die Lösung für dieses Problem ist die Speicherhierarchie. Man stattet den Rechner mit Speichern unterschiedlicher Größe und Geschwindigkeit aus. Abb. 2.3 zeigt eine Speicherhierarchie mit drei Ebenen. Im Prozessor befinden sich die Register und ein sogenannter Level-1-Cache. Dieser arbeitet typischerweise mit der selben Taktrate wie der Prozessor (typische Größe sind wenige KByte). Außerhalb des Prozessors befindet sich der etwas größere Level-2-Cache mit einer typischen Größe von 512 KByte bis 2 MByte. Darunter befindet sich der Hauptspeicher, der typischerweise noch einmal um den Faktor 100 bis 500 größer ist.

Greift der Prozessor lesend auf eine Speicheradresse zu, so wird überprüft ob sich eine Kopie der Speicherzelle im Cache befindet (wir betrachten nur zwei Ebenen, im Prinzip läuft auf jeder Ebene das selbe Schema ab). Wenn ja, wird die Kopie im Cache verwendet, ansonsten wird der Inhalt der Speicherzelle aus dem Speicher geholt. Genau genommen wird nicht nur die gewünschte Speicherzelle sondern ein ganzer Speicherblock, der die genünschte Adresse enthält, in den Cache kopiert. Dabei nimmt man an, dass typische Programme mit hoher Wahrscheinlichkeit auch auf Speicherzellen in der Umgebung der ursprünglichen Adresse zugreifen. Man spricht vom Lokalitätsprinzip. Die Blöcke in denen Daten zwischen Speicher und Cache transferiert werden heißen auch *cache lines*. Typische Größe einer Cache–Line ist 16 Bytes bis 128 Bytes.

Vier Strategien bestimmen die Funktionsweise eines Cache–Speichers: Platzierungsstrategie, Identifikationsstrategie, Ersetzungsstrategie und Schreibstrategie. Diese wollen wir nun der Reihe nach betrachten.

Platzierung

Nehmen wir an der Hauptspeicher besteht aus N Blöcken und der Cache–Speicher aus M Blöcken. Bei einem Cache mit direkter Zuordnung (*engl.* direct mapped cache) wird Block i des Haupspeichers in Block $j = i \mod M$ des Cache geladen. Dies bedeutet, dass zwei Blöcke i_1, i_2 mit $i_1 \mod M = i_2 \mod M$ in den selben Cache–Block geladen werden müssen. Dieser Konflikt wird vermieden bei einem voll assoziativen Cache (*engl.* fully associative cache). Hier kann ein Hauptspeicherblock in jeden beliebigen Cache–Block kopiert werden. Als Kompromiss ergibt sich der Cache–Speicher mit k–facher Assoziativität (*engl.* k-way associative cache). Hier besteht der Cache aus M Mengen (*engl.* sets) zu je k Blöcken. Block i des Haupspeichers kann in einen der k Blöcke in der Menge $i \mod M$ kopiert werden. Typische Werte für k sind 2 oder 4.

Abb. 2.4 illustriert die verschiedenen Formen der Cache Organisation. Der Hauptspeicher besteht aus 32 Blöcken, der Cache jeweils aus 8 Blöcken (die Größe der Cache–Lines ist unwichtig!). Im voll assoziativen Fall kann Block Nr. 12 in jeden Cache–Block geladen werden, bei direkter Zuordnung nur in den mit der Nr. 4 (wegen $12 \mod 8 = 4$). Im Fall 2-facher Assoziativität besteht der Cache aus 4 Teilmengen zu je 2 Blöcken. Wegen $12 \mod 4 = 0$ geht Block 12 in Menge 0 und kann dort in jeden der beiden Blöcke kopiert werden.

Identifizierung

Wie findet man nun heraus ob ein bestimmter Block bereits im Cache ist? Dazu wird zusätzlich für jeden Cache–Block die Blockrahmenadresse (*engl.* block frame address) im Cache–Tag abgespeichert. Die Blockrahmenadresse des Blockes i im Hauptspeicher ist $i \div M$. Bei einem Cache

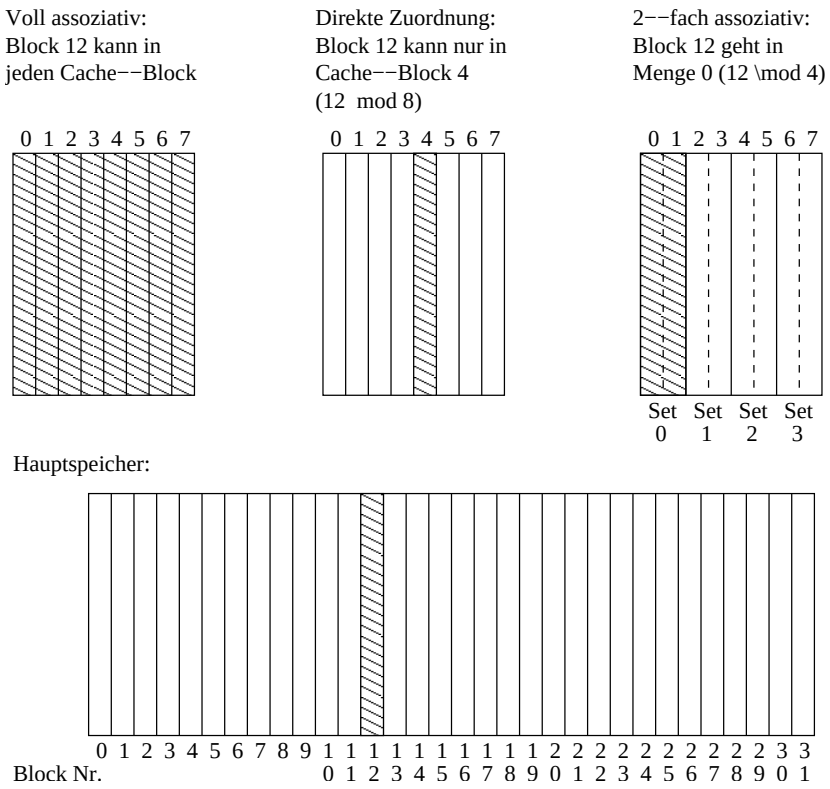


Abbildung 2.4: Illustration verschiedener Cache–Organisationen.

mit direkter Zuordnung ist nur genau ein Vergleich der Blockrahmenadresse mit dem zugehörigen Cache–Tag notwendig. Im Falle eines k–fach assoziativen Cache sind k Vergleiche notwendig, die von der Hardware parallel durchgeführt werden. Bei einem voll assozativen Cache sind M Vergleiche notwendig (dies erklärt warum diese Variante praktisch nicht verwendet wird).

Zusätzlich zum Cache–Tag muss man noch wissen ob die im Cache–Block enthaltene Information überhaupt gültig ist. Z.B. ist ja nach dem Einschalten des Rechners noch kein Block im Cache gültig. Dazu erhält jeder Cache–Block noch ein sog. Valid–Flag. Im Rahmen der später zu besprechenden Multiprozessorsysteme mit Cache kann es auch vorkommen, dass ein anderer Prozessor einen Hauptspeicherblock modifiziert und somit die Cache–Speicher aller übrigen Prozessoren eventuell einen Cache–Eintrag invalidieren müssen.

Ersetzung

Irgendwann wird unweigerlich die Situation eintreten, dass alle Cache–Blöcke gültige Information enthalten und ein noch nicht vorhandener Block in den Cache geladen werden muss. In diesem Fall muss ein anderer gültiger Block entfernt werden. Welcher Block entfernt wird hängt wieder von der Organisation des Cache ab. Bei einem Cache mit direkter Zuordnung gibt es nur einen möglichen Block der entfernt werden kann und die Entscheidung ist trivial. Bei einem Cache mit k–facher Assoziativität stehen natürlich k Blöcke zur Auswahl. Aufgrund des Lokalitätsprinzips würde man den Block entfernen auf den am längsten nicht mehr zugegriffen wurde (*engl.* least recently used, LRU). Eine andere mögliche Strategie ist die zufällige Auswahl eines Blockes. Untersuchungen in (Hennessy und Patterson 1996) zeigen, dass LRU und zufällige Auswahl für große Caches praktisch gleich sind (unabhängig von k, dem Assoziativitätsfaktor).

Schreiben

Obwohl Lesezugriffe auf den Speicher wesentlich häufiger sind als Schreibzugriffe wird jedes vernünftige Programm auch einmal Werte zurückschreiben. Hier gibt es zwei übliche Strategien. Zum einen kann man bei Schreibzugriffen immer das entsprechende Wort im Cache (angenommen der Block befindet sich bereits im Cache) und im Hauptspeicher modifizieren (*engl.* write through) oder man modifiziert den Eintrag im Cache und wartet mit dem Zurückschreiben in den Haupspeicher bis der Cache–Block aus dem Cache verdrängt wird (*engl.* write back). Beide Strategien haben Vor- und Nachteile. Im ersten Fall sind Hauptspeicher und Cache immer konsistent (gut für Multiprozessorsysteme) und falls der Block noch nicht im Cache ist ist kein zusätzliches Lesen des Blockes notwendig (*engl.* cold write, write miss). Im zweiten Fall erfordern mehrere Schreibzugriffe auf den selben Block nur ein Zurückschreiben des ganzen Blockes (dies ist schneller, da hierfür spezielle Transfermodi der DRAM Speicher verwendet werden). Bei einem Write–Back–Cache enthält jeder Cache–Block ein sog. Dirty–Bit welches anzeigt ob der Inhalt des Cache mit dem entsprechenden Block des Hauptspeichers übereinstimmt oder nicht.

Das Lokalitätsprinzip und seine Auswirkungen

Das klassische von–Neumann–Programmiermodell geht von gleicher Zugriffszeit auf alle Speicherzellen aus. Bei Systemen mit Cache–Speicher ist diese Annahme nicht mehr erfüllt. Beim Entwurf von Programmen muss man darauf achten einmal in den Cache geladene Daten so oft wie möglich wiederzuverwenden. Dazu betrachten wir nun das Beispiel der Multiplikation zweier Matrizen. Für weitere Beispiele zu diesem Thema sei auf die Übungsaufgaben verwiesen.

Wir betrachten das Produkt zweier $n \times n$ Matrizen $C = AB$. Alle Matrizen seien in zweidimensionalen Feldern gespeichert. Legen wir die Programmiersprache C zugrunde, so sind zweidi-

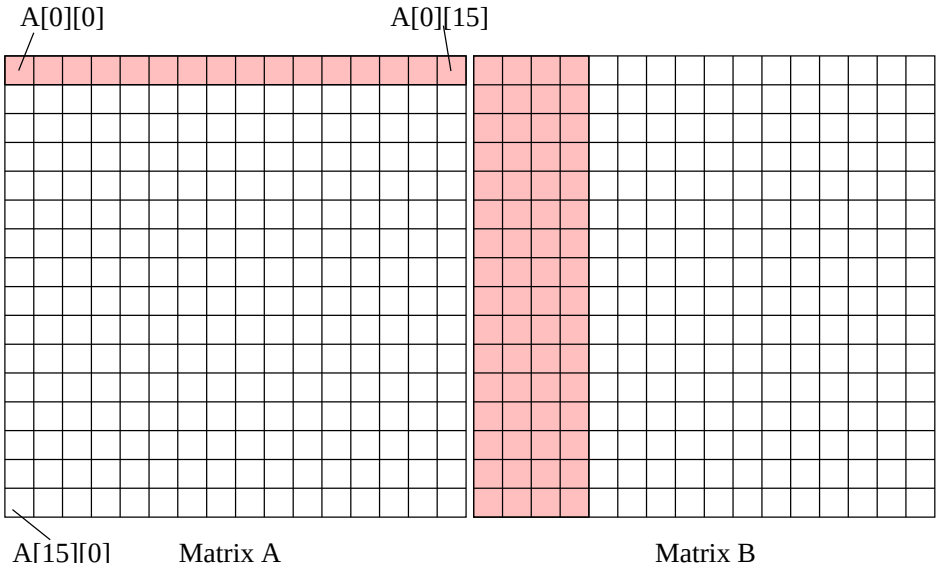


Abbildung 2.5: Daten im Cache nach Berechnung von $C[0][0]$.

mensionale Felder zeilenweise abgespeichert. Der naive Algorithmus für die Matrixmultiplikation lautet:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

(C sei schon mit Nullen initialisiert) Überlegen wir nun wie ein Cache–Speicher bei der Matrixmultiplikation verwendet wird. Nehmen wir an der verwendete Rechner habe einen Cache mit 32 Byte Cache–Lines. Bei 8 Byte pro Fließkommazahl passen genau vier Matrixelemente in eine Cache–Line. Abb. 2.5 zeigt welche Daten sich nach der Berechnung von $C[0][0]$ im Cache befinden. Auf A wird in der innersten Schleife zeilenweise zugegriffen, auf B spaltenweise. Da immer ganze Cache–Lines geladen werden, befinden sich die ersten vier Elemente jeder Zeile von B im Cache. Solange n klein genug ist, dass eine Zeile von A und vier Spalten von B in den Cache passen läuft unser Programm recht effizient. So kann etwa der Wert von $C[0][1]$ in Abb. 2.5 ohne einen weiteren Hauptspeicherzugriff berechnet werden. Nimmt man an, dass A, B und C vollständig in den Cache passen, so muss man für die $2n^3$ Fließkommaoperationen nur $2n^2$ 64–Bit Worte lesen und n^2 Worte schreiben. Für genügend großes n hat man also deutlich mehr Fließkommaoperationen wie Speicherzugriffe.

Allerdings wird ein Level–1–Cache nicht viel mehr als 2000 Fließkommazahlen speichern können (16 KByte) und somit ist $n < 25$. Bei $n > 400$ (2000/5) werden wir nicht einmal für einen Durchlauf der innersten Schleife alle Daten im Cache halten können. Dann ist bei beinahe jedem Zugriff auf ein Matrixelement das Laden einer Cache–Line notwendig. Die Datenlokalität des Algorithmus kann entscheidend verbessert werden indem man die beteiligten Matrizen in (kompatible) Blöcke zerlegt, die jeweils in den Cache passen. Angenommen die Blockgröße sei $m \times m$ und n ein vielfaches von m so lautet der Algorithmus:

```
for (i = 0; i < n; i+=m)
    for (j = 0; j < n; j+=m)
        for (k = 0; k < n; k+=m)
            for (s = 0; s < m; s++)
                for (t = 0; t < m; t++)
                    for (u = 0; u < m; u++)
                        C[i + s][j + t] += A[i + s][k + u] * B[k + u][j + t];
```

Diese Technik wird auch als „Kacheln“ (*engl.* tiling) bezeichnet. Hier ist diese Programmtransformation noch vergleichsweise einfach, da die Datenstruktur regelmäßig ist und keinerlei Datenabhängigkeiten vorhanden sind.

2.5 RISC und CISC

Die Abkürzungen RISC und CISC stehen für „reduced instruction set computer“ bzw. „complex instruction set computer“ und somit für gegensätzliche Designprinzipien von Mikroprozessoren. Im Laufe der Jahrzehnte der Rechnerentwicklung wurden die Maschinenbefehlssätze der Rechner immer umfangreicher, da man meinte immer exotischere Operationen und Adressierungsarten per Maschinenbefehl zur Vefügung stellen zu müssen. Leider hat dies auch die ganz einfachen Befehle verlangsamt. Compiler waren oft nicht in der Lage diese exotischen Befehle zu verwenden. Die Instruktionswerke waren so komplex, dass man eine zweite Ebene der Programmierung, die sog. Mikroprogrammierung, eingeführt hatte.

Anfang der 80er–Jahre hat man wieder Prozessoren mit einfachen aber schnell ausführbaren Befehlen konstruiert. Die wichtigsten Designprinzipien von RISC Rechnern sind:

- Alle Befehle werden in Hardware dekodiert, keine Mikroprogrammierung.
- Agressive Ausnutzung von Instruktionspipelining (Parallelismus auf Instruktionsebene).
- Möglichst ein Befehl/Takt ausführen (oder mehr bei superskalaren Maschinen). Dies erfordert einen möglichst einfachen, homogenen Befehlssatz.
- Speicherzugriffe nur mit speziellen Load/Store–Befehlen, keine komplizierten Adressierungsarten.
- Viele Allzweckregister bereitstellen um Speicherzugriffe zu minimieren. Die gesparte Chipfläche im Instruktionswerk wird für Register oder Caches verwendet.
- Folge dem Designprinzip „Mache den oft auftretenden Fall schnell“.

2.6 PRAXIS: Der Intel Pentium II

In diesem Abschnitt wollen wir kurz ausführen welche Designmerkmale einer der bekanntesten Mikroprozessoren, der Intel Pentium II, bietet. Für eine etwas ausführlichere Beschreibung sei auf (A. S. Tanenbaum 1999) verwiesen.

Der Pentium II ist eine aufwärtskompatible Weiterentwicklung des 8086 Prozessors, der von Intel 1978 eingeführt worden war. Der Maschinenbefehlssatz ist sehr heterogen mit vielen Formaten und Adressierungsarten. Es handelt sich also um einen CISC–Prozessor. Um trotzdem die

Vorteile des RISC–Konzeptes nutzen zu können werden die Maschinenbefehle intern in einfachere sog. Mikrooperationen zerlegt, die von einem RISC–Kern ausgeführt werden.

Insgesamt besteht die Instruktionspipeline aus 12 Stufen, die Maschinenbefehle werden in der Ausgangsreihenfolge beendet, die Mikroinstruktion können jedoch vom RISC–Kern in dynamisch bestimmter Reihenfolge ausgeführt werden (out–of–order execution).

Der Pentium II besitzt eine superskalare Architektur. Bis zu drei PII–Maschinenbefehle können in einem Taktzyklus dekodiert werden. Allerdings benötigen komplexere Maschinenbefehle mehr als einen Takt zur Dekodierung und natürlich kann es zu Resourcenkonflikten bezüglich der Ausführungseinheiten kommen. Immerhin gibt es zwei Fließkomma und zwei Integereinheiten, die parallel arbeiten können. Fließkommaaddition und Multiplikation verwenden selbst wieder Pipelining mit 3 bzw. 5 Stufen. Für bedingte Sprungbefehle steht eine Sprungvorhersageeinheit (branch prediction unit) zur Verfügung.

Das Cache–Konzept ist zweistufig. Auf dem Chip stehen zwei 16 KByte Level–1–Caches getrennt für Instruktionen und Daten zur Verfügung. Die Länge einer Cache–Line beträgt 32 Bytes, die Organisation ist 4–fach assoziativ mit verzögertem Schreiben (write back) und einer pseudo–LRU Ersetzungsstrategie. Die Größe des Level–2–Cache beträgt 512 KByte ebenfalls mit 32 Byte Cache–Lines. Der Level–2–Cache läuft mit halber Taktrate.

Als Hauptspeicher werden üblicherweise 100 MHz SDRAMs verwendet. Der Datenbus ist 64 Bit breit und prinzipiell können 64 GByte adressiert werden (übliche Motherboards können allerdings nur mit 1–2 GByte bestückt werden). Die Pentium II Architektur unterstützt den Zugriff mehrerer Prozessoren auf den Hauptspeicher. Cache–Kohärenz wird mit dem MESI–Protokoll sichergestellt, das wir weiter unten besprechen werden.

2.7 Übungen

Übung 2.1 Wir betrachten einen hypothetischen Rechner mit einem einstufigen Cache. Cache–Lines bestehen aus $L = 2^l$ Worten, der Cache umfaßt $M = 2^m$ Cache–Lines und sei 4–fach assoziativ. Der Haupspeicher habe $N = 2^n$ Worte. Beschreiben Sie die Organisation des Cache, die Länge des Cache–Tags in Bits und wie die einzelnen Abschnitte einer Wortadresse zur Adressierung des Cache verwendet werden.

Übung 2.2 Wir betrachten die Berechnung des Skalarproduktes zweier Vektoren der Länge n: $s = \sum_{i=0}^{n-1} x_i y_i$. Die Vektoren werden in eindimensionalen Feldern mit den Anfangsadressen a_x bzw a_y gespeichert. Der Algorithmus werde auf einer Maschine mit einem Cache mit direkter Zuordnung ausgeführt. Was passiert, wenn $|a_x - a_y|$ ein Vielfaches der Cachegröße ist? Kann die Skalarproduktoperation bei langen Vektoren effizient (im Sinne von maximaler Prozessorleistung) auf so einer Maschine ausgeführt werden?

Übung 2.3 Wieder betrachten wir das Skalarprodukt zwei Vektoren der Länge n. Nun seien die Vektorelemente in einer einfach verketteten Liste abgespeichert. Vergleichen Sie diese Datenstruktur mit einem eindimensionalen Feld hinsichtlich der Cacheausnutzung. Der Cache sei 2–fach assoziativ ausgeführt.

Übung 2.4 Betrachten sie folgendes Programmsegment:

for($k = 0$; $k < 1000$; k++)
    for($i = 1$; $i < n - 1$; i++)

for$(j = 1;\ j < n - 1;\ j{+}{+})$
$\quad u[i][j] = \frac{1}{4}(u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]);$

Dabei sei u ein Feld der Dimension $n \times n$ mit n so groß, daß eine Zeile des Feldes nicht mehr in den Cache passt. Überlegen Sie wie man die Schleifen umstrukturieren könnte um eine bessere Cachenutzung zu erreichen.

Übung 2.5 Implementieren Sie den geblockten Algorithmus zur Matrixmultiplikation auf einem Rechner mit Cache. Experimentieren Sie mit verschiedenen Blockgrößen und bestimmen Sie die MFLOP/s–Raten.

3 Skalierbare Rechnerarchitekturen

Die im letzten Kapitel betrachteten Rechnerarchitekturen nutzten bereits Parallelismus in der Form von Instruktionspipelining und mehrfachen Funktionseinheiten. Hiermit lässt sich selten mehr als eine Beschleunigung um den Faktor 5 erreichen. In diesem Kapitel geht es um Architekturen bei denen sich durch Hinzufügen weiterer Elemente (im Prinzip) beliebig hohe Leistung erreichen lässt. Eine ausführliche Einführung in parallele Rechnerarchitekturen gibt Culler, Singh und Gupta (1999).

3.1 Klassifikation von Parallelrechnern

Welche Ansätze gibt es für skalierbare Rechnerarchitekturen? Dieser Abschnitt gibt eine Übersicht.

Eine betagte aber immer noch gerne verwendete Klassifikation ist die nach Flynn (1972). Seine Einteilung kennt nur zwei Dimensionen: die Anzahl der Instruktionsströme und die Anzahl der Datenströme. In jeder Dimension können auch nur zwei Werte angenommen werden: einfach (1) und mehrfach (> 1). Die Anzahl der Instruktionsströme entspricht der Anzahl von Befehlszählern im System. Die Anzahl der Datenströme meint die Zahl der unabhängigen Datenpfade von den Speichern zu den Rechenwerken.

Somit gibt es nach Flynn vier Klassen von Rechnern (wovon eine leer ist):

- *SISD – single instruction single data.* Dies sind sequentielle Rechner aus Kapitel 2.
- *SIMD – single instruction multiple data.* Diese Rechner, auch Feldrechner genannt, verfügen über ein Instruktionswerk und mehrere unabhängige Rechenwerke von denen jedes mit einem eigenen Speicher verbunden ist. Die Rechenwerke werden taktsynchron vom Instruktionswerk angesteuert und führen die selbe Operation auf unterschiedlichen Daten aus. Hierbei denke man etwa an die Addition zweier Vektoren. Jedes Rechenwerk könnte die Addition zweier Vektorelemente durchführen.
- *MISD – multiple instruction single data.* Diese Klasse ist leer.
- *MIMD – multiple instruction multiple data.* Dies entspricht einer Kollektion eigenständiger Rechner, jeder mit einem Instruktions– und einem Rechenwerk ausgestattet.

SIMD–Rechner sind inflexibel. Bei einer **if**–Anweisung führen alle Verarbeitungseinheiten die die Bedingung zu wahr auswerten zuerst den **if**–Zweig aus und in einem zweiten Schritt führen die restlichen Verarbeitungseinheiten den **else**–Zweig aus. Aufgrund der Taktsynchronität sind aber keine Koordinationskonstrukte (wie z.B. wechselseitiger Ausschluss) notwendig. Der erste Parallelrechner, ILLIAC IV, war eine SIMD–Maschine. Allerdings war er nicht erfolgreich. In den 80er Jahren führten die CM–2 und MasPar das Konzept (nun erfolgreich) fort. Zur Zeit (1999) hat keine Architektur dieses Typs einen nennenswerten Marktanteil. Dies liegt daran, dass SIMD–Prozessoren Spezialanfertigungen sind, die nicht von der stürmischen Entwicklung der Mikroprozessoren im Massenmarkt profitieren können. Wir werden deshalb diese Rechnerklasse nicht weiter betrachten.

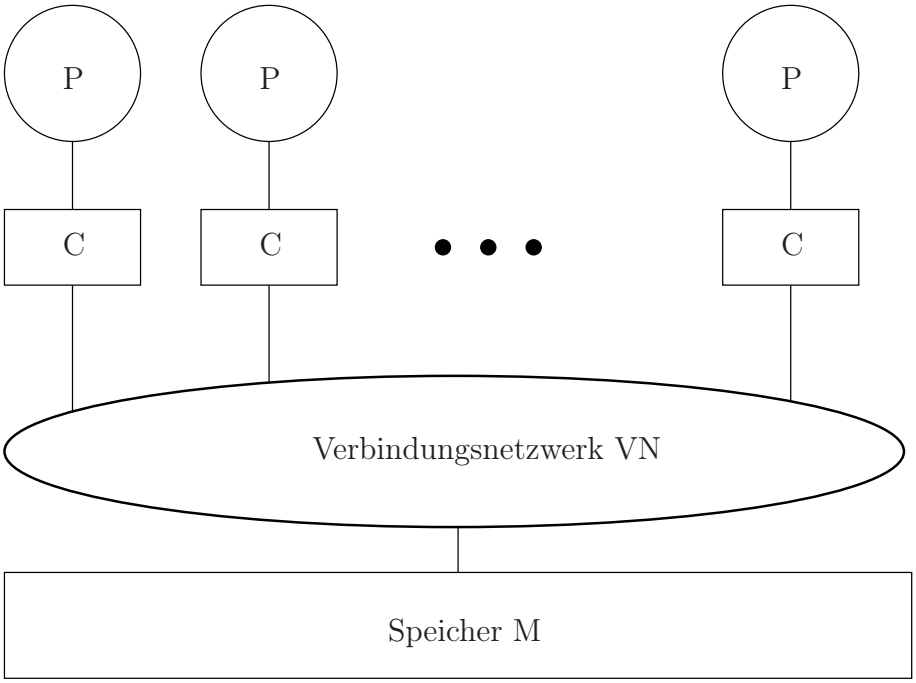


Abbildung 3.1: Rechner mit zentralem, gemeinsamen Speicher.

Rechner vom MIMD–Typ entsprechen genau der Vorstellung von einer parallelen Berechnung wie wir sie im ersten Kapitel eingeführt haben, d.h. einer Kollektion miteinander kommunizierender, sequentieller Prozesse. Jeder Prozess ist in der Lage ein anderes Programm auszuführen.

Die Klassifikation von Flynn berücksichtigt jedoch nicht wie der Datenaustausch im System stattfindet. Hierfür gibt es zwei prinzipielle Möglichkeiten:

1. Kommunikation über gemeinsamen Speicher, und
2. Kommunikation durch Austausch von Nachrichten.

Rechner vom MIMD–Typ mit gemeinsamen Speicher verfügen über einen *globalen Adressraum*. Dies bedeutet: jede Speicherzelle im System hat eine global eindeutige Adresse und kann von allen Prozessoren gelesen und geschrieben werden (unter der Annahme entsprechend gesetzter Zugriffsrechte).

Der gemeinsame Speicher kann als zentraler Speicher realisiert sein auf den alle Prozessoren über ein *dynamisches Verbindungsnetzwerk* zugreifen. Man spricht in diesem Fall auch von *Multiprozessoren* (*engl.* multiprocessors). Diese Variante ist in Abbildung 3.1 dargestellt.

Das Verbindungsnetzwerk leitet Anfragen eines Prozessors (Caches) an den Speicher weiter. Der Speicher kann als verschränkter Speicher mit mehreren Modulen aufgebaut sein um mehrere Anfragen gleichzeitig befriedigen zu können (bei entsprechender Auslegung des Verbindungsnetzwerkes). Die Zugriffszeit von jedem Prozessor zu jeder Speicherstelle ist gleich groß. Daher werden solche Rechner auch als Maschinen mit uniformem Speicherzugriff (*engl.* uniform memory access, UMA) bezeichnet. Jeder schnelle Prozessor benötigt einen Cache–Speicher (oder gar mehrere) um die Geschwindigkeit von Prozessor und Hauptspeicher auszugleichen. Der Cache wurde in Abb. 3.1 schon eingezeichnet. Dies wirft das Problem auf einen Block im Hauptspeicher und evtl. *mehrere* Kopien des Blockes kohärent zu halten (*engl.* cache coherence). Schreibzugriffe auf

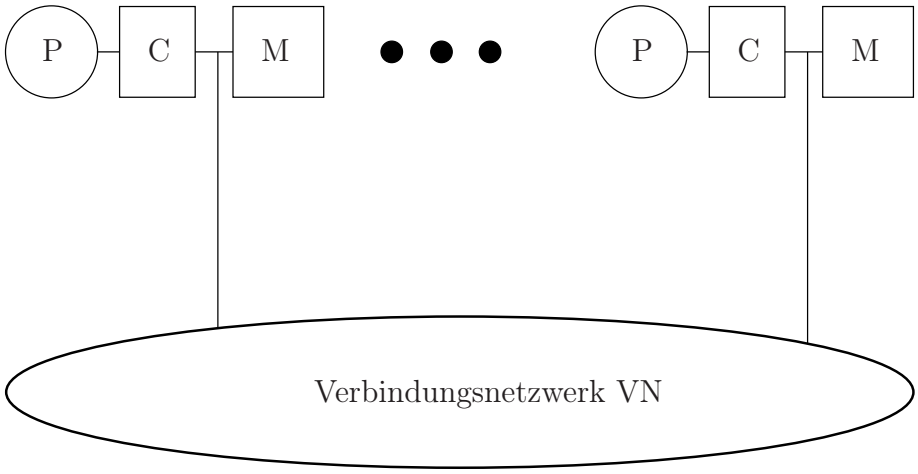


Abbildung 3.2: Rechner mit verteiltem, gemeinsamen Speicher.

den selben Block in verschiedenen Caches müssen sequentialisiert werden. Lesezugriffe müssen stets aktuelle Daten liefern.

Ein sogenanntes Protokoll zur Cache–Kohärenz stellt dies sicher (*engl.* cache coherence protocol). Cache–Kohärenz lässt sich sehr effizient implementieren falls das Verbindungsnetzwerk ein Bus ist. Ein Beispielprotokoll wird im nächsten Abschnitt ausführlich besprochen.

Leider können busbasierte UMA–Systeme nicht über wenige zehn Prozessoren hinaus skaliert werden. Dies gelingt nur mit Systemen mit verteiltem, gemeinsamen Speicher (*engl.* distributed shared memory, DSM). Den prinzipiellen Aufbau zeigt Abbildung 3.2.

Jedes Modul besteht aus Prozessor, Cache und lokalem Speicher. Über das Verbindungsnetzwerk kann auch auf entfernte Speicherstellen zugegriffen werden. Allerdings dauern entfernte Zugriffe (wesentlich) länger als lokale: die Zugriffszeit ist nicht mehr uniform (*engl.* nonuniform memory access, NUMA). Im Fall eines zweistufigen Cache–Speichers gibt es somit vier unterschiedliche Zugriffszeiten für Speicherzellen je nachdem ob sie sich im Level–1–Cache, Level–2–Cache, lokalem Speicher oder entferntem Speicher befinden! Die Lösung des Cache–Kohärenz–Problems erfordert hier weit höheren Aufwand als bei busbasierten Systemen. Cache–Kohärente NUMA–Rechner werden als ccNUMA–Maschinen abgekürzt. Systeme mit bis zu 1024 Prozessoren sind kommerziell erhältlich (SGI Origin).

Rechner mit Nachrichtenaustausch verfügen auch über verteilte Speicher haben aber einen *lokalen Adressraum*. Gleiche Adressen auf verschiedenen Rechnern sprechen unterschiedliche Speicherstellen an. Ein Prozessor kann nur auf seinen lokalen Speicher zugreifen. Den prinzipiellen Aufbau zeigt Abbildung 3.3. Da diese Rechner als Ansammlung eigenständiger Computer erscheinen werden sie auch als „Multicomputer“ bezeichnet.

Die einzelnen Knoten, bestehend aus Prozessor, Cache und Speicher sind Massenprodukte und können daher voll von der Entwicklung der Halbleiterindustrie (Moore'sches Gesetz: Verdopplung der Komplexität alle 18 Monate) profitieren. Das ist eine der Hauptstärken dieses Ansatzes. Das (statische) Verbindungsnetzwerk kann je nach Leistungsbedarf ausgelegt werden und kann unabhängig vom Rechenknoten entwickelt werden. Dieser Ansatz verspricht zur Zeit die größte Skalierbarkeit. Ein System mit beinahe 10000 Prozessoren wurde gebaut (Intel ASCI Option Red mit 9472 Prozessoren – der erste TFLOP/s–Rechner der Welt).

Zusammenfassend kann man sagen, dass die Skalierbarkeit von busbasierten UMA Systemen

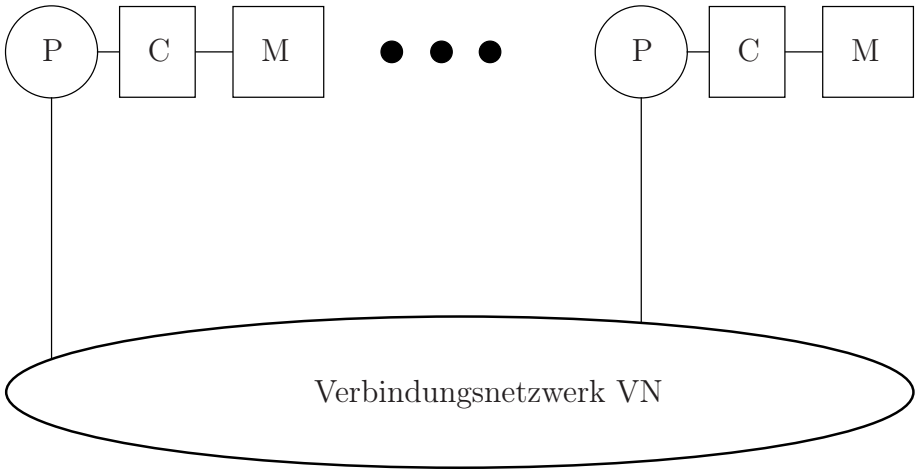


Abbildung 3.3: Rechner mit verteiltem, lokalem Speicher.

über ccNUMA hin zu Rechnern mit Nachrichtenaustausch zunimmt. In der selben Reihenfolge steigt jedoch auch die Schwierigkeit existierende Programmpakete für die jeweilige Rechnerarchitektur anzupassen. In den folgenden Abschnitten werden verschiedene Konstruktionsdetails der genannten Architekturen näher beschrieben.

3.2 Gemeinsamer Speicher

Viele Teile dieses Abschnittes folgen Kapitel 8 in Hennessy und Patterson (1996).

3.2.1 Dynamische Verbindungsnetzwerke

Dynamische Verbindungsnetzwerke sind leitungsvermittelnd, d.h. sie schalten eine elektrische Verbindung von der Quelle zum Ziel durch. Abb. 3.4 zeigt drei verschiedene dynamische Verbindungsnetzwerke.

Das einfachste dynamische Verbindungsnetzwerk ist der sogenannte Bus, siehe dazu Abb. 3.4(a). Bei einem Bussystem können immer nur je zwei Einheiten zu einer Zeit miteinander verbunden werden. Die Übertragungsgeschwindigkeit wird durch die Taktfrequenz und die Busbreite bestimmt. Ein Bus ist zwar billig aber nicht skalierbar.

Das andere Extrem stellt der Crossbar in Abb. 3.4(b) dar. Hier kann jeder Prozessor mit jedem Speicher gleichzeitig verbunden werden (vollständige Permutation). Dafür sind aber auch P^2 Schaltelemente notwendig bei P Prozessoren bzw. Speichern.

Einen Kompromiss stellen die sog. Mehrstufennetzwerke dar. Einen Vertreter dieser Klasse, das Ω–Netzwerk, zeigt Abb. 3.4(c). Die elementare Schaltzelle eines Mehrstufennetzwerkes besteht aus einem Element mit zwei Eingängen und zwei Ausgängen. Diese können entweder parallel oder über Kreuz verschaltet werden. Ein Ω–Netzwerk mit $P = 2^d$ Eingängen besitzt $d = \mathrm{ld}P$ Stufen mit je $P/2$ solcher Elementarzellen. Zwei aufeinanderfolgende Stufen sind in der Form eines *perfect shuffle* verschaltet.

Das Ω–Netzwerk erlaubt ein einfaches routing. Sucht man den Weg von einer Quelle a zu einem Ziel b so wird in der i-ten Stufe des Netzwerkes die $d - i$-te Bitstelle in der Binärdarstellung von a und b verglichen. Sind beide Bits gleich so schaltet der Schalter auf Durchgang sonst auf Kreuzung. Zur Illustration ist in Abb. 3.4 der Weg von 001 nach 101 eingezeichnet. Betrachtet

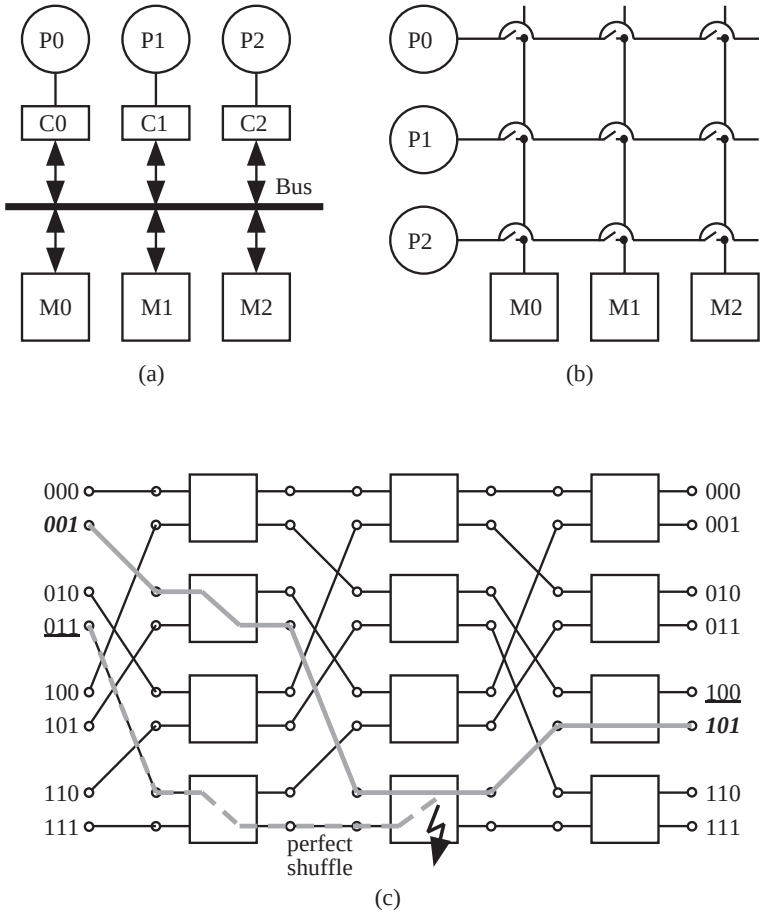


Abbildung 3.4: Beispiele für dynamische Verbindungsnetzwerke: Bus (a), Crossbar (b) und dreistufiges Ω–Netzwerk.

Tabelle 3.1: Zustände und Aktionen im MESI-Protokoll.

Zustand	Bedeutung
I	Eintrag ist nicht gültig
E	E. gültig, Speicher aktuell, keine Kopien vorhanden
S	E. gültig, Speicher aktuell, weitere Kopien vorhanden
M	E. gültig, Speicher ungültig, keine Kopien vorhanden
Aktion	**Beschreibung**
read miss	Prozessor liest Zeile, die nicht im Cache ist
write miss	Prozessor schreibt Zeile, die nicht im Cache ist
read hit	Prozessor liest aus Zeile, die im Cache ist
write hit	Prozessor schreibt Zeile, die im Cache ist
remote read miss	Anderer Prozessor liest Zeile, die er nicht hat, die lokal aber vorhanden ist
invalidate	Anderer Prozessor schreibt Zeile

man den Weg 011–100 stellt man fest das das Ω–Netzwerk nicht alle beliebigen Permutationen gleichzeit realisieren kann.

3.2.2 Cache–Kohärenz mit Schnüffeln

Wie bereits erwähnt gibt es bei cache–basierten Multiprozessorsystemen das Problem den Inhalt des Hauptspeichers und evtl. mehrerer Caches kohärent zu halten (Kohärenz bedeutet Zusammenhang). Bei busbasierten Systemen gibt es dafür eine effiziente und kostengünstige Lösung bei der jeder Cache ständig den Verkehr auf dem Bus abhorcht, man sagt „er schnüffelt“ (*engl.* bus snooping).

Der Cache in einem Multiprozessorsystem hat zwei Aufgaben: zum einen stellt er einem Prozessor lokal eine Kopie der Daten zur Verfügung (*engl.* migration) und reduziert dadurch den Verkehr auf dem Bus, zum anderen können mehrere Caches eine Kopie des selben Blockes enthalten (*engl.* replication) und dadurch Zugriffskonflikte vermeiden helfen. Probleme treten beim Schreiben auf, da dann die Kopien in den anderen Caches ungültig werden.

Grundsätzlich gibt es zwei Möglichkeiten wie man beim Schreiben eines Wortes im Cache verfahren kann. Man kann die Kopien in den anderen Caches für ungültig erklären (*engl.* write invalidate) oder man überträgt sofort die Änderung in alle anderen Caches (*engl.* write broadcast). Im ersten Fall führt mehrfaches Schreiben der selben Cache–Line zu keiner weiteren Kommunikation. Daher hat sich diese Möglichkeit (insbesondere mit write–back Caches) durchgesetzt.

Als Beispiel betrachten wir das Cache–Kohärenzprotokoll MESI. MESI ist benannt nach den vier Zuständen in denen sich ein im Cache gespeicherter Block befinden kann: modified (M), exclusive (E), shared (S) und invalid (I). Die Bedeutung dieser Zustände gibt Tabelle 3.1.

MESI erweitert einen write–back Cache um Cache–Kohärenz und wird etwa im Pentium II verwendet. Je nach Aktion kann der Zustand eines Cache–Blockes wechseln. Die in MESI zu verarbeitenden Aktionen sind ebenfalls in Tabelle 3.1 beschrieben.

Die möglichen Zustandsübergänge zeigt Abbildung 3.5. Nachfolgende Tabelle 3.2 beschreibt die verschiedenen Zustandsübergange aus der Sicht einer Cache–Verwaltung. Es ist wichtig zu beachten, dass jeder Cache einen eigenen Zustand für die jeweilige Cache–Zeile führt. Alle Cache–Verwaltungen hören ständig den Verkehr auf dem Bus ab. Stellt etwa ein Cache nach einem

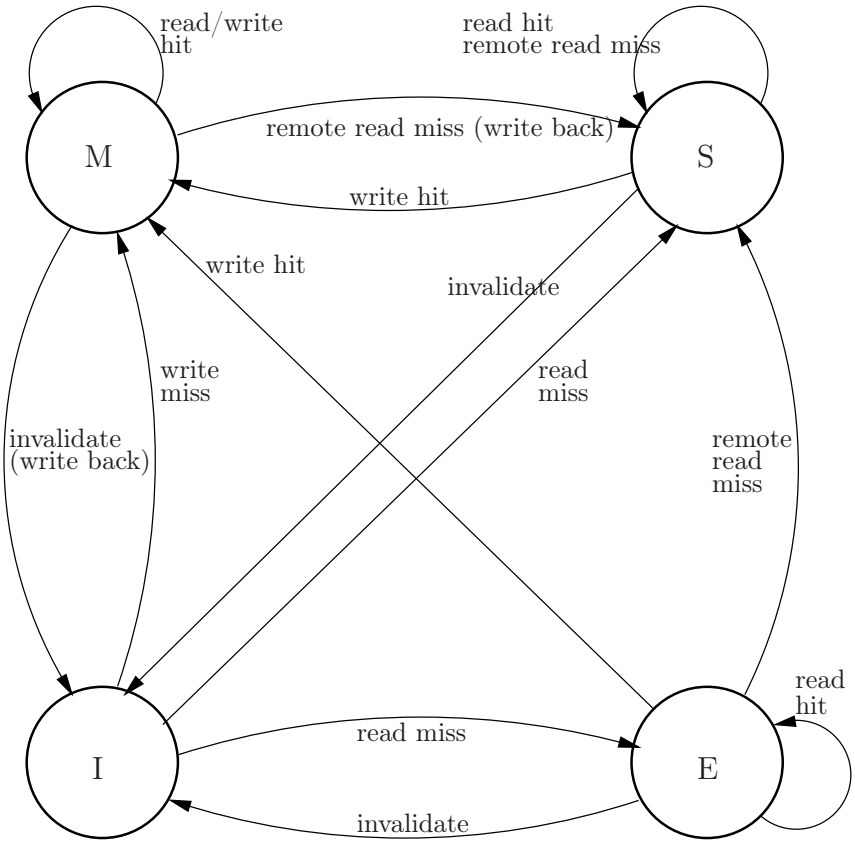


Abbildung 3.5: Zustandsübergangsdiagram für das Cache–Kohärenzprotokoll MESI.

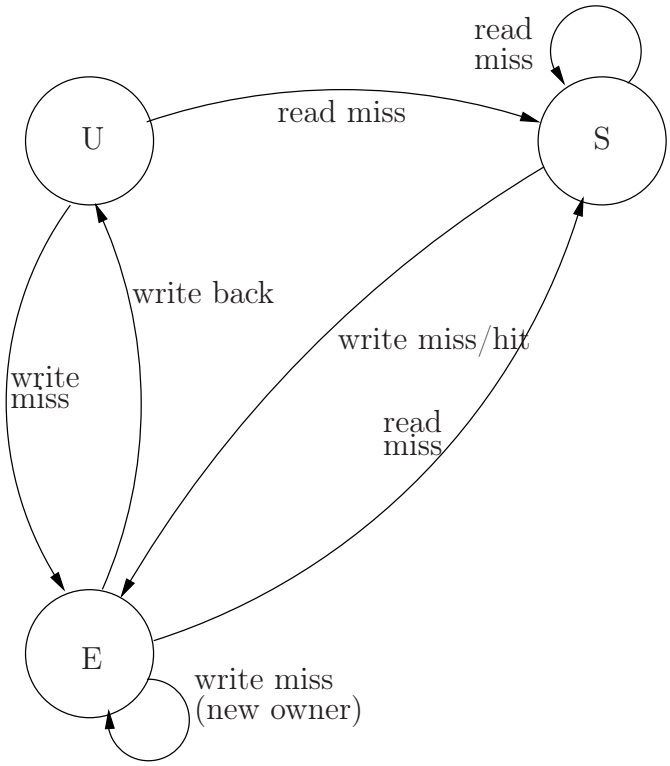


Abbildung 3.6: Zustandsübergangsdiagram für ein verzeichnisbasiertes Cache–Kohärenzprotokoll (Verzeichnisseite).

read miss eine Anfrage nach einer Cache–Zeile auf den Bus, so kann ein anderer der die Zeile im Zustand M hat entsprechend reagieren und eine Antwort auf den Bus legen.

3.2.3 Cache–Kohärenz mit Verzeichnissen

Busbasierte Systeme mit gemeinsamen Speicher skalieren nur bis wenige 10 Prozessoren. Bei größeren Prozessorzahlen verwendet man verteilte Speicher. Um trotzdem Cache–Kohärenz zu unterstützen verteilt man mit dem Hauptspeicher auch den sog. Verzeichnisspeicher (*engl.* directory).

Das Verzeichnis auf einem Knoten enthält für jeden Hauptspeicherblock dessen Zustand und welcher Cache eine Kopie dieses Blockes enthält (*engl.* sharers). Üblicherweise ist dies ein Feld mit einem Bit pro Prozessor. Der Rechenknoten, der einen Hauptspeicherblock und dessen Verzeichniseintrag speichert wird als dessen Heimatknoten (*engl.* home node) bezeichnet. Zusätzlich führt natürlich auch noch jeder Cache einen Zustand für jeden Knoten.

Die möglichen Zustände für Cache–Blöcke als auch Hauptspeicherblöcke und ihre Bedeutung gibt Tabelle 3.3 (das betrachtete Protokoll ist etwas einfacher als MESI).

Die möglichen Aktionen sind die selben wie oben. Tabelle 3.4 zeigt nun wie *das Verzeichnis* auf verschiedene Aktionen reagiert. Das zugehörige Zustandsübergangsdiagramm zeigt Abbildung 3.6. Das Protokoll aus Sicht eines Cache–Blockes wird nicht dargestellt (es ist ziemlich analog zu MESI, nur dass dort M dem Zustand E hier entspricht und der E–Zustand von MESI wegfällt).

Offensichtlich erzeugen manche Aktionen eine Menge Verkehr. So sind bei einem write–miss in Zustand E vier Nachrichten unterwegs: Neuer Eigentümer an Verzeichnis, Verzeichnis an alten

Tabelle 3.2: Beschreibung der Zustandsübergänge im MESI-Protokoll.

Zust.	Aktion	Folgt	Beschreibung
I	read miss	E,S	Adresse wird auf den Bus gelegt. Es gibt drei Möglichkeiten: a) Kein anderer Cache hat den Block, neuer Zustand ist E. b) Anderer Cache hat den Block im Zustand E oder S, neuer Zustand ist S bei allen. c) Anderer Cache hat Block im Zustand M: dieser schreibt den Block zurück und setzt seine Kopie in den Zustand S, lokale Kopie hat Zustand S.
	write miss	M	Adresse wird auf den Bus gelegt. Existieren Kopien in anderen Caches im Zustand E oder S werden diese invalidiert (I). Gibt es eine Kopie im Zustand M wird diese zurückgeschrieben und invalidiert.
E	invalidate	I	Block wird verdrängt oder anderer will schreiben.
	rem. r miss	S	Anderer liest den Block auch.
	write hit	M	Prozessor schreibt. Da keiner eine Kopie hat ist kein Buszugriff erforderlich.
	read hit	E	Zustand bleibt.
S	invalidate	I	Block wird verdrängt oder anderer will schreiben.
	write hit	M	Invalidiere andere Kopien.
	read hit	S	Keine Änderung des Zustandes.
M	r/w hit	M	Keine Änderung nötig.
	invalidate	I	Block wird verdrängt oder anderer will schreiben. Block muss zurückgeschrieben werden.
	rem. r miss	S	Anderer will Block lesen. Block muss zurückgeschrieben werden.

Tabelle 3.3: Zustände für verzeichnisbasierte Kohärenz.

Cache–Block		Hauptspeicherblock	
Zust.	Beschreibung	Zust.	Beschreibung
I	Block ungültig	U	niemand hat den Block
S	≥ 1 Kopien vorhanden, Caches und Speicher sind aktuell	S	wie links
E	genau einer hat den Block geschrieben (entspricht M in MESI)	E	wie links

Tabelle 3.4: Beschreibung der Zustandsübergänge im verzeichnisbasierten Kohärenzprotokoll.

Z	Aktion	Folgt	Beschreibung
U	read miss	S	Block wird an Cache übertragen, Bitvektor enthält wer die Kopie hat.
	write miss	E	Block wird an den anfragenden Cache übertragen, Bitvektor enthält wer die gültige Kopie hat.
S	read miss	S	anfragender Cache bekommt Kopie aus dem Speicher und wird in Bitvektor vermerkt.
	w miss/hit	E	Anfrager bekommt (falls miss) eine Kopie des Speichers, Verzeichnis sendet invalidate an alle übrigen Besitzer einer Kopie.
E	read miss	S	Eigentümer des Blockes wird informiert, dieser sendet Block zurück zum Heimatknoten und geht in Zustand S, Verzeichnis sendet Block an anfragenden Cache.
	write back	U	Eigentümer will Cache–Block ersetzen, Daten werden zurückgeschrieben, niemand hat den Block.
	write miss	E	Der Eigentümer wechselt. Alter Eigentümer wird informiert und sendet Block an Heimatknoten, dieser sendet Block an neuen Eigentümer.

Eigentümer, alter Eigentümer an Verzeichnis und Verzeichnis an neuen Eigentümer. Verbindungsnetzwerke in ccNUMA–Maschinen sind statische, paketvermittelnde Netzwerke wie wir sie ausführlicher im nächsten Abschnitt besprechen werden. Für weitere Details zur verzeichnisbasierten Cache–Kohärenz verweisen wir auf (Hennessy und Patterson 1996, Kap. 8.4 und Anhang E).

Es gibt noch weitere Varianten von verteiltem, gemeinsamen Speicher. Ein Problem bei ccNUMA tritt auf, wenn ein Prozessor große Datenmengen bearbeitet, die nicht in seinen Cache passen (*engl.* capacity misses) und die auch nicht in seinem lokalen Speicher sind. In diesem Fall werden ständig Cache–Blöcke von einem entfernten Knoten geladen. Das Problem hier ist, dass die Daten nicht richtig auf die Speicher aufgeteilt sind. Abhilfe kann man auf zweierlei Arten schaffen: 1. Der Programmierer muss die Daten besser aufteilen. Dies läuft auf das hinaus, was man für einen Rechner mit privatem Speicher auch tun müsste. 2. Man verwaltet den Hauptspeicher wie einen Cache, d.h. man migriert und repliziert Hauptspeicherblöcke (evtl. in größeren Blöcken als Cache–Lines) in den Hauptspeicher anderer Prozessoren. Hierfür hat sich der Name COMA (*engl.* cache only memory architecture) eingebürgert. Hierbei wird angenommen, dass die Hardware dies mit einer verzeichnisbasierten Struktur unterstützt. Ist der Mechanismus in Software implementiert, etwa unter Benutzung der Speicherverwaltungseinheit (*engl.* memory management unit, MMU), so spricht man von Software–DSM oder shared virtual memory.

3.3 Nachrichtenaustausch

Wir betrachten nun den Nachrichtenaustausch in Systemen mit privatem Speicher.

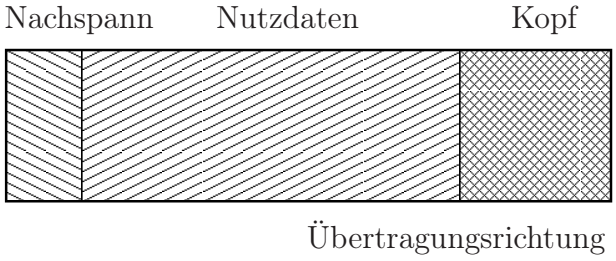


Abbildung 3.7: Aufbau eines Paketes.

3.3.1 Statische Verbindungsnetzwerke

Nachrichtenaustausch bedeutet das Kopieren eines Blockes vom Speicher eines Prozessors in den eines anderen. Da die Prozessoren dies nicht selbst tun können muss diese Aufgabe von einer zusätzlichen Hardware übernommen werden. Im einfachsten Fall stellt man sich dazu ein spezielles Register im Prozessor vor, in das der eine Prozessor den Inhalt des Speicherblockes Wort für Wort hineinschreibt und der andere liest entsprechend aus. Zwischen beiden Prozessoren übernimmt das Verbindungsnetzwerk den Transport der Daten.

Es gibt natürlich verbesserte Varianten dieses Schemas. So wird häufig das Kopieren durch einen Spezialprozessor übernommen damit der Hauptprozessor in der Zwischenzeit andere sinnvolle Dinge tun kann. Der Rest dieses Abschnittes beschäftigt sich hauptsächlich mit den verschiedenen Aspekten des Verbindungsnetzwerkes.

Die Verbindungsnetzwerke in Rechnern mit Nachrichtenaustausch sind statisch, bzw. paketvermittelnd. Das Netzwerk besteht aus Knoten, die über Leitungen miteinander verbunden sind. Die (beliebig langen) Nachrichten zerlegt man in *Pakete* fester Länge (z.B. 32 Byte bis 4 KByte), die dann von Knoten zu Knoten übertragen werden. Den prinzipiellen Aufbau eines Paketes zeigt Abbildung 3.7. Es besteht aus einem Kopf, den Nutzdaten und dem Nachspann, die in dieser Reihenfolge übertragen werden. Der Kopf dient zur Identifikation der nachfolgenden Nutzdaten und enthält etwa die Nummer des Zielprozessors. Der Nachspann kann eine Prüfsumme enthalten um feststellen zu können ob das Paket richtig übertragen wurde. Sieht man diese Möglichkeit vor, muss man sich überlegen was passieren soll wenn ein Paket fehlerhaft übertragen wurde. Man wird eine entsprechende Nachricht an den Absender zurückschicken damit dieser das Paket nochmal übertragen kann. Es kann auch passieren, dass der Zielprozessor gerade sehr beschäftigt ist und die Daten nicht in der Geschwindigkeit annehmen kann wie sie der Sender abschicken möchte. Der Sender muss dann durch weitere „Steuerpakete“gebremst werden, was man als Flusskontrolle bezeichnet. Das Zusammenspiel der Partner wird durch das Kommunikationsprotokoll (*engl.* communication protocol) geregelt.

Meist wird ein mehr oder weniger großer Anteil des Protokolls durch Software im Betriebssystem realisiert. Laufen etwa mehrere Prozesse auf einem Prozessor so sorgt das Betriebssystem dafür, dass die Nachricht den richtigen Prozess erreicht. Abb. 3.8 zeigt das Durchlaufen der verschiedenen Schichten bis eine Nachricht von einem Prozess zum anderen gelangt.

Damit kommen wir zu der Frage: „Wie leistungsfähig ist ein Netzwerk?“. Aus der Sicht zweier Benutzerprozesse kann die Zeit, die für die Übertragung einer Nachricht mit n Bytes benötigt wird näherungsweise mit einem linearen Ansatz beschrieben werden:

$$t_{mess}(n) = t_s + n * t_b.$$

Dabei ist t_s die Latenzzeit oder Aufsetzzeit (*engl.* latency, setup time). Dies ist die Zeit, die vom

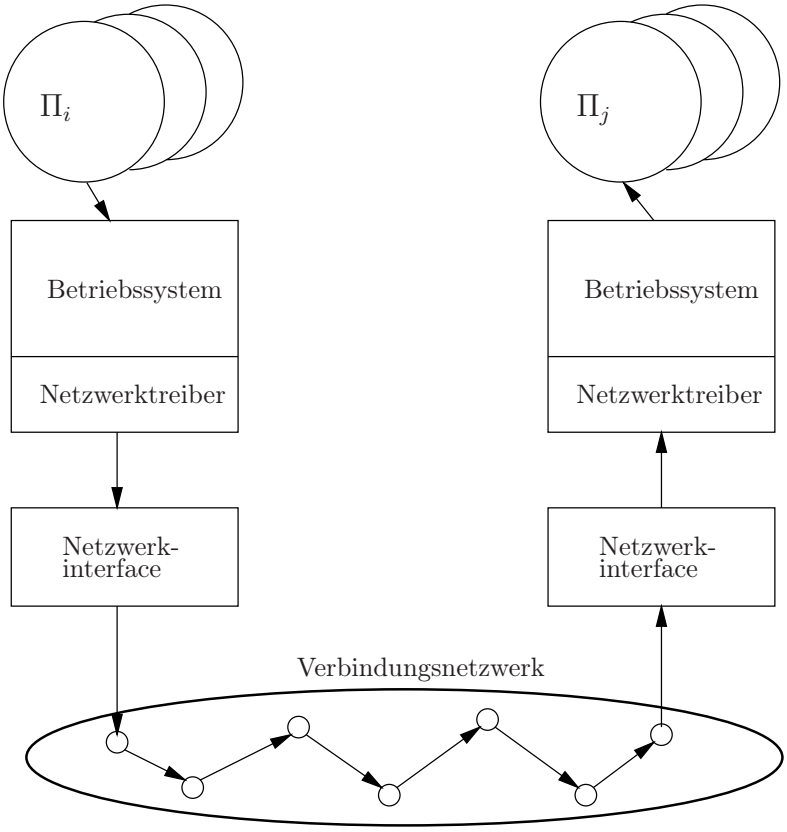


Abbildung 3.8: Das Schichtenmodell.

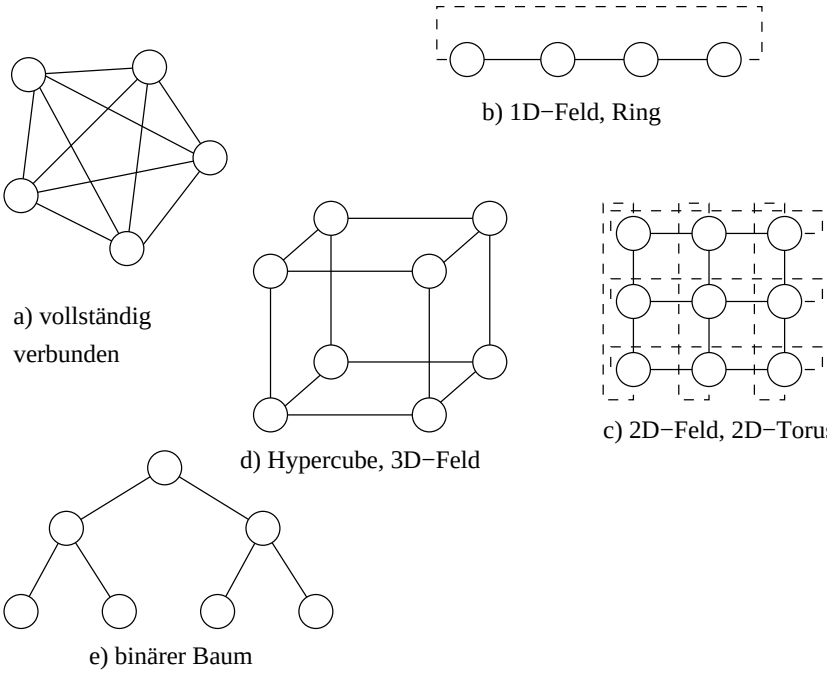


Abbildung 3.9: Beispiele verschiedener Verbindungsstrukturen.

Absetzen des **send**–Befehls in Π_i bis zur Beendigung des **receive**–Befehls in Π_j vergeht wenn die Nachrichtenlänge 0 Bytes beträgt. Für jedes weitere Byte wird dann nur noch die Zeit t_b benötigt. Den Wert $1/t_b$ bezeichnet man als Bandbreite (*engl.* bandwidth) in Bytes/Sekunde. Je nach Protokoll kann der Softwareanteil in t_s sehr hoch sein.

3.3.2 Netzwerktopologien

Die Leistung (und die Kosten) des Verbindungsnetzwerkes wird wesentlich durch dessen Verbindungsstruktur bestimmt. Abbildung 3.9 zeigt verschiedene Beispiele. Bild (a) zeigt die volle Verbindungsstruktur: jeder ist mit jedem verbunden. Bilder (b)–(d) zeigen ein–, zwei– und dreidimensionale Felder. Sind auch gegenüberliegende Randknoten jeweils miteinander verbunden, so spricht man von Ring (1D) und Torus (2D, 3D). Die Hypercubestruktur verdient nähere Erläuterung. Ein Hypercube der Dimension d, $d \geq 0$, besteht aus $P = 2^d$ Prozessoren. Die Nummern der Prozessoren lauten im Dualsystem $\underbrace{00\ldots0}_{d \text{ Stellen}}$ bis $\underbrace{11\ldots1}_{d \text{ Stellen}}$. Prozessor p ist mit Prozessor q direkt verbunden, wenn sich die Zahlen p und q im Dualsystem in genau einem Bit unterscheiden. Im d–dimensionalen Hypercube ist jeder Prozessor mit d anderen verbunden. Der Hypercube der Dimension $d = 3$ enspricht einem 3D–Feld der Größe $2 \times 2 \times 2$. Höherdimensionale Hypercubes lassen sich rekursiv aus zwei Hypercubes der nächst niedrigeren Dimension aufbauen indem man den Nummern der einen Hälfte eine 0 und der anderen Hälfte eine 1 voranstellt. Prozessoren mit ursprünglich gleichen Nummern sind dann zu verbinden.

Die wichtigsten Merkmale der verschiedenen Topologien sind (Tabelle 3.5 zeigt die Abhängigkeit von der Prozessorzahl P):

Knotengrad k. Der Knotengrad k gibt die Zahl von Nachbarn an, mit denen ein Netzwerkknoten direkt verbunden ist. Bezogen auf ein Netzwerk gibt man den maximalen Knotengrad an.

Tabelle 3.5: Parameter einiger Verbindungsnetzwerke.

Topologie	k	D	l	B	Sym
Volle Verb.	$P-1$	1	$\frac{P(P-1)}{2}$	$\left(\frac{P}{2}\right)^2$	ja
Hypercube	d	d	$d\frac{P}{2}$	$\frac{P}{2}$	ja
2D-Torus	4	$2\left\lfloor\frac{r}{2}\right\rfloor$	$2P$	$2r$	ja
2D-Feld	4	$2(r-1)$	$2P-2r$	r	nein
Ring	2	$\left\lfloor\frac{P}{2}\right\rfloor$	P	2	ja
Binärer Baum	3	$2(h-1)$	$P-1$	1	nein

$d = \log_2 P, P = r \times r, h = \lceil \log_2 P \rceil$

Der Knotengrad bestimmt die Kosten eines Netzwerkknotens wesentlich und sollte daher einigermaßen klein sein. Besonders interessant sind Netzwerktopologien mit konstantem Knotengrad, da nur sie eine beliebige Skalierbarkeit des Gesamtsystems erlauben.

Netzwerkdurchmesser D. Unter dem Netzwerkdurchmesser D versteht man die maximale Länge des kürzesten Pfades zwischen zwei beliebigen Knoten im Netzwerk. Ein kleiner Durchmesser ist wichtig für Situationen, in denen ein globaler Datenaustausch erforderlich ist.

Zahl der Verbindungen l. Die Gesamtzahl aller Verbindungen wird mit l bezeichnet. Sie bestimmt wesentlich den Kostenaufwand für die Verbindungen (Kabel). Die Zahl der Verbindungen sollte möglichst proportional zur Prozessorzahl sein.

Bisektionsbreite B. Minimale Zahl von Kanten ohne die das Netzzwerk in zwei gleichgroße Teile zerfällt. Eine große Bisektionsbreite ist wünschenswert, wenn viele Prozesse gleichzeitig miteinander kommunizieren wollen, da sich dann die Nachrichten weniger gegenseitig behindern.

Symmetrie. Ein Netzwerk ist symmetrisch, wenn es von jedem Knoten aus „gleich“aussieht. Symmetrische Netzwerke sind einfacher zu implementieren und programmieren.

Die „optimale“Netzwerktopologie gibt es natürlich nicht, da sich die Kriterien gegenseitig widersprechen. Jede Entscheidung für eine physikalische Topologie stellt somit einen Kompromiß zwischen Kosten und Leistungsfähigkeit dar. Tabelle 3.5 zeigt wie die einzelnen Größen von der Prozessorzahl abhängen.

Gute Kompromisse stellen die Hypercube und die 2D bzw. 3D Feldstruktur dar. Aufgrund der schnellen Verbindungsnetzwerke hat die Feldstruktur die Hypercubestruktur inzwischen verdrängt. Der Hypercube findet allerdings noch häufig als logische Verbindungsstruktur in verschiedenen Algorithmen Verwendung.

3.3.3 Routing

Wir betrachten nun genauer wie die einzelnen Pakete einer Nachricht von einem Netzwerkknoten zum anderen weitergereicht werden und wie sie ihren Weg zum Ziel finden.

Wie in Abschnitt 3.3.1 beschrieben werden Nachrichten in Pakete fester Länge zerlegt und mit Kopf und Nachspann versehen. Bei *store–and–forward routing* wird jeweils ein komplettes Paket von einem Netzwerkknoten zum anderen übertragen und dort zwischengespeichert. Erst wenn es komplett empfangen wurde, wird mittels der Adresse im Kopf der nächste Netzwerkknoten bestimmt an den das Paket weitergeleitet werden muss. Jeder Netzwerkknoten besitzt dazu eine Tabelle (*engl.* routing table), die Adressen auf Nachbarn abbildet. Im Fall mehrerer möglicher Wege zum Ziel ist dieser Vorgang aufwendiger und hängt von der Netzlast ab (adaptives

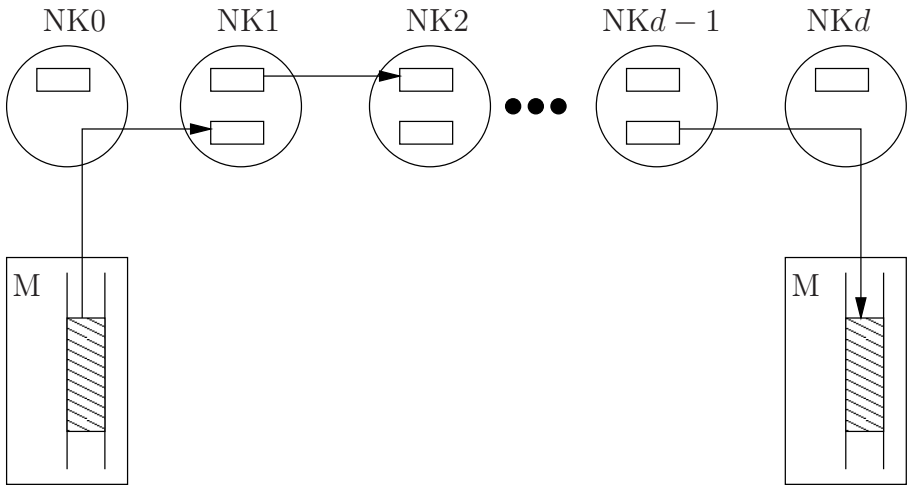


Abbildung 3.10: Pipelining im Kommunikationsnetzwerk.

Routing).

Bei *cut–through* oder *wormhole routing* wird ein Paket sofort nach lesen des Kopfes an den nächsten Netzwerkknoten weitergeleitet. Es wird nur ein kleiner Teil des Paketes, ein sog. *flit* (z.B. 4–16 Bits, je nach Breite der Verbindung zweier Netzwerkknoten), zwischengespeichert. In diesem Fall kann ein Paket über mehrere Netzwerkknoten verstreut sein. Cut–through routing wird von speziellen Prozessoren (*engl.* routing chips) komplett in Hardware ausgeführt.

Wir wollen nun abschätzen wie sich die Übertragungszeit für die verschiedenen Routingarten verhält. Dazu sei eine Nachricht von n Bytes zu übertragen. Die Paketgröße sei N Bytes und es sind d Verbindungen von der Quelle bis zum Ziel zu durchlaufen. Abbildung 3.10 zeigt die Kette von Netzwerkknoten die zu durchlaufen sind.

Wir benötigen folgende Größen:

t_s: Zeit, die auf Quell– und Zielrechner vergeht bis das Netzwerk mit der Nachrichenübertragung beauftragt wird, bzw. bis der empfangende Prozess benachrichtigt wird. Dies ist der Softwareanteil des Protokolls.

t_h: Zeit die benötigt wird um das erste Byte einer Nachricht von einem Netzwerkknoten zum anderen zu übertragen (*engl.* node latency, hop–time).

t_b: Zeit für die Übertragung eines Byte von Netzwerkknoten zu Netzwerkknoten.

Für store–and–forward Routing erhalten wir:

$$\begin{aligned} t_{SF}(n,N,d) &= t_s + d(t_h + Nt_b) + \left(\frac{n}{N} - 1\right)(t_h + Nt_b) \\ &= t_s + t_h\left(d + \frac{n}{N} - 1\right) + t_b\left(n + N(d-1)\right). \end{aligned} \tag{3.1}$$

Durch die Paketierung ist ein Pipelining möglich. Während Netzwerkknoten 1 ein Paket an Netzwerkknoten 2 schickt kann gleichzeitig eines von Netzwerkknoten 2 an Netzwerkknoten 3 übertragen werden (bei entsprechend paralleler Auslegung der Hardware). Somit setzt sich die Übertragungszeit zusammen aus der Aufsetzzeit t_s, der Zeit bis das erste Paket ankommt und der Zeit bis die restlichen Pakete angekommen sind. Im Fall $n = N$ erhalten wir $t_{SF} =$

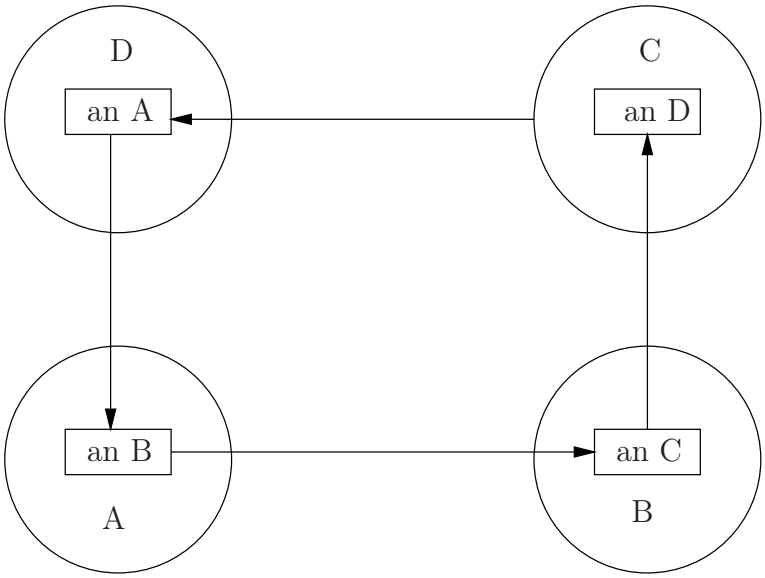


Abbildung 3.11: Verklemmung im Kommunikationsnetzwerk.

$t_s + d(t_h + Nt_b)$, für kurze Nachrichten ist die Zeit also proportional zur Entfernung von Quelle und Ziel. Bei langen Nachrichten dominiert der Term nt_b und die Entfernung spielt keine Rolle mehr.
Für cut–through Routing bekommen wir

$$t_{CT}(n, N, d) = t_s + t_h d + t_b n \tag{3.2}$$

Hier vergeht die Zeit $t_s + dt_h$ bis das erste Byte am Ziel ankommt (unabhängig von N). Da cut–through Routing in Hardware implementiert wird ist t_h in der Größenordnung weniger 10 ns und es kann in guter Näherung $dt_h \ll t_s$ angenommen werden. Somit ist die Latenzzeit in der Praxis unabhängig von der Entfernung d der Prozessoren.

3.3.4 Verklemmungen

In einem Kommunikationsnetzwerk besteht potentiell die Gefahr von Verklemmungen (*engl.* deadlock). Abb. 3.11 zeigt eine solche Situation. Die Prozessoren warten zyklisch auf ihre Kommunikationspartner bis diese einen freien Puffer haben. Die Situation läßt sich auch mit mehr Puffern nicht prinzipiell ausschließen (sie tritt dann nur etwas später ein).

Schnelle Verbindungsnetzwerke in Parallelrechnern benutzen daher prinzipiell verklemmungsfreie Routingalgorithmen. Dies wird durch den Ausschluß von zyklischen Abhängigkeiten erreicht. Als Beispiel für das sog. *Dimensionsrouting* diene eine zweidimensionale Feldstruktur. Für die vier verschiedenen Richtungen $+x$, $-x$, $+y$ und $-y$ werden vollkommen getrennte Netzwerke mit eigenen Puffern aufgebaut. Eine Nachricht läuft dann immer erst in $+x$ oder $-x$–Richtung bis die richtige Spalte erreicht ist und wechselt dann in das $+y$ oder $-y$–Netzwerk bis sie am Ziel ankommt. Es kann keine Verklemmung mehr auftreten, da sowohl die einzelnen Teilnetzwerke als auch die Verknüpfungen der Teilnetzwerke keine Zyklen enthalten.

Abbildung 3.12: Ein PC Cluster mit 80 Prozessoren am IWR der Universität Heidelberg.

3.4 PRAXIS: Cluster von LINUX PCs

Bei PC Clustern ist die Idee einen Superrechner aus handelsüblichen Standardkomponenten zusammenzubauen. Durch die große Stückzahl sind diese Komponenten sehr preisgünstig. Urvater aller PC Cluster ist das Beowulf System[1] das an einem Institut der NASA aufgebaut wurde. Deshalb spricht man auch oft von „Beowulfs“. Der Unterschied zu einem Cluster von Workstations besteht darin, dass die Rechenknoten eines Beowulf Systems keine regulären Arbeitsplätze sind (d.h. ohne Tastatur und Bildschirm), dass sie an einem separatem Netzwerk angeschlossen sind (private ethernet) und dass die UNIX–üblichen Sicherheitsvorkehrungen abgeschaltet sind. Abbildung 3.12 zeigt ein solches Cluster.

Warum hat sich dieser Ansatz bewährt? Hierfür gibt es die folgenden Gründe:

- PC Systeme mit Pentium Prozessor haben inzwischen die Rechenleistung typischer Workstations erreicht. Nur sehr teuere Workstations sind schneller als ein PC.
- Systeme mit 2–4 Pentium CPUs und gemeinsamem Speicher auf einer Platine sind billig erhältlich.
- Fast Ethernet (100 MBit/s) Vernetzung ist für viele Anwendungen ausreichend und kostet nicht viel. Leistungsfähige Switches sind erhältlich.

[1] `http://www.beowulf.org`

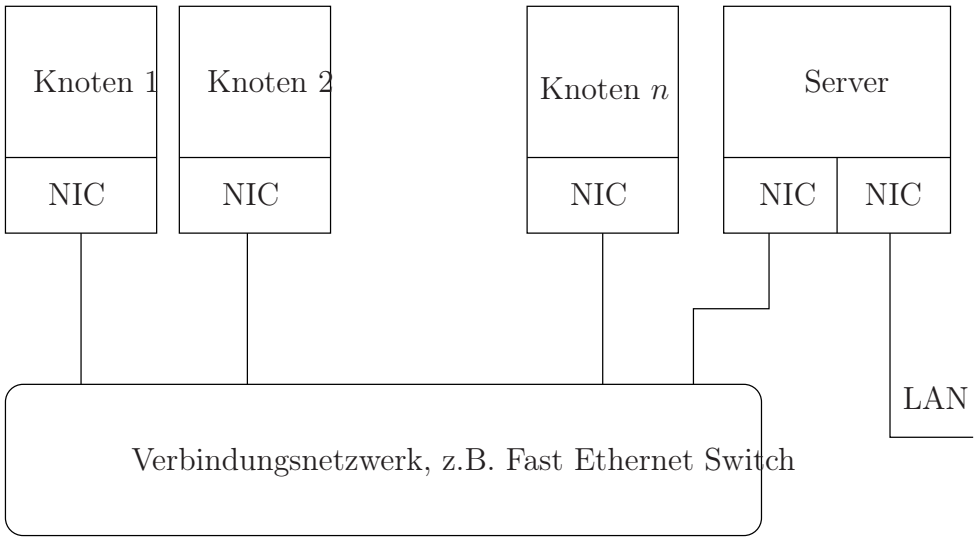


Abbildung 3.13: Prinzipieller Aufbau eines PC Clusters.

- Mit LINUX ist ein sehr stabiles und leistungsfähiges UNIX System praktisch umsonst verfügbar.

Durch den Einsatz von Standardkomponenten kann ein PC Cluster je nach Erfordernissen aufgebaut werden. Im folgenden werden wir den prinzipiellen Aufbau so eines Systems und die verschiedenen Komponenten erläutern.

3.4.1 Prinzipieller Aufbau

Abbildung 3.13 zeigt den prinzipiellen Aufbau eines PC Clusters. Es besteht aus den Rechenknoten mit je 1–4 Prozessoren und gemeinsamem Speicher sowie dem Serverknoten. Sowohl Server als auch Rechenknoten sind handelsübliche PCs. Die Rechenknoten sind über eine Netzwerkkarte (NIC) an das Verbindungsnetzwerk angeschlossen, im einfachsten Fall ein Fast Ethernet Switch. Der Server ist über eine zweite Netzwerkkarte an das lokale Netz (LAN) angeschlossen. Im lokalen Netzwerk ist nur der Server, nicht jedoch die Rechenknoten sichtbar. Um das Cluster zu benutzen muss man sich am Server anmelden. Der Server agiert als Firewall und somit kann die Sicherheit im Cluster niedrig gehalten werden.

3.4.2 Rechenknoten

Da die Rechenknoten handelsübliche PCs sind kann man sie sehr flexibel entsprechend den Anforderungen ausstatten. Die wichtigsten Optionen sind:

- Einen oder mehrere Prozessoren. Zumindest zwei Prozessoren als UMA–System sind sehr billig erhältlich. Für Anwendungen die den Cache nutzen können ist diese Variante zu empfehlen, da sich alle Kosten für Netzwerk, Gehäuse, Platten usw. halbieren. Für Anwendungen die den Cache schlecht nutzen sollte man besser eine CPU pro Rechenknoten verwenden.
- Pentium oder Alpha CPU. Als Alternative zu den Intel CPUs gibt es den Alpha Prozessor von DEC. Auch hier werden UMA System mit mehreren CPUs angeboten. Der Preis ist

höher, dafür aber auch die Leistung, zumindest wenn die Anwendung den Cache nutzen kann.

- Mit Platte oder ohne. Es besteht die Möglichkeit in jeden Knoten eine Festplatte einzubauen oder nicht. Im ersten Fall kann man diese Platten für ein verteiltes File–System nutzen. Im zweiten Fall muss man für jeden Knoten das Betriebssystem über Netz laden, was zusätzlich Bandbreite und Hauptspeicherplatz wegnimmt. Dafür hat es den Vorteil, dass jeder die selbe Version des Betriebssystemes hat. Bei vielen Knoten kann das Einspielen neuer Software auf alle Platten durchaus zum Problem werden.

In der Praxis sollte man auch die Ausfallwahrscheinlichkeit eines solchen Systemes beachten. Fällt ein Rechenknoten mit der Wahrscheinlichkeit ϵ innerhalb einer bestimmten Zeit aus, so ist die Wahrscheinlichkeit dafür, dass ein System aus P Komponenten funktioniert eben $(1-\epsilon)^P$. Daher sollte man durchaus auf die Qualität der Komponenten achten oder auch die Möglichkeit von fehlerkorrigierendem Speicher vorsehen. Bei größeren Systemen kann auch die Kühlung und der Platzbedarf zum Problem werden.

Verbindungsnetzwerk

Auch hier gibt es verschiedene Optionen. Das an manchen Orten noch vorhandene 10 MBit/s Ethernet ist sicher nicht ausreichend für die meisten Anwendungen. Die wichtigsten brauchbaren Netzwerke sind:

- Fast Ethernet bietet eine Bandbreite von 100 MBit/s. In der Praxis lassen sich damit von Prozess zu Prozess etwa 10 MByte/s und eine Latenzzeit von etwa 100 μs erzielen. Diese Angaben beziehen sich auf das standardmässig verwendete TCP/IP Kommunikationsprotokoll und MPI (message passing interface) zum Nachrichtenaustausch. Andere Protokolle (z.B. M-VIA[2]) versprechen kürzere Latenzzeiten. Fast Ethernet arbeitet mit einer sternförmigen Verkabelung. Alle Rechenknoten werden an einen sog. Switch angeschlossen, der viele Ein–/Ausgänge gleichzeitig mit einander verbinden kann. Switches mit einer Bandbreite von 4 GBit/s (entspricht 40 Fast Ethernet Verbindungen) sind für relativ wenig Geld erhältlich. Dies ist für viele Anwendungen bereits ausreichend.

- Gigabit Ethernet ist die Erweiterung von Ethernet auf 1 GBit/s. Einsteckkarten (NICs) sind relativ günstig, jedoch sind entsprechende Switches noch sehr teuer. Verwendet man das TCP/IP Protokoll, so lässt sich kaum eine Verbesserung in der Latenzzeit von Prozess zu Prozess erzielen, da der Softwareteil des Protokolls einen hohen Anteil an der Latenzzeit hat.

- SCI (Scalable Coherent Interface) und Myrinet sind schnelle Netzwerktechnologien, die Bandbreiten im Bereich 70–80 MByte/s und Latenzzeiten im Bereich von 10 μs ermöglichen. Dafür beträgt der Anteil des Netzwerks an den Gesamtkosten des Clusters dann aber auch bis zu 50% (je nach Ausbau der Knoten).

3.4.3 Software

Als Betriebssystem für ein PC Cluster bietet sich das frei erhältliche LINUX an. Als echtes UNIX Betriebssystem bietet es bereits genügend Funktionalität um ein Cluster zu betreiben. Jeder

[2] `http://www.nersc.gov/research/FTG/via/`

Benutzer der Maschine erhält eine Zugangsberechtigung auf allen Knoten des Clusters und dem Server. Da alle Clusterknoten durch den Server vom übrigen LAN abgeschottet sind kann man die Verwendung der r–Kommandos (`rlogin, rsh, ...`) getrost erlauben. Jeder Rechenknoten hat ausserdem Zugriff auf das Filesystem des Servers, so dass dieser ausschliesslich für das Übersetzen von Programmen benutzt wird. Damit kann der Benutzer bereits Jobs auf verschiedene Rechenknoten verteilen.

Parallele Anwendungen nutzen Nachrichtenaustausch als Kommunikationsmittel da ja die Knoten nur Zugriff auf den lokalen Speicher haben[3]. Für den Nachrichtenaustausch hat sich das standardisierte MPI (message passing interface) durchgesetzt von dem es mehrere frei erhältliche Implementierungen gibt. Eine so entsprechend vorbereitete Anwendung wird dann auf dem Server etwa mittels des Befehls

```
mpirun -np 8 test
```

auf 8 Prozessoren gestartet.

Zusätzlich gibt es auch (meist frei erhältliche) Erweiterungen des LINUX–Kernels die eine bequemere Nutzung des Clusters ermöglichen. So bietet MOSIX[4] die Möglichkeit zur transparenten Prozessmigration und automatischer Lastverteilung. Verschiedene Ansätze zur Implementierung eines verteilten gemeinsamen Speichers in Software[5] sind auch unternommen worden.

3.4.4 Zusammenfassung

Mit LINUX Clustern auf PC Basis lassen sich sehr einfach Systeme bis etwa 128 Knoten aufbauen. Das Preis/Leisungsverhältnis ist sehr gut, sofern man entsprechend parallelisierte Anwendungsprogramme besitzt. Es ist allerdings noch unklar wie sich diese Systeme auf die Größenordnung klassischer Superrechner erweitern lassen. Systeme mit bis zu 1000 Prozessoren sind geplant bzw. werden gebaut. Dies ist sicher nicht ohne Anpassungen des Betriebssystem und der Administrationssoftware zu machen. Jedoch hat die bisherige Entwicklung schon bewirkt, dass große Firmen wie Cray/SGI und IBM deutlich weniger kleine Systeme ihrer Superrechner verkaufen. Damit sind auch sie gezwungen in diese Technologie einzusteigen was wiederum Auswirkungen auf die großen Systeme haben wird. Eine gute Einführung in das „Cluster Computing“ bietet Buyya (1999).

3.5 Übungen

Übung 3.1 (Programmierübung) Schreiben Sie auf einem Ihnen zugänglichen Rechner ein paralleles Programm mit zwei Prozessen, die eine Nachricht vorgebbarer Länge immer hin– und herschicken (sog. Ping–Pong Test). Bestimmen Sie damit Latenzzeit und Bandbreite für das verwendete System.

Übung 3.2 Wir betrachten ein System mit einem Hypercube Verbindungsnetzwerk. Wie lässt sich ein 1–, 2– oder 3–dimensionales Feld so in den Hypercube einbetten, dass nur direkte Verbindungen zweier Knoten verwendet werden. Tipp: Verwenden Sie dazu binär reflektierte Gray–Codes (binary reflected Gray codes, BRGC). Der BRGC der Länge 2 verwendet die Folge $(0, 1)$. Um daraus die Folge der Länge 4 zu erhalten fügt man die Folge der Länge 2 in umgekehrter Reihenfolge hinten an und ergänzt bei der ersten Hälfte eine 0 und bei der zweiten eine

[3] SCI ermöglicht auch den Zugriff auf entfernte Speicher.
[4] `http://www.mosix.cs.huji.ac.il/`
[5] `http://www.cs.umd.edu/~keleher/dsm.html`

1, also: $(00, 01, 11, 10)$. Offensichtlich unterscheiden sich zwei benachbarte Zahlen in der Folge in genau einem Bit. Längere Folgen erhält man rekursiv.

Übung 3.3 Überlegen Sie sich einen verklemmungsfreien Routingalgorithmus im Hypercube. Tipp: Zerlegen Sie das Netzwerk in d Teilnetzwerke.

Teil II

Programmiermodelle

4 Kommunikation über gemeinsame Variablen

In diesem Kapitel beschreiben und lösen wir verschiedene Probleme, die entstehen wenn zwei oder mehr Prozesse gemeinsame Variablen benutzen. Abschnitt 4.1 beschäftigt sich nochmals eingehend mit dem schon mehrfach erwähnten wechselseiten Ausschluss. Abschnitt 4.2 diskutiert das Problem der Synchronisation aller Prozesse (eine spezielle Form der Bedingungssynchronisation) und dann stellt Abschnitt 4.3 ein allgemeines Werkzeug zur Lösung vieler Synchronisationsprobleme vor, die Semaphore. Schließlich werden in Abschnitt 4.4 komplexere Synchronisationsaufgaben betrachtet bei denen sich mehrere kritische Abschnitte überlappen. Dieses Kapitel stützt sich wesentlich auf (Andrews 1991, Teil II).

4.1 Kritischer Abschnitt

4.1.1 Problemdefinition

Das Problem des kritischen Abschnitts, bzw. des wechselseitigen Ausschlusses, trat nun schon mehrfach auf. Wir stellen uns eine parallele Anwendung bestehend aus P Prozessen vor. Die von einem Prozess ausgeführten Anweisungen werden in zwei zusammenhängende Gruppen eingeteilt: einen kritischen Abschnitt und einen unkritischen Abschnitt. Ein kritischer Abschnitt ist eine Folge von Anweisungen, die auf gemeinsame Variablen zugreift. Die Anweisungen des kritischen Abschnitts dürfen *nicht* von zwei oder mehr Prozessen gleichzeitig bearbeitet werden, man sagt es darf sich nur ein Prozess im kritischen Abschnitt befinden.

Um dies sicherzustellen wird vor dem kritischen Abschnitt ein Eingangsprotokoll ausgeführt und am Ende ein Ausgangsprotokoll. Alle Prozesse führen abwechselnd kritische und unkritische Abschnitte aus wobei der wechselseitige Ausschluss auch bei verschieden langer Ausführungsdauer sichergestellt werden muss. Dies zeigt das folgende

Programm 4.1 (Einführung wechselseitiger Ausschluss)

```
parallel critical-section
{
    process Π [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            Eingangsprotokoll;
            kritischer Abschnitt;
            Ausgangsprotokoll;
            unkritischer Abschnitt;
        }
    }
}
```

Nun geht es darum Eingangs– und Ausgangsprotokolle zu finden, die die folgenden Eigenschaften erfüllen:

1. *Wechselseitiger Ausschluss.* Höchstens ein Prozess führt den kritischen Abschnitt zu einer Zeit aus.

2. *Verklemmungsfreiheit.* Falls zwei oder mehr Prozesse versuchen in den kritischen Abschnitt einzutreten muss es genau einer nach endlicher Zeit auch schaffen.

3. *Keine unnötige Verzögerung.* Will ein Prozess in den kritischen Abschnitt eintreten während alle anderen ihre unkritischen Abschnitte bearbeiten so darf er nicht am Eintreten gehindert werden.

4. *Schließliches Eintreten.* Will ein Prozess in den kritischen Abschnitt eintreten so muss er dies nach endlicher Wartezeit auch dürfen (dazu nimmt man an, daß jeder den kritischen Abschnitt auch wieder verlässt).

In Abschnitt 1.3 haben wir schon erläutert wie wechselseitiger Ausschluss mit speziellen Maschinenbefehlen realisiert werden kann. Die dort vorgestellte Lösung besitzt sicher die Eigenschaften 1–3 aber die Eigenschaft 4 ist nicht ohne weiteres erfüllt. Bei drei Prozessen könnte es passieren, dass immer abwechselnd zwei Prozesse ihren kritischen Abschnitt bearbeiten und der dritte unendlich lange wartet. Zugegebenermassen ist dies sehr unwahrscheinlich aber andererseits wird diese Möglichkeit auch nicht explizit ausgeschlossen. Nun wollen wir zunächst betrachten, wie wechselseitiger Ausschluss auch ohne spezielle Maschinenbefehle realisiert werden kann.

4.1.2 Petersons Algorithmus für zwei Prozesse

Dieser Algorithmus realisiert wechselseitigen Ausschluss inklusive schließlichem Eintreten (Bedingung 4 von oben). Außerdem werden keine speziellen Maschinenbefehle benötigt.

Wir beschränken uns zunächst auf zwei Prozesse und erweitern die Lösung dann auf P Prozesse. Betrachten wir folgende erste Idee für ein Eingangsprotokoll:

int *in1*=0, *in2*=0;

Π_1:	Π_2:
while (*in2*) ;	**while** (*in1*) ;
in1=1;	*in2*=1;
krit. Abschnitt;	krit. Abschnitt;

Es gibt zwei Variablen *in1*, *in2* die anzeigen ob sich der betreffende Prozess im kritischen Abschnitt befindet. Will ein Prozess eintreten, so wartet er bis der andere nicht im kritischen Abschnitt ist und führt dann seinen kritischen Abschnitt aus. Die vorgeschlagene „Lösung“ funktioniert natürlich so nicht, da das Lesen und Schreiben der Variablen nicht atomar ist. Hier können beide Prozesse in den kritischen Abschnitt eintreten wenn sie beide erst die Bedingung der **while**–Schleife als falsch auswerten bevor der jeweils andere seine Variable schreibt.
Vertauschen wir nun die ersten beiden Zeilen in jedem Prozess:

```
int in1=0, in2=0;

Π1:                 Π2:
in1=1;              in2=1;
while (in2) ;       while (in1) ;
krit. Abschnitt;    krit. Abschnitt;
```

Es wird als erstes die Besetzt–Variable gesetzt und dann gewartet. Nun ist zwar wechselseitiger Ausschluss sichergestellt (sequentielle Speicherkonsistenz vorausgesetzt, siehe Abschnitt 4.1.3) aber es kann zu einer Verklemmung kommen. Setzt nämlich erst Π_1 seine Variable auf 1 und dann Π_2 seine (bevor Π_1 sie lesen kann) so warten beide in der nachfolgenden **while**–Schleife. Diese Verklemmung können wir lösen indem wir eine zusätzliche Bedingung einführen, die nur einer der beiden Prozesse wahr auswerten kann. In einer zusätzlichen Variablen *last* vermerken wir wer als zweiter an dem kritischen Abschnitt kommt und dieser muss dann warten. Die vollständige Lösung enthält folgendes

Programm 4.2 (Petersons Algorithmus für zwei Prozesse)

```
parallel Peterson-2
{
    int in1=0, in2=0, last=1;

    process Π1                      process Π2
    {                               {
        while (1) {                     while (1) {
            in1=1;                          in2=1;
            last=1;                         last=2;
            while (in2 ∧ last==1) ;         while (in1 ∧ last==2) ;
            krit. Abschnitt;                krit. Abschnitt;
            in1=0;                          in2=0;
            unkrit. Abschnitt;              unkrit. Abschnitt;
        }                               }
    }                               }
}
```

Überlegen wir warum diese Lösung funktioniert. Nehmen wir an beide Prozesse hätten ihren kritischen Abschnitt betreten, d.h. beide haben die Bedingung der **while**–Schleife als falsch ausgewertet. Betrachten wir o. B. d. A. den Prozess Π_1. Da beide den kritischen Abschnitt betreten haben, muss ($in2$==1) gelten. Somit kann die **while**–Schleife nur wegen ($last$==2) verlassen worden sein. Damit wird aber die Bedingung der **while**–Schleife in Prozess Π_2 wahr und dieser kann nicht in den kritischen Abschnitt eingetreten sein. Somit war die Behauptung falsch und der wechselseitige Ausschluss wird sichergestellt.

Darüberhinaus ist der schließliche Eintritt sichergestellt, da Π_2 in jedem Fall als nächster eintreten darf.

4.1.3 Speicherkonsistenz

Kohärenzprotokolle regeln die Reihenfolge von Zugriffen verschiedener Prozessoren auf die selbe Speicherstelle (bzw. Cache–Block). Konsistenz (Folgerichtigkeit) regelt die Reihenfolge von Lese–

und Schreibzugriffen *eines* Prozessors auf verschiedene Speicherstellen. Warum ist das wichtig? Betrachten wir ein Beispiel:

Programm 4.3 (Beispiel zur Speicherkonsistenz)

```
parallel memory–consistency
{
    int a = 0, b = 0;

    process Π1                process Π2
    {                         {
        ...                       ...
        a = 0;                    b = 0;
        ...                       ...
        a = 1;                    b = 1;
        if (b == 0) ...           if (a == 0) ...
    }                         }
}
```

Nehmen wir an sowohl a als auch b befinden sich im Cache beider Prozessoren (Π_1 und Π_2 werden auf verschiedenen Prozessoren ausgeführt) mit den Cache–Blöcken im Zustand S. Weiterhin führt jeder Prozessor seine Operationen streng der Reihe nach aus, d.h. jede Anweisung ist beendet bevor die nächste gestartet wird. Anweisungen verschiedener Prozessoren sind aber relativ zueinander nicht geordnet. Dieses Modell bezeichnet man als *sequentielle Konsistenz*.

Setzt man sequentielle Konsistenz voraus so kann nur einer der beiden Prozessoren seine **if**–Anweisung als wahr auswerten. Überlegen wir kurz: Angenommen beide werten ihre **if**–Bedingung als wahr aus. O. B. d. A. betrachten wir Prozess Π_1. Er hat für b den Wert 0 vorgefunden. Dies bedeutet, dass Π_2 die Anweisung $b = 1$ noch nicht ausgeführt haben kann, denn bevor Π_2 die Variable b schreibt würde er den Cache–Block in Prozessor 1 invalidiert haben. Damit kann, wegen der sequentiellen Konsistenz, Π_2 auch seine **if**–Bedingung noch nicht ausgewertet haben. Dies ist ein Widerspruch zur Voraussetzung.

Sequentielle Konsistenz ist jedoch eine starke Einschränkung. Bei Prozessoren mit Instruktionspipelining möchte man durch außer–der–Reihe–Ausführung Lücken in der Pipeline vermeiden. Bei cache–basierten Systemen möchte man Ladezeiten des Caches verstecken. Da insbesondere Schreibzugriffe sehr lange dauern (invalidate Meldung) möchte der Prozessor einen nachfolgenden Lesebefehl (wie in obigem Beispiel) gerne ausführen während der Schreibbefehl auf den Bus wartet. Aus der Sicht des Prozessors ist dies auch kein Problem, da keinerlei Datenabhängigkeiten zwischen a und b bestehen.

Somit muss man festlegen welche „Überholvorgänge“erlaubt sind und welche nicht. Im Laufe der Zeit wurden viele verschiedene Konsistenzmodelle vorgestellt. Ziel ist es dabei immer einerseits eine effiziente Abarbeitung durch den Prozessor zu ermöglichen und andererseits dem Programmierer möglichst einfache Regeln zu geben an die er sich halten kann.

Unterscheidet man zwischen Lesezugriffen R und Schreibzugriffen W so gibt es vier verschiedene Möglichkeiten zweier aufeinanderfolgender Zugriffe (wir nehmen an es handelt sich um Zugriffe auf *verschiedene* Speicherstellen, ansonsten ist die Reihenfolge immer einzuhalten, da es sich um echte Datenabhängigkeiten handelt):

1. $R \rightarrow R$: Lesezugriff folgt Lesezugriff,
2. $R \rightarrow W$: Schreibzugriff folgt einem Lesezugriff,

3. $W \rightarrow W$: Schreibzugriff folgt Schreibzugriff,

4. $W \rightarrow R$: Lesezugriff folgt Schreibzugriff.

Bei der sequentiellen Konsistenz werden alle vier Reihenfolgen strikt eingehalten. Schwächere Konsistenzmodelle erlauben, dass gewisse Reihenfolgen nicht eingehalten werden müssen. Ist es notwendig eine gewisse Reihenfolge zu erzwingen (wie in unserem Beispiel) muss der Programmierer dies durch spezielle *Synchronisationszugriffe* kennzeichnen.

Bei der *Prozessorkonsistenz* (*engl.* processor consistency, total store ordering) erlaubt man, dass ein Lesezugriff einen Schreibzugriff überholen darf, d.h. man verlangt nicht dass $W \rightarrow R$ eingehalten wird. Zusätzlich führt man spezielle (lesende und schreibende) Zugriffe S auf Synchronisationsvariable ein und verlangt $S \rightarrow R$, $S \rightarrow W$, $R \rightarrow S$, $W \rightarrow S$ und $S \rightarrow S$. Will man in diesem Modell die Reihenfolge $W \rightarrow R$ erzwingen, so ist zwischen die beiden Zugriffe ein Synchronisationszugriff einzufügen, denn dann gilt wegen der Transitivität $W \rightarrow S$ und $S \rightarrow R$.

Bei der *schwachen Konsistenz* (*engl.* weak consistency) dürfen alle Zugriffe einander überholen. Nur bei den Synchronisationszugriffen werden alle ausstehenden Zugriffe beendet und keine neuen gestartet bevor nicht der Synchronisationszugriff beendet ist.

Bei der noch weiter gehenden *Freigabekonsistenz* (*engl.* release consistency) unterscheidet man zwei verschiedene Synchronisationszugriffe: eine Anforderung S_A (*engl.* acquire) und eine Freigabe S_R (*engl.* release). Jeder Zugriff auf gemeinsam benutzte Daten verschiedener Prozesse wird üblicherweise durch ein S_A eingeleitet und durch ein S_R beendet. In diesem Fall müssen *alle* Lese– und Schreibzugriffe warten bis ein vorausgehendes S_A bzw. ein nachfolgendes S_R beendet ist, d.h. $S_A \rightarrow R$, $S_A \rightarrow W$, $R \rightarrow S_R$ und $W \rightarrow S_R$ sowie $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$ werden erhalten. Wegen $S_A \rightarrow \{R, W\} \rightarrow S_R \rightarrow S_A \rightarrow \ldots$ können aufeinanderfolgende Zugriffe auf gemeinsame Daten richtig geordnet werden. Als Beispiel betrachten wir die Realisierung von Programm 4.3 auf einer Maschine mit Freigabekonsistenz:

Programm 4.4 (Beispiel zur Freigabekonsistenz)

```
parallel release–consistency
{
    int a = 0, b = 0;

    process Π1                  process Π2
    {                           {
        ...                         ...
        acquire;                    acquire;
        a = 0;                      b = 0;
        release;                    release;
        ...                         ...
        acquire;                    acquire;
        a = 1;                      b = 1;
        release;                    release;
        acquire;                    acquire;
        if (b == 0) {               if (a == 0) {
            c = 3;                      c = 5;
            d = a;                      d = b;
        }                           }
        release;                    release;
    }                           }
```

}

Mit den aquire/release Paaren ist sichergestellt, dass der Lesezugriff ($b == 0$) den vorausgehenden Schreibzugriff $a = 1$ nicht überholen darf. Man beachte, dass aquire/release nur die Reihenfolge von Befehlen innerhalb eines Prozessors betrifft. Ausserdem ist die Reihenfolge der Zugriffe $b == 0$, $c = 3$ und $d = a$ nicht vorgeschrieben. Alle drei dürfen aber erst starten wenn das vorausgehende aquire beendet ist und müssen vor dem nachfolgenden release beendet werden.

4.1.4 Verallgemeinerung des Peterson auf P Prozesse

Hier ist die Idee in $P-1$ Stufen vorzugehen. Um in den kritischen Abschnitt zu gelangen muss ein Prozess alle Stufen durchlaufen. Der *letzte* auf einer Stufe ankommende Prozess muss auf dieser Stufe warten bis alle Prozesse auf höheren Stufen weitergekommen sind.

Wir verwenden zwei Felder $in[P]$ und $last[P]$ der Länge P. In $in[i]$ vermerkt Prozess Π_i die erreichte Stufe, d.h. eine Zahl in $\{1,\ldots,P-1\}$, $last[j]$ enthält die Nummer des Prozesses, der zuletzt auf der Stufe j angekommen ist. Die vollständige Lösung enthält folgendes Programm:

Programm 4.5 (Petersons Algorithmus für P Prozesse)

```
parallel Peterson-P
{
    const int P=8;
    int in[P] = {0[P]};
    int last[P] = {0[P]};

    process Π [int i ∈ {0, ..., P − 1}]
    {
        int j,k;
        while (1)
        {
            for (j=1; j ≤ P − 1; j++)  // durchlaufe Stufen
            {
                in[i] = j;                  // ich bin auf Stufe j
                last[j] = i;                // ich bin der letzte auf Stufe j
                for (k = 0; k < P; k++)// teste alle anderen
                    if (k ≠ i)
                        while (in[k]≥in[i] ∧ last[j]==i) ;
            }
            kritischer Abschnitt;
            in[i] = 0;                      // Ausgangsprotokoll
            unkritischer Abschnitt;
        }
    }
}
```

Auch dieser Algorithmus stellt eine faire Behandlungsweise sicher. Allerdings müssen $O(P^2)$ Bedingungen ausgewertet werden bevor ein Prozess in den kritischen Abschnitt eintreten darf.

4.1.5 Hardware Locks

Wie wir in Abschnitt 1.3 festgestellt haben, benötigt man für die korrekte Benutzung gemeinsamer Variable zusätzliche Operationen, die den exklusiven Zugriff eines Prozessors sicherstellen. In diesem Unterabschnitt zeigen wir welche Unterstützung die Hardware für diese Operation bereitstellt. Üblicherweise verwendet ein Anwendungsprogrammierer diese Operationen nur indirekt über die Funktionen einer Systembibliothek.

Zur effizienten Realisierung des wechselseitigen Ausschluß stellen viele Prozessoren einen der folgenden Maschinenbefehle zur Verfügung:

- *atomic-swap*: Vertausche den Inhalt eines Registers mit dem Inhalt einer Speicherzelle in einer unteilbaren Operation.
- *test-and-set*: Überprüfe ob eine Speicherstelle den Wert 0 hat, wenn ja schreibe den Inhalt eines Registers in die Speicherstelle (als unteilbare Operation).
- *fetch-and-increment*: Hole den Inhalt einer Speicherstelle in ein Register und erhöhe den Inhalt der Speicherstelle um eins (als unteilbare Operation).

In all diesen Befehlen muss ein Lesezugriff gefolgt von einem Schreibzugriff ohne Unterbrechung durchgeführt werden. In einem Multiprozessorsystem müssen zusätzlich Zugriffe anderer Prozessoren auf die selbe Speicherstelle verhindert werden.

Wir wollen nun betrachten wie ein solcher Maschinenbefehl zusammen mit dem Mechanismus für die Cache–Kohärenz das Eintreten eines einzigen Prozessors in den kritischen Abschnitt sicherstellt und gleichzeitig relativ wenig Verkehr auf dem Verbindungsnetzwerk erzeugt.

Da wir nicht auf Assemblerebene absteigen wollen, stehe dazu eine Funktion

int $atomic - swap$ (**int** $*a$);

zur Verfügung, die den Inhalt der Speicherstelle mit Adresse a liefert und unteilbar den Wert 1 in die Speicherstelle schreibt. Ein kritischer Abschnitt wird dann wie folgt realisiert:

Programm 4.6 (Spin Lock)

```
parallel spin–lock
{
    const int P = 8;          // Anzahl Prozesse
    int lock=0;               // Variable zur Absicherung

    process Π [int p ∈ {0, ..., P − 1}]
    {
        ...
        while ( atomic − swap(& lock))
            ;
        ...                   // kritischer Abschnitt
        lock = 0;
        ...
    }
}
```

Der Eintritt in den kritischen Abschnitt wird durch die globale Variable *lock* abgesichert. Hat sie den Wert 1 so befindet sich genau ein Prozess im kritischen Abschnitt, ist sie 0 so ist der kritische Abschnitt frei. Das Warten auf des Freiwerden des kritischen Abschnittes wird durch eine Schleife realisiert, daher heisst diese Lösung im Englischen *spin lock*.

Zu Beginn hat die Variable *lock* den Wert 0 und ist in keinem der Caches. Angenommen Prozess Π_0 führt die $atomic - swap$–Operation aus. Der Lesezugriff führt zu einem read miss, der Block wird aus dem Speicher geholt und bekommt den Zustand E (wir legen MESI zugrunde). Der nachfolgende Schreibzugriff findet (angenommen kein anderer hat inzwischen zugegriffen) ohne weiteren Buszugriff statt und der Zustand geht von E nach M. Führt ein anderer Prozess Π_1 nun seine $atomic - swap$–Operation aus, führt der read miss zum Zurückschreiben des Blockes durch Π_0, der Zustand beider Kopien ist nun, nach dem Lesezugriff, S. Der Schreibzugriff von Π_1 invalidiert die Kopie von Π_0 und der Zustand in Π_1 ist M. Die Funktion $atomic - swap$ liefert 1 in Π_1 und der kritische Abschnitt wird von Π_1 nicht betreten.

Führen beide Prozesse die $atomic - swap$–Operation gleichzeitig aus entscheidet letztendlich der Bus wer gewinnt. Angenommen sowohl C_0 (der Cache von Prozessor P_0 auf dem Π_0 läuft) als auch C_1 habe je eine Kopie des Cache–Blockes im Zustand S bevor sie ihre $atomic - swap$–Operation ausführen. Ist keiner im kritischen Abschnitt, so lesen beide zunächst den Wert 0 aus der Variable *lock*. Beim nachfolgenden Schreibzugriff konkurrieren beide um den Bus um eine invalidate Meldung zu platzieren. Der Gewinner setzt seine Kopie in den Zustand M, der Verlierer seine in den Zustand I. Die Cache–Logik des Verlierers findet beim Schreiben den Zustand I vor und muss den $atomic - swap$–Befehl veranlassen doch den Wert 1 zurückzuliefern (der atomic-swap–Befehl ist zu dieser Zeit ja noch nicht beendet).

Programm 4.6 kann noch weiter verbessert werden. Angenommen ein Prozessor befindet sich im kritischen Abschnitt und mehrere andere bewerben sich um den Eintritt. In diesem Fall wird durch die ständigen Schreibzugriffe der Bewerber ein enormer Busverkehr erzeugt. Dies lässt sich jedoch leicht vermeiden wie folgende verbesserte Version zeigt:

PROGRAMM 4.7 (VERBESSERTER SPIN LOCK)

```
parallel improved–spin–lock
{
    const int P = 8;          // Anzahl Prozesse
    int lock=0;               // Variable zur Absicherung

    process Π [int p ∈ {0, ..., P − 1}]
    {
        ...
        while (1)
        {
            if (lock==0)
                if (atomic − swap(& lock)==0 )
                    break;
        }
        ...                   // kritischer Abschnitt
        lock = 0;
        ...
    }
}
```

Nun testen alle Prozesse erst einmal den Wert der Variablen *lock* auf den Wert 0. Nur wenn der

kritische Abschnitt frei ist (*lock*=0) bewerben sie sich um den Eintritt. Ist der kritische Abschnitt besetzt, so lesen die Bewerber nur ihre Kopie im Cache bis diese durch die Zuweisung *lock*=0 ungültig wird. Sind die Berechnungen in einem kritischen Abschnitt sehr lange, so sollte man kein spin lock mit aktivem Warten verwenden. Es gibt dann bessere Operationen (Semaphore, siehe nächstes Kapitel), die es anderen Prozessen auf dem selben Prozessor währenddessen erlauben sinnvolle Dinge zu tun.

Wie skaliert ein spin lock bei größeren Prozessorzahlen? Schließlich könnten sich in einem ccNUMA–System Hunderte von Prozessen um den Eintritt in einen kritischen Abschnitt bemühen. In Programm 4.7 führt die Zuweisung lock=0 dazu, dass sich alle n restlichen Bewerber eine neue Kopie des Cache–Blockes holen müssen. Dies erfordert n Transaktionen auf dem Bus. Müssen P Prozesse (auf P Prozessoren) durch den kritischen Abschnitt, so führt dies zu einem Aufwand $O(P^2)$. Mit einer Warteschlange (*engl.* queueing locks) lässt sich der Aufwand wieder auf $O(P)$ drücken. Hier vermerkt jeder Bewerber seinen Eintrittswunsch in einer Datenstruktur (dies muss er nur einmal machen). Verlässt ein Prozess den kritischen Abschnitt, so „weckt er“ den nächsten Bewerber (siehe Übungsaufgabe 4.1).

4.1.6 Ticket Algorithmus

Petersons Algorithmus für P Prozesse ist relativ schwierig zu verstehen. Wie können wir das Verhalten bezüglich schließlichem Eintreten bei der Lösung mit Maschinenbefehlen verbessern? Die Wahrscheinlichkeit, dass ein Prozess am Eintreten gehindert wird hängt davon ab wie lange die kritischen Abschnitte dauern. Sind sie sehr kurz, so ist die Wahrscheinlichkeit sehr gering dass ein Prozess beliebig lange verzögert wird. Dies nutzt man beim sog. Ticket–Algorithmus aus. Die Idee ist von Verkaufstheken in Supermärkten abgeschaut: Der Kunde zieht bei Eintritt in den Laden eine Nummer und wartet dann bis seine Nummer aufgerufen wird. Ist ein Kunde fertig bedient wird die nächste Nummer aufgerufen. Wir simulieren dies mit zwei globalen Variablen *number* und *next*. Die Lösung zeigt

Programm 4.8 (Ticket Algorithmus für P Prozesse)

```
parallel Ticket
{
    const int P=8;
    int number=0;
    int next=0;

    process Π [int i ∈ {0, ..., P − 1}]
    {
        int mynumber;
        while (1)
        {
            [mynumber=number; number=number+1;]
            while (mynumber≠next) ;
            kritischer Abschnitt;
            next = next+1;
            unkritischer Abschnitt;
        }
    }
}
```

Jeder Prozess speichert die gezogene Nummer in der Variablen *mynumber* und wartet bis *next* diesen Wert erreicht. Das Erhöhen der Variable *next* kann ohne wechselseitigen Ausschluss erfolgen, da nur der Prozess im kritischen Abschnitt dies tut. Der kritische Abschnitt [...] kann mit den Maschinenbefehlen aus Abschnitt 4.1.5 realisiert werden. Man überlege sich, dass der Zahlenüberlauf das korrekte Funktionieren das Algorithmus nicht beeinflusst.

4.1.7 Bedingter kritischer Abschnitt

Wir betrachten folgendes Problem mit m Erzeugern und n Verbrauchern. Eine Menge von Prozessen P_i, $0 \leq i < m$ (producer) erzeugt Aufträge, die von einer Menge von Prozessen C_j, $0 \leq j < n$, (consumer) abgearbeitet werden sollen. Die Aufträge werden in eine zentrale Warteschlange mit k Plätzen eingereiht. Ist die Warteschlange voll so muss ein P_i solange warten bis ein Platz frei ist. Ist die Warteschlange leer so muss ein C_j solange warten bis ein Auftrag ankommt. Der Zugriff auf die Warteschlange muss selbstverständlich unter wechselseitigem Ausschluss erfolgen. Programm 4.9 zeigt eine Lösung für dieses Problem. Die Warteschlange selbst wurde weggelassen, nur die Anzahl der gespeicherten Aufträge wird gespeichert.

Programm 4.9 (Erzeuger–Verbraucher / aktives Warten)

```
parallel producer-consumer-busy-wait
{
    const int m = 8; n = 6; k = 10;
    int orders=0;
    process P [int 0 ≤ i < m]
    {
        while (1) {
            produziere Auftrag;
            CSenter;
            while (orders==k){
                CSexit;
                CSenter;
            }
            speichere Auftrag;
            orders=orders+1;
            CSexit;

        }
    }
    process C [int 0 ≤ j < n]
    {
        while (1) {

            CSenter;
            while (orders==0){
                CSexit;
                CSenter;
            }
            lese Auftrag;
            orders=orders-1;
            CSexit;
            bearbeite Auftrag;
        }
    }
}
```

Hierbei stehen die Anweisungen *CSenter* bzw. *CSexit* für ein beliebiges Eingangs– bzw. Ausgangsprotokoll für kritische Abschnitte (*CSenter* entspricht „[“und *CSexit* dem „]“).

Entscheidend ist, dass ein Prozess den eigentlichen kritischen Abschnitt „speichere Auftrag“ bzw. „lese Auftrag“ nur ausführen kann wenn die Bedingungen „Warteschlange nicht voll“ bzw. „Warteschlange nicht leer“ erfüllt sind. Dazu betritt er probeweise den kritischen Abschnitt und testet die Bedingung. Ist die Bedingung erfüllt kann er weitermachen, ist sie jedoch nicht erfüllt so muss er den kritischen Abschnitt wieder verlassen um anderen Prozessen die Gelegenheit zu geben die Bedingung zu verändern.

Programm 4.9 kann sehr ineffizient sein, da ständig kritische Abschnitte betreten und wieder verlassen werden. Man denke an den Fall, dass mehrere Erzeuger Aufträge absetzen wollen

aber alle Verbraucher beschäftigt sind (und die Warteschlange voll ist). In diesem Fall wird ein enormer Busverkehr erzeugt, auch bei einer Maschine mit Cache–Kohärenz.

Um dafür Abhilfe zu schaffen,kann man eine Verzögerungsschleife zwischen das *CSexit*/*CSenter*–Paar in der **while**–Schleife einbauen. Die Verzögerungszeit sollte man zufällig wählen um zu verhindern, dass alle Prozesse wieder zugleich eintreten wollen (dies bringt uns auf das Problem paralleler Zufallsgeneratoren . . .). Verdoppelt man die maximale Verzögerungszeit in jedem Durchlauf der **while**–Schleife so spricht man von einem „exponential back–off protocol“.

4.2 Globale Synchronisation

Eine globale Synchronisation, oder auch „Barriere“ veranlasst alle Prozesse zu warten bis alle an diesem Punkt im Programm angekommen sind. Erst dann dürfen alle weitermachen. Barrieren werden üblicherweise wiederholt ausgeführt wie etwa in

```
while (1) {
    eine Berechnung;
    Barriere;
}
```

Codesequenzen dieser Art treten oft in sog. „datenparallelen Programmen“ auf. Wir betrachten drei verschiedene Lösungen dieses Problems.

4.2.1 Zentraler Zähler

Hier ist die Idee in einer gemeinsamen Variable *count* die ankommenden Prozesse zu zählen. Ist die Zahl P erreicht so dürfen alle weitermachen. Das Erhöhen des Zählers muss unter wechselseitigem Ausschluss erfolgen.
Betrachten wir folgenden Vorschlag:

Programm 4.10 (Erster Vorschlag einer Barriere)

```
parallel barrier-1
{
    const int P=8;
    int count=0;
    int release=0;

    process Π [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            Berechnung;
            CSenter;                    // Eintritt
            if (count==0) release=0;    // Zurücksetzen
            count=count+1;              // Zähler erhöhen
            CSexit;                     // Verlassen
            if (count==P) {
                count=0;                // letzter löscht
```

```
            release=1;                  // und gibt frei
        } else
            while (release==0) ;        // warten
    }
  }
}
```

Dieser Vorschlag funktioniert noch nicht ganz. Stellen wir uns einen Prozess vor der in der **while**–Schleife wartet. Der letzte Prozess setzt *release*=1. Läuft nun ein Prozess los, kommt wieder an der Barriere und setzt *release*=0 *bevor* alle das *release*==1 gesehen haben so werden die wartenden Prozesse nie aus der **while**–Schleife befreit. Da *count* nie mehr den Wert P erreicht werden schließlich alle Prozesse warten und eine Verklemmung ist eingetreten.

Irgendwie muss man also verhindern, dass zwei aufeinanderfolgende Durchläufe der Barriere miteinander verwechselt werden. Dies gelingt indem abwechselnd die Werte 0 und 1 die Freigabe der Barriere anzeigen wie im folgenden

Programm 4.11 (Barriere mit Richtungsumkehr)

```
parallel sense-reversing-barrier
{
  const int P=8;
  int count=0;
  int release=0;

  process Π [int p ∈ {0, ..., P − 1}]
  {
    int local_sense = release;
    while (1)
    {
      Berechnung;
      local_sense = 1-local_sense; // Richtung wechseln
      CSenter;                    // Eintritt
      count=count+1;              // Zähler erhöhen
      CSexit;                     // Verlassen
      if (count==P) {
        count=0;                  // letzter löscht
        release=local_sense;      // und gibt frei
      } else
        while (release≠local_sense) ;
    }
  }
}
```

Nun merkt sich jeder Prozess in der Variable *local_sense* den Wert den *release* annehmen muss um weitermachen zu dürfen. Im ersten Durchlauf warten alle auf *release*==1, dann auf *release*==0, usw.

Auch mit Programm 4.11 sind wir noch nicht zufrieden. Es funktioniert zwar korrekt, ist jedoch für große Prozessorzahlen ineffizient. Kommen alle Prozesse gleichzeitig an der Barriere an, was aus Gründen optimaler Lastverteilung zu erwarten ist, so beträgt der Aufwand für den kritischen Abschnitt $O(P^2)$, wie in Abschnitt 4.1.5 gezeigt. Selbst ein Aufwand $O(P)$ wäre bei

größeren Prozessorzahlen nicht mehr tolerierbar, da der Aufwand für die Berechnungsphase in vielen Fällen konstant bleibt. Wir suchen also nach schnelleren Lösungen.

4.2.2 Barriere mit Baum und Flaggen

Wir beginnen zunächst mit zwei Prozessen und erweitern diese Lösung dann mit einem hierarchischen Ansatz auf 2^d Prozesse.

Die beiden Prozesse Π_0, Π_1 verwenden zwei gemeinsame Variable *arrived* und *continue*. In *arrived* vermerkt Prozess Π_1 wenn er an der Barriere angekommen ist. Prozess Π_0 wartet auf dieses Ereignis und informiert Π_1 dann mit Hilfe der Variable *continue* dass er weitermachen kann:

```
int arrived=0, continue=0;

Π0:                     Π1:
                        arrived=1;
while (¬arrived) ;
arrived=0;
continue=1;
                        while (¬continue) ;
                        continue=0;
```

Offensichtlich benötigt diese Lösung keine kritischen Abschnitte mehr. Es wird nur noch mit Bedingungssynchronisation gearbeitet. Die dabei verwendeten Synchronisationsvariablen werden manchmal auch als *Flaggen* bezeichnet. Man beachte wie die Flaggen sofort nach Eintreten der Bedingung wieder zurückgesetzt werden. Hier gilt die Regel: *Der Prozess der auf eine Flagge wartet setzt sie auch zurück. Eine Flagge darf erst erneut gesetzt werden wenn sie sicher zurückgesetzt wurde.* Dies wird im Beispiel dadurch gewährleistet, dass abwechselnd auf die eine und dann auf die andere gewartet wird.

Um diese Lösung auf 2^d Prozesse auszubauen verwenden wir eine Baumstruktur wie in Abbildung 4.1 dargestellt. Der Algorithmus läuft in $2d$ Stufen ab. In der ersten Stufe warten alle Prozesse deren letztes Bit der Prozessnummer 0 ist auf die andere Hälfte. Dann warten die deren letzte zwei Bits 00 sind auf die, deren letzte beiden Bits 10 sind usw. bis Prozess 0 weiss, dass alle angekommen sind. Dann läuft der Vorgang rückwärts ab um allen Bescheid zu geben, dass die Barriere verlassen werden kann. Die vollständige Implementierung zeigt

Programm 4.12 (Barriere mit Baum)

```
parallel tree-barrier
{
    const int d = 4;
    const int P = 2^d;
    int arrived[P]={0,...,0};
    int continue[P]={0,...,0};

    process Π [int p ∈ {0,...,P−1}]
    {
        int i, r, m, k;
```

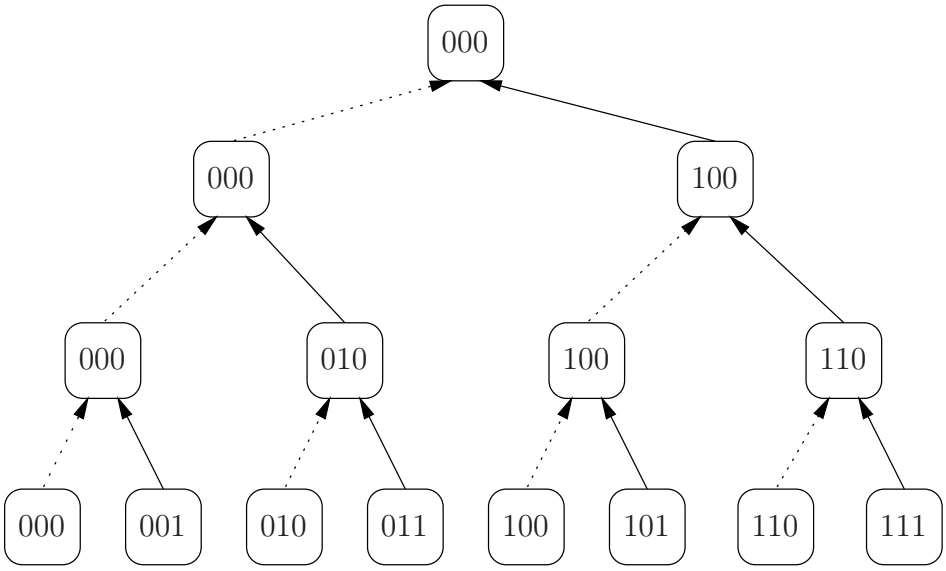


Abbildung 4.1: Baumstruktur für die Barriere.

```
while (1) {
    Berechnung;
    for (i = 0; i < d; i++)                    // aufwärts
    {
        r = p & [~ (Σ_{k=0}^{i} 2^k)];         // Bits 0 bis i löschen
        m = r | 2^i;                           // Bit i setzen
        if (p == m) arrived[m]=1;
        if (p == r)
        {
            while(¬arrived[m]) ;               // warte
            arrived[m]=0;
        }
    }
    for (i = d − 1; i ≥ 0; i–)                 // abwärts
    {
        r = p & [~ (Σ_{k=0}^{i} 2^k)];         // Bits 0 bis i löschen
        m = r | 2^i;
        if (p == m)
        {
            while(¬continue[m]) ;
            continue[m]=0;
        }
        if (p == r) continue[m]=1;
    }
}
```

```
    }
}
```

Der Aufwand für die in Programm 4.12 implementierte Barriere beträgt $O(\mathrm{ld}\, P)$. Die paarweisen Synchronisationen behindern sich gegenseitig nicht, wenn die *arrived*/*continue* Variablen in verschiedenen Cache–Lines gespeichert werden (dies beachtet obige Implementierung nicht!). Eine weitere Variante einer hierarchischen Barriere zeigt der nächste Abschnitt.

4.2.3 Barriere mit rekursiver Verdopplung

Programm 4.12 ist etwas inelegant, da nicht alle Prozesse identische Anweisungen ausführen. Deshalb betrachten wir erst einmal folgende symmetrische Variante zur Synchronisation zweier Prozesse Π_i, Π_j:

Π_i:	Π_j:
while (*arrived*[i]) ;	**while** (*arrived*[j]) ;
arrived[i]=1;	*arrived*[j]=1;
while ($\neg$*arrived*[j]) ;	**while** ($\neg$*arrived*[i]) ;
arrived[j]=0;	*arrived*[i]=0;

Dem Prozess Π_i ist eine Variable *arrived*[i] zugeordnet, die zu Beginn 0 ist. Betrachten wir erst nur die Zeilen 2–4, so setzt jeder *seine* Flagge auf 1, wartet dann bis die des anderen gesetzt ist (nun wissen beide, dass beide da sind) und setzt dann die Flagge des anderen zurück. Die erste Zeile ist notwendig um abzuwarten bis der andere die Flagge zurückgesetzt hat (sonst würde das nachfolgende Setzen übersehen werden).

Nun können wir diese Lösung mit d Stufen auf 2^d Prozesse erweitern. Die paarweise Synchronisation folgt dem Muster in Abbildung 4.2. Ein Pfeil von i nach j bedeutet, dass Prozess i auf Prozess j wartet. An der Abbildung kann man auch erkennen warum diese Lösung „rekursives Verdoppeln“ (*engl.* recursive doubling) heisst: Der Abstand zwischen den zu synchronisierenden Prozessen verdoppelt sich in jeder Stufe.

Wesentlich ist hierbei, dass jeder Prozess nach seiner Synchronisation auf der letzten Stufe $d-1$ bereits weiss, dass alle Prozesse angekommen sind. Das folgende Programm zeigt die elegante Implementierung:

Programm 4.13 (Barriere mit rekursiver Verdopplung)

```
parallel recursive-doubling-barrier
{
    const int d = 4;
    const int P = 2^d;
    int arrived[d][P]={0[P · d]};

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, q;

        while (1) {
```

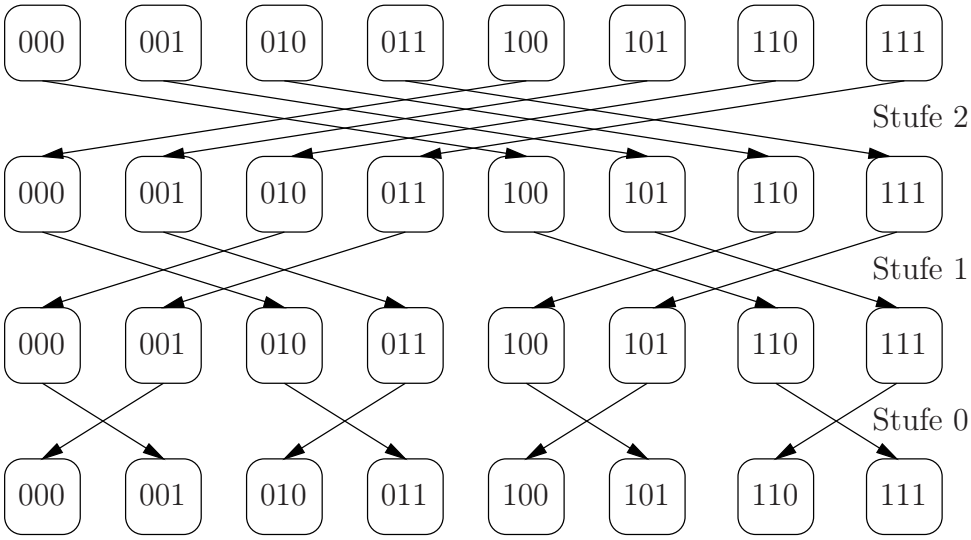


Abbildung 4.2: Struktur der rekursiven Verdopplung für die Barriere.

```
            Berechnung;
            for (i = 0; i < d; i++)              // alle Stufen
            {
                q = p ⊕ 2^i;                     // Bit i umschalten
                while (arrived[i][p]) ;
                arrived[i][p]=1;
                while (¬arrived[i][q]) ;
                arrived[i][q]=0;
            }
        }
    }
}
```

Hierbei bezeichnet der Operator $\oplus$ die bitweise exklusiv–oder–Verknüpfung.

4.3 Semaphore

Wir haben nun eine Reihe von Problemen der Prozesskoordination behandelt. Jedes dieser Probleme hat eine separate Lösung erfordert. Es wäre praktisch ein allgemeines Werkzeug zu besitzen mit dem sich verschiedene Prozesskoordinationsprobleme behandeln lassen. Semaphore sind ein solches Werkzeug.

Alle bisher in diesem Kapitel vorgestellten Programme haben aktives Warten verwendet um einen Prozess zu verzögern. Dies kann sehr ineffizient sein, insbesondere dann wenn Prozesse zeitverzahnt auf einem Prozessor ablaufen. In diesem Fall sollten wartende Prozesse keine Rechenzeit verbrauchen. Auch dieses Problem wird von der Semaphore gelöst.

4.3.1 Eigenschaften einer Semaphore

Die Semaphore kann man als abstrakten Datentyp einführen, d.h. als Datenstruktur mit Operation, die gewisse Eigenschaften erfüllen. Der Zustand der Semaphore kann nur mit Hilfe der darauf definierten Operationen verändert werden. Im objektorientierten Sinne würde man eine Semaphore als Klasse mit privaten Daten verstehen. Eine Semaphore S hat eine ganzzahlige, nichtnegative Komponente $value(S)$, die mit dem Anfangswert *init* initialisiert wird. Auf der Semaphore sind zwei Operationen definiert: $\mathbf{P}(S)$ und $\mathbf{V}(S)$. Es sei n_P die Zahl der auf S ausgeführten **P**–Operationen und entsprechend n_V die Zahl der auf S ausgeführten **V**–Operationen. Die Semaphore stellt sicher, dass folgende Bedingung gilt:

$$n_P \leq n_V + init. \tag{4.1}$$

Im Falle $n_P = n_V + init$ werden nachfolgend ausgeführte **P**–Operationen solange verzögert bis eine entsprechende **V**–Operation ausgeführt wird.

Für den Wert einer Semaphore gilt

$$value(S) = n_V + init - n_P \geq 0. \tag{4.2}$$

Entsprechend beschreiben die folgenden drei Regeln das Verhalten einer Semaphore S:

- Die Semaphore hat zu Beginn den Wert $value(S) = init$.
- $\mathbf{P}(S)$ erniedrigt den Wert von S um eins falls $value(S) > 0$, sonst blockiert der Prozess solange bis ein anderer Prozess eine **V**–Operation auf S ausführt.
- $\mathbf{V}(S)$ befreit einen anderen Prozess aus seiner **P**–Operation falls einer wartet (warten mehrere wird einer ausgewählt), ansonsten wird der Wert von S um eins erhöht. **V**–Operationen blockieren nie!

Der Wert einer Semaphore ist nach aussen nicht sichtbar. Er äußert sich nur durch die Ausführbarkeit der **P**–Operation. Das Erhöhen bzw. Erniedrigen einer Semaphore erfolgt atomar, mehrere Prozesse können also **P**/**V**–Operationen gleichzeitig durchführen. Darüberhinaus können mehrere Prozesse in einer **P**–Operation blockieren. Jede **V**–Operation weckt dann einen Prozess auf.

Semaphore, die einen Wert größer als eins annehmen können bezeichnet man als *allgemeine Semaphore*, Semaphore die nur die Werte $\{0, 1\}$ annehmen können heißen *binäre Semaphore*.

In unserer hypothetischen Programmiersprache führen wir einen neuen Datentyp **Semaphore** ein. Variablen dieses Typs müssen immer initialisiert werden:

Semaphore S=1;

Auch Felder sind möglich:

Semaphore *forks*[5] = {1 [5]};

Semaphore können mit aktivem Warten und kritischen Abschnitten implementiert werden (siehe Übungen). Aufwendigere Implementierungen arbeiten mit der Scheduler–Komponente des Betriebsystems zusammen und versetzen wartende Prozesse in den inaktiven Zustand. Bei Ausführung einer **V**–Operation wird einer der wartenden Prozesse (üblicherweise in der Reihenfolge der Ankunft, d.h. *first come first served*) wieder in die Schlange der rechenbereiten Prozesse eingereiht.

Wir stellen nun vor wie die uns schon bekannten Koordinationsaufgaben mit Semaphoren gelöst werden können.

4.3.2 Wechselseitiger Ausschluss

Der Zutritt zu einem kritischen Abschnitt kann mit einer binären Semaphore abgesichert werden. Die Semaphore wird mit dem Wert 1 initialisiert. Hier ist das sehr simple Programm:

Programm 4.14 (Wechselseitiger Ausschluss mit Semaphore)

```
parallel cs-semaphore
{
    const int P=8;
    Semaphore mutex=1;
    process Π [int i ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            P(mutex);
            kritischer Abschnitt;
            V(mutex);
            unkritischer Abschnitt;
        }
    }
}
```

4.3.3 Barriere

Wir betrachten nur eine Barriere mit zwei Prozessen. Erweiterung auf 2^d Prozesse kann mit rekursiver Verdopplung erfolgen. Eine Barriere muss zwei Bedingungen erfüllen:

- Jeder Prozess muss verzögert werden bis der andere an der Barriere ankommt.
- Die Barriere muss wiederverwendbar sein, da sie in der Regel wiederholt ausgeführt wird.

Die simple Lösung für zwei Prozesse zeigt

Programm 4.15 (Barriere mit Semaphore für zwei Prozesse)

```
parallel barrier-2-semaphore
{
    Semaphore b1=0, b2=0;
    process Π1                          process Π2
    {                                   {
        while (1) {                         while (1) {
            Berechnung;                         Berechnung;
```

```
            V(b1);                          V(b2);
            P(b2);                          P(b1);
        }                               }
    }                               }
}
```

Zu Beginn haben beide Semaphore den Wert 0. Durch die **V**–Operation auf *b1* signalisiert Prozess Π_1 dem Π_2 dass er angekommen ist. Π_2 wartet in der **P**–Operation auf dieses Signal. Der symmetrische Vorgang gilt für *b2*.

Überlegen wir nun, dass ein Prozess den Anderen nicht überholen kann. Rollen wir die Schleife ab, dann sieht es so aus:

Π_1:	Π_2:
Berechnung 1;	Berechnung 1;
V(*b1*);	**V**(*b2*);
P(*b2*);	**P**(*b1*);
Berechnung 2;	Berechnung 2;
V(*b1*);	**V**(*b2*);
P(*b2*);	**P**(*b1*);
Berechnung 3;	Berechnung 3;
V(*b1*);	**V**(*b2*);
P(*b2*);	**P**(*b1*);
...	...

Angenommen Prozess Π_1 arbeitet an Berechnungsphase i, heisst das er hat **P**(*b2*) $i-1$-mal ausgeführt. Angenommen Π_2 *arbeitet an* Berechnungphase $j < i$, heisst das er hat **V**(*b2*) $j-1$ mal ausgeführt, somit gilt

$$n_P(b2) = i - 1 > j - 1 = n_V(b2).$$

Dies ist aber ein Widerspruch zur Semaphorenregel

$$n_P(b2) \leq n_V(b2) + 0$$

und somit kann $j < i$ nicht gelten. Das Argument ist symmetrisch und gilt auch bei Vertauschen der Prozessornummern was dann zeigt, dass $j > i$ auch nicht gelten kann. Somit kann nur $j = i$ gelten und beide Prozessuren müssen immer in der selben Iteration arbeiten.

4.3.4 Erzeuger–Verbraucher–Probleme

Ein oder mehrere Erzeugerprozesse generieren Aufträge die von einem oder mehreren Verbrauchern abgearbeitet werden sollen. Erzeuger und Verbraucher kommunizieren über einen Puffer. Wir behandeln drei Varianten:

- m Erzeuger, n Verbraucher, 1 Pufferplatz,
- 1 Erzeuger, 1 Verbraucher, k Pufferplätze,
- m Erzeuger, n Verbraucher, k Pufferplätze.

Ein Erzeuger–Verbraucher–Problem trat bereits in Abschnitt 4.1.7 auf. Die Lösung erforderte bedingte kritische Abschnitte, die nur recht ineffizient mit aktivem Warten implementiert werden können. Auch hier werden Semaphore eine elegante und effiziente Lösung ermöglichen.

m Erzeuger, n Verbraucher, 1 Pufferplatz

Hier sind Erzeuger und Verbraucher wie folgt zu koordinieren: Ist der Puffer leer, so kann ihn ein Erzeuger füllen, ansonsten muss er warten bis er leer wird. Ist der Puffer belegt (voll) so kann ein Verbraucher den Auftrag entnehmen, ansonsten muss er warten bis ein Auftrag kommt. Wir verwenden somit eine (binäre) Semaphore *empty*, die die freien Pufferplätze zählt und eine (binäre) Semaphore *full*, die die abgelegten Aufträge zählt.

Programm 4.16 (m Erzeuger, n Verbraucher, 1 Pufferplatz)

```
parallel prod-con-nm1
{
    const int m = 3, n = 5;
    Semaphore empty=1;          // freier Pufferplatz
    Semaphore full=0;           // abgelegter Auftrag
    T buf;                      // der Puffer

    process P [int i ∈ {0, ..., m − 1}]
    {
        while (1)
        {
            Erzeuge Auftrag t;
            P(empty);           // Ist Puffer frei?
            buf = t;            // speichere Auftrag
            V(full);            // Auftrag abgelegt
        }
    }

    process C [int j ∈ {0, ..., n − 1}]
    {
        while (1)
        {
            P(full);            // Ist Auftrag da?
            t = buf;            // entferne Auftrag
            V(empty);           // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

Der Puffer wird durch die Variable *buf* vom Typ T realisiert. Ist der Puffer frei, d.h. $value(empty)=1$, so betritt ein P_i den kritischen Abschnitt, legt den Auftrag ab und signalisiert, dass ein Auftrag vorhanden ist, d.h. $value(full)=1$. In diesem Fall betritt ein C_j den kritischen Abschnitt, liest den Auftrag und signalisiert, dass der Puffer frei ist. Somit gilt für die beiden Semaphore die Eigenschaft

$$0 \leq empty + full \leq 1. \tag{4.3}$$

Semaphore mit dieser Eigenschaft werden als *geteilte binäre Semaphore* bezeichnet (*engl.* split binary semaphore).

1 Erzeuger, 1 Verbraucher, k Pufferplätze

Der Puffer wird durch ein Feld der Länge k realisiert. Neue Aufträge werden am Index *front* eingefügt während gespeicherte Aufträge am Index *rear* entnommen werden. Das Einfügen geschieht mittels

$$buf[front]; \quad front = (front + 1) \mod k;$$

und das Entfernen mit

$$buf[rear]; \quad rear = (rear + 1) \mod k;$$

Die Struktur von Programm 4.16 kann vollkommen beibehalten werden. Die Semaphore *empty* wird nun mit dem Wert k initialisiert. Damit können dann k Aufträge im Puffer abgelegt werden bevor der Erzeuger blockiert wird. Die Implementierung zeigt

Programm 4.17 (1 Erzeuger, 1 Verbraucher, k Pufferplätze)

```
parallel prod-con-11k
{
    const int k = 20;
    Semaphore empty=k;          // zählt freie Pufferplätze
    Semaphore full=0;           // zählt abgelegte Aufträge
    T buf[k];                   // der Puffer
    int front=0;                // neuester Auftrag
    int rear=0;                 // ältester Auftrag

    process P
    {
        while (1)
        {
            Erzeuge Auftrag t;
            P(empty);           // Ist Puffer frei?
            buf[front] = t;     // speichere Auftrag
            front = (front+1) mod k; // nächster freier Platz
            V(full);            // Auftrag abgelegt
        }
    }

    process C
    {
        while (1)
        {
            P(full);            // Ist Auftrag da?
            t = buf[rear];      // entferne Auftrag
            rear = (rear+1) mod k; // nächster Auftrag
            V(empty);           // Puffer ist frei
            Bearbeite Auftrag t;
        }
    }
}
```

Das Erhöhen der Zeiger *front* und *rear* muss nicht unter wechselseitigen Ausschluss erfolgen, da P nur *front* bearbeitet und C nur *rear*. Man beachte dass P und C gleichzeitig Aufträge in den Puffer schreiben und entfernen können.

m Erzeuger, n Verbraucher, k Pufferplätze

Dieses Programm ist eine einfache Erweiterung von Programm 4.17. Wir müssen nur sicherstellen, dass Erzeuger untereinander und Verbraucher untereinander nicht gleichzeitig den Puffer manipulieren. Dazu verwenden wir zwei zusätzliche binäre Semaphore:

Programm 4.18 (m Erzeuger, n Verbraucher, k Pufferplätze)

```
parallel prod-con-mnk
{
    const int k = 20, m = 3, n = 6;
    Semaphore empty=k;          // zählt freie Pufferplätze
    Semaphore full=0;           // zählt abgelegte Aufträge
    T buf[k];                   // der Puffer
    int front=0;                // neuester Auftrag
    int rear=0;                 // ältester Auftrag
    Semaphore mutexP=1;         // Zugriff der Erzeuger
    Semaphore mutexC=1;         // Zugriff der Verbraucher

    process P [int i ∈ {0, ..., m − 1}]
    {
        while (1)
        {
            Erzeuge Auftrag t;
            P(empty);           // Ist Puffer frei?
            P(mutexP);          // manipuliere Puffer
            buf[front] = t;     // speichere Auftrag
            front = (front+1) mod k; // nächster freier Platz
            V(mutexP);          // fertig mit Puffer
            V(full);            // Auftrag abgelegt
        }
    }

    process C [int j ∈ {0, ..., n − 1}]
    {
        while (1)
        {
            P(full);            // Ist Auftrag da?
            P(mutexC);          // manipuliere Puffer
            t = buf[rear];      // entferne Auftrag
            rear = (rear+1) mod k; // nächster Auftrag
            V(mutexC);          // fertig mit Puffer
            V(empty);           // Puffer ist frei
            Bearbeite Auftrag t;
        }
```

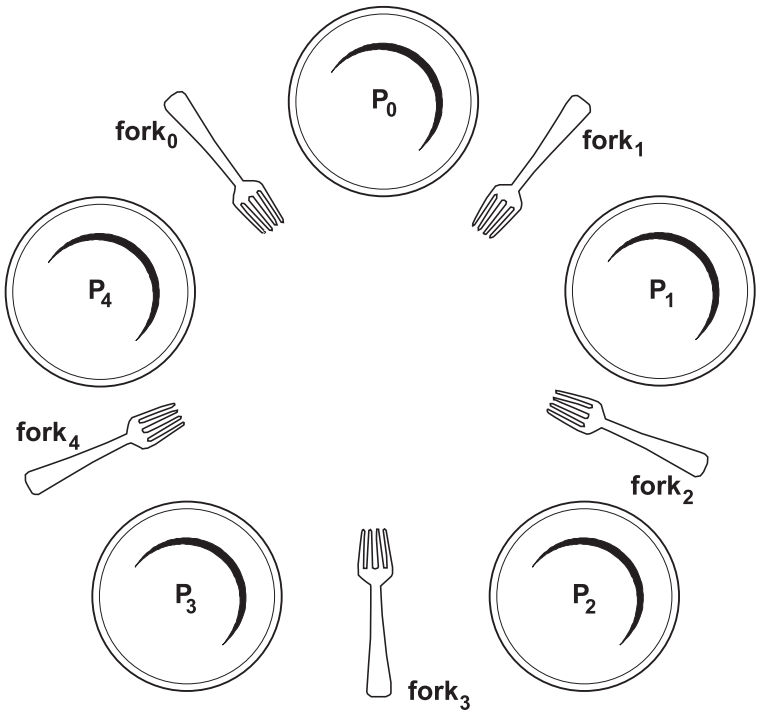


Abbildung 4.3: Illustration des Philosophenproblems.

```
    }
}
```

4.4 Überlappende kritische Abschnitte

In diesem Abschnitt betrachten wir zwei komplexere Probleme der Prozesskoordination.

4.4.1 Die speisenden Philosophen

Das Problem der speisenden Philosophen ist ein echter Klassiker in der parallelen Programmierung. Hier geht es darum, dass ein Prozess exklusiven Zugriff auf mehrere Resourcen benötigt bevor er seine Aufgabe erfüllen kann. Das Problem wird in folgender Weise beschrieben:

> Fünf Philosophen sitzen an einem runden Tisch. Die Tätigkeit jedes Philosophen besteht aus den sich abwechselnden Phasen des Denkens und des Essens. Zwischen je zwei Philosophen liegt eine Gabel und in der Mitte steht ein Berg Spaghetti. Zum Essen benötigt ein Philosoph zwei Gabeln – die links *und* rechts von ihm liegende. Das Problem besteht darin, ein Programm zu schreiben, mit einem Prozess pro Philosoph, das die obige Aufgabe löst.

Abbildung 4.3 illustriert das Philosophenproblem. Jeder Philosoph führt also abwechselnd folgende Tätigkeiten durch:

```
while (1)
{
    Denke;
    Nehme Gabeln;
    Esse;
    Lege Gabeln zurück;
}
```

Betrachten wir zunächst folgende naive Lösung bei der jeder Philosoph erst seine rechte und dann seine linke Gabel ergreift. Jede Gabel wird dabei durch eine binäre Semaphore realisiert. Der Wert 1 zeigt an, dass die Gabel frei ist:

Programm 4.19 (Naive Lösung des Philosophenproblems)

```
parallel philosphers–1
{
    const int P = 5;                        // Anzahl Philosophen
    Semaphore forks[P] = { 1 [P] };         // Gabeln

    process Philosopher [int p ∈ {0, ..., P − 1}]
    {
        while (1)
        {
            Denke;
            P(fork[p]);                     // rechte Gabel
            P(fork[(p + 1) mod P]);         // linke Gabel
            Esse;
            V(fork[p]);                     // rechte Gabel
            V(fork[(p + 1) mod P]);         // linke Gabel
        }
    }
}
```

Bei dieser Lösung besteht die Gefahr der Verklemmung. Wollen alle Philosophen zugleich essen so nehmen sie zuerst alle die linke Gabel und warten dann auf die rechte Gabel, die sie nie bekommen können. Verklemmungen können nur bei einer zyklischen Abhängigkeit vorkommen. Eine einfache Möglichkeit eine Verklemmung zu vermeiden besteht darin zyklische Abhängigkeiten auszuschließen, z.B. indem ein Prozess die Reihenfolge der Aufnahme der Gabeln umdreht. Nehmen wir an dies sei Prozess 0. Wollen nun alle zugleich essen so greifen alle bis auf Prozess 0 zuerst nach rechts und Prozess 0 greift nach links. Angenommen Prozess 0 ist schneller als Prozess 1 und erhält seine linke Gabel. Nehmen wir an er ist auch schneller als Prozess 4 (der seine rechte Gabel ergriffen hat) und somit kann Prozess 0 essen (ansonsten würde Prozess 4 essen). Die Lösung hat nur einen Haken: Nur Prozess 0 kann essen, alle anderen haben höchstens eine Gabel und warten. Dies ist nicht optimal, da immer zwei von fünf Philosophen essen könnten!

Der Grund für dieses Verhalten ist, dass eine Gabel sofort ergriffen wird sobald sie frei ist ohne vorher zu prüfen ob auch die Chance besteht die andere Gabel zu bekommen. Die nachfolgend beschriebene Lösung ermöglicht immer die maximale Zahl essender Philosophen. Es wird von der Tatsache ausgegangen, dass ein Philosoph nur essen kann wenn sein linker *und* rechter Nachbar *nicht* essen.

Programm 4.20 (Lösung des Philosophenproblems)

```
parallel philosphers–2
{
    const int P = 5;                        // Anzahl Philosophen
    const int think=0, hungry=1, eat=2;
    Semaphore mutex=1;
    Semaphore s[P] = { 0 [P] };             // essender Philosoph
    int state[P] = { think [P] };           // Zustand

    process Philosopher [int p ∈ {0, ..., P − 1}]
    {
        void test (int i)
        {
            int r=(i + P − 1) mod P, l=(i + 1) mod P;

            if (state[i]==hungry ∧ state[l]≠eat ∧ state[r]≠eat)
            {
                state[i] = eat;
                V(s[i]);
            }
        }

        while (1)
        {
            Denke;

            P(mutex);                       // Gabeln nehmen
            state[p] = hungry;
            test(p);
            V(mutex);
            P(s[p]);

            Esse;

            P(mutex);                       // Gabeln weglegen
            state[p] = think;
            test((p + P − 1) mod P);
            test((p + 1) mod P);
            V(mutex);
        }
    }
}
```

Jeder Philosoph befindet sich in einem der Zustände *think*, *hungry* oder *eat*. Der Zustand *hungry* zeigt an, dass der Philosoph gerne essen möchte jedoch die Gabeln noch nicht ergriffen hat. Die Funktion $test(i)$ überprüft in diesem Fall ob beide Nachbarn von Philosoph i nicht gerade essen und setzt ihn, wenn ja, in den Zustand *eat*. Die binäre Semaphore $s[i]$ wird dann erhöht um den wartenden Prozess zu wecken, oder, falls $p == i$, ihn in der nachfolgenden **P**–

Operation nicht zu verzögern. Um Interferenzen zu vermeiden muss die Zustandsmanipulation unter wechselseitigem Ausschluss durchgeführt werden. Dazu dient die Semaphore *mutex*.

Will ein Philosoph in der **while**–Schleife zum Essen übergehen, so setzt er im kritischen Abschnitt seinen Zustand auf *hungry*. In der Funktion *test* werden dann die beiden Nachbarn geprüft und die Semaphore $s[p]$ erhöht falls beide nicht essen. Dann verlässt Prozess p den kritischen Abschnitt um anderen Prozessen wieder eine Zustandsmanipulation zu ermöglichen und führt eine **P**–Operation auf $s[i]$ aus. Ist keiner der Nachbarn im Zustand *eat* so wird diese nicht blockieren, ansonsten muss der Prozess warten bis ihn einer der beiden Nachbarn weckt.

Nach dem Essen wird der Zustand wieder auf *think* gesetzt und geprüft ob einer der beiden Nachbarn essen will (und kann) und gegebenenfalls geweckt.

Diese Lösung ist verklemmungsfrei da die Zustandsmanipulation immer exklusiv erfolgt, allerdings werden dadurch eventuell Prozesse verzögert obwohl sie sich gegenseitig nicht beeinflussen würden. Außerdem besteht die Gefahr des Verhungerns (*engl.* starvation) wenn beide Nachbarn eines Philosophen immer abwechselnd essen.

Praktische Anwendung findet das Philosophenproblem bei der Zuteilung von Resourcen innerhalb des Betriebssystems. Dort kann ein Prozess gleichzeitigen, exklusiven Zugriff auf verschiedene Klassen von Resourcen benötigen (z.B. Drucker, Bandlaufwerke).

4.4.2 Leser–Schreiber–Problem

Ein anderes klassisches Synchronisationsproblem ist das Leser–Schreiber–Problem, das wie folgt beschrieben wird:

> Zwei Klassen von Prozessen, Leser und Schreiber, greifen auf eine gemeinsame Datenbank zu. Leser führen Transaktionen aus, die die Datenbank nicht verändern. Schreiber verändern die Datenbank und benötigen exklusiven Zugriff. Falls kein Schreiber Zugriff hat können beliebig viele Leser gleichzeitig zugreifen.

Wir verwenden eine binäre Semaphore *rw*, die den exklusiven Zugriff der individuellen Schreiber und der Leser als Gruppe sicherstellt. Nur der erste Leser muss den exklusiven Zugriff der Leser gewährleisten. Der exklusive Zugriff auf die Zahl der zugreifenden Leser wird durch eine weitere binäre Semaphore *mutexR* abgesichert. Hier ist ein Lösungsvorschlag:

Programm 4.21 (Leser–Schreiber–Problem, erste Lösung)

```
parallel readers–writers–1
{
    const int m = 8, n = 4;          // Anzahl Leser und Schreiber
    Semaphore rw=1;                  // Zugriff auf Datenbank
    Semaphore mutexR=1;              // Anzahl Leser absichern
    int nr=0;                        // Anzahl zugreifender Leser

    process Reader [int i ∈ {0, ..., m − 1}]
    {
        while (1)
        {
            P(mutexR);               // Zugriff Leserzähler
            nr = nr+1;               // Ein Leser mehr
            if (nr==1) P(rw);        // Erster wartet auf DB
            V(mutexR);               // nächster Leser kann rein
```

```
            lese Datenbank;

            P(mutexR);                  // Zugriff Leserzähler
            nr = nr-1;                  // Ein Leser weniger
            if (nr==0) V(rw);           // Letzter gibt DB frei
            V(mutexR);                  // nächster Leser kann rein
        }
    }
    process Writer [int j ∈ {0, ..., n − 1}]
    {
        while (1)
        {
            P(rw);                      // Zugriff auf DB
            schreibe Datenbank;
            V(rw);                      // gebe DB frei
        }
    }
}
```

Die vorgestellte Lösung gibt Lesern den Vorzug (*engl.* reader preference solution). Für die Schreiber besteht die Gefahr des Verhungerns wenn nie alle Leser gleichzeitig den Zugriff auf die Datenbank aufgeben.

Um eine faire Behandlung zu erreichen sollen nun alle Anfragen streng der Reihe nach bearbeitet werden. Weiterhin können beliebig viele Leser gleichzeitig lesen. Meldet sich jedoch ein Schreiber an, so werden alle weiteren Leser blockiert bis der Schreiber dran war. Die Implementierung zeigt das folgende

Programm 4.22 (Leser–Schreiber–Problem, faire Lösung)

```
parallel readers–writers–2
{
    const int m = 8, n = 4;             // Anzahl Leser und Schreiber
    int nr=0, nw=0, dr=0, dw=0;         // Zustand
    Semaphore e=1;                      // Zugriff auf Warteschlange
    Semaphore r=0;                      // Verzögern der Leser
    Semaphore w=0;                      // Verzögern der Schreiber
    const int reader=1, writer=2;       // Marken
    int buf[n + m];                     // Wer wartet?
    int front=0, rear=0;                // Zeiger

    int wake_up (void)                  // darf genau einer ausführen
    {
        if (nw==0 ∧ dr>0 ∧ buf[rear]==reader){
            dr = dr-1;
            rear = (rear+1) mod (n + m);
            V(r);
            return 1;                   // habe einen Leser geweckt
        }
        if (nw==0 ∧ nr==0 ∧ dw>0 ∧ buf[rear]==writer){
```

```
        dw = dw-1;
        rear = (rear+1) mod (n + m);
        V(w);
        return 1;                    // habe einen Schreiber geweckt
    }
    return 0;                        // habe keinen geweckt
}

process Reader [int i ∈ {0, ..., m − 1}]{
    while (1) {
        P(e);                        // will Zustand verändern
        if(nw>0 ∨ dw>0){
            buf[front] = reader;     // in Warteschlange
            front = (front+1) mod (n + m);
            dr = dr+1;
            V(e);                    // Zustand freigeben
            P(r);                    // warte bis Leser dran sind
                                     // hier ist e = 0 !
        }
        nr = nr+1;                   // hier ist nur einer
        if (wake_up()==0)            // kann einer geweckt werden?
            V(e);                    // nein, setze e = 1

        lese Datenbank;

        P(e);                        // will Zustand verändern
        nr = nr-1;
        if (wake_up()==0)            // kann einer geweckt werden?
            V(e);                    // nein, setze e = 1
    }
}

process Writer [int j ∈ {0, ..., n − 1}]
{
    while (1){
        P(e);                        // will Zustand verändern
        if(nr>0 ∨ nw>0){
            buf[front] = writer;     // in Warteschlange
            front = (front+1) mod (n + m);
            dw = dw+1;
            V(e);                    // Zustand freigeben
            P(w);                    // warte bis an der Reihe
                                     // hier ist e = 0 !
        }
        nw = nw+1;                   // hier ist nur einer
        V(e);                        // hier braucht keiner geweckt werden
```

```
            schreibe Datenbank;          // exklusiver Zugriff

            P(e);                        // will Zustand verändern
            nw = nw-1;
            if (wake_up()==0)            // kann einer geweckt werden?
                V(e);                    // nein, setze e = 1
        }
    }
}
```

Es werden folgende Variable verwendet: *nr* zählt die Zahl der aktiven Leser, *nw* die Zahl der aktiven Schreiber (≤ 1), *dr* die Zahl der wartenden Leser und *dw* die Zahl der wartenden Schreiber. Muss ein Prozess warten so fügt er eine Marke in den Puffer *buf* ein. Somit ist erkennbar in welcher Reihenfolge die wartenden Prozesse angekommen sind. Die binäre Semaphore *e* sichert den exklusiven Zugriff auf die Zustandsinformation (alle oben genannten Variablen) ab. Semaphore *r* und *w* werden zum blockieren der Prozesse verwendet.

Wartende Prozesse werden mit der Funktion *wake_up* geweckt. Diese prüft ob ein Leser oder Schreiber geweckt werden kann. Sie liefert 1 falls ein Prozess geweckt wurde und sonst 0. Es gibt nur einen Prozess im kritischen Abschnitt dem eine Zustandsmanipulation erlaubt ist. Nur dieser Prozess kann genau einen anderen Prozess wecken, der sich dann im kritischen Abschnitt befindet. Der weckende Prozess führt dann keine weiteren kritischen Zugriffe auf gemeinsame Variable mehr aus. Diese Programmiertechnik bezeichnet man als *Weiterreichen des Staffelstabes* (*engl.* passing the baton).

Betrachten wir einen Leser. Dieser tritt in den kritischen Abschnitt ein. War ein Schreiber vor ihm da ($nw > 0 \vee dw > 0$) so trägt er sich in die Warteschlange ein und blockiert sich unter vorherigem Verlassen des kritischen Abschnittes. Gibt es keine Schreiber so wird die Zahl der Leser erhöht und möglicherweise ein anderer Prozess geweckt (dies kann wegen $nr > 0$ nur ein Leser sein). Falls niemand wartet muss der kritische Abschnitt verlassen werden. Nach dem Lesen wird innerhalb des kritischen Abschnittes die Zahl der Leser erniedrigt und möglicherweise ein anderer Prozess geweckt (dies kann hier nur ein Schreiber sein, da andere Leser bereits oben geweckt worden wären oder gar nicht erst gewartet hätten).

Der Schreiberprozess folgt dem selben Muster, nur müssen bei erlangtem Zugriff keine weiteren Prozesse geweckt werden (Schreiben ist exklusiv). Nach dem Schreibzugriff kann sowohl ein Leser als auch ein Schreiber geweckt werden, je nachdem wer als nächstes in der Warteschlange steht.

4.5 PRAXIS: Posix Threads

POSIX steht für Portable Operating System Interface, ein Standard für UNIXartige Betriebssysteme. Prozesse haben in UNIX immer einen *getrennten* Adressraum. Das Betriebssystem braucht relativ viel Zeit um von der Ausführung eines Prozesses auf einen anderen zu wechseln. Deshalb hat man sogenannte leichtgewichtige Prozesse oder „threads“eingeführt. Wir wollen hier den sogennanten PThreads Standard, siehe (Nichols, Buttlar und Farell 1996) oder (Stevens 1999), verwenden der Teil des POSIX ist. Nach diesem Modell kann sich ein UNIX-Prozess in mehrere Ausführungspfade verzweigen, die dann echt (bei Mehrprozessorbetrieb) oder quasiparallel ausgeführt werden.

Wie wir sehen werden lassen sich unsere bisherigen Programme sehr leicht mit PThreads realisieren. Stehen mehrere Prozessoren zur Verfügung können die Threads echt parallel ausgeführt werden, ansonsten werden sie zeitverzahnt auf einem Prozessor abgearbeitet.

Wir zeigen nun wie ein Erzeuger–Verbraucher–Problem mit einem Erzeuger, einem Verbraucher und einem Pufferplatz realisiert werden kann. Das Programm ist in C geschrieben und sollte sich auf nahezu jedem UNIX System, z.B. LINUX, übersetzen lassen. Wir gehen nun durch den Programmcode und erläutern die einzelnen Schritte.

POSIX Threads und Semaphore werden durch eine Bibliothek realisiert die zu dem Programm hinzugebunden werden muss. Um die Funktionen zu benutzen müssen zwei Header–Dateien geladen werden:

```
/*****************************************************
 * Producer/Consumer with POSIX threads and semaphores
 *****************************************************/
#include <stdio.h>       /* for printf      */
#include <pthread.h>     /* for threads     */
#include <semaphore.h>   /* for semaphores  */
```

Die Semaphore wird durch den Datentyp `sem_t` realisiert. Der Puffer ist einfach ein Integer–Wert in unserem Beispiel:

```
sem_t empty; /* init 1, counts free buffer slots */
sem_t full;  /* init 0, counts tasks stored      */

int buf;     /* the buffer */
```

Mit den POSIX Threads kann eine C–Funktion als eigener Thread gestartet werden. Threads können Parameter erhalten und auch Ergebnisse zurückliefern, allerdings nur in eingeschränkter Art und Weise: Als Parameter kann man einen (und nur einen) Zeiger übergeben und als Ergebnis kann ein Thread nur einen Zeiger liefern (oder gar nichts wie in unserem Beispiel). Die nun folgende Prozedur `P` realisiert einen Erzeugerprozess und dürfte selbsterklärend sein:

```
/* Producer thread */
void P (int *i)
{
    int count=0;

    while (1)
    {
        count++;
        printf ("prod: %d\n",count);
        sem_wait(&empty);
        buf=count;
        sem_post(&full);
    }
}
```

Die Funktionen `sem_wait` und `sem_post` entsprechen genau den Funktionen **P** und **V**. Als Argument erhalten sie einen Zeiger auf eine Semaphore.

Nun kommt der Verbraucher `C`:

```
/* Consumer thread */
void C (int *j)
{
    int count=0;

    while (1)
    {
        sem_wait(&full);
        count=buf;
        sem_post(&empty);
        printf ("con: %d\n",count);
    }
}
```

Das nun folgende Hauptprogramm initialisiert die beiden Semaphore mit dem Anfangswert, startet die beiden Threads, wartet bis beide Threads beendet sind (was hier leider nie passiert) und vernichtet dann die beiden Semaphore:

```
int main (int argc, char *argv[])
{
    pthread_t thread_p, thread_c;
    int i,j;

    /* initialize semaphores */
    sem_init(&empty,0,1);
    sem_init(&full ,0,0);

    /* start threads */
    i = 1;
    pthread_create(&thread_p,
                   NULL,
                   (void *) P,
                   (void *) &i);

    j = 1;
    pthread_create(&thread_c,
                   NULL,
                   (void *) C,
                   (void *) &j);

    /* wait for threads to terminate ... */
    pthread_join(thread_p, NULL);
    pthread_join(thread_c, NULL);

    /* destroy semaphores */
    sem_destroy(&empty);
    sem_destroy(&full);
```

```
    return(0);
}
```

Die Funktion `sem_init` initialisiert eine Semaphore. Der erste Parameter ist ein Zeiger auf die Semaphore und der dritte Parameter ist der Anfangswert. Der zweite Parameter ist immer 0 wenn nur Threads eines Prozesses auf die Semaphore zugreifen.

Die Funktion `pthread_create` startet eine C–Funktion als eigenständigen Thread. Das erste Argument ist ein Zeiger auf eine Variable vom Typ `pthread_t`, der Verwaltungsinformation enthält. Das dritte Argument ist die C–Funktion, während das vierte Argument das Argument für den zu startenden Thread ist.

Die Funktion `pthread_join` wartet bis der angegebene Thread beendet ist. Der Thread wird über die `pthread_t`–Datenstruktur identifiziert. Schließlich führt `sem_destroy` noch eventuelle Aufräumarbeiten für eine nicht mehr benötigte Semaphore durch.

Mit Semaphoren lassen sich, wie oben gezeigt, praktisch alle Koordinationsprobleme lösen. Speziell für kritische Abschnitte bietet PThreads die mutex-Variablen, wie folgendes Programmstück zeigt:

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void foo (void)
{
    pthread_mutex_t mutex2;

    pthread_mutex_init(&mutex2,NULL);

    pthread_mutex_lock(&mutex1);
    ... kritischer Abschnitt hier ...;
    pthread_mutex_unlock(&mutex1);
}
```

`pthread_mutex_t`-Variablen müssen initialisiert werden, entweder statisch, wie `mutex1`, oder dynamisch wie `mutex2`. Nach Initialisierung ist der kritische Abschnitt frei.

Ein ausführliches Beispiel zu den pthreads findet sich im Anhang.

4.6 Übungen

Übung 4.1 Implementieren Sie den wechselseitigen Ausschluss mit einer Warteschlange, d.h. findet ein Prozess den Abschnitt besetzt vor soll er sich in ein Feld eintragen. Der Prozess im kritischen Abschnitt wählt vor Verlassen des kritischen Abschnittes den nächsten Prozess aus und teilt ihm mit, dass er eintreten kann.

Übung 4.2 Würden Sie Programm 4.3 zur Sicherung von wechselseitigem Ausschluss verwenden? Überlegen Sie ob einer der beiden Prozesse *immer* seine **if**–Bedingung als wahr auswertet.

Übung 4.3 Betrachten Sie folgendes Programmsegment:

1: $A = 3$;
2: **if** $(B == 5)$ $C = 7$;

3: aquire;
4: $D = 6$;
5: $A = D$;
6: $E = 7$;
7: release;
8: $F = 9$;

Stellen Sie mit Hilfe eines gerichteten Graphen dar welche Zugriffe bei den verschiedenen Konsistenzmodellen (sequentielle Konsistenz, Prozessorkonsistenz, schwache Konsistenz und Freigabekonsistenz) aufeinander warten müssen.

Übung 4.4 Spielen Sie Programm 4.5 mit 3 Prozessen durch.

Übung 4.5 Erweitern Sie Programm 4.2 mittels eines hierarchischen Ansatzes auf $P = 2^d$ Prozesse. Können Sie schließliches Eintreten sicherstellen?

Übung 4.6 Realisieren Sie die Bildung einer globalen Summe (siehe Programm 1.6) mit rekursiver Verdopplung.

Übung 4.7 Schreiben Sie eine C++ Klassendefinition für eine Semaphore. Implementieren Sie die **P**– und **V**–Operationen mit aktivem Warten.

Übung 4.8 Schreiben Sie ein Programm für das Erzeuger–Verbraucher–Problem mit m Erzeugern, n Verbrauchern und k Pufferplätze unter POSIX Threads. Realisieren Sie das Generieren und Verarbeiten von Aufträgen durch zufällig gewählte Wartezeiten. Experimentieren Sie mit verschiedenen Pufferlängen und messen Sie die Programmlaufzeit.

Übung 4.9 *Das Saunaproblem.* Frauen und Männer wollen eine Sauna besuchen. Personen eines Geschlechtes können die Sauna gleichzeitig benutzen, Personen verschiedener Geschlechter jedoch nicht. Entwickeln Sie eine faire Lösung für dieses Problem. Erweitern Sie die Lösung so, dass nur k Personen (eines Geschlechtes) gleichzeitig die Sauna benutzen.

5 Nachrichtenaustausch

In diesem Kapitel behandeln wir zunächst Syntax und Semantik der Funktionen für den Nachrichtenaustausch und zeigen dann eine Reihe von grundlegenden Programmiertechniken.

5.1 Funktionen für den Nachrichtenaustausch

Die ersten Mechanismen zum Nachrichtenaustausch zwischen Prozessen wurde Ende der 60er Jahre entwickelt. Damals dienten sie nicht zum Transport der Information in Netzwerken (die gab es noch gar nicht!), sondern der besseren Strukturierung von parallelen Programmen bzw. der Aufgaben innerhalb eines Betriebssystemes. Seit dem wurden viele verschiedene Mechanismen zum Nachrichtenaustausch vorgeschlagen, die im Prinzip alle gleich mächtig sind, sich jedoch hinsichtlich der Eleganz der Lösung bestimmter Kommunikationsaufgaben deutlich unterscheiden können.

5.1.1 Synchrone Kommunikation

Wir betrachten zunächst Befehle für den synchronen Punkt-zu-Punkt Nachrichtenaustausch. Dazu stehen die beiden Befehle

- **send**($dest - process$,$expr_1$,...,$expr_n$)
- **recv**($src - process$, var_1,..., var_n)

zur Verfügung.

Der Befehl **send** sendet eine Nachricht an den Prozess $dest - process$, die die Werte der Ausdrücke $expr_1$ bis $expr_n$ enthält. Der Prozess $dest - process$ empfängt diese Nachricht mit dem entsprechenden **recv**-Befehl und speichert die Werte der Ausdrücke in den Variablen var_1 bis var_n ab. Die Variablen müssen von passendem Typ sein.

Sowohl **send** als auch **recv** sind blockierend, d.h. werden erst beendet, wenn die Kommunikation stattgefunden hat. Die beteiligten Prozesse werden dadurch synchronisiert, daher der Name. Sende– und Empfangsprozess müssen ein passendes **send**/**recv**-Paar ausführen, sonst entsteht eine Verklemmung.

Abbildung 5.1(a) illustriert graphisch die Synchronisation zweier Prozesse durch blockierende Kommunikationsoperationen, und Teil (b) gibt ein Beispiel für eine Verklemmung.

Blockierende Punkt-zu-Punkt-Kommunikation ist nicht ausreichend um alle Aufgaben zu lösen. In einigen Situationen kann ein Prozess nicht wissen, welcher von mehreren möglichen Partnerprozessen als nächstes zu einem Datenaustausch bereit ist.

Eine Möglichkeit dieses Problem zu lösen besteht in der Bereitstellung zusätzlicher Funktionen, die es ermöglichen festzustellen, ob ein Partnerprozess bereit ist, eine Nachricht zu senden oder zu empfangen, ohne selbst zu blockieren:

- **int sprobe**($dest - process$)
- **int rprobe**($src - process$).

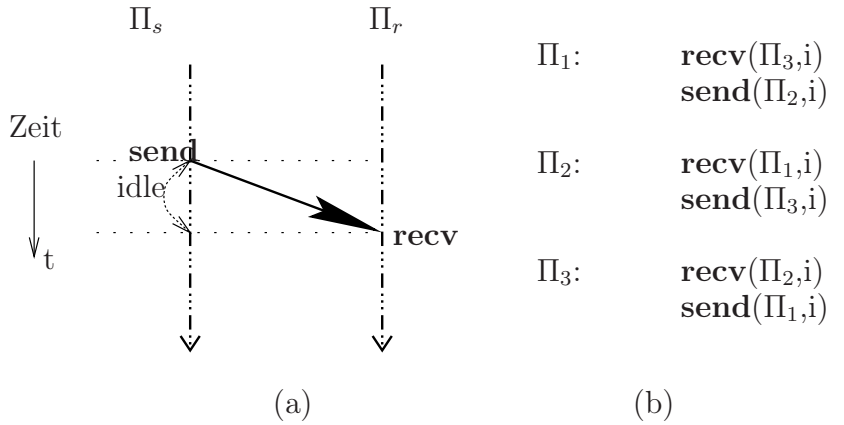


Abbildung 5.1: (a) Synchronisation zweier Prozesse durch ein **send**/**recv** Paar. (b) Beispiel für eine Verklemmung.

Die Funktion **sprobe** liefert 1 (*true*), falls der angegebene Prozess bereit ist, eine **recv**-Operation durchzuführen, d.h. ein **send**-Befehl würde *nicht* blockieren. Entsprechend testet **rprobe**, ob ein **recv**-Befehl nicht blockieren würde. Um ein Blockieren eines Prozesses zu vermeiden, würde man einem Kommunikationsbefehl eine entsprechende probe-Anweisung voranstellen:

- **if** (**sprobe**(Π_d)) **send**(Π_d,...);

In diesem Zusammenhang bezeichnet man die Befehle sprobe/rprobe auch als *Wächter* (*engl.* „guards“).

Die Implementierung von **rprobe** ist relativ einfach. Ein **send**-Befehl schickt die Nachricht (oder einen Teil davon) zum Empfängerprozessor, wo sie zwischengespeichert wird. Der Empfangsprozess braucht nun nur lokal nachzuschauen, ob eine entsprechende Nachricht (oder Teil davon) angekommen ist.

Eine Implementierung von **sprobe** ist aufwendiger, da normalerweise keine Information vom Empfängerprozess zum Zielprozess zurücklaufen muss. Glücklicherweise kommt man (im Prinzip) mit einem der Befehle **sprobe**/**rprobe** aus, weshalb man sich in der Praxis meist auf **rprobe** beschränkt.

Eine ähnliche Wirklung wie **rprobe** hat ein Befehl der Art

- **recv_any**(who,var_1,...,var_n)

der es erlaubt, eine Nachricht von einem beliebigen Prozess zu empfangen. Der Absender wird in der Variablen *who* abgespeichert.

5.1.2 Asynchrone Kommunikation

Die Befehle

- **asend**($dest-process$,$expr_1$,...,$expr_n$)
- **arecv**($src-process$,var_1,...,var_n)

haben die gleiche Semantik wie **send**/**recv** mit dem Unterschied, dass sie *nicht blockieren*. Ein Prozess kann nun (im Prinzip) beliebig viele **asend**-Befehle ausführen ohne zu verzögern. Die beteiligten Prozesse werden also *nicht* implizit synchronisiert. Man kann sich am Anfang des

Kommunikationskanals von Sendeprozess zu Empfangsprozess eine Warteschlange vorstellen, die alle Nachrichten zwischenpuffert, bis sie vom Empfangsprozess abgenommen werden.

Dies zeigt auch schon die Schwachstelle dieses Konzeptes: In der Praxis muss jeglicher Pufferplatz endlich sein und somit kann es zur Situation kommen, dass kein weiterer **asend**-Befehl mehr ausgeführt werden kann. In der *Praxis* muss man also testen können, ob ein früher ausgeführter Kommunikationsbefehl inzwischen bearbeitet werden konnte. Dazu liefern **asend**/**arecv** einen eindeutigen Identifikator zurück,

- **msgid asend**(...)
- **msgid arecv**(...)

mittels dem dann die Funktion

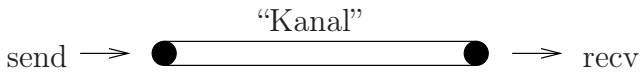
- **int** success(**msgid** m)

den Status des Kommunikationsauftrages liefert.

Wir werden auch erlauben, die Funktionen für asynchronen und synchronen Nachrichtenaustausch zu mischen. So ist es etwa sinnvoll, asynchrones Senden mit synchronem Empfangen zu mischen.

5.1.3 Virtuelle Kanäle

Die bisher vorgestellten Kommunikationsfunktionen werden als *verbindungslos* bezeichnet. Alternativ dazu kann man sich vorstellen, dass die Kommunikation über sogenannte „Kanäle“ erfolgt, wie in der folgenden Abbildung dargelegt:



Diese Kommunikationskanäle (*engl.* communication channels) können als gemeinsame Objekte verstanden werden. Bevor ein Kanal benutzt werden kann, muss er aufgebaut werden. Dies kann statisch (zur Ladezeit) oder dynamisch, während des Programmablaufes geschehen. Sende– und Empfangsbefehle operieren dann auf Kanälen:

- **send**($channel$,$expr_1$,...,$expr_n$)
- **recv**($channel$,var_1,...,var_n).

Eine mit der **recv_any**-Anweisung vergleichbare Funktionalität erhält man, wenn man erlaubt, dass mehrere Prozesse auf einem Kanal senden dürfen, aber nur einer empfängt. Wir werden in den folgenden Beispielen keinen Gebrauch von kanalorientierten Kommunikationsfunktionen machen.

5.1.4 Der MPI Standard

Das *Message Passing Interface*, oder *MPI*[1] ist eine portable Bibliothek von Funktionen zum Nachrichtenaustausch zwischen Prozessen. MPI wurde 1993-94 von einem internationalen Gre-

[1] `http://www-unix.mcs.anl.gov/mpi`

mium entwickelt und ist heute auf praktisch allen Plattformen verfügbar. Freie Implementierungen für LINUX Cluster (und weitere Rechner) sind *MPICH*[2] und *LAM*[3]. MPI hat die folgenden Merkmale:

- Bibliothek zum Binden mit C-, C++- und FORTRAN-Programmen (keine Spracherweiterung).
- Große Auswahl an Punkt-zu-Punkt Kommunikationsfunktionen.
- Globale Kommunikation.
- Datenkonversion für heterogene Systeme.
- Teilmengenbildung und Topologien.

MPI besteht aus über 125 Funktionen, die auf über 200 Seiten im Standard beschrieben werden. Daher können wir nur eine kleine Auswahl der Funktionalität betrachten.

Ein simples Beispiel

MPI ist eine Bibliothek, die zu einem Standard-C(C++/FORTRAN) Programm hinzugebunden wird. Ein erstes Beispiel in C zeigt folgender Code:

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"


int main (int argc, char *argv[])
{
    int my_rank;
    int P;
    int dest;
    int source;
    int tag=50;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&P);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    if (my_rank!=0)
    {
        sprintf(message,"I am process %d\n",my_rank);
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,
                 dest,tag,MPI_COMM_WORLD);
    }
```

[2] http://www-unix.mcs.anl.gov/mpi/mpich
[3] http://www.mpi.nd.edu/lam

```
    else
    {
        puts("I am process 0\n");
        for (source=1; source<P; source++)
        {
            MPI_Recv(message,100,MPI_CHAR,source,tag,
                     MPI_COMM_WORLD,&status);
            puts(message);
        }
    }

    MPI_Finalize();

    return 0;
}
```

Dieses Beispielprogram ist im SPMD-Stil geschrieben. Dies ist von MPI nicht zwingend vorgeschrieben, es erleichtert nur das Starten des parallelen Programmes etwas. Übersetzen, binden und ausführen des Programms unterscheiden sich von Implementierung zu Implementierung. Viele Implementierungen enthalten eine Reihe von Shell-Skripten, die den Installationsort der Bibliotheken verbergen. So benötigt man etwa in MPICH die Kommandos

```
mpicc -o hello hello.c
mpirun -machinefile machines -np 8 hello
```

um das Programm zu übersetzen und acht Prozesse zu starten. Dabei werden die Namen der zu benutzenden Rechner aus der Datei `machines` genommen.

Das Programm selbst erklärt sich fast von alleine. Die MPI-Funktionen und -Macros werden durch die Datei `mpi.h` zur Verfügung gestellt. Jedes MPI-Programm beginnt mit `MPI_Init` und endet mit `MPI_Finalize`.

Die Funktion `MPI_Comm_size` liefert die Anzahl der beteiligten Prozesse P und `MPI_Comm_rank` liefert die Nummer des Prozesses, in MPI „rank“ genannt. Die Nummer eines Prozesses ist zwischen 0 und $p-1$. Schließlich werden noch die Funktionen `MPI_Send` und `MPI_Recv` zum senden/empfangen von Nachrichten verwendet. Dabei handelt es sich um blockierende Kommunikationsfunktionen. Die genaue Bedeutung der Parameter wird unten besprochen.

Blockierende Kommunikation

MPI unterstützt verschiedene Varianten blockierender und nicht-blockierender Kommunikation, Wächter für die **receive**-Funktion, sowie Datenkonversion bei Kommunikation zwischen Rechnern mit unterschiedlichen Datenformaten.

Die grundlegenden blockierenden Kommunikationsfunktionen lauten:

```
int MPI_Send(void *message, int count, MPI_Datatype dt,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *message, int count, MPI_Datatype dt,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status);
```

Eine Nachricht in MPI besteht aus den eigentlichen *Daten* und einer Hülle (*engl.* envelope). Die Hülle besteht aus

1. Nummer des Senders
2. Nummer des Empfängers
3. Tag
4. Einem Communicator

Nummer (Rank) des Senders und Empfängers sind selbsterklärend. Das sog. Tag ist auch eine Integer-Zahl und dient der Kennzeichnung verschiedener Nachrichten zwischen identischen Kommunikationspartnern.

Ein Communicator ist gegeben durch eine Teilmenge der Prozesse und einen Kommunikationskontext. Nachrichten, die zu verschiedenen Kontexten gehören, beeinflussen einander nicht, bzw. Sender und Empfänger müssen den selben Communicator verwenden. Wir werden das Konzept des Communicator später noch genauer erläutern. Für den Moment genügt es, dass MPI den Communicator `MPI_COMM_WORLD` vordefiniert, der aus allen Prozessen besteht, die gestartet wurden.

Betrachten wir den Befehl `MPI_Send` näher. Die ersten drei Argumente `message`, `count`, und `dt`, beschreiben die zu sendenden Daten. Dabei zeigt `message` auf einen zusammenhängenden Speicherbereich der `count` Elemente des Datentyps `dt` enthält. Die Kenntnis des Datentyps gibt MPI die Gelegenheit zur Datenkonversion. Die Argumente `dest`, `tag` und `comm` bilden die Hülle der Nachricht (wobei die Nummer des Senders implizit durch den Aufruf gegeben ist).

`MPI_Send` ist grundsätzlich eine blockierende (synchrone) Sendefunktion. Dabei gibt es verschiedene Varianten:

- *buffered send* (`B`): Falls der Empfänger noch keine korrespondierende **recv**-Funktion ausgeführt hat, wird die Nachricht auf Senderseite gepuffer. Ein „buffered send“ wird, genügend Pufferplatz vorausgesetzt, immer sofort beendet. Im Unterschied zur asynchronen Kommunikation kann der Sendepuffer `message` sofort wiederverwendet werden.
- *synchronous send* (`S`): Ende des synchronous send zeigt an, dass der Empfänger eine **recv**-Funktion ausführt und begonnen hat, die Daten zu lesen.
- *ready send* (`R`): Ein ready send darf nur ausgeführt werden, falls der Empfänger bereits das korresponierende **recv** ausgeführt hat. Ansonsten führt der Aufruf zum Fehler.

Die entsprechenden Aufrufe zu diesen verschiedenen Varianten lauten `MPI_Bsend`, `MPI_Ssend` und `MPI_Rsend`. Der `MPI_Send`-Befehl hat entweder die Semantik von `MPI_Bsend` oder die von `MPI_Ssend`, je nach Implementierung. `MPI_Send` kann also, muss aber nicht blockieren. In jedem Fall kann der Sendepuffer `message` sofort nach beenden wieder verwendet werden.

Der Befehl `MPI_Recv` ist in jedem Fall blockierend, d.h. wird erst beendet, wenn `message` die Daten enthält. Das Argument `status` enthält Quelle, Tag, und Fehlerstatus der empfangenen Nachricht. Für die Argumente `src` und `tag` können die Werte `MPI_ANY_SOURCE` bzw. `MPI_ANY_TAG` eingesetzt werden. Somit beinhaltet `MPI_Recv` die Funktionalität von **recv_any**.

Eine nicht-blockierende Wächterfunktion für das Empfangen von Nachrichten steht mittels

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status);
```

zur Verfügung. Diese Funktion setzt `flag` auf `true` ($\neq 0$), falls eine Nachricht mit passender Quelle und Tag empfangen werden kann. Die Argumente `MPI_ANY_SOURCE` und `MPI_ANY_TAG` sind möglich.

Datenkonversion

MPI Implementierungen für heterogene Systeme sind in der Lage, eine automatische Konversion der Datenrepräsentation durchzuführen. Wie dies geschieht, ist Sache der Implementierung (z.B. durch XDR). Der (MPI-) Datentyp `MPI_BYTE` erfährt in jedem Fall *keine* Konversion.

Nichblockierende Kommunikation

Hierfür stehen die Funktionen

```
int MPI_ISend(void *buf, int count, MPI_Datatype dt,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *req);
int MPI_IRecv(void *buf, int count, MPI_Datatype dt,
              int src, int tag, MPI_Comm comm,
              MPI_Request *req);
```

zur Verfügung. Sie imitieren nur eine entsprechende Kommunikationsoperation. Mittels der `MPI_Request`-Objekte ist es möglich, den Zustand des Kommunikationsauftrages zu ermitteln (entspricht unserer **msgid**). Dazu gibt es (unter anderem) die Funktion

```
int MPI_Test(MPI_Request *req,int *flag, MPI_Status *status);
```

Das `flag` wird auf `true` ($\neq 0$) gesetzt, falls die durch `req` bezeichnete Kommunikationsoperation abgeschlossen ist. In diesem Fall enthält `status` Angaben über Sender, Empfänger und Fehlerstatus.

Zu beachten ist dabei, dass das `MPI_Request`-Objekt ungültig wird, sobald `MPI_Test` mit `flag==true` zurückkehrt. Es darf dann nicht mehr verwedet werden. Der Grund dafür ist, dass die `MPI_Request`-Objekte von der MPI-Implementierung verwaltet werden (sog. opaque objects).

Globale Kommunikation

MPI bietet verschiedene Funktionen zur globalen Kommunikation an der alle Prozesse eines Communicator (siehe nächster Abschnitt) teilnehmen. So blockiert

```
int MPI_Barrier(MPI_Comm comm);
```

alle Prozesse, bis alle angekommen sind (d.h. diese Funktion ausgeführt haben). Die Funktion

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dt,
              int root, MPI_Comm comm);
```

verteilt die Nachricht in Prozess `root` an alle anderen Prozesse des Communicator.

Für das Einsammeln von Daten stehen verschiedene Operationen zur Verfügung. Wir beschreiben nur eine davon:

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
       MPI_Datatype dt, MPI_Op op, int root, MPI_Comm comm);
```

kombiniert die Daten im Eingangspuffer `sbuf` aller Prozesse mittels der assoziativen Operation `op`. Das Endergebnis steht im Empfangspuffer `rbuf` des Prozesses `root` zur Verfügung. Beispiele für `op` sind `MPI_SUM`, `MPI_MAX` und `MPI_MIN`, womit sich Summen, Maxima und Minima über alle Prozesse berechnen lassen.

Bemerkung 5.1 Globale Kommunikationen müssen von allen Prozessen eines Communicators mit passenden Argumenten (z.B. `root` in `MPI_Reduce` aufgerufen werden!

Communicators und Topologien

In allen bisher betrachteten MPI Kommunikationsfunktionen trat ein Argument vom Typ `MPI_Comm` auf. Ein solcher *Communicator* beinhaltet die folgenden Abstraktionen:

- *Prozessgruppe:* Ein Communicator kann benutzt werden, um eine Teilmenge aller Prozesse zu bilden. Nur diese nehmen dann etwa an einer globalen Kommunikation teil. Der Vordefinierte Communicator `MPI_COMM_WORLD` besteht aus allen gestarteten Prozessen.
- *Kontext:* Jeder Communicator definiert einen eigenen Kommunikationskontext. Nachrichten können nur innerhalb des selben Kontextes empfangen werden, in dem sie abgeschickt wurden. So kann etwa eine Bibliothek numerischer Funktionen einen eigenen Communicator verwenden. Nachrichten der Bibliothek sind dann vollkommen von Nachrichten im Benutzerprogramm abgeschottet, und Nachrichten der Bibliothek können nicht fälschlicherweise vom Benutzerprogramm empfangen werden und umgekehrt.
- *Virtuelle Topologien:* Ein Communicator steht nur für eine Menge von Prozessen $\{0, \ldots, P-1\}$. Optional kann man diese Menge mit einer zusätzlichen Struktur, etwa einem mehrdimensionalen Feld oder einem allgemeinen Graphen, versehen.
- *Zusätzliche Attribute:* Eine Anwendung (z.B. eine Bibliothek) kann mit einem Communicator beliebige statische Daten assoziieren. Der Communicator dient dann als Vehikel, um diese Daten von einem Aufruf der Bibliothek zum nächsten hinüberzuretten.

Der so beschriebene Communicator stellt einen sogenannten *Intra-Communicator* dar. Dieser ist dadurch gekennzeichnet, dass er nur Kommunikation *innerhalb* einer Prozessgruppe erlaubt. Für die Kommunikation zwischen *verschiedenen* Prozessgruppen gibt es sogenannte *Inter-Communicators*, auf die wir nicht näher eingehen wollen.

Als eine Möglichkeit zur Bildung neuer (Intra-) Communicators stellen wir die Funktion

```
int MPI_Comm_split(MPI_Comm comm, int color,
                   int key, MPI_Comm *newcomm);
```

vor. `MPI_Comm_split` ist eine kollektive Operation, die von *allen* Prozessen des Communicators `comm` gerufen werden muss. Alle Prozesse mit gleichem Wert für das Argument `color` bilden jeweils einen neuen Communicator. Die Reihenfolge (rank) innerhalb des neuen Communicator wird durch das Argument `key` geregelt.

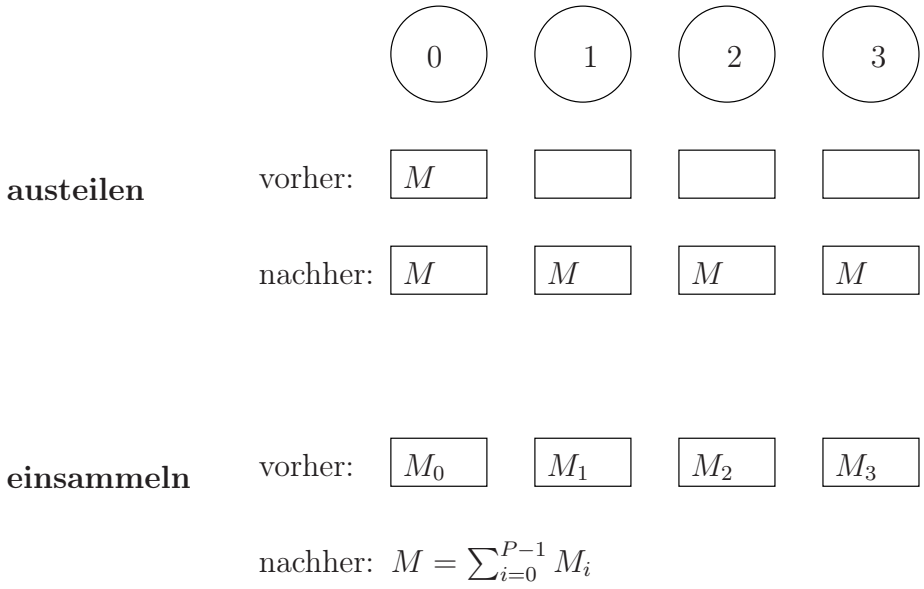
5.2 Globale Kommunikation

In diesem Abschnitt behandeln wir verschiedene Situationen in denen *alle* Prozesse Daten miteinander austauschen müssen. Optimale Lösung dieser Probleme setzt immer eine hierarchische Kommunikationstopologie voraus.

5.2.1 Einer an alle (und alle an einen zusammenfassen)

Wir betrachen das Problem, dass ein Prozess ein identisches Datum an alle anderen Prozesse versenden muss (*engl.* „one-to-all broadcast“ oder „mingle-node-broadcast“).

Die dazu duale Operation ist das Zusammenfassen von individuellen Resultaten auf einem Prozess, z.B. Summen- oder Maximumbildung (alle assoziativen Operatoren sind möglich).



Wir betrachten das Austeilen auf verschiedenen Topologien (Ring, Feld, Hypercube) und berechnen den notwendigen Zeitbedarf im Falle von store & forward routing. Anschließend wenden wir uns den Cut-through Netzwerken zu.

Wir geben nur Algorithmen für das Austeilen an, solche für Einsammeln ergeben sich durch umdrehen der Reihenfolge und Richtung der Kommunikationen.

Ring mit store & forward routing

Auf dem Ring nutzen wir die Kommunikation in beiden Richtungen. Alle Prozesse kleiner gleich $P/2$ werden von kleineren Nummern her versorgt, die anderen von größeren Nummern her (s.Abb. 5.2).

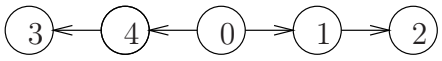


Abbildung 5.2: Einer an alle im Ring

Programm 5.2 (Einer-an-alle auf dem Ring)

```
parallel one-to-all-ring
{
    const int P;

    process Π[int p ∈ {0, ..., P − 1}]
    {
        void one_to_all_broadcast(msg *mptr)
        {
            // Nachrichten empfangen
            if (p > 0  ∧  p ≤ P/2)   recv(Π_{p−1}, *mptr);
            if (p > P/2)             recv(Π_{(p+1)%P}, *mptr);
```

```
            // Nachrichten an Nachfolger weitergeben
            if (p ≤ P/2 − 1)              send(Π_{p+1}, *mptr);
            if (p > P/2 + 1 ∨ p == 0)     send(Π_{(p+P−1)%P}, *mptr);
        }
        ...;
        m=...;
        one_to_all_broadcast(&m);
    }
}
```

Der Zeitbedarf für die Operation beträgt

$$T_{one-to-all-ring} = (t_s + t_h + t_w \cdot n) \left\lceil \frac{P}{2} \right\rceil ,$$

wobei $n = |*mptr|$ immer die Länge der Nachricht bezeichne.

2D-Feld mit store & forward routing

Auf einer 2D-Feldtopologie (ohne Rückverbindungen) würde man wie in Abb. 5.3 vorgehen. Wichtig ist es, die Nachricht erst in der ersten Spalte nach oben zu schicken. Das Programm

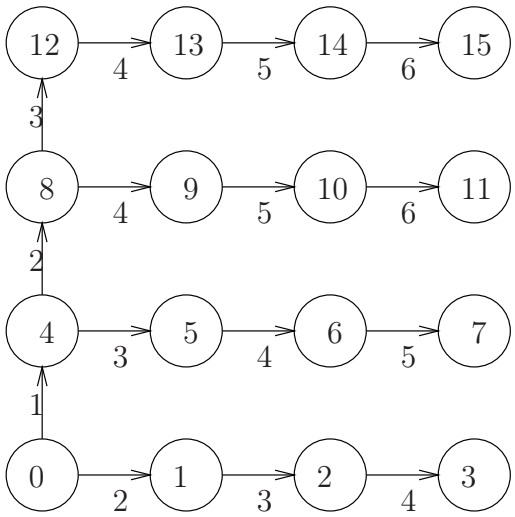


Abbildung 5.3: Einer an alle im Feld

lautet:

Programm 5.3 (Einer an alle auf dem Feld)

```
parallel one-to-all-array
{
    int P, Q;                          // Feldgröße in x- und y-Richtung
    process Π[int[2] (p, q) ∈ {0, ..., P − 1} × {0, ..., Q − 1}]
    {
        void one_to_all_broadcast(msg *mptr)
        {
```

```
        if (p == 0)
        {                          // erste Spalte
            if (q > 0)      recv(Π_(p,q-1),*mptr);
            if (q < Q - 1) send(Π_(p,q+1),*mptr);
        }
        else            recv(Π_(p-1,q),*mptr);
        if (p < P - 1) send(Π_(p+1,q),*mptr);
    }

    msg m=...;
    one_to_all_broadcast(&m);
  }
}
```

Die Ausführungszeit beträgt für ein 2D-Feld

$$T_{one-to-all-array2D} = 2(t_s + t_h + t_w \cdot n)(\sqrt{P} - 1)$$

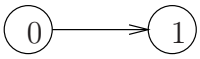
und für ein 3D-Feld

$$T_{one-to-all-array3D} = 3(t_s + t_h + t_w \cdot n)(P^{1/3} - 1),$$

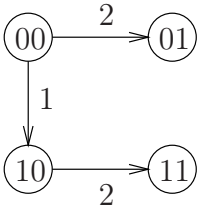
wobei wir ein Feld mit gleicher Prozessorzahl in jede Richtung und insgesamt P Prozessoren angenommen haben.

Hypercube mit store & forward routing

Auf einem Hypercube der Dimension $d = 1$ ist das Problem trivial zu lösen:



Auf einem Hypercube der Dimension $d = 2$ schickt 0 erst an 2 und das Problem ist auf 2 Hypercubes der Dimension 1 reduziert:



Allgemein schicken im Schritt $k = 0, \ldots d-1$ die Prozesse

$$\text{je eine Nachricht an } \underbrace{p_{d-1} \ldots p_{d-k}}_{k \text{ Dimens.}} \; 0 \; \underbrace{0 \ldots 0}_{d-k-1 \text{ Dimens.}}$$
$$\text{je eine Nachricht an } \underbrace{p_{d-1} \ldots p_{d-k}}_{k \text{ Dimens.}} \; 1 \; \underbrace{0 \ldots 0}_{d-k-1 \text{ Dimens}}$$

Es folgt ein Programm.[4]

[4] $\oplus$=`xor`

Programm 5.4 (Einer an alle auf dem Hypercube)

```
parallel one-to-all-hypercube
{
    int d, P = 2^d;

    process Π[int p ∈ {0,...,P−1}]
    {

        void one_to_all_broadcast(msg *mptr)
        {
            int i, mask = 2^d − 1;
            for (i = d − 1; i ≥ 0; i − −)
            {
                mask = mask ⊕ 2^i;
                if (p&mask == 0)
                {
                    if (p&2^i == 0)          //die letzten i Bits sind 0
                        send(Π_{p⊕2^i},*mptr);
                    else
                        recv(Π_{p⊕2^i},*mptr);
                }
            }
        }

        msg m = „bla“;
        one_to_all_broadcast(&m);
    }
}
```

Der Zeitbedarf ist

$$T_{one-to-all-HC} = (t_s + t_h + t_w \cdot n) \operatorname{ld} P$$

Als Variante können wir noch den Fall einer beliebigen Quelle $src \in \{0, \ldots, P-1\}$ betrachten, die ein Datum an alle anderen Prozesse schicken will. Der Hypercube-Algorithmus lässt sich dafür leicht modifizieren, indem wir alle Prozesse (virtuell) umnummerieren:

$$p^{neu} = p \oplus src.$$

Nun gilt $p^{neu} = 0 \iff p = src$ und in der Funktion *one_to_all_broadcast* von Programm 5.4 sind einfach alle Vorkommen von p durch p^{neu} zu ersetzen.

Ring mit cut-through routing

Hier verwendet man vorteilhafterweise den Hypercubealgorithmus auf dem Ring. Dies sieht für $P = 8$ so aus wie in Abb. 5.4. Wie man sieht, werden keine Kommunikationsleitungen doppelt verwendet. Für die Laufzeit erhalten wir:

$$\begin{aligned} T_{one-to-all-ring-ct} &= \sum_{i=0}^{\operatorname{ld} P - 1} (t_s + t_w \cdot n + t_h \cdot 2^i) = \\ &= (t_s + t_w \cdot n) \operatorname{ld} P + t_h (P - 1) \end{aligned}$$

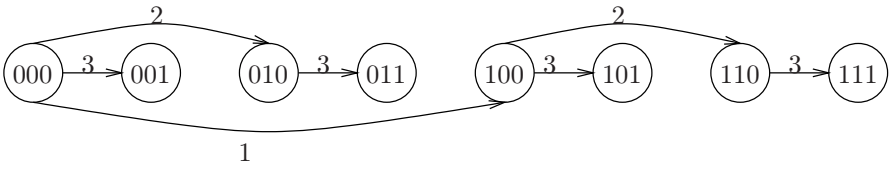


Abbildung 5.4: Hypercube-Baum auf Ring mit cut-throug routing

Der Term mit t_h ist in der Praxis vernachlässigbar, und wir erhalten Hypercube-Perfomance auf dem Ring!

Feld mit cut-through routing

Auch hier kann man den Hypercubealgorithmus vorteilhaft verwenden. Dazu verwendet man eine Nummerierung der Prozesse wie in Abb. 5.5. Wieder kommt man ohne Leitungskonflikte

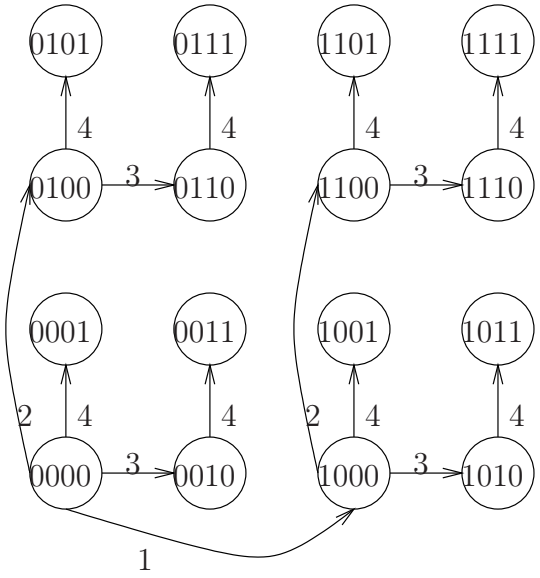


Abbildung 5.5: Prinzip der Hypercube-Baumes auf dem Feld

aus, und die Laufzeit beträgt:

$$
\begin{aligned}
T_{one-to-all-field-ct} &= \underbrace{2}_{\substack{\text{jede}\\ \text{Entfernung}\\ \text{2 mal}}} \sum_{i=0}^{\frac{\text{ld}\,P}{2}-1} (t_s + t_w \cdot n + t_h \cdot 2^i) = \\
&= (t_s + t_w \cdot n) 2 \frac{\text{ld}\,P}{2} + t_h \cdot 2 \underbrace{\sum_{i=0}^{\frac{\text{ld}\,P}{2}-1} 2^i}_{=\underbrace{2^{\frac{\text{ld}\,P}{2}}}_{=\sqrt{P}}-1} = \\
&= \text{ld}\,P(t_s + t_w \cdot n) + t_h \cdot 2(\sqrt{P} - 1)
\end{aligned}
$$

Selbst für $P = 1024 = 32 \times 32$ ist der Term mit t_h vernachlässigbar. Dies erklärt, warum man mit cut-through routing keine physikalische Hypercube-Struktur mehr benötigt.

5.2.2 Alle an alle austeilen

Jeder Prozess sendet ein Datum an alle anderen Prozesse (wie in einer an alle mit beliebiger Quelle). Unterschiedliche Prozesse senden unterschiedliche Daten, und am Ende soll jeder alle Nachrichten kennen (Abb. 5.6). In einer Variante mit Akkumulieren wird ein assoziativer Ope-

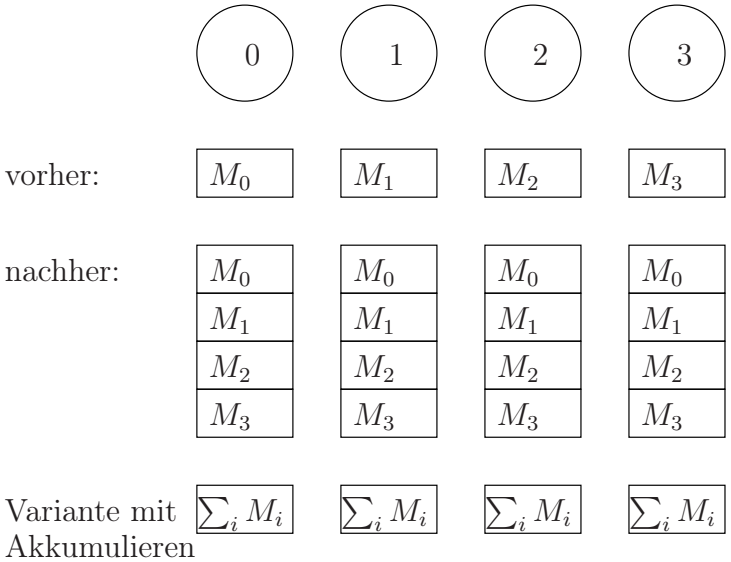


Abbildung 5.6: Alle an alle

rator (Summe, Max, Min,...) auf alle Daten angewandt, und am Ende soll jeder das Ergebnis kennen (im Gegensatz zu alle-an-einen).

Ring mit store & forward routing

Alle Prozesse schieben das Datum, das sie zuletzt erhalten haben, zyklisch im Ring weiter (Abb. 5.7)

Programm 5.5 (Alle an alle Austeilen auf dem Ring)

```
parallel all-to-all-ring
{
    const int P;
    process Π[int p ∈ {0,...,P − 1}]
    {
        void all_to_all_broadcast(msg m[P])
        {
            int i,
                item = p;                    // M[item] wird geschickt/empfangen
            msgid ids, idr;
            for (i = 0; i < P; i++)          // P − 1 mal schieben
            {
```

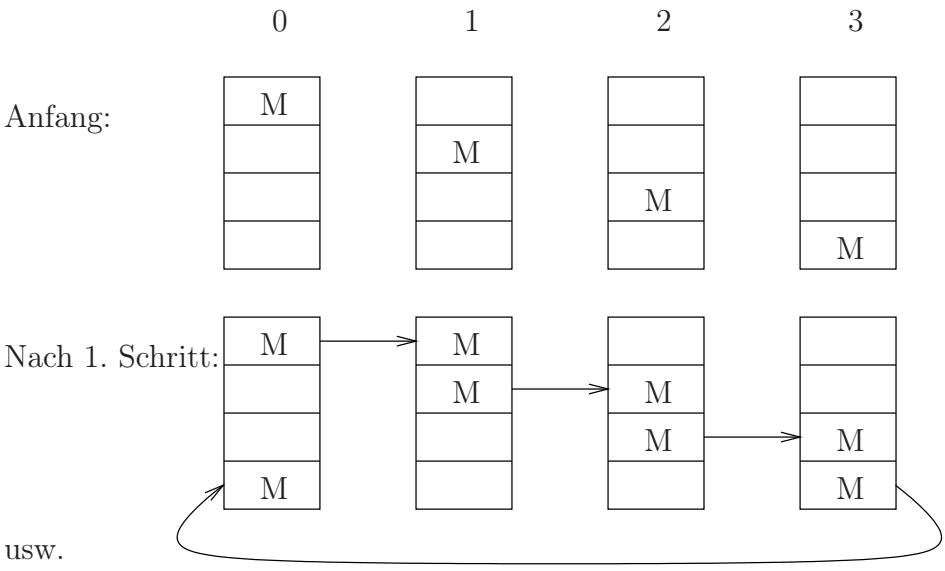


Abbildung 5.7: Alle an alle im Ring

```
            ids =asend(Π(p+1)%P,m[item]);// an nächsten
            item = (item + P − 1)%P;        // zähle rückwärts
            idr =arecv(Π(p+P−1)%P,m[item]);
            while(¬success(ids));
            while(¬success(idr));
        }
    }
    ...
    m[p] =„Das ist von p!“;
    all_to_all_broadcast(m);
    ...
  }
}
```

Bemerkung 5.6 Hier wird asynchrone Kommunikation verwendet, um das Schieben parallel durchzuführen.

Der Zeitbedarf berechnet sich zu

$$T_{all-to-all-ring-sf} = \underbrace{2}_{\substack{\text{send u.}\\ \text{recv in}\\ \text{jedem}\\ \text{Schritt}}} (t_s + t_h + t_w \cdot n)(P-1).$$

1D-Feld mit store & forward routing

Nun betrachten wir den Ring ohne die Rückverbindung. In diesem Fall sendet jeder Prozess seine Daten nach links und rechts (Abb. 5.8):

Programm 5.7 (Alle an alle auf 1D-Feld)

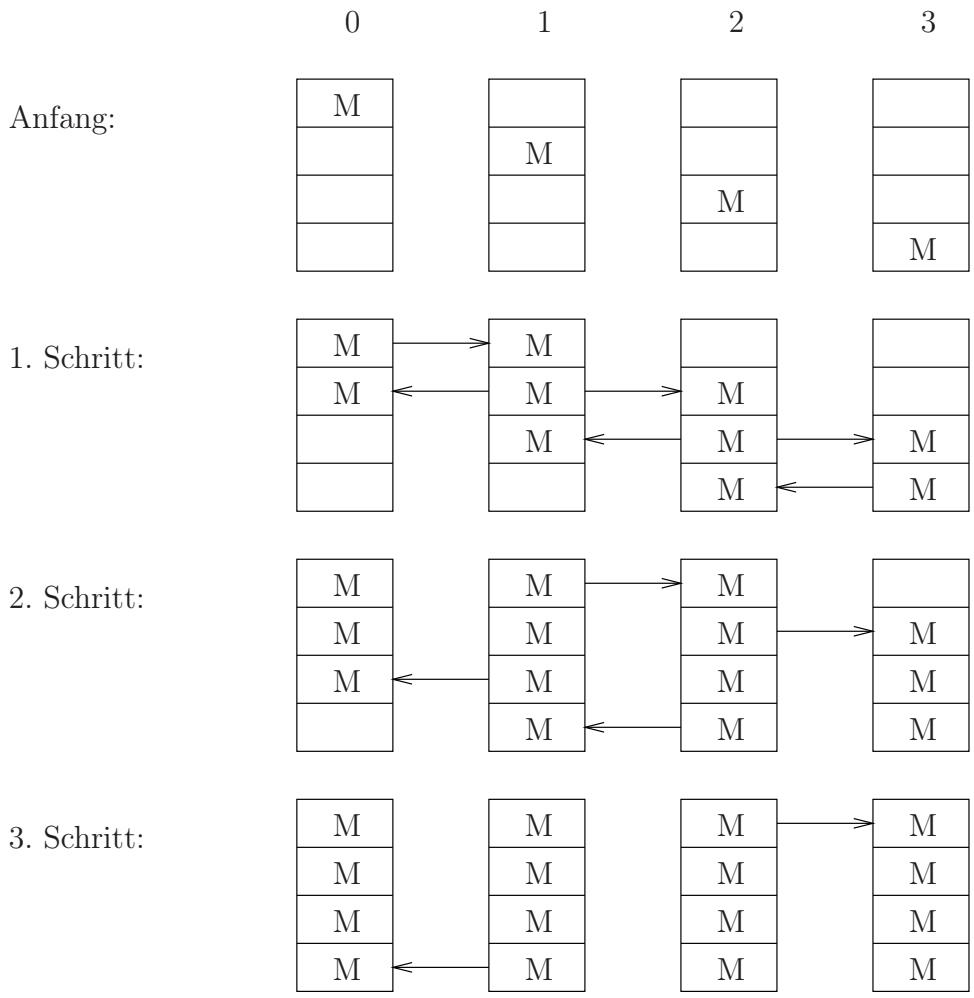


Abbildung 5.8: Alle an alle im 1D-Feld mit store & forward routing

```
parallel all-to-all-1D-feld
{
    const int P;
    process Π[int p ∈ {0, ..., P − 1}]
    {
        void all_to_all_broadcast(msg m[P])
        {
            int i,
                from_left= p − 1, from_right= p + 1,
                                                  // das empfange ich
                to_left= p, to_right= p;          // das verschicke ich
            for (i = 1; i < P; i++)               // P − 1 Schritte
            {
                if ((p%2) == 1)                   // schwarz/weiss Färbung
                {
                    if (from_left ≥ 0) recv(Π_{p−1}, m[from_left]);
                    if (to_right ≥ 0) send(Π_{p+1}, m[to_right]);
                    if (from_right < P) recv(Π_{p+1}, m[from_right]);
                    if (to_left < P) send(Π_{p−1}, m[to_left]);
                }
                else
                {
                    if (to_right ≥ 0) send(Π_{p+1}, m[to_right]);
                    if (from_left ≥ 0) recv(Π_{p−1}, m[from_left]);
                    if (to_left < P) send(Π_{p−1},m[to_left]);
                    if (from_right < P) recv(Π_{p+1}, m[from_right]);
                }
                from_left−−; to_right−−;
                from_right++; to_left++;
            }
        }
        ...
        m[p] =„Das ist von p!“;
        all_to_all_broadcast(m);
        ...
    }
}
```

Für die Laufzeitanalyse betrachte P ungerade, $P = 2k + 1$:

$$\underbrace{\Pi_0, \ldots, \Pi_{k-1},}_{k}\ \Pi_k,\ \underbrace{\Pi_{k+1}, \ldots, \Pi_{2k}}_{k}$$

Prozess Π_k	empfängt	k	von links
	schickt	$k+1$	nach rechts
	empfängt	k	von rechts
	schickt	$k+1$	nach links.
	$\sum =$	$4k+2$	
		$= 2P$	

Danach hat Π_k alle Nachrichten. Nun muss die Nachricht von 0 noch zu $2k$ und umgedreht. Dies dauert nochmal

$$(\underbrace{k}_{\text{Entfernung}} - 1) \cdot \underbrace{2}_{\substack{\text{senden u.}\\ \text{empfangen}}} + \underbrace{1}_{\substack{\text{der Letzte}\\ \text{empfängt nur}}} = 2k - 1 = P - 2$$

also haben wir

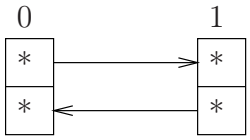
$$T_{all-to-all-array-1d} = (t_s + t_h + t_w \cdot n)(3P - 2)$$

also 50% mehr als der Ring.

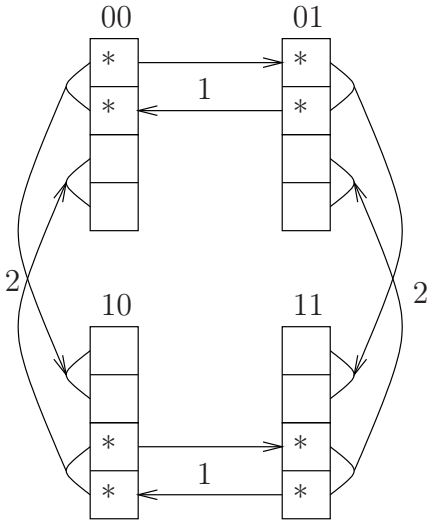
Hypercube

Der folgende Algorithmus für den Hypercube ist als *Dimensionsaustausch* bekannt.

Beginne mit $d = 1$



Bei vier Prozessen tauschen erst 00 und 01 bzw 10 und 11 ihre Daten aus, dann tauschen 00 und 10 bzw 01 und 11 jeweils zwei Informationen aus:



Allgemein kennt nach dem i-ten Schritt ein Prozess p die Daten aller Prozesse q, die sich genau in den letzten i Bitstellen von ihm unterscheiden.

Programm 5.8 (Alle an Alle auf dem Hypercube)

```
parallel all-to-all-hypercube
{
    const int d, P = 2^d;
    process Π[int p ∈ {0, ..., P − 1}]
    {
        void all_to_all_broadcast(msg m[P]) {
            int i, mask = 2^d − 1, q;
```

```
        for (i = 0; i < d; i++) {
            q = p ⊕ 2^i;
            if (p < q) {                    // wer zuerst?
                send(Π_q,m[p&mask],...,m[p&mask + 2^i − 1]);
                recv(Π_q,m[q&mask],...,m[q&mask + 2^i − 1]);
            }
            else {
                recv(Π_q,m[q&mask],...,m[q&mask + 2^i − 1]);
                send(Π_q,m[p&mask],...,m[p&mask + 2^i − 1]);
            }
            mask = mask ⊕ 2^i;
        }
    }
    ...
    m[p] =„Das ist von mir!“;
    all_to_all_broadcast(m);
    ...
  }
}
```

Laufzeitanalyse:

$$\begin{aligned} T_{all-to-all-bc-hc} &= \underbrace{2}_{\substack{\text{send u.}\\ \text{receive}}} \sum_{i=0}^{\operatorname{ld} P-1} t_s + t_h + t_w \cdot n \cdot 2^i = \\ &= 2 \operatorname{ld} P(t_s + t_h) + 2t_w n(P-1). \end{aligned}$$

Für grosse Nachrichtenlänge n ist der Hypercube also nicht besser als der Ring! Warum? Ein Prozess muss ein individuelles Datum von *jedem* anderen Prozess empfangen. Dies dauert immer mindestens $t_w n(P-1)$, unabhängig von der Topologie!

Bemerkung 5.9 Summe bilden (akkumulieren) geht in Zeit

$$T_{all-to-all-sum-hc} = 2 \sum_{i=0}^{\operatorname{ld} P-1} t_s + t_h + t_w n \cdot 1 = 2(t_s + t_h + nt_w) \operatorname{ld} P.$$

Bemerkung 5.10 Bei cut-through routing auf dem Ring wird durch Abbilden des HC-Algorithmus keine Einsparung erzielt, da Leitungskonflikte auftreten (also auch nicht im Term mit $t_s + t_h$!).

5.2.3 Einer an alle (alle an einen) mit individuellen Nachrichten

Nun betrachten wir das Problem, dass Prozess 0 an jeden Prozess eine individuelle Nachricht schicken muss (Abb. 5.9).

In der dualen Version „alle an einen individuell“ schickt jeder Prozess seine individuelle Nachricht an Prozess 0. Vorher und Nachher sind also im obigen Bild zu vertauschen. (Dieser Teil ist auch in „alle an alle“ enthalten.)

Eine *Anwendung* ist die Ein-/Ausgabe für parallele Anwendungen.

Wir betrachten einen Algorithmus für den Hypercube, der nur jeweils eine Nachricht zwischenpuffert. Er benutzt Pipelining und kann durch Paketierung leicht auf beliebig lange Nachrichten erweitert werden.

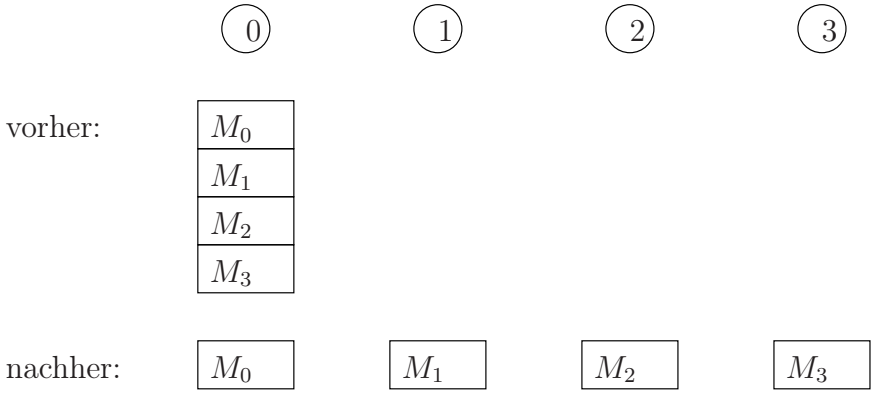


Abbildung 5.9: 0 schickt an jeden Prozess seine persönliche Nachricht (für *alle an einen* vertausche man „vorher“ und „nachher“)

Wie benötigen dazu eine Baumstruktur als Teil des Hypercube, wie in Abb. 5.10.

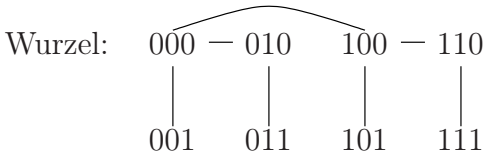


Abbildung 5.10: Altbekannte Baumstruktur auf dem Hypercube

Programm 5.11 (*Einsammeln* persönlicher Nachrichten (HC))

```
parallel all-to-one-personalized
{
    const int d, P = 2^d;
    process Π[int p ∈ {0, ..., P − 1}]
    {
        void all_to_one_pers(msg m)
        {
            int mask, i, q, root;
            // bestimme Papa-Knoten des Baumes (p's Wurzel):
            mask = 2^d − 1;
            for (i = 0; i < d; i++)
            {
                mask = mask ⊕ 2^i;
                if (p&mask ≠ p) break;
            } // p = p_{d−1} ... p_{i+1}  1 (zuletzt 0 gesetzt in mask)  0...0 (i−1,...,0)
            if (i < d) root = p ⊕ 2^i;          // meine Wurzelrichtung

            // eigene Daten
            if (p == 0) selber-verarbeiten(m);
```

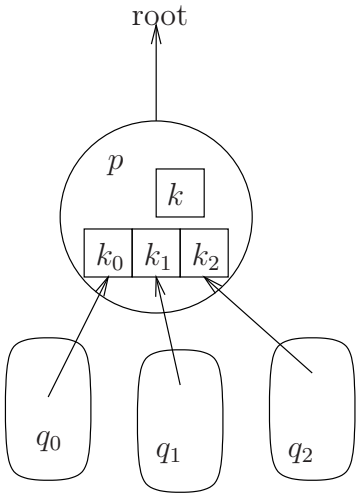


Abbildung 5.11: Sortierte Ausgabe individueller Daten.

```
            else send(Π_root,m);                    // hochgeben

            // arbeite Unterbäume ab:
            mask = 2^d − 1;
            for (i = 0; i < d; i++)
            {
                mask = mask ⊕ 2^i; q = p ⊕ 2^i;
                if (p&mask == p)
                                                    p =  p_{d−1} ... p_{i+1}  0  0...0 (i−1,...,0),
                {                                   //
                                                    q =  p_{d−1} ... p_{i+1}  1  0...0 (i−1,...,0)
                                                    // ⇒ Wurzel HC Dim. i + 1!
                    for (k = 0; k < 2^i; k++)   // Daten aus anderer Hälfte
                    {
                        recv(Π_q,m);
                        if (p == 0) verarbeite(m);
                        else send(Π_root,m);
                    }
                }
            }
        }
        ...
    }
}
```

Für die *Laufzeit* hat man für grosse (n) Nachrichten

$$T_{all-to-one-pers} \geq t_w n(P-1)$$

wegen dem Pipelining.

Einige Varianten von Programm 5.11 sind denkbar:

- *Individuelle Länge der Nachricht:* Hier sendet man vor verschicken der eigentlichen Nachricht nur die Längeninformation. Ist aber Käse, weil wir das sowieso ohne Längenangaben formulieren.
- *beliebig lange Nachricht* (aber nur endlicher Zwischenpuffer!): zerlege Nachrichten in Pakete fester Länge (s.o.).
- *sortierte Ausgabe:* Es sei jeder Nachricht M_i (von Prozess i) ein Sortierschlüssel k_i zugeordnet. Die Nachrichten sollen von Prozess 0 in aufsteigender Reihenfolge der Schlüssel verarbeitet werden, *ohne* dass alle Nachrichten zwischengepuffert werden.

  Dabei folgt man der folgenden Idee: p habe drei „Untergebene", q_0, q_1, q_2, die für ganze Unterbäume stehen. Jeder q_i sendet seinen nächst kleinsten Schlüssel an p, der den kleinsten Schlüssel raussucht und ihn seinerseits weitergibt. Abbildung 5.11 illustriert dies. Die Implementierung sei dem geneigten Leser überlassen.

5.2.4 Alle an alle mit individuellen Nachrichten

Hier hat jeder Prozess $P-1$ Nachrichten, je eine für jeden anderen Prozess. Es sind also $(P-1)^2$ individuelle Nachrichten zu verschlicken (s.Abb. 5.12). Das Bild zeigt auch schon eine Anwen-

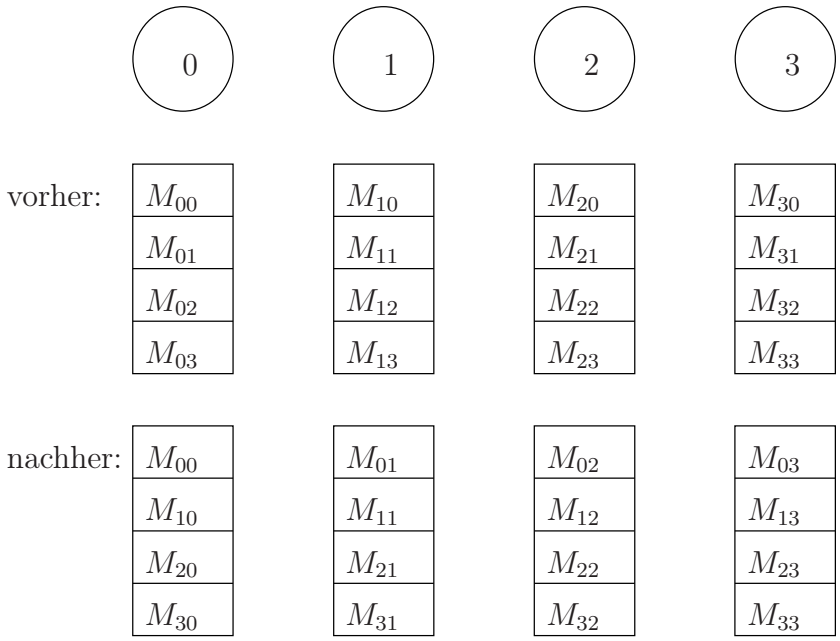


Abbildung 5.12: Alle an alle mit individuellen Nachrichten

dung: Matrixtransposition bei spaltenweiser Aufteilung.

Wir wollen wieder einen Algorithmus für den Hypercube angeben. Im Schritt i kommuniziert p mit $q = p \oplus 2^i$ – wie immer. Die Abb. 5.13 zeigt die Idee für $P = 4$.

Allgemein haben wir folgende Situation im Schritt $i = 0, \ldots, d-1$: Prozess p kommuniziert mit $q = p \oplus 2^i$ und sendet ihm

$$\begin{array}{rcll} \text{alle Daten der Prozesse } p_{d-1} \ldots p_{i+1} & p_i & x_{i-1} \ldots x_0 \\ \text{für die Prozesse } y_{d-1} \ldots y_{i+1} & \overline{p_i} & p_{i-1} \ldots p_0, \end{array}$$

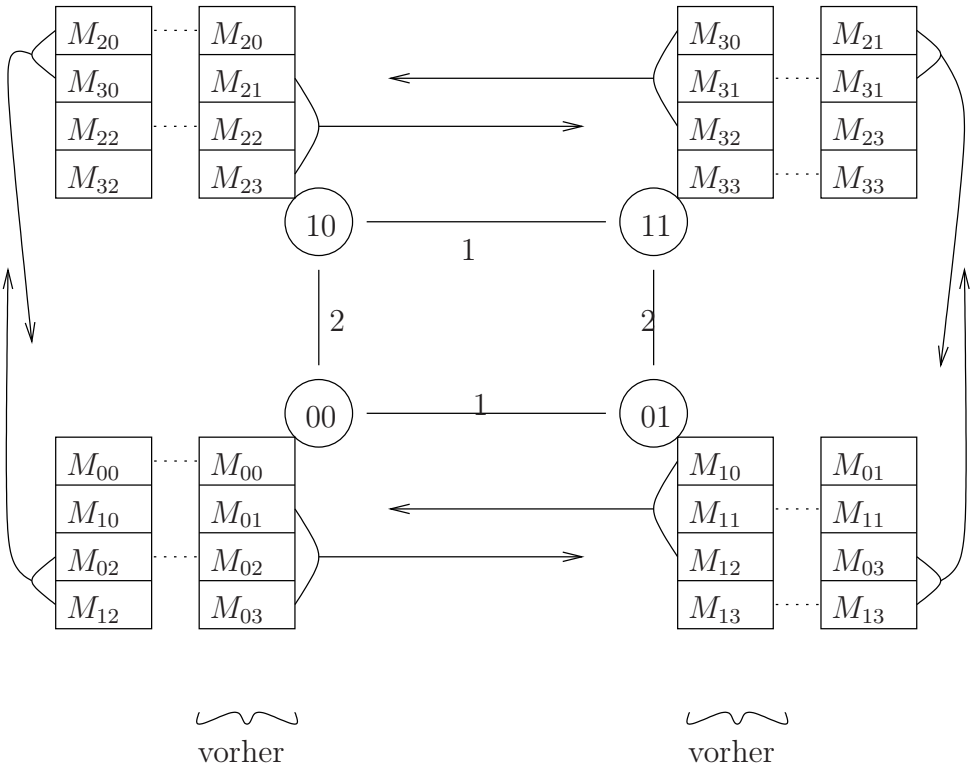


Abbildung 5.13: Illustration für HC mit Dimension 2.

wobei die xe und $y$psilons für alle möglichen Einträge stehen. Es werden also in jeder Kommunikation $P/2$ Nachrichten (nur Bit i ist fest) gesendet.

Nun wollen wir rekursiv herleiten, welche Daten ein Prozess p in jedem Schritt besitzt. Es sei $p = p_{d-1} \dots p_0$ in Binärdarstellung. Prozess p besitzt zu jedem Zeitpunkt P Daten, wobei ein individuelles Datum von Prozess r zu Prozess s unterwegs ist. Jedes Datum können wir also durch das Paar $(r, s) \in \{0, \dots, P-1\} \times \{0, \dots, P-1\}$ identifizieren. Entsprechend schreiben wir $\mathcal{M}_p^i \subset \{0, \dots, P-1\} \times \{0, \dots, P-1\}$ für die Daten, die Prozess p im Schritt i besitzt, d.h.

$$(r, s) \in \mathcal{M}_p^i \iff \begin{array}{l}\text{im Schritt } i \text{ hat Prozess} \\ p \text{ die Daten, die} \\ \text{Prozess } r \text{ an Prozess } s \\ \text{senden will.}\end{array}$$

Zu Beginn besitzt Prozess p die Daten

$$\mathcal{M}_p^0 = \{(p_{d-1} \dots p_0, y_{d-1} \dots y_0) \mid y_{d-1}, \dots, y_0 \in \{0,1\}\}$$

Nach Schritt $i = 0, \dots, d-1$ hat p die Daten $\mathcal{M}_p^{i+1}$, die sich aus $\mathcal{M}_p^i$ und obiger Regel ergeben ($q = p_{d-1} \dots p_{i+1} \overline{p_i} p_{i-1} \dots p_0$):

$$\begin{aligned}\mathcal{M}_p^{i+1} = \mathcal{M}_p^i & \\ \underbrace{\setminus}_{\substack{\text{schickt } p \\ \text{an } q}} & \{(p_{d-1} \dots p_{i+1} p_i x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} \overline{p_i} \dots p_0) \mid x_j, y_j \in \{0,1\}\} \\ \underbrace{\cup}_{\substack{\text{kriegt } p \\ \text{von } q}} & \{(p_{d-1} \dots p_{i+1} \overline{p_i} x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} p_i \dots p_0) \mid x_j, y_j \in \{0,1\}\}\end{aligned}$$

Per Induktion über i zeigt man nun, dass für alle p

$$\mathcal{M}_p^{i+1} = \{(p_{d-1} \dots p_{i+1} x_i \dots x_0, y_{d-1} \dots y_{i+1} p_i \dots p_0) \mid x_j, y_j \in \{0,1\}\ \forall j\}$$

Für $i = -1$ hat M_p^i (für alle p) diese Form.
Ansonsten gilt: $\mathcal{M}_p^{i+1} =$

$$\begin{array}{rlllllll}
 & \{(p_{d-1} \dots p_{i+1} & p_i & x_{i-1} \dots x_0, & y_{d-1} \dots & y_i & p_{i-1} \dots p_0) & \mid \dots\} \\
\cup & \{(p_{d-1} \dots p_{i+1} & \overline{p_i} & x_{i-1} \dots x_0, & y_{d-1} \dots y_{i+1} & p_i & \dots p_0) & \mid \dots\} \\
\setminus & \underbrace{\{\dots\}}_{\substack{\text{was ich nicht} \\ \text{brauche}}} & & & & & & \\
= & \{(p_{d-1} \dots p_{i+1} & x_i & x_{i-1} \dots x_0, & y_{d-1} \dots y_{i+1} & p_i & \dots p_0) & \mid \dots\}
\end{array}$$

Programm 5.12 (Alle an alle individuell mit HC)

```
parallel all-to-all-personalized
{
    const int d, P = 2^d;
    process Π[int p ∈ {0, ..., P − 1}]
    {
        void all_to_all_pers(msg m[P])
        {
            int i, x, y, q, index;
            msg sbuf[P/2], rbuf[P/2];
            for (i = 0; i < d; i++)
            {
                q = p ⊕ 2^i;                    // mein Partner

                // Sendepuffer assemblieren:
                for (y = 0; y < 2^(d−i−1); y++)
                    for (x = 0; x < 2^i; x++)
                        sbuf[y · 2^i + x] = m[y · 2^(i+1) + (q&2^i) + x];
                                  <P/2 (!)

                // Messetsches austauschen:
                if (p < q)
                {
                    send(Π_q,sbuf[0], ..., sbuf[P/2 − 1]);
                    recv(Π_q,rbuf[0], ..., rbuf[P/2 − 1]);
                }
                else
                {
                    recv(Π_q,rbuf[0], ..., rbuf[P/2 − 1]);
                    send(Π_q,sbuf[0], ..., sbuf[P/2 − 1]);
                }

                // Empfangpuffer disassemblieren:
                for (y = 0; y < 2^(d−i−1); y++)
                    for (x = 0; x < 2^i; x++)
```

$$m[\underbrace{y \cdot 2^{i+1} + (q \& 2^i) + x}_{\substack{\text{genau das, was} \\ \text{gesendet wurde, wird} \\ \text{ersetzt}}}] = sbuf[y \cdot 2^i + x];$$

```
        }
    } // ende all_to_all_pers
    ...
    m[q] =„Grüß q von p!“;
    all_to_all_pers(m);
    ...
  }
}
```

Komplexitätsanalyse:

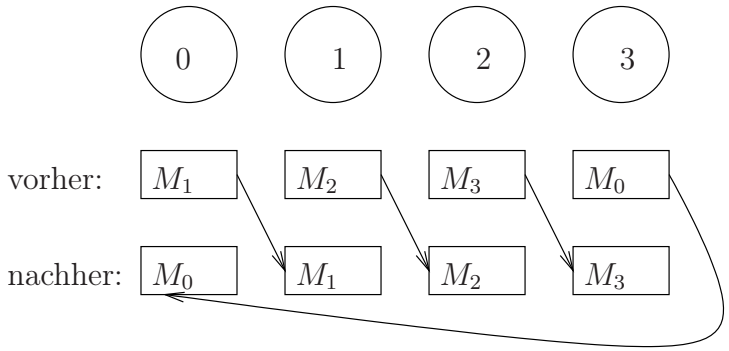
$$\begin{aligned} T_{all-to-all-pers} &= \sum_{i=0}^{\operatorname{ld} P-1} \underbrace{2}_{\substack{\text{send u.} \\ \text{receive}}} (t_s + t_h + t_w \underbrace{\frac{P}{2}}_{\substack{\text{in jedem} \\ \text{Schritt}}} n) = \\ &= 2(t_s + t_h) \operatorname{ld} P + t_w n P \operatorname{ld} P. \end{aligned}$$

5.3 Lokaler Austausch

Bei diesem Problem muss ein Prozess jeweils mit einer festen Menge anderer Prozesse Daten austauschen (d. h. Kommunikation in beide Richtungen). Wir behandeln erst den einfacheren Fall des Ringes und dann den allgemeinen Fall.

5.3.1 Schieben im Ring

Betrachte folgendes Problem: Jeder Prozess $p \in \{0, \ldots, P-1\}$ muss ein Datum an $(p+1)\%P$ schicken:



Folgendes Vorgehen mit *blockierenden* Kommunikationsoperationen produziert einen Deadlock, ähnlich dem im Problem der Philosophen:

```
...
send(Π_{(p+1)%P}, msg);
recv(Π_{(p+P-1)%P}, msg);
...
```

Man kann das Problem lösen, indem ein Prozess (z.B. $p = 0$) die Reihenfolge umdreht (d.h. erst von $P - 1$ empfängt und danach an 1 sendet). Dann werden die Kommunikationen allerdings sequentiell nacheinander abgearbeitet.

Einen optimalen Parallelitätsgrad erreicht man durch „Färben“. Genauer gesagt sei $G = (V, E)$ ein Graph mit

$$\begin{aligned} V &= \{0, \ldots, P-1\} \\ E &= \{e = (p, q) | \text{Prozess } p \text{ muß mit Prozess } q \text{ kommunizieren}\} \end{aligned}$$

Es sind die *Kanten* so einzufärben, dass an einem Knoten nur Kanten unterschiedlicher Farben anliegen. Die Zuordnung der Farben sei durch die Abbildung

$$c\colon E \to \{0, \ldots, C-1\}$$

gegeben, wobei C die Zahl der benötigten Farben ist.

Für das oben beschriebene Problem des Schiebens im Ring genügen zwei Farben für gerades P, wohingegen man drei Farben für ungerades P benötigt (s.Abb. 5.14).

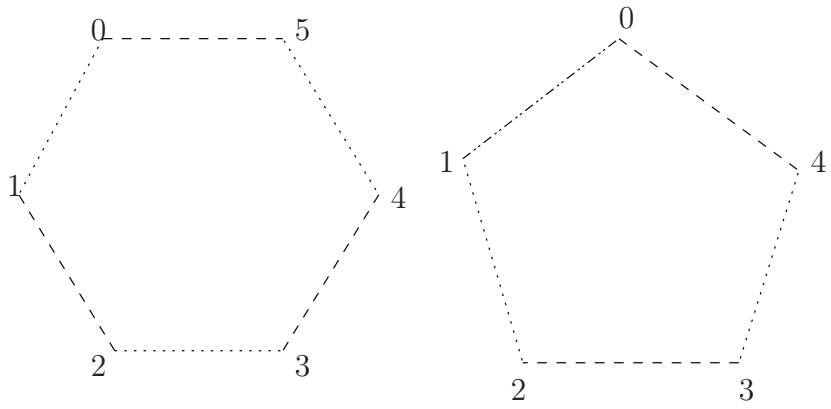


Abbildung 5.14: Im Ring genügen zwei bzw. drei Farben

Die Kommunikation läuft nun in C Schritten ab, wobei in Schritt $0 \leq i < C$ die Nachrichten auf den Kanten der Farbei i ausgetauscht werden.

Bemerkung 5.13 Beim Schieben im $1D$-Feld kommt man immer mit genau zwei Farben aus ($P \geq 3$).

Bemerkung 5.14 Obiger Algorithmus kann leicht dahingehend modifiziert werden, dass Nachrichten in beide Richtungen geschoben werden. Jeder tauscht also eine Nachricht mit all seinen Nachbarn im Graphen aus (s.Abb. 5.15).

Hierzu benötigt man genauso viele Farben wie beim einfachen Schicken. Auf jeder Kante laufen nun zwei Nachrichten. Die kann man dadurch sequentialisieren, dass etwa der Prozess mit der kleineren Nummer zuerst sendet.

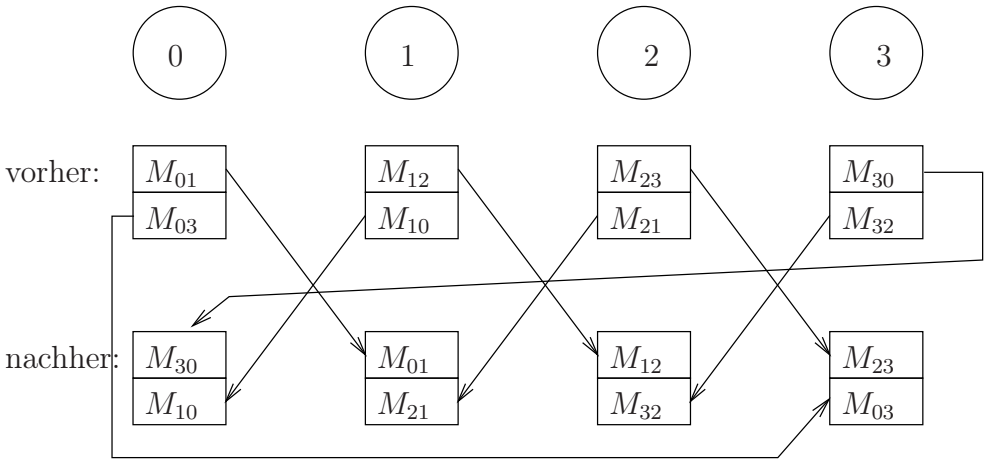


Abbildung 5.15: Schieben im Ring in beide Richtungen

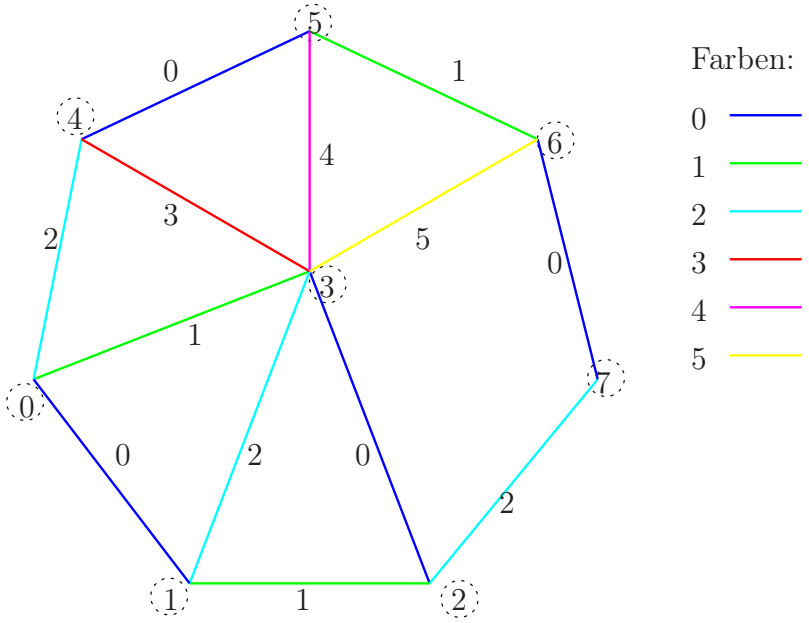


Abbildung 5.16: Farben in einem allgemeinen Graphen

5.3.2 Allgemeiner Graph

Es sei nun ein ungerichteter Graph $G = (V, E)$ gegeben, der bestimmt, welche Prozesse miteinander kommunizieren müssen (s.Abb. 5.16 z.B.).

In diesem Fall kann die Färbung der Kanten nicht mehr so einfach angegeben werden. Insbesondere könnte sich die Kommunikationsstruktur auch erst zur Laufzeit ergeben. Wir benötigen also einen verteilten Algorithmus, der aus der vorgegebenen Nachbarschaftsstruktur die Färbung der Kanten bestimmt. Jeder Prozess habe eine Liste aller Nachbarn in einem Feld

int $nb[N_p]$;

Die Einträge des Feldes seien aufsteigend sortiert, d.h. es gelte

$$nb[i] < nb[i+1].$$

Die Idee des Algorithmus ist einfach: In aufsteigender Reihenfolge handeln die Prozessoren die nächste freie Farbe aus.

Programm 5.15 (Verteiltes Färben)

```
parallel coloring
{
    const int P;
    process Π[int p ∈ {0,...,P−1}]
    {
        int nbs;                                  // Anzahl Nachbarn
        int nb[nbs];                              // nb[i] < nb[i+1]
        int color[nbs];                           // das Ergebnis
        int index[MAXCOLORS];                     // freie Farben
        int i, c, d;

        for (i = 0; i < nbs; i++)
            index[i]=-1;                          // alles frei
        for (i = 0; i < nbs; i++)
        {                                         // Farbe für nb[i]
            c = 0;                                // beginne mit Farbe 0
            while(1)
            {
                c=min{k ≥ c|index[k] < 0};        // nächste freie Farbe ≥ c
                if (p < nb[i])                    // kleinerer sendet zuerst
                {
                    send(Π_nb[i],c);              // Farben
                    recv(Π_nb[i],d);              // austauschen
                }
                else
                {
                    recv(Π_nb[i],c);              // Farben
                    send(Π_nb[i],d);              // austauschen
                }
                if (c == d)
                {                                 // beide sind sich einig
```

```
                    index[c] = i; color[i] = c; break;
                }
                else c = max(c, d);
            }
        }
    }
}
```

Die Farben in der Abbildung 5.16 oben zeigen das Ergebnis des Algorithmus für diesen Graphen. Der Algorithmus findet auch die Färbung für einen Ring mit zwei bzw. drei Farben, benötigt dazu aber $O(P)$ Schritte. Dies ist akzeptabel, da in vielen Anwendungen (z.B. FE) viele Male über den selben Graphen kommuniziert werden muss.

BEMERKUNG 5.16 Alternative: asynchrone Kommunikation.

5.4 Zeitmarken

Im Kapitel 4.1 haben wir die Durchsetzung des wechselseitigen Ausschlusses in einem System mit gemeinsamem Speicher betrachtet. Nun betrachten wir eine Menge nur über **send**/**receive** kommunizierender Prozesse. Diese sind so zu koordinieren, dass nur genau ein Prozessor einen kritischen Abschnitt bearbeiten darf.

5.4.1 Lamport-Zeitmarken

In einer Lösung mit Zeitmarken versucht man mit Hilfe sogenannter „logischer Uhren“ jedem Ereignis (z.B. ich will in kritischen Abschnitt eintreten) in einem beliebigen Prozess eine *global eindeutige Sequenznummer* bzw. eine *Zeitmarke* zuzuordnen. Diese Nummer dient dann der Konfliktlösung: kleinere Nummern, entsprechend „früheren“ Ereignissen, haben Vorrang vor größeren Nummern, die späteren Ereignissen entsprechen.

Das im folgenden vorgestellte Konzept der Zeitmarken stammt von Lamport ((LAMPORT 1978), ja, das ist der von LaTeX). Jeder Prozess besitzt eine lokale Uhr C_p. Die mit einem Ereignis a im Prozess p assoziierte „Uhrzeit“ wird mit $C_p(a)$ bezeichnet. Kommunikation der Prozesse findet nur über **send**/**receive** statt. Die Uhr erfülle folgende Eigenschaften:

1. Seien a und b zwei Ereignisse im selben Prozess p, wobei a vor b stattfinde, so soll $C_p(a) < C_p(b)$ gelten.
2. Es sende p eine Nachricht an q, so soll $C_p(\textbf{send}) < C_q(\textbf{receive})$ sein.
3. Für zwei beliebige Ereignisse a und b in beliebigen Prozessen p bzw. q gelte $C_p(a) \neq C_q(b)$.

Die Bedingungen 1 und 2 sorgen dafür, dass die logische Uhr die Kausalität von Ereignissen widerspiegelt, d.h. wenn in einem parallelen Programm sicher gesagt werden kann, dass a in p *vor* b in q stattfindet, dann gilt auch $C_p(a) < C_q(b)$.

Die Relation $a \leq_C b :\iff C_p(a) < C_q(b)$ – für Ereignis a im Prozess p und Ereignis b im Prozess q – definiert eine partielle Ordnung auf der Menge aller Ereignisse, falls C nur die Eigenschaften 1 und 2 hat. Durch die zusätzliche Bedingung 3 induziert C eine totale Ordnung auf der Menge aller Ereignisse.

Wir sehen uns im folgenden Programm an, wie Lamport-Zeitmarken implementiert werden können.

PROGRAMM 5.17 (LAMPORT-ZEITMARKEN)

```
parallel Lamport-timestamps
{
    const int P;                          // was wohl?
    int d = min{i|2^i ≥ P};               // wieviele Bitstellen hat P.

    process Π[int p ∈ {0,...,P−1}]
    {
        int C=0;                          // die Uhr
        int t, s, r;                      // nur für das Beispiel
        int Lclock(int c)                 // Ausgabe einer neuen Zeitmarke
        {
            C=max(C, c/2^d);              // Regel 2
            C++;                          // Regel 1
            return C · 2^d + p;           // Regel 3
                    // die letzten d Bits enthalten p
        }

        // Verwendung:
        // Ein lokales Ereignis passiert
        t=Lclock(0);

        // send
        s=Lclock(0);
        send(Π_q,message,s);              // Zeitmarke verschicken

        // receive
        recv(Π_q,message,r);              // Zeitmarke empfangen
        r=Lclock(r);                      // so gilt C_p(r) > C_q(s)!
    }
}
```

BEMERKUNG 5.18 Die Verwaltung der Zeitmarken obliegt dem Benutzer, da nur dieser weiss, welche Ereignisse mit Marken versehen werden müssen. Was ist bei Integer-Überlauf?!?

5.4.2 Verteilter wechselseitiger Ausschluss mit Zeitmarken

Lamport Zeitmarken können nun verwendet werden, um das Problem des verteilten wechselseitigen Ausschlusses (VWE) zu lösen.

Die Idee ist wie folgt: Will ein Prozess in den kritischen Abschnitt eintreten, so sendet er eine entsprechende *Anforderungsnachricht* an alle anderen Prozesse. Sobald er eine Antwort von allen anderen Prozessen erhalten hat, kann er in den kritischen Abschnitt eintreten (die Antwort ist also die positive Bestätigung, eine „Nein“-Meldung gibt es nicht). Will ein anderer Prozess auch gerade in den kritischen Abschnitt eintreten, so antwortet er nur, wenn seine Zeitmarke größer (älter) als die empfangene, ansonsten verzögert er die Antwort, bis der kritische Abschnitt abgearbeitet wurde.

Die Anforderungen müssen von den Partnerprozessen „asynchron“ beantwortet werden. Dazu wird jedem Rechenprozess ein Monitor (oder Dämon) zur Seite gestellt, der den Eintritt in den

kritischen Abschnitt überwacht. Zum Empfang der Nachrichten wird die **recv_any**-Funktion verwendet, da man ja nicht weiss, wer als nächstes kommt.

PROGRAMM 5.19 (VWE MIT LAMPORT-ZEITMARKEN)

```
parallel DME-timestamp
{
    int P;                                      // ...
    const int REQUEST=1, REPLY=2;               // Nachrichten

    process Π[int p ∈ {0,...,P − 1}]
    {
        int C=0, mytime;                        // Uhr
        int is_requesting=0, reply_pending;
        int reply_deferred[P]={0,...,0};        // abgewiesene Prozesse

        process M [int p' = p]                  // der Monitor
        {
            int msg, time;
            while(1) {
                recv_any(π,q,msg,time);         // Empfange von q's Monitor
                if (msg==REQUEST)               // q will eintreten
                {
                    [Lclock(time);]             // Erhöhe eigene Uhr
                                                // Koordiniere mit Π
                    [ if(is_requesting ∧ mytime < time)
                        reply_deferred[q]=1;    // q soll warten
                    else
                        asend(M_q,p,REPLY,0); ]// q darf 'rein
                }
                else
                    reply_pending−−;            // es war ein REPLY
            }
        }

        void enter_cs()                         // kritischen Abschnitt
        {
            int i;
            [ mytime=Lclock(0); is_requesting=1; ]
                                                // kritischer Abschnitt
            reply_pending=P − 1;                // Anzahl Antworten
            for (i=0; i < P; i++)
                if (i ≠ p) send(M_i,p,REQUEST,mytime);
            while (reply_pending> 0);           // bisi wait
        }

        void leave_cs()
        {
            int i;
```

```
            [ is_requesting=0;                    //krit. Abschnitt !
            for (i=0; i < P; i++)                 // Wecke Wartende
                if (reply_deferred[i])
                {
                    send(M_i,p,REPLY,0);
                    reply_deferred[i]=0;
                } ]
        }

        ...
        enter_cs();
            /* critical section */
        leave_cs();
        ...
    } // end process
}
```

Abbildung 5.17 zeigt ein Ablaufschema, wenn zwei Prozesse 0 und 1 annähernd gleichzeitig in den kritischen Abschnitt eintreten wollen.

5.4.3 Verteilter wechselseitiger Ausschluss mit Wählen

Beim Zeitmarkenalgorithmus werden $2(P-1)$ Nachrichten für jedes Eintreten in den kritischen Abschnitt verschickt. Der nachfolgende Algorithmus kommt mit $O(\sqrt{P})$ Nachrichten aus. Insbesondere muss ein Prozess *nicht* alle anderen Prozesse fragen, bevor er in den kritischen Abschnitt eintreten kann!

Wie bei einer Wahl bewerben sich mehrere Prozesse um den Eintritt in den kritischen Abschnitt. Diese Prozesse heissen *Kandidaten.* Alle Prozesse (oder nur ein Teil, wie wir sehen werden) stimmen nun darüber ab, wer eintreten darf. Jeder Wähler hat *genau eine Stimme* zu vergeben.

In einer naiven Form könnte jeder Kandidat alle Wähler fragen. Erhält er mehr als die Hälfte aller Stimmen, so kann er eintreten. Wenn man statt absoluter Mehrheit nur eine relative Mehrheit verlangt, so erhält man einen effizienteren Algorithmus: Ein Prozess kann in den kritischen Abschnitt eintreten, sobald er weiss, dass kein anderer mehr Stimmen haben kann als er. Dazu ordnen wir jedem Prozess p einen *Wahlbezirk* $S_p \subseteq \{0,\ldots,P-1\}$ mit folgender Überdeckungseigenschaft zu:

$$S_p \cap S_q \neq \emptyset \quad \forall p,q \in \{0,\ldots,P-1\}.$$

Erhält ein Prozess alle Stimmen seines Wahlbezirkes, so darf er eintreten, denn angenommen q hätte auch alle Stimmen erhalten, so gibt es einen $r \in S_p \cap S_q$, der sich nur für p *oder* q entschieden haben kann.

Wenden wir uns zunächst der Frage zu, wie die Wahlbezirke konstruiert werden. Natürlich möchte man die Wahlbezirke so klein wie möglich machen.

Es sei K die Größe eines Wahlbezirks (alle seien gleich groß) und

$$D = |\{q | p \in S_q\}|$$

die Anzahl der Wahlbezirke in denen ein Prozess Mitglied ist. Wir versuchen nun für gegebene K, D die Zahl der Wahlbezirke M und Prozesse P zu maximieren. Jeder Prozess ist Mitglied in

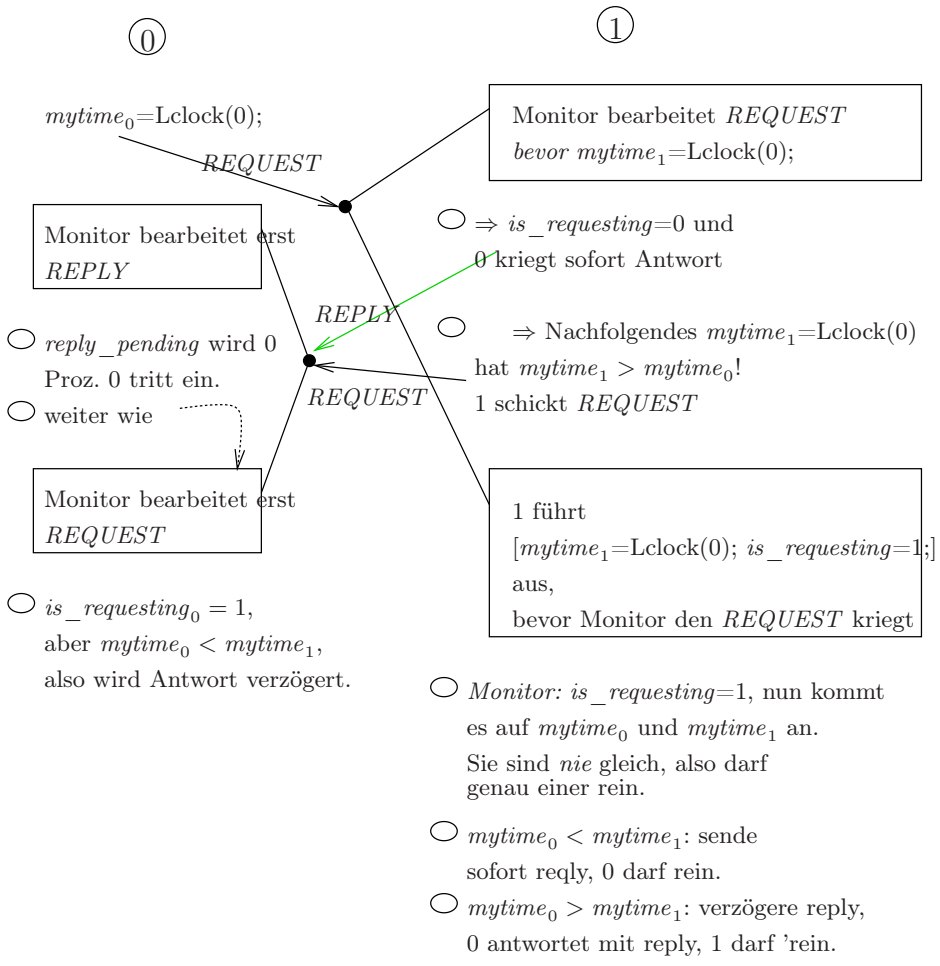


Abbildung 5.17: Prozesse 0 und 1 wollen gleichzeitig in den kritischen Abschnitt

D Wahlbezirken, die Menge

$$A = \{(p, S) | p \text{ Prozess mit } 0 \leq p < P, \text{ S Wahlbezirk mit } p \in S\}$$

hat also $|A| = P \cdot D$ Elemente. Andererseits hat jeder Wahlbezirk K Mitglieder, d.h. die Menge

$$B = \{(S, p) | S \text{ Wahlbezirk, } p \text{ Prozess mit } p \in S\}$$

hat $|B| = M \cdot K$ Elemente. Wegen $(p, S) \in A \iff (S, p) \in B$ gilt $|A| = |B|$, $P \cdot D = M \cdot K$. Pro Prozess gibt es genau einen Wahlbezirk, d.h. P=M, somit erhalten wir

$$D = K.$$

Ein beliebiger Wahlbezirk S hat K Mitglieder. Jedes Mitglied von S ist Mitglied von $D - 1$ anderen Wahlbezirken. Somit kann es bei gegebenem D, K höchstens

$$M \leq K(D - 1) + 1$$

Wahlbezirke geben. Wegen $P = M$ und $D = K$ erhalten wir:

$$\boxed{P \leq K(K - 1) + 1}$$

oder

$$K \geq \frac{1}{2} + \sqrt{P - \frac{3}{4}}.$$

Eine mögliche Konstruktion für $P = 16$ zeigt folgende Abbildung (Wer findet die Lösung mit $D = K = 5$?):

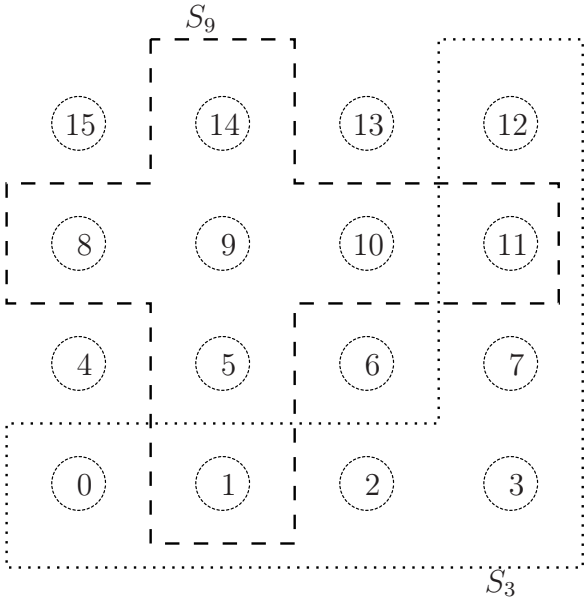


Abbildung 5.18: Konstruktion der Wahlbezirke für $P = 16$.

Der bis jetzt skizzierte Algorithmus enthält die Gefahr eines Deadlocks. Betrachte die Kandidaten 3 bzw 9 mit ihren Wahlbezirke S_3 und S_9. Es könnte folgende Situation entstehen:

- 9 gewählt von $1, 5, 8, 10, 13 \in S_9$
- 3 gewählt von $0, 2, 7, 11, 15 \in S_3$

Keiner der beiden hat alle Stimmen seines Wahlbezirkes, aber alle Wähler haben ihre Stimme bereits abgegeben. Das Protokoll muss also noch eine Komponente zur Deadlockauflösung enthalten. Dies geschieht wieder mit Hilfe von Zeitmarken. Das Problem liegt darin, dass ein Wähler seine Stimme abgeben muss *bevor* er alle Kandidaten kennt. So hat sich 1 für 9 entschieden, da er von diesem zuerst gefragt wurde. Versieht man die Anfragen mit Zeitmarken, so könnte 1 erkennen, dass die Anfrage von 3 älter ist (kleinere Zeitmarke), und seine Stimme von 9 zurückfordern und sie der 1 geben. Allerdings könnte 9 auch schon in dem kritischen Abschnitt eingetreten sein bevor 1 seine Stimme zurückhaben will ...

PROGRAMM 5.20 (VME MIT WÄHLEN)

```
parallel DME-Voting
{
    const int P = 7.962;
    const int REQUEST=1, YES=2, INQUIRE=3,
              RELINQUISH=4, RELEASE=5;
          // „inquire“ = „sich erkundigen“; „relinquish“ = „verzichten“
    process Π[int p ∈ {0, ..., P − 1}]
    {
        int C=0, mytime;

        void enter_cs()                    // will eintreten
        {
            int i, msg, time, yes_votes=0;
            [ mytime=Lclock(0); ]          // Zeit meiner Anfrage
            for (i ∈ S_p) asend(V_i,p, REQUEST,mytime);
                                           // Anfrage an Wahlbezirke
            while (yes_votes < |S_p|)
            {
                recv_any(π,q,msg,time);    // empfange von q
                if (msg==YES) yes_votes++; // q wählt mich
                if (msg==INQUIRE)          // q will Stimme zurück
                    if (mytime==time)      // nur aktuelle Anfrage
                    {
                        asend(V_q,p,RELINQUISH,0);
                                           // gib zurück
                        yes_votes−−;
                    }

            }

        }// end enter_cs


        void leave_cs()
        {
```

```
    int i;
    for (i ∈ S_p) asend(V_i,p,RELEASE,0);
    // Es könnten noch nicht bearbeitete INQUIRE-Messages für diesen
    // kritischen Abschnitt anstehen, die nun obsolet sind.
    // Diese werden dann in enter_cs ignoriert.
}

// Beispiel:
enter_cs();
...; // kritischer Abschnitt
leave_cs();

process V[int p' = p]                    // der Voter zu Π_p
{
    int q, candidate;                    // Prozesse
    int msg, time, have_voted=0,
        candidate_time, have_inquired=0;
    while(1)                             // läuft für immer
    {
        recv_any(π,q,msg,time);          // empfange sie mit Absender
        if (msg==REQUEST)                // Anfrage eines Kandidaten
        {
            [ Lclock(time); ]            // Uhr weiterschalten für spätere Anfragen
            if (¬have_voted)             // ich habe meine Stimme noch zu vergeben
            {
                asend(Π_q,p,YES,0);      // zurück an Kandidaten-Prozess
                candidate_time=time;     // merke, wem ich meine
                candidate=q;             // Stimme gegeben habe.
                have_voted=1;            // ja, ich habe schon gewählt
            }
            else                         // ich habe schon gewählt
            {
                Speichere (q, time) in Liste;
                if (time < candidate_time ∧ ¬have_inquired)
                {                        // hole Stimme vom Kandidaten zurück!
                    asend(Π_candidate,p,INQUIRE,candidate_time);
                            // an der candidate_time erkennt er, um welche Anfrage
                            // es geht: es könnte sein, dass er schon eingetreten ist.
                    have_inquired=1;
                }
            }
        }
        else if (msg==RELINQUISH)
                                         // q ist der Kandidat, der mir meine
                                         // Stimme zurück gibt.
        {
            Speichere (candidate, candidate_time) in Liste;
            Entnehme und Lösche
```

```
                    den Eintrag mit der kleinsten time aus der Liste: (q, time)
                                                // Es könnte schon mehrere geben
                asend(Π_q,p,YES,0);             // gebe dem q meine Stimme
                candidate_time=time;            // neuer Kandidat
                candidate=q;
                have_inquired=0;                // kein INQUIRE mehr unterwegs
            }
            else if (msg==RELEASE)              // q verlässt den kritischen Abschnitt
            {
                if (Liste ist nicht leer)
                {                               // vergebe Stimme neu
                    Entnehme und Lösche
                        den Eintrag mit der kleinsten time aus der Liste: (q, time)
                    asend(Π_q,p,YES,0);
                    candidate_time=time;        // neuer Kandidat
                    candidate=q;
                    have_inquired=0;            // vergiss alle INQUIREs weil obsolet
                }
                else
                    have_voted=0;               // niemand mehr zu wählen
            }
        }
    } // end Voter
}
}
```

5.5 Markenbasierte Algorithmen

Wir wenden uns nun Algorithmen zu, die Prozesse koordinieren indem sie eine Marke (engl. *token*) herumreichen. Nur der Prozess der die Marke besitzt darf etwas bestimmtes tun.

5.5.1 Verteilter wechselseitiger Ausschluß

Zur Illustration der Idee mit den Marken betrachten wir noch einmal das Problem des verteilten wechselseitigen Ausschluß. Die zunächst vorgestellte Lösung benötigt im Mittel $P/2$ Kommunikationen. Dann werden wir aber eine Lösung konstruieren, die mit nur $O(\mathrm{ld}\, P)$ Schritten auskommt!

Die Idee zur $O(P)$-Lösung ist einfach: In einem Ring wird eine Marke ständig herumgereicht. Ein Prozess darf in den kritischen Abschnitt eintreten, wenn er die Marke besitzt. In diesem Fall hält er sie fest. Das Herumreichen wird von einem zusätzlich zu dem Rechenprozess vorhandenen Prozess realisiert:

Programm 5.21 (VME mit Marken)

```
parallel DME-token-ring
{
    const int P = 10;

    process Π_i [int i ∈ {0, ..., P − 1}]
```

```
    {
        while (1) {
            ...;
            send(M_i,1);                    // Eintritt
            kritischer Abschnitt;
            send(M_i,1);                    // Austritt
        }
    }

    process M_i [int i ∈ {0,...,P−1}]
    {
        int msg;
        if (i == 0) send(M_1,1);            // initialisiere Marke
        while (1) {
            recv(M_{(i+P−1) mod P},msg);    // empfange Marke
            if (rprobe(Π_i)) {
                recv(Π_i,msg);              //blockiert nicht
                recv(Π_i,msg);              //blockiert
            }
            delay;                          // bremse Marke
            send(M_{(i+1) mod P},1);        // sende Marken
        }
    }
}
```

Offensichtlich ist die Lösung fair, da nach Bearbeiten eines kritischen Abschnitts die Marke erst einmal im Ring rumlaufen muss. Nachteil dieser Lösung ist, dass relative viele Nachrichten herumlaufen auch wenn kein Prozess den kritischen Abschnitt betreten will.

Um nun eine Lösung mit $O(\operatorname{ld} P)$ Schritten zu erhalten seien alle Prozesse in einem vollständigen binären Baum angeordnet. Wir betrachten also $P = 2^d - 1$ Prozesse. Prozess M_0 sei die Wurzel des Baumes und jeder kennt seinen linken und rechten Nachfolger M_l und M_r sowie seinen Vorgänger M_u.

Wieder kann nur der in den kritischen Abschnitt eintreten der die Marke besitzt. Die Marke wird aber nur auf Anforderung herumgereicht. Dazu muss jeder wissen wer die Marke gerade hat. Da es im Baum nur genau einen Weg von einem Knoten zu einem anderen Knoten gibt muss man sich nur merken in welcher Richtung sich die Marke befindet. Fordern mehrere Prozesse die Marke an muss man sich alle Anforderungen merken. Um Fairness sicherzustellen werden die Anforderungen mit einer Lamport-Zeitmarke versehen.

Hier nun das vollständige Programm:

Programm 5.22 (VME mit Marken und binärem Baum)

```
parallel DME-token-tree
{
    const int d = 4; P = 2^d − 1;
    const Int REQUEST=1, TOKEN=2;

    process Π_i [int i ∈ {0,...,P−1}]
    {
```

```
        int ts, msg;
        while (1) {
            ...;
            ts = Lclock(0);              // neue Zeitmarke
            send(M_i,REQUEST,ts);        // Anforderung
            recv(M_i,msg,ts);            // erhalten
            Lclock(ts);                  // Uhr weiter
            kritischer Abschnitt;
            send(M_i,TOKEN,1);           // Austritt
        }
    }

    process M_i [int i ∈ {0,...,P-1}]
    {
        process M_l, M_r, M_u;           // Der binäre Baum;
        process who, where, Π;
        int msg,ts,tr;
        List requests;

        if (i == 0) where=M_0; else where=M_u;
        while (1) {
            recv_any(who,msg,ts);        // von M_u, M_l, M_r oder Π_i
            if (msg==REQUEST)
                put_min(requests,(who,ts));
            if (msg==TOKEN) {
                where = M_i;             // ich hab es
                tr = 0;                  // request bearbeitet
            }
            if (where==M_i ∧ ¬empty(requests))
            {
                (Π,ts) = get_min(requests);
                where = Π;
                send(Π,TOKEN,ts);
            }
            if (where ≠ M_i ∧ where ≠ Π_i ∧ ¬empty(requests) ∧ tr == 0)
            {
                (Π,ts) = get_min(requests);
                put_min(requests,(Π,ts));
                send(where,REQUEST,ts);
                tr = 1;
            }
        }
    }
}
```

Das Programm verwendet eine Liste mit den Operationen

- *put_min*: Ein Paar (Π, t) wird in die Liste aufgenommen falls für den Prozess Π noch kein Eintrag exisiert. Ein vorhandener Eintrag für Π wird ersetzt wenn die Zeitmarke t kleiner

ist als die schon gespeicherte Zeitmarke.

- *get_min*: Liefert das Paar (Π, t) mit der kleinsten Zeitmarke t und entfernt dieses aus der Liste.

5.5.2 Verteilte Terminierung

Es seien die Prozesse $\Pi_0, \ldots, \Pi_{P-1}$ gegeben, welche über einen Kommunikationsgraphen

$$\begin{aligned} G &= (V, E) \\ V &= \{\Pi_0, \ldots, \Pi_{P-1}\} \\ E &\subseteq V \times V \end{aligned}$$

kommunizieren.

Dabei schickt Prozess Π_i Nachrichten an die Prozesse

$$N_i = \{j \in \mathbb{N} \mid (\Pi_i, \Pi_j) \in E\}$$

Das geschieht auf folgende Weise:

```
process Π_i [ int i ∈ {0,...,P-1}]
{
    while (1)
    {
        recv_any(who,msg),          // Π_i ist idle
        compute(msg);
        for ( p ∈ N_msg ⊆ N_i )
        {
            msg_p = ... ;
            asend(Π_p, msg_p);      // vernachlässige Pufferproblem

        }
    }
}
```

Das Terminierungsproblem besteht darin, daß das Programm beendet ist, falls gelten:

1. Alle warten auf eine Nachricht (sind idle)
2. Keine Nachrichten sind unterwegs

Dabei werden folgende Annahmen über die Nachrichten gemacht:

1. Vernachlässigung von Problemen mit Pufferüberlauf
2. Die Nachrichten zwischen zwei Prozessen werden in der Reihenfolge des Absendens bearbeitet

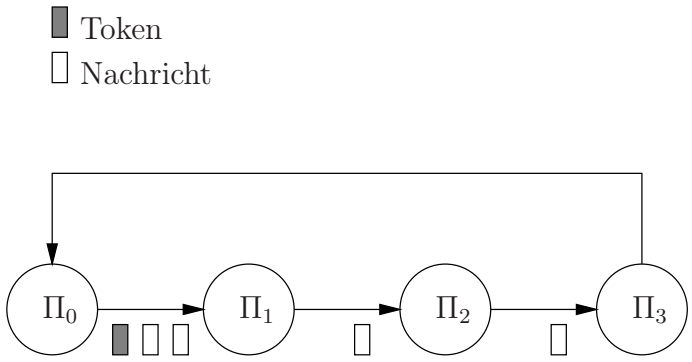


Abbildung 5.19: Terminierung im Ring

1.Variante: Ring

Jeder Prozess hat einen von zwei möglichen Zuständen: rot (aktiv) oder blau (idle). Zur Terminierungserkennung wird eine Marke im Ring herumgeschickt.

Angenommen, Prozess Π_0 startet den Terminierungsprozess, wird also als erster blau. Weiter angenommen,

1. Π_0 ist im Zustand blau
2. Marke ist bei Π_i angekommen und Π_i hat sich blau gefärbt

Dann kann gefolgert werden, daß die Prozesse $\Pi_0, \ldots, \Pi_i$ idle sind und die Kanäle $(\Pi_0, \Pi_1), \ldots, (\Pi_{i-1}, \Pi_i)$ leer sind.

Ist die Marke wieder bei Π_0 und ist dieser immer noch blau (was er ja feststellen kann), so gilt offenbar:

1. $\Pi_0, \ldots, \Pi_{P-1}$ sind idle
2. Alle Kanäle sind leer

Damit ist die Terminierung erkannt.

2.Variante: Allgemeiner Graph mit gerichteten Kanten

Idee: Über den Graph wird ein Ring gelegt, der alle Knoten erfasst, wobei ein Knoten auch mehrmals besucht werden kann.

Allerdings reicht hier der Algorithmus vom Ring nicht mehr aus, denn stellen wir uns für das obige Beispiel vor, Π_0 starte den Terminierungsprozess , und die Marke habe Π_2 erreicht, dann kann nicht gefolgert werden, daß $\Pi_0, \ldots, \Pi_2$ idle sind, denn inzwischen könnte Π_1 eine Nachricht von Π_3 bekommen haben.

Es gilt aber: Kommt die Marke wieder bei Π_0 an und Π_0 ist immer noch idle und die Marke erreicht jetzt Π_i und alle $\Pi_0, \ldots, \Pi_i$ sind idle geblieben, so sind die letzten i Kanäle leer.

Idee für den Algorithmus: Wähle einen Pfad $\pi = (\Pi_{i_1}, \Pi_{i_2}, \ldots, \Pi_{i_n})$ der Länge n von Prozessoren derart aus, daß gelten:

1. Jede Kante $(\Pi_p, \Pi_q) \in E$ kommt mindestens einmal im Pfad vor

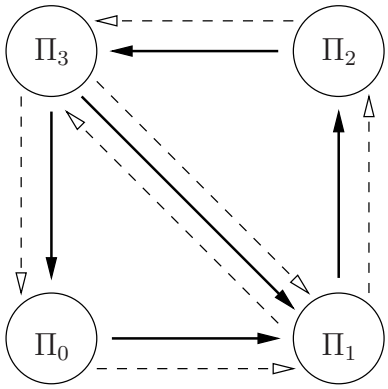


Abbildung 5.20: Allgemeiner Graph mit darübergelegtem Ring

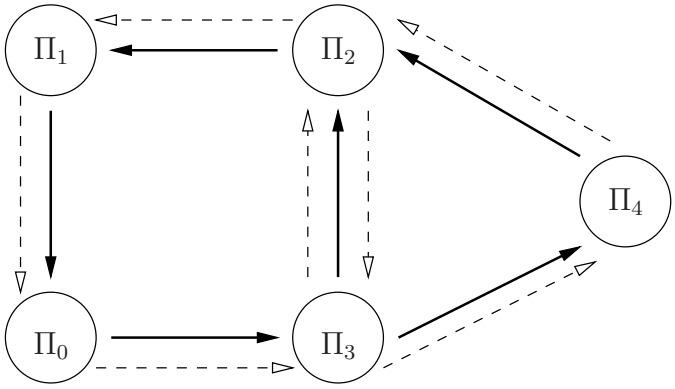


Abbildung 5.21: In diesem Beispiel ist $\pi = (\Pi_0, \Pi_3, \Pi_4, \Pi_2, \Pi_3, \Pi_2, \Pi_1, \Pi_0)$

2. Eine Sequenz (Π_p, Π_q, Π_r) kommt höchstens einmal im Pfad vor. Erreicht man also q von p aus, so geht es immer nach r weiter. r hängt also von Π_p und Π_q ab: $r = r(\Pi_p, \Pi_q)$

Man beachte, daß das Token diesmal einen Wert hat.

```
process Π [ int i ∈ {0,...,P − 1}]
{
    int color = red , token;
    if (Π_i == Π_{i_1})
    {   // Initialisierung des Tokens
        color = blue;
        token = 0 ,
        asend(Π_{i_2}, TOKEN, token)

    }
    while(1)
    {
        recv_any(who,tag,msg);
        if ( tag != TOKEN ) { color = red; Rechne weiter }
        else    // msg = Token
        {
            if ( msg == n ) { break; „hurra, fertig! “}
            if ( color == red )
            {
                color = blue ;
                token = 0 ;
                rcvd = who ;

            }
            else

                if ( who == rcvd ) token++ ;
                                // ein kompletter Zyklus

            asend(Π_{r(who,Π_i)}, TOKEN , token );

        }
    }
}
```

Ist das Token = l , so haben die letzten l Prozesse das Token im Zustand blau erhalten und weitergeschickt. Die letzten l Kanten enthielten keine Nachrichten mehr.

Ist l = n , so sind alle idle und keine Nachrichten mehr in Puffern.

3.Variante: Synchrone Kommunikation

Da die Nachrichten mit synchroner Kommunikation abgegeben werden, die Kommunikationskanäle also keine Pakete speichern können, reicht es, ein Token im Ring herumzuschicken.

Wieder befindet sich jeder Prozess in einem von zwei Zuständen: rot (aktiv) oder blau (inaktiv), wobei zu Beginn alle rot sind. Wird nun Prozess 0 zum ersten Mal frei, geht er in den Zustand blau über und sendet das Token mit Wert 0 zu Prozess 1. Empfängt ein Prozess das Token im freien Zustand und ist rot, so wird er blau und sendet es mit Wert 0 weiter. Ist er bei Empfang im Zustand blau (und frei), so erhöht er den Wert des Tokens um 1 und sendet es weiter. Ist er bei Empfang im Zustand aktiv, so bleibt er rot und behält das Token solange, bis er inaktiv wird (dann schickt er es mit Wert 0 weiter und wird blau). Bekommt ein Prozess eine neue Nachricht, so wird er rot. Empfängt ein Prozess im Zustand blau das Token mit Wert P (Anzahl der Prozessoren), so ist das Programm beendet.

5.5.3 Verteilte Philosophen

Wir betrachten noch einmal das Philosophenproblem, diesmal jedoch mit message passing.

- Lasse eine Marke im Ring herumlaufen. Nur wer die Marke hat, darf eventuell essen.
- Zustandsänderungen werden dem Nachbarn mitgeteilt, **bevor** die Marke weitergeschickt wird.
- Jedem Philospohen P_i ist ein Diener W_i zugeordnet, der die Zustandsmanipulation vornimmt.
- Wir verwenden nur synchrone Kommunikation

```
process P_i [ int i ∈ {0,...,P-1}]
{
    while (1) {
        think;
        send(W_i, HUNGRY );
        recv( W_i, msg );
        eat;
        send( W_i, THINK );

    }
}
```

```
process W_i [ int i ∈ {0,...,P-1}]
{
    int L = (i+1)%P;
    int R = (i+p-1)%P ;
    int state = stateL = stateR = THINK ;
    int stateTemp;
    if ( i == 0 ) send( W_L , TOKEN );
    while (1) {
        recv_any( who, tag );
        if ( who == P_i ) stateTemp = tag ;
```

```
                    // Mein Philospoph
        if ( who == W_L && tag ≠ TOKEN ) stateL = tag ;
                    // state change
        if ( who == W_R && tag ≠ TOKEN ) stateR = tag ;
                    // in Nachbarn
        if ( tag == TOKEN){
            if ( state ≠ EAT && stateTemp == HUNGRY
                && stateL == THINK && stateR == THINK ){
                    state = EAT;
                    send( W_l , EAT );
                    send( W_R , EAT );
                    send( P_i , EAT );

            }
            if ( state == EAT && stateTemp ≠ EAT ){
                state = THINK;
                send( W_L , THINK );
                send( W_R , THINK );
            }
            send( W_L , TOKEN );
        }
    }
}
```

Der Übergang von HUNGRY auf EAT ist einem Philosphen nur dann möglich, falls sein Diener das Token hat (damit ist er der einzige, der etwas verändert) und weder linker noch rechter Nachbar essen und außerdem der Philosoph selber essen will.

Ebenfalls ist der Übergang von EAT auf THINK nur möglich, falls der Diener das Token hat. Dann hat der Philosoph bereits aufgehört zu essen, also ist $stateTemp \in \{$ THINK , HUNGRY $\}$.

Das Weitersenden des Tokens bedeutet, daß die Nachbarn die Zustandsänderung mitbekommen haben.

5.6 Client-Server Paradigma

...welches wir schon x-mal benutzt haben.

- **Server:** Prozess, der in einer Endlosschleife Anfragen (Aufträge) von Clients bearbeitet.
- **Client:** Stellt in unregelmäßigen Abständen Anfragen an einen Server.

Zum Beispiel waren im Fall der verteilten Philospohen die Philosophen die Clients und die Diener die Server (die auch untereinander kommunizieren).

Praktische Beispiele:

- File Server (NFS: Network File Server)
- Datenbank-Server
- HTML-Server

Weiteres Beispiel: File Server, Conversational Continutity

Über das Netzwerk soll der Zugriff auf Dateien realisiert werden.

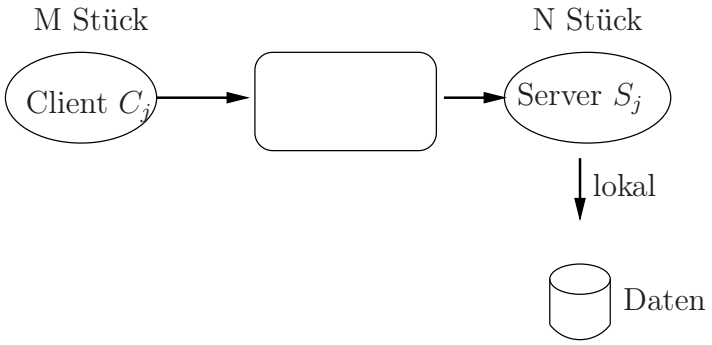


Abbildung 5.22: Datenzugriff Client über Server

- Client : öffnet Datei; macht beliebig viele Lese-/ Schreibzugriffe; schließt Datei.
- Server : bedient genau einen Client, bis dieser die Datei wieder schließt. Wird erst nach Beendigung der Kommunikation wieder frei.
- Allocator : ordnet dem Client einen Server zu.

```
process C [ int i ∈ {0, ..., M − 1}]
{
    send( A, OPEN , „foo.txt “);
    recv( A , ok , j );
    send( S_j , READ , where );
    recv( S_j , buf );
    send( S_j , WRITE , buf , where );
    recv( S_j , ok );
    send( S_j , CLOSE );
    recv( S_j , ok );
}
```

```
process A                       // Allocator
{
    int free [N] = {1[N]} ;     // alle Server frei
    int cut = 0;                // wieviel Server belegt?
    while (1) {
        if ( rprobe( who ) ) {          // von wem kann ich empfangen?
            if ( who ∈ {C_0, ..., C_{M−1}} & & cut == N )
                continue;                       // keine Server frei
```

```
            recv( who , tag , msg );
            if ( tag == OPEN ){
                Finde freien Server j ;
                free [j] = 0 ;
                cut++;
                send( S_j , tag , msg , who );
                recv( S_j , ok );
                send( who , ok , j );
            }
            if ( tag == CLOSE )
                for ( j ∈ {0, ..., N − 1} )
                    if ( S_j == who ) {
                        free [j] = 1;
                        cut = cut - 1 ;
                    }
        }
    }
}
```

```
process S [ int j ∈ {0, ..., N − 1}]
{
    while (1) {
        // warte auf Nachricht von A
        recv( A , tag , msg , C );// mein Client
        if ( tag ≠ OPEN ) → error;
        öffne Datei msg
        send( A , ok );
        while (1) {
            recv( C , tag , msg );
            if ( tag == READ ) {
                ...
                send( C , buf );
            }
            if ( tag == WRITE ) {
                ...
                send( C , ok ); }
            }
            if ( tag == CLOSE ){
                close file;
                send( C , ok );
                send( A , CLOSE , dummy );
                break;
            }
        }
    }
}
```

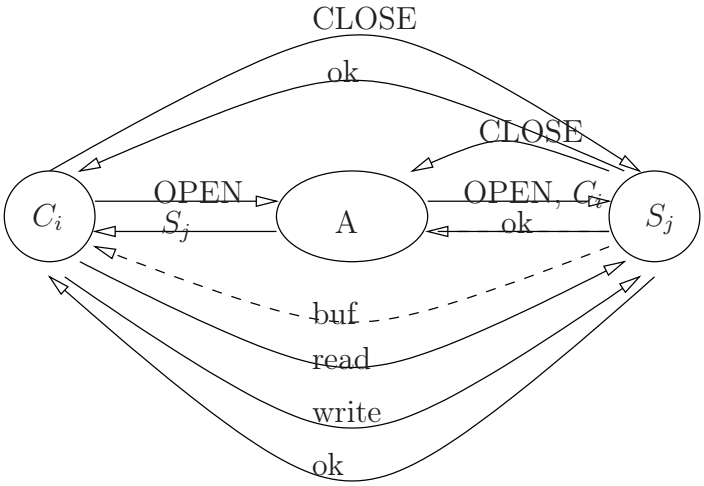


Abbildung 5.23: Kommunikation Client/Allocator/Server

6 Entfernter Prozeduraufruf

6.1 Einführung

Wird abgekürzt mit RPC (Remote Procedure Call). Ein Prozess ruft eine Prozedur in einem anderen Prozess auf.

Π_1:	Π_2:
⋮	**int** Square(**int** x)
y = Square(x);	{
⋮	return x · x;
	}

Es gilt dabei:

- Die Prozesse können auf verschiedenen Prozessoren sein.
- Der Aufrufer blockiert solange, bis die Ergebnisse da sind.
- Es findet eine Zweiweg-Kommunikation statt, d.h. Argumente werden hin-/ und Ergebnisse zurückgeschickt. Für das Client-Server Paradigma ist das die ideale Konfiguration.
- Viele Clients können gleichzeitig eine entfernte Prozedur aufrufen.

In unserer simplen Programmiersprache realiesieren wir den RPC dadurch, dass Prozeduren durch das Schlüsselwort `remote` gekennzeichnet werden. Diese können dann von anderen Prozessen aufgerufen werden.

Programm 6.1 (RPC-Syntax)

```
parallel rpc-example
{
    process Server
    {
        remote int Square(int x)
        {
            return x · x;
        }
        remote long Time ( void )
        {
            return time_of_day;
        }
        ... Initialisierungscode
    }
```

```
    process Client
    {
        y = Server.Square(5);
    }
}
```

Abbildung 6.1 zeigt was beim Aufruf einer Funktion in einem anderen Prozess technisch passiert. Die Argumente werden auf Aufruferseite in eine Nachricht verpackt, über das Netzwerk geschickt und auf der anderen Seite wieder ausgepackt. Nun kann die Funktion ganz normal aufgerufen werden. Der Rückgabewert der Funktion wird auf dieselbe Weise zum Aufrufer zurückgeschickt.

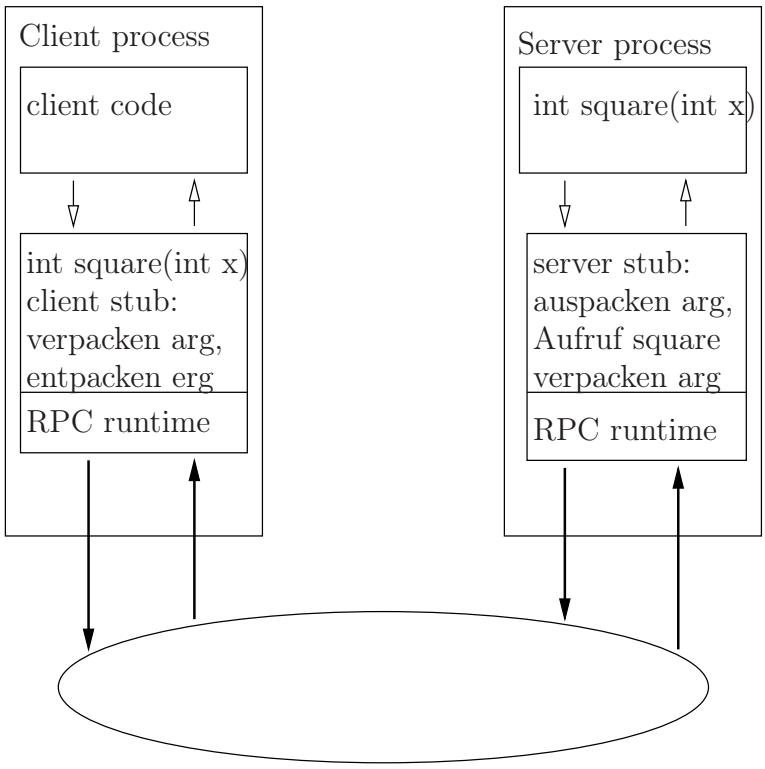


Abbildung 6.1: Realisierung RPC

6.2 Praxis: SUN RPC

Eine sehr häufig verwendete Implementierung des RPC stammt von der Firma SUN. Die wichtigsten Eigenschaften sind:

- Portabilität (Client/Server Anwendungen auf verschiedenen Architekturen). Dies bedeutet, dass die Argumente und Rückgabewerte in einer Rechnerunabhängigen Form über das Netzwerk transportiert werden müssen. Dazu dient die `XDR`-Bibliothek (external data representation).
- Es sind wenig Kenntnisse über Netzwerkprogrammierung erforderlich.

Wir gehen nun schrittweise durch wie man obiges Beispiel mittels SUN's RPC realisiert.

Erstelle RPC Spezifikation

In die Datei `square.x` packen wir eine Beschreibung aller remote aufrufbaren Funktionen und ihrer Argumente.

```
struct square_in {  /* first argument */
    int arg1;
} ;

struct square_out { /* return value */
    int res1;
} ;

program SQUARE_PROG {
    version SQUARE_VERS {
        square_out SQUAREPROC(square_in) = 1;
        /*procedure number */
    } = 1;  /* version number */
} = 0x31230000 ; /* program number */
```

Übersetzen der Beschreibung

Mit dem RPC-Compiler wird die Datei `square.x` nun verarbeitet:

```
rpcgen -C square.x
```

und heraus kommen vier Dateien mit folgender Bedeutung:

1. `square.h`: Datentypen für Argumente, Prozedurköpfe (Ausschnitt)

   ```
   #define SQUAREPROC 1

   /* die ruft Client */
   extern square_out *
   squareproc_1(square_in *, CLIENT *);

   /* Serverseite */
   extern square_out *
   squareproc_1_svc(square_in *, struct svc_req *);
   ```

2. `square_clnt.c`: Client-Seite der Funktion, Verpacken der Argumente

   ```
   #include <memory.h>     /* for memset */
   #include "square.h"

   /* Default timeout can be changed using clnt_control() */
   static struct timeval TIMEOUT = { 25, 0 };
   ```

```
square_out  * squareproc_1(square_in *argp, CLIENT *clnt)
{
    static square_out clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, SQUAREPROC,
        (xdrproc_t) xdr_square_in, (caddr_t) argp,
        (xdrproc_t) xdr_square_out, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
            return (NULL);
    }
    return (&clnt_res);
}
```

3. `square_svc.c`: Kompletter Server, der auf den Prozeduraufruf reagiert.
4. `square_xdr.c`: Funktion für Datenkonversion in heterogener Umgebung:

```
#include "square.h"

bool_t xdr_square_in (XDR *xdrs, square_in *objp)
{
    register int32_t *buf;

    if (!xdr_int (xdrs, &objp->arg1))
        return FALSE;
    return TRUE;
}

bool_t xdr_square_out (XDR *xdrs, square_out *objp]
{
    register int32_t *buf;

    if (!xdr_int (xdrs, &objp->res1))
        return FALSE;
    return TRUE;
}
```

Client

Nun ist der Client zu schreiben, der die Prozedur aufruft (`client.c`):

```
#include "square.h" /* includes also rpc/rpc.h */
int main ( int argc, char **argv)
{
    CLIENT *cl;
    square_in in;
```

```
    square_out *outp;

    if ( argc!=3) {
        printf("usage: client <hostname>
                <integer-value>\n");
        exit(1);
    }

    cl = clnt_create(argv[1],
        SQUARE_PROG,SQUARE_VERS,"tcp");
    if (cl== NULL) {
        printf("clnt_create failed\n");
        exit(1);
    }

    in.arg1 = atoi(argv[2]);
    outp = squareproc_1(&in,cl);/* RPC */
    if (outp==NULL) {
        printf("%s",clnt_sperror(cl,argv[1]));
        exit(1);
    }

    printf("%d\n", outp->res1);
    exit(0);
}
```

Das Übersetzen des Client geschieht mittels:

```
gcc -g -c client.c
gcc -g -c square_xdr.c
gcc -g -c square_clnt.c
gcc -o client client.o square_xdr.o square_clnt.o
```

Remote Funktionen auf Serverseite

Schlussendlich ist die Funktion auf der Serverseite zu schreiben (`server.c`):

```
square_out *
squareproc_1_svc(square_in *inp,
                 struct svc_req *rqstp)
{
    /* weil wir pointer zurueckgeben werden ! */
    static square_out out;

    out.res1 = inp->arg1 * inp->arg1;
    return (&out);
}
```

Das Übersetzen des Servers geschieht mittels:

```
gcc -g -c server.c
gcc -g -c square_xdr.c
gcc -g -c square_svc.c
gcc -o server.o square_xdr.o square_svc.o
```

Test

Als erstes ist sicherzustellen, dass der portmapper läuft:

```
rpcinfo -p
```

Starte server mittels

```
server &
```

Starte Client:

```
peter@troll:~/TeX/vorlesung/pr1_ueb/rpc > client troll 123
15129
```

Per default beantwortet der Server die Anfragen sequentiell nacheinander. Einen multithreaded Server kriegt man so:

- generiere RPC code mittels `rpcgen -C -M ...`
- Mache die Prozeduren reentrant. Trick mit static Varianble geht dann nicht mehr. Lösung: Gebe Ergebnis in einem call-by-value Parameter zurück.

6.3 CORBA

In der objektorientierten Programmierung kapseln Klassen sowohl Daten als auch Methoden. Nun können wir uns die Objekte verteilt über mehrere Rechner vorstellen. Der Aufruf einer Methode eines Objekts das nicht lokal vorliegt entspricht dabei einem RPC.

Das von der OMG (Object Managment Group) entwickelte CORBA (Common Objekt Request Broker Architecture) realisiert diese Idee verteilter Objekte in einer sehr allgemeinen und portablen Weise. Dabei dürfen sowohl die Rechnerarchitektur, als auch das Betriebssystem, sowie die Programmiersprache auf den unterschiedlichen Maschinen verschieden sein. (*engl.* heterogeneous distributed environment). Wichtigste Eigenschaften von CORBA sind:

- objektorientiert
- transparente Verteilung
- unabhängig von Hardware, Betriebssystem und Programmiersprache
- herstellerunabhängig

Konkurrenzprodukte zur CORBA mit ähnlicher Zielsetzung sind DCOM (Microsoft) und RMI (Java, also nicht sprachunabhängig).

Eine freie Implementierung von CORBA 2.3 ist MICO[1], welches an der Uni Frankfurt für C++ entwickelt wurde und mit dem wir nun die Konzepte von CORBA illustrieren wollen.

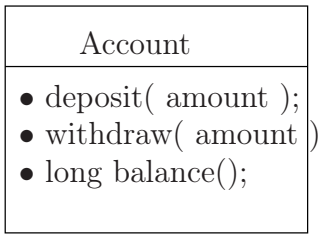


Abbildung 6.2: Account Klasse

Abbildung 6.2 zeigt eine Klasse `Account`. Mittels CORBA wollen wir nun ein Objekt dieser Klasse auf einem entfernten Rechner platzieren und dessen Methoden aufrufen können.

Zunächst sind wie beim RPC alle entfernt aufrufbaren Methoden zu beschreiben. Hier sind das alle Methoden eines Objektes (Puristen würden öffentliche Datenmitglieder ohnehin vermeiden). Da CORBA sprachunabhängig ist wurde dafür eine eigene Sprache, die sogenannte IDL (interface definition language) geschaffen.

Die Idee der Realisierung verteilter Objekte ist in Abbildung Abb. 6.3 skizziert. `Account` wird zunächst als abstrakte Basisklasse realisiert. Davon wird dann einerseits der Stummel `Account_stub` abgeleitet und andererseits `Account_skel`, wovon wieder die eigentliche Implementierung des Kontos `Account_impl` abgeleitet wird. Der Stummel ist ein Stellvertreter des entfernten Objektes auf dem lokalen Rechner und leitet alle Anfragen an das entfernte Objekt weiter. Der Benutzer muss nur die Implementierung das Kontos in `Account_impl` liefern, alle anderen Klassen werden vom IDL-Compiler automatisch generiertt.

Wir gehen nun alle Schritte im Details durch

IDL-Definition erstellen

Die kommt in die Datei `account.idl`:

```
interface Account {
    void deposit( in unsigned long amount );
    void withdraw( in unsigned long amount );
    long balance();
};
```

Automatisches Generieren der Client/Server Klassen

gelingt mittels

```
idl account.idl
```

Das Ergebnis sind die Dateien `account.h` (enthält die Klassendefinitionen) und `account.cc` (Implementierung auf der Client-Seite).

[1] `www.mico.org`

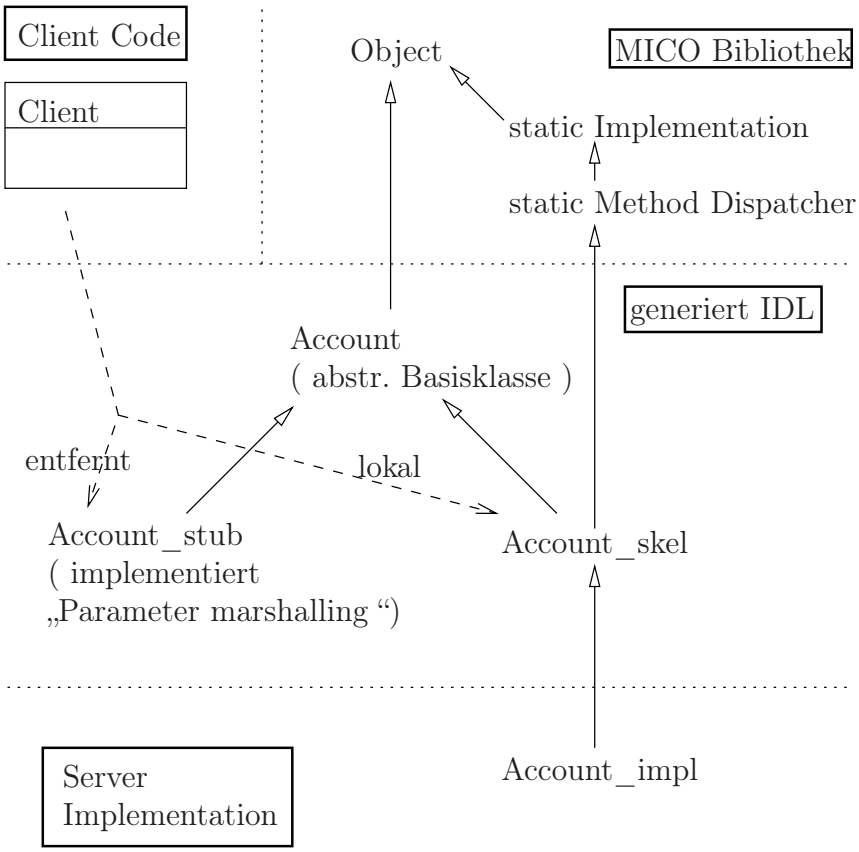


Abbildung 6.3: Account mit IDL

Aufruf auf der Client-Seite

Siehe Datei `client.cc`:

```
#include <CORBA-SMALL.h>
#include <iostream.h>
#include <fstream.h>
#include "account.h"

int main( int argc, char *argv[] )
{
    //ORB initialization
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv,
                    "mico-local-orb" );
    CORBA::BOA_var boa = orb->BOA_init (argc, argv,
                    "mico-local-boa");

    //read stringified object reference
    ifstream in ("account.objid");
    char ref[1000];
    in » ref;
    in.close();

    // client side
    CORBA::Object_var obj = orb->string_to_object(ref);
    assert (!CORBA::is_nil (obj));
    Account_var client = Account::_narrow( obj );

    client->deposit( 100 );
    client->deposit( 100 );
    client->deposit( 100 );
    client->deposit( 100 );
    client->deposit( 100 );
    client->withdraw( 240 );
    client->withdraw( 10 );
    cout « "Balance is " « client->balanca() « endl;

    return 0;
}
```

Serverseite

Server erhält Implementierung der Klasse, Erzeugen eines Objektes und den eigentlichen Server: `server.cc`:

```
#define MICO_CONF_IMR
#include <CORBA-SMALL.h>
#include <iostraem.h>
#include <fstraem.h>
```

```
#include <unistd.h>
#include "account.h"

class Account_impl : virtual public Account_skel {
    CORBA::Long _current_balance;
public:
    Account_impl ()
    {
        _current_balance = 0;
    }
    void deposit( CORBA::ULong amount )
    {
        _current_balance += amount;
    }
    void withdraw( CORBA::ULong amount )
    {
        _current_balance -= amount;
    }
    CORBA::Long balance()
    {
        return _current_balance;
    }
};

int main( int argc, char *argv[] )
{
    cout « "server init" « endl;

    //initialisiere CORBA
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv,
                    "mico-local-orb" );
    CORBA::BOA_var boa = orb->BOA_init (argc, argv,
                    "mico-local-boa");

    // create object, produce global reference
    Account_impl *server = new Account_impl;
    CORBA::String_var ref = orb->object_to_string( server );
    ofstream out ("account.objid");
    out « ref « endl;
    out.close();

    // start server
    boa->impl_is_ready( CORBA::ImplementationDef::_nil() );
    orb->run ();

    CORBA::release( server );
    return 0;
}
```

Test

Zum Start wird wieder der Server gestartet:

```
server &
```

und der Client wird aufgerufen:

```
peter@troll:~/TeX/vorlesung/pr1_ueb/corba > client
Balance is 250
peter@troll:~/TeX/vorlesung/pr1_ueb/corba > client
Balance is 500
peter@troll:~/TeX/vorlesung/pr1_ueb/corba > client
Balance is 750
```

Object-Naming: Hier über „stringified object reference“. Austausch über gemeinsam lesbare Datei, email, etc., ist Global eindeutig und enthält IP-Nummer, Serverprozess, Objekt.

Alternativ: Separate naming services.

Teil III

Algorithmen

7 Grundlagen paralleler Algorithmen

7.1 Parallelisierungsansätze

In diesem Kapitel stellen wir einige grundsätzliche Überlegungen zum Entwurf paralleler Algorithmen an. Dazu unterscheiden wir die folgenden Schritte, denen je ein Unterabschnitt gewidmet wird:

1. *Zerlegen* eines Problems in unabhängige Teilaufgaben. Dies dient der Identifikation des maximal möglichen Parallelismus.
2. Kontrolle der *Granularität* um Rechenaufwand und Kommunikation auszubalancieren.
3. Abbilden der Prozesse auf Prozessoren. Ziel ist eine optimale Abstimmung der logischen Kommunikationsstruktur mit der Maschinenstruktur.

7.1.1 Zerlegen

Ziel des Zerlegungsschrittes ist das Aufzeigen der maximalen Parallelität.

Datenzerlegung

Bei vielen Algorithmen sind die Berechnungen direkt an eine bestimmte Datenstruktur geknüpft. Für jedes Datenobjekt sind gewisse Operationen auszuführen, oftmals ist dieselbe Folge von Operationen auf unterschiedliche Daten anzuwenden. In diesem Fall empfiehlt es sich jedem Prozess einen Teil der Daten zuzuordnen, für die dieser verantwortlich ist.

Als Beispiele für Datenstrukturen betrachten wir eine Matrix und eine Triangulierung wie sie in Abbildung 7.1 dargestellt sind. Bei der Matrixaddition $C = A + B$ können alle Elemente c_{ij} vollkommen parallel bearbeitet werden. In diesem Fall würde man jedem Prozess Π_{ij} die Matrixelemente a_{ij}, b_{ij} und c_{ij} zuordnen. Eine Triangulierung ergibt sich bei der numerischen Lösung partieller Differentialgleichungen. Hier treten Berechnungen pro Dreieck auf, die alle gleichzeitig durchgeführt werden können, jedem Prozess würde man somit ein Dreieck zuordnen.

Datenabhängigkeiten

Oft können die Operationen nicht für alle Datenobjekte gleichzeitig durchgeführt werden können. Als Beispiel dient die Gauß-Seidel-Iteration mit lexikographischer Nummerierung. Diese führt Berechnungen auf einem Gitter durch, wobei die Berechnung am Gitterpunkt (i, j) vom Ergebnis der Berechnungen an den Gitterpunkten $(i-1, j)$ und $(i, j-1)$ abhängt. Der Gitterpunkt $(0, 0)$ kann ohne jede Voraussetzung berechnet werden. Abbildung 7.2 zeigt die Datenabhängigkeiten mit Pfeilen. In diesem Fall können alle Gitterpunkte auf den Diagonalen $i + j = const$ parallel bearbeitet werden. Datenabhängigkeiten machen die Datenzerlegung wesentlich komplizierter.

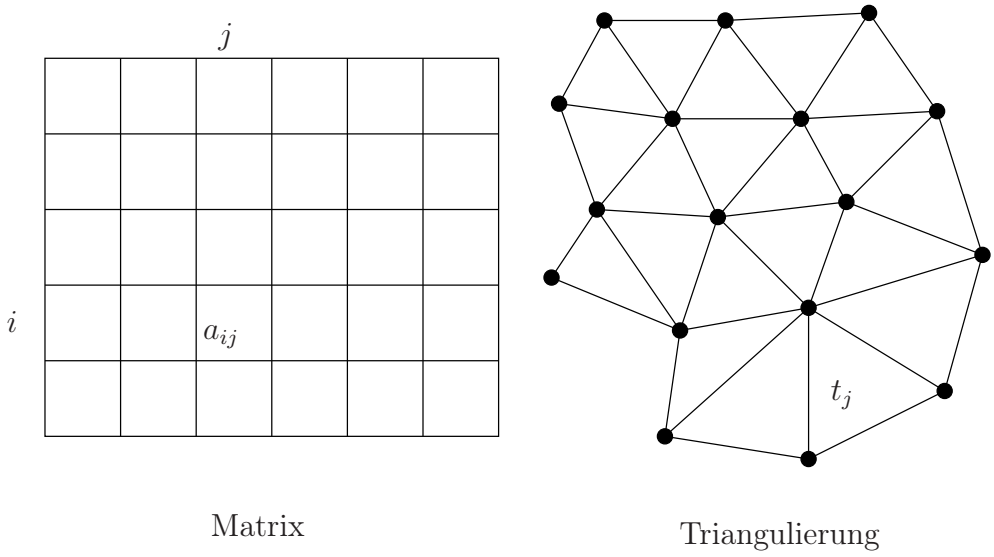


Abbildung 7.1: Beispiele für Datenstrukturen.

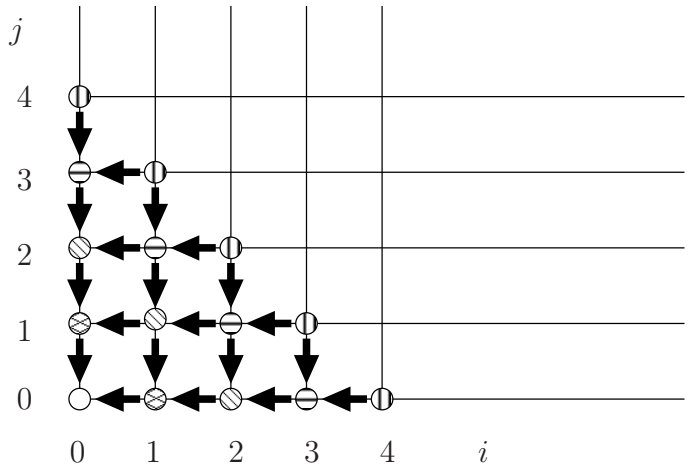


Abbildung 7.2: Datenabhängigkeiten im Gauß–Seidel–Verfahren.

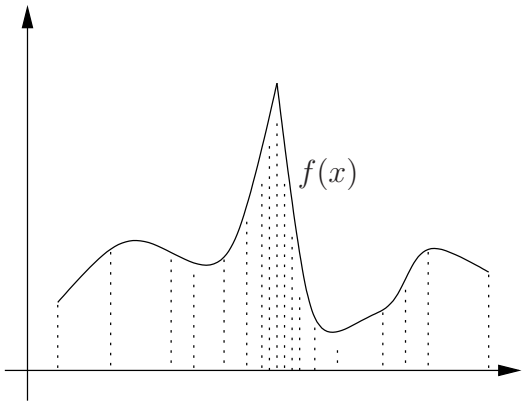


Abbildung 7.3: Zur adaptiven Quadratur.

Funktionale Zerlegung

Müssen unterschiedliche Operationen auf einer Datenmenge ausgeführt werden, so kann man an eine funktionale Zerlegung denken. Als Beispiel sei ein Compiler erwähnt. Dieser besteht aus den Schritten lexikalische Analyse, Parser, Codegenerierung, Optimierung und Assemblierung. Jeder dieser Schritte könnte einem separaten Prozess zugeordnet werden. Die Daten könnten die Stufen in Portionen durchlaufen. In diesem Fall spricht man auch von „Makropipelining“.

Irreguläre Probleme

Bei manchen Problemen kann man die Zerlegung nicht a priori festlegen. Als Beispiel dient die Berechnung des Integrals einer Funktion $f(x)$ durch adaptive Quadratur wie in Abbildung 7.3 angedeutet. Die Wahl der Intervalle hängt von der Funktion f ab und ergibt sich erst während der Berechnung durch Auswertung von Fehlerschätzern.

7.1.2 Agglomeration

Der Zerlegungsschritt zeigt die maximale Parallelität auf. Diese wirklich zu nutzen (im Sinne von ein Datenobjekt pro Prozess) ist meist nicht sinnvoll, da der Kommunikationsaufwand dann überhand nimmt. Deswegen ordnet man einem Prozess mehrere Teilaufgaben zu und fasst die Kommunikation für all diese Teilaufgaben in möglichst wenigen Nachrichten zusammen. Dies bezeichnet man als Agglomeration. Damit reduziert man zumindest die Anzahl der zu sendenden Nachrichten. Weisen die Berechnungen eine gewisse *Datenlokalität* auf, so ergeben sich noch weitere Einsparungen wie unten erläutert wird. Als *Granularität* eines parallelen Algorithmus bezeichnet man das Verhältnis:

$$\text{Granularität} = \frac{\text{Anzahl Nachrichten}}{\text{Rechenzeit}}.$$

Agglomeration reduziert also die Granularität.

Als Beispiel betrachten wir gitterbasierte Berechnungen, siehe Abbildung 7.4. Hierbei können die Berechnungen für alle Gitterpunkte parallel durchgeführt werden. Jedem Prozess wird nun eine ganze Menge von Gitterpunkten zugewiesen, etwa alle schwarzen Gitterpunkte dem Prozess Π in Abbildung 7.4. In vielen solchen Anwendungen werden die Berechnungen *iterativ* durchgeführt, d.h. an jedem Gitterpunkt wird ein Wert gespeichert, der nun modifiziert werden soll.

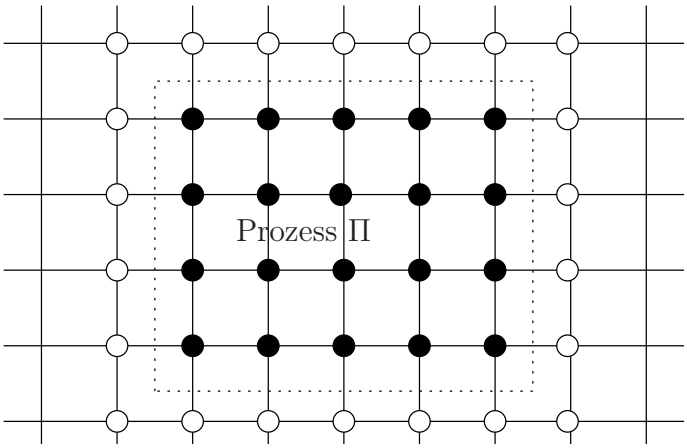


Abbildung 7.4: Gitterbasierte Berechnungen.

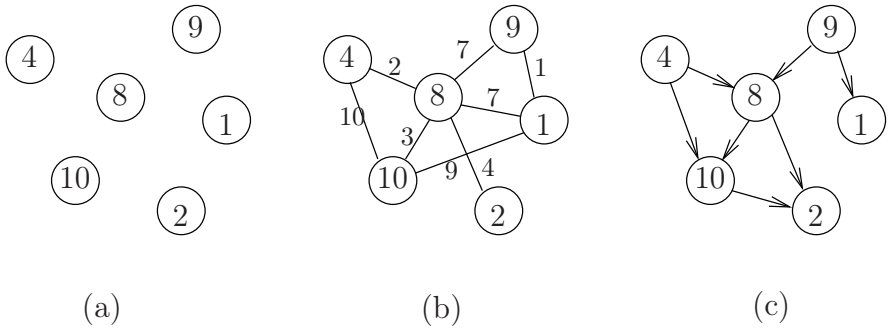


Abbildung 7.5: Verschiedene Agglomerationsprobleme.

Diese Modifikation berechnet sich aus dem aktuellen Wert des Gitterpunktes und dem seiner vier direkten Nachbarpunkte. Alle Modifikationen können gleichzeitig durchgeführt werden. Es bestehen also keine Datenabhängigkeiten wie im Beispiel des Gauß–Seidel–Verfahrens oben.

Durch die Zusammenfassung von Gitterpukten in einem Prozess tritt nun der folgende Effekt ein. Ein Prozess besitze N Gitterpunkte und muss somit $O(N)$ Rechenoperationen ausführen. Falls die Datenaufteilung so gewählt wird wie in Abbildung 7.4 ist jedoch nur für die Gitterpunkte am Rand der Partition, d.h. für solche mit Nachbarn in anderen Prozessen eine Kommunikation notwendig. Es ist also somit nur für insgesamt $4\sqrt{N}$ Gitterpunkte Kommunikation notwendig. Das Verhältnis von Kommunikationsaufwand zu Rechenaufwand verhält sich somit wie $O(N^{-1/2})$ und wir können durch erhöhen der Anzahl der Gitterpunkte pro Prozessor den Aufwand für die Kommunikation relativ zur Rechnung beliebig klein machen. Dies bezeichnet man als *Oberfläche–zu–Volumen–Effekt*. Er setzt voraus, dass die Berechnung *Datenlokalität* besitzt und die Agglomeration miteinander kommunizierende Teilaufgaben möglichst in den selben Prozessor packt.

Nun geht es darum wie die Agglomeration durchzuführen ist. Wir stellen hier nur die Problematik anhand dreier verschiedener Modelle vor, die in Abbildung 7.5 schematisch dargestellt sind. Die eigentlichen Methoden zur Agglomeration werden in Abschnitt 7.2 besprochen.

(a) Ungekoppelte Berechnungen

Im ersten Fall besteht die Berechnung aus Teilproblemen, die völlig unabhängig voneinander berechnet werden können. Jedes Teilproblem kann jedoch eine andere Rechenzeit erfordern. Dies ist in Abbildung 7.5(a) durch eine Menge von Knoten mit Gewichten dargestellt. Die Gewichte sind ein Maß für die erforderliche Rechenzeit.

Hier ist die Agglomeration sehr einfach. Man weist die Knoten der Reihe nach (z.B. der Größe nach geordnet oder zufällig) jeweils dem Prozess zu der am wenigsten Arbeit hat (dies ist die Summe all seiner Knotengewichte).

Dieser Fall wird komplizierter wenn die Anzahl der Knoten sich erst während der Berechnung ergibt (wie bei der adaptiven Quadratur) und/oder die Knotengewichte a priori nicht bekannt sind (wie z.B. bei depth first search). In diesem Fall ist eine dynamische Lastverteilung erforderlich, die entweder zentral (ein Prozess nimmt die Lastverteilung vor) oder dezentral (ein Prozess holt sich Arbeit von anderen, die zuviel haben) sein kann.

(b) Gekoppelte Berechnungen

Dies ist das Standardmodell für statische, datenlokale Berechnungen. Die Berechnung wird in diesem Fall durch einen ungerichteten Graphen beschrieben. Ein Beispiel zeigt Abbildung 7.5(b). Hier ist die Vorstellung, dass in einem ersten Schritt eine Berechnung pro Knoten erforderlich ist, deren Länge vom Knotengewicht modelliert wird. In einem zweiten Schritt tauscht jeder Knoten Daten mit seinen Nachbarknoten aus. Die Anzahl der zu sendenden Daten ist proportional zum jeweiligen Kantengewicht.

Ist der Graph regelmäßig (und die Gewichte konstant), wie in Abbildung 7.4, so ist die Agglomeration trivial. Im Fall eines allgemeinen Graphen $G = (V, E)$ und P Prozessoren ist die Knotenmenge V so zu partititonieren, dass einerseits

$$\bigcup_{i=1}^{P} V_i = V, \quad V_i \cap V_j = \emptyset, \quad \sum_{v \in V_i} g(v) = \sum_{v \in V} g(v)/|V|$$

und andererseits die Separatorkosten

$$\sum_{(v,w) \in S} g(v,w) \to \min, \; S = \{(v,w) \in E | v \in V_i, w \in V_j, i \neq j\}$$

minimal sind. Dieses Problem wird als *Graphpartitionierungsproblem* bezeichnet und ist schon im Fall konstanter Gewichte und $P = 2$ (Graphbisektion) $\mathcal{NP}$–vollständig. Es gibt aber eine ganze Reihe guter Heuristiken, die in linearer Zeit (in der Anzahl der Knoten, $O(1)$ Nachbarn) eine (ausreichend) gute Partitionierung finden können.

(c) Gekoppelte Berechnungen mit zeitlicher Abhängigkeit

Hier ist das Modell ein gerichteter Graph wie in Abbildung 7.5(c) dargestellt. Die Vorstellung dahinter ist, dass ein Knoten erst berechnet werden kann wenn alle Knoten, die über eingehende Kanten erreicht werden bereits berechnet sind. Ist ein Knoten berechnet (Rechenzeit durch Knotengewicht gegeben) so wird das Ergebnis über die ausgehenden Kanten weitergegeben (Kommunikationszeit gegeben durch die Kantengewichte).

Dieses Problem ist im allgemeinen sehr schwer zu lösen. Theoretisch ist es zwar nicht „schwieriger als Graphpartitionierung“ (beide sind $\mathcal{NP}$–vollständig) aber es sind keine einfachen und guten Heuristiken bekannt. Für spezielle Probleme dieser Art, wie sie etwa z.B. in adaptiven Mehrgitterverfahren auftreten, kann man jedoch ganz gute Heuristiken finden.

7.1.3 Abbilden der Prozesse auf Prozessoren

Die Menge der Prozesse Π bildet einen ungerichteten Graphen $G_\Pi = (\Pi, K)$. Dabei sind zwei Prozesse miteinander verbunden, falls sie miteinander kommunizieren (man könnte auch noch Kantengewichte einführen, die modellieren wieviel sie kommunizieren).

Andererseits haben wir die Menge der Prozessoren P, die zusammen mit dem Kommunikationsnetzwerk (Hypercube, Feld, etc.) einen Graphen $G_P = (P, N)$ bilden. Wir nehmen an, dass $|\Pi| = |P|$ und stehen nun vor der Frage welcher Prozess auf welchem Prozessor ausgeführt werden soll. Im allgemeinen wollen wir die Abbildung so durchführen, dass Prozesse, die miteinander kommunizieren möglichst auf (nahe) benachbarte Prozessoren abgebildet werden. Dieses Optimierungsproblem bezeichnet man als *Graphabbildungsproblem* und auch dieses ist (leider wieder) $\mathcal{NP}$–vollständig.

Heutige Multiprozessoren besitzen sehr leistungsfähige Kommunikationsnetzwerke. Insbesondere ist in cut-through-Netzwerken die Übertragungszeit einer Nachricht praktisch entfernungsunabhängig. Diese Tatsache hat heute das Problem der optimalen Prozessplatzierung etwas in den Hintergrund gedrängt. Eine gute Platzierung der Prozesse ist jedoch wichtig falls viele Prozesse *gleichzeitig* miteinander kommunizieren wollen (was aber durchaus häufig vorkommt!).

In den unten beschriebenen Algorithmen ist die logische Kommunikationsstruktur meist identisch mit der Maschinenstruktur, so dass wir auf dieses Problem nicht weiter eingehen.

7.2 Lastverteilung

Wir untersuchen in diesem Kapitel die Aufgabe, die anfallende Arbeit auf die verschiedenen Prozessoren aufzuteilen. Dies entspricht dem Agglomerationsschritt vom vorhergehenden Abschnitt bei dem man die parallel abarbeitbaren Teilprobleme wieder zusammengefasst hat.

7.2.1 Statische Verteilung ungekoppelter Probleme

Wir machen die Annahme, daß das Maß für die Arbeit bekannt ist.

Bin Packing

Am Anfang sind alle Prozessoren leer. Die Knoten, die in beliebiger Reihenfolge oder sortiert (z.B. der Grösse nach) vorliegen, werden nacheinander jeweils auf den Prozessor gepackt, der (gerade) am wenigsten Arbeit hat. Das funktioniert auch dynamisch, wenn während des Rechenvorganges neue Arbeit entsteht.

Rekursive Bisektion

Hierbei wird die zusätzliche Annahme gemacht, daß die Knoten eine Position im Raum besitzen. Der Raum wird jetzt so geteilt, daß in den Teilen etwa gleich viel Arbeit besteht. Dieses Vorgehen wird dann rekursiv auf die entstandenen Teilräume angewandt.

7.2.2 Dynamische Verteilung ungekoppelter Probleme

Nun sei das Maß für die Arbeit nicht bekannt. Ein Prozess ist entweder aktiv (hat Arbeit) oder frei (hat keine Arbeit). Die Arbeit kann geteilt werden (falls es sich lohnt).

Die Aktivitäten und Zustände eines Prozesses zeigt die Abb. 7.7

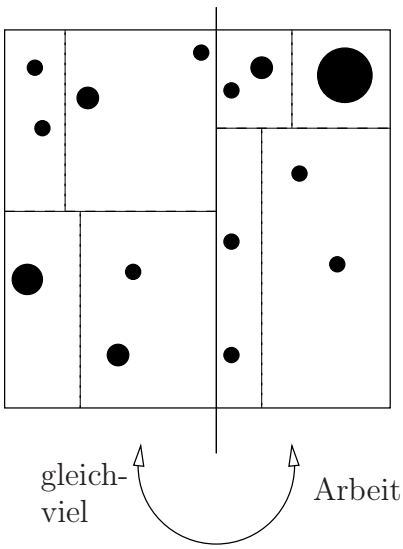


Abbildung 7.6: Rekursive Bisektion

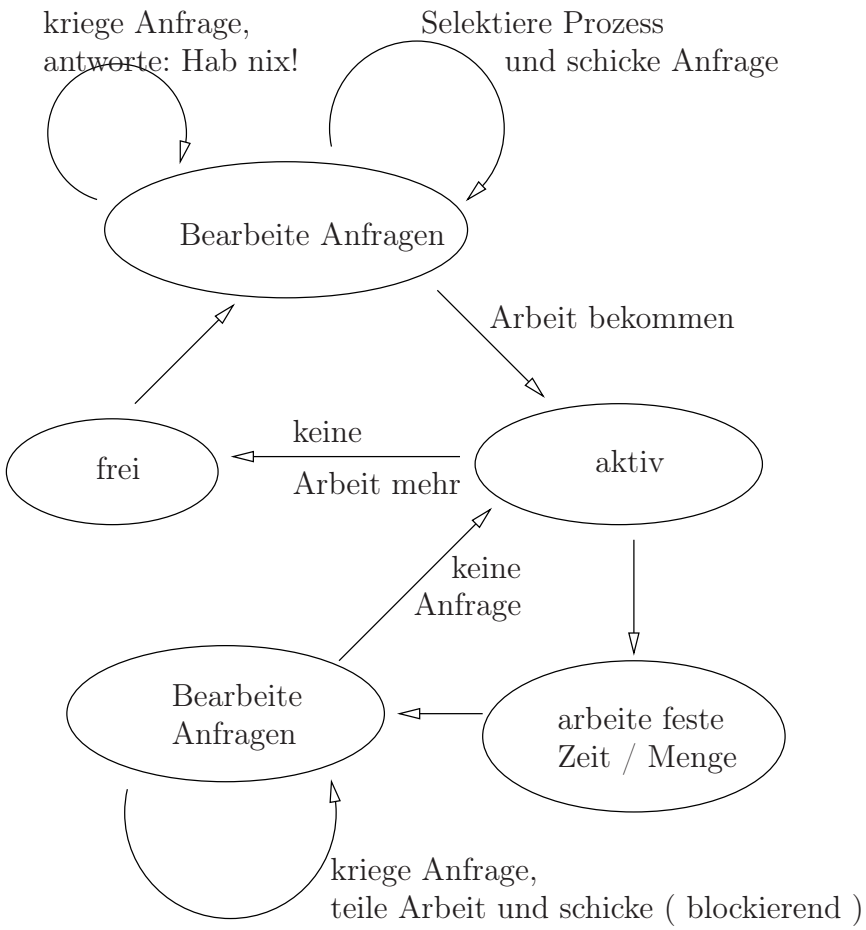


Abbildung 7.7: Aktivitäts-/Zustandsdiagramm

Beim Teilen der Arbeit sind folgende Fragen zu berücksichtigen: Was will ich abgeben (beim Travelling-Salesman Problem z.B. möglichst Knoten aus dem Stack, die weit unten liegen) und wieviel will ich abgeben (z.B. die Hälfte der Arbeit (half split))?

Zusätzlich zur Arbeitsverteilung kann noch weitere Kommunikation stattfinden (z.B. die Bildung des globalen Minimums bei branch-and-bound beim Travelling-Salesman). Außerdem kommt noch das Problem der Terminierungserkennung hinzu. Wann sind alle Prozesse idle?

Für die Frage, welcher Prozess auf der Suche nach Arbeit als nächster angesprochen wird, betrachten wir verschiedene Selektionsstrategien:

- **Master/Slave(Worker) Prinzip**

  Ein Prozess wird abgestellt, um die Arbeit zu verteilen. Er weiß, wer aktiv bzw. frei ist und leitet die Anfrage weiter. Er regelt (da er weiß, wer frei ist) auch das Terminierungsproblem. Der Nachteil dieser Methode ist, daß sie nicht skaliert. Als Alternative könnte man eine hierarchische Struktur von Mastern einfügen.

- **Asynchrones Round Robin**

  Der Prozess Π_i hat eine Variable target_i. Er schickt seine Anfrage an Π_{target_i} und setzt dann $\text{target}_i = (\text{target}_i + 1)\%P$.

- **Globales Round Robin**

  Es gibt nur eine globale Variable target. Daran ist positiv, daß keine gleichzeitigen Anfragen an denselben Prozess gestellt werden können. Negativ ist jedoch der nötige Zugriff auf eine globale Variable (was z.B. ein Server Prozess machen kann).

- **Random Polling**

  Jeder wählt zufällig einen Prozess mit gleicher Wahrscheinlichkeit (→ Paralleler Zufallsgenerator, achte zumindest auf das Austeilen von seeds o.Ä.). Dieses Vorgehen bietet eine gleichmäßige Verteilung der Anfragen und benötigt keine globale Resource.

7.2.3 Graphpartitionierung

Betrachten wir ein Finite-Elemente-Netz :

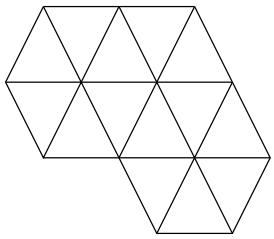


Abbildung 7.8: Finite-Elemente-Netz

Es besteht aus Dreiecken $T = \{t_1, \ldots, t_N\}$ mit

$$\bar{t_i} \cap \bar{t_j} = \begin{cases} \emptyset \\ \text{ein Knoten} \\ \text{eine Kante} \end{cases}$$

Wir haben also einen Graphen. Die Arbeit findet beim Verfahren der Finiten Elemente in den Knoten statt. Alternativ kann man auch in jedes Dreieck einen Knoten legen, diese mit Kanten verbinden und den so entstehenden Dualgraphen betrachten.

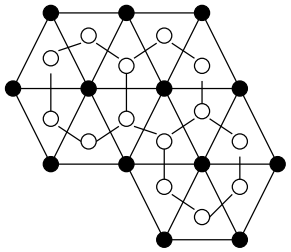


Abbildung 7.9: Graph und Dualgraph

Die Aufteilung des Graphen auf Prozessoren führt zum Graphpartitionierungsproblem. Dazu machen wir die folgenden Notationen:

$$\begin{aligned} G &= (V, E) && \text{(Graph oder Dualgraph)} \\ E &\subseteq V \times V && \text{symmetrisch (ungerichtet)} \end{aligned}$$

Die Gewichtsfunktionen

$$\begin{aligned} w &: V \longrightarrow \mathbb{N} && \text{(Rechenaufwand)} \\ w &: E \longrightarrow \mathbb{N} && \text{(Kommunikation)} \end{aligned}$$

Die Gesamtarbeit

$$W = \sum_{v \in V} w(v)$$

Außerdem sei k die Anzahl der zu bildenden Partitionen, wobei $k \in \mathbb{N}$ und $k \geq 2$ sei. Gesucht ist nun eine Partitionsabbildung

$$\pi : V \longrightarrow \{0, \ldots, k-1\}$$

und der dazugehörige Kantenseparator

$$X_\pi := \{(v, v') \in E \mid \pi(v) \neq \pi(v')\} \subseteq E$$

Das Graphpartitionierungproblem besteht nun darin, die Abbildung π so zu finden, daß das Kostenfunktional (Kommunikationskosten)

$$\sum_{e \in X_\pi} w(e) \rightarrow \min$$

minimal wird unter der Nebenbedingung (Gleichverteilung der Arbeit)

$$\sum_{v, \pi(v)=i} w(v) \leq \delta \frac{W}{k} \quad \text{für alle } i \in \{0, \ldots, k-1\}$$

wobei δ das erlaubte Ungleichgewicht bestimmt (z.B. $\delta = 1.1$ für erlaubte 10% Abweichung vom Durchschnittswert).

Wir machen hierbei die Annahme, daß die Berechnung dominiert, weil sonst wegen der hohen Kommunikationskosten evtl. die Partitionierung unnötig wäre. Das ist allerdings nur ein Modell für die Laufzeit!

Für den Fall k = 2 spricht man vom Graphbisektionsproblem. Durch rekursive Bisektion lassen sich 2^d-Wege-Partitionierungen erzeugen.

Problematischerweise ist die Graphpartitionierung für k $\geq$ 2 NP-vollständig. Eine optimale Lösung würde also normalerweise die eigentliche Rechnung dominieren ($\rightarrow$ paralleler Overhead) und ist daher nicht annehmbar, weshalb man schnelle Heuristiken benötigt.

Abb. 7.10 illustriert die Partitionierung eines Finite-Elemente-Netzes

Rekursive Koordinatenbisektion(RCB)

Dazu benötigt man die Positionen der Knoten im Raum (bei Finite-Elemente Anwendungen sind die vorhanden).

Weiter oben haben wir das Verfahren schon unter dem Namen **Rekursive Bisektion** gesehen. Diesmal haben wir aber ein gekoppeltes Problem vorliegen. Daher ist es wichtig, daß der Raum, in dessen Koordinaten die Bisektion durchgeführt wird, mit dem Raum, in dem die Knoten liegen, übereinstimmt. In der Abb. 7.11 ist das nicht der Fall. Zwei Knoten mögen zwar räumlich dicht beieinanderliegen, eine Koordinatenbisektion macht aber keinen Sinn, da die Punkte hier nicht gekoppelt sind, es also einem Prozessor gar nichts nützt, beide zu speichern.

Rekursive Spektralbisektion(RSB)

Hier werden die Positionen der Knoten im Raum nicht benötigt. Man stellt zunächt die Laplacematrix A(G) zum vorliegenden Graphen G auf. Diese ist folgendermaßen definiert:

$$A(G) = \{a_{ij}\}_{i,j=1}^{|V|} \quad \text{mit} \quad a_{ij} = \begin{cases} \text{grad}(v_i) & i = j \\ -1 & (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

Dann löse das Eigenwertproblem

$$Ae = \lambda e$$

Der kleinste Eigenwert λ_1 ist gleich Null, denn mit $e_1 = (1, \ldots, 1)^T$ gilt $Ae_1 = 0 \cdot e_1$. Der zweitkleinste Eigenwert λ_2 allerdings ist ungleich Null, falls der Graph zusammenhängend ist.

Die Bisektion findet nun anhand der Komponenten des Eigenvektors e_2 statt, und zwar setzt man für $c \in \mathbb{R}$ die beiden Indizeemengen

$$I_0 = \{i \in \{1, \ldots, |V|\} \mid (e_2)_i \leq c\}$$
$$I_1 = \{i \in \{1, \ldots, |V|\} \mid (e_2)_i > c\}$$

und die Partitionsabbildung

$$\pi(v) = \begin{cases} 0 & \text{falls } v = v_i \wedge i \in I_0 \\ 1 & \text{falls } v = v_i \wedge i \in I_1 \end{cases}$$

Dabei wählt man das c so, daß die Arbeit gleichverteilt ist.

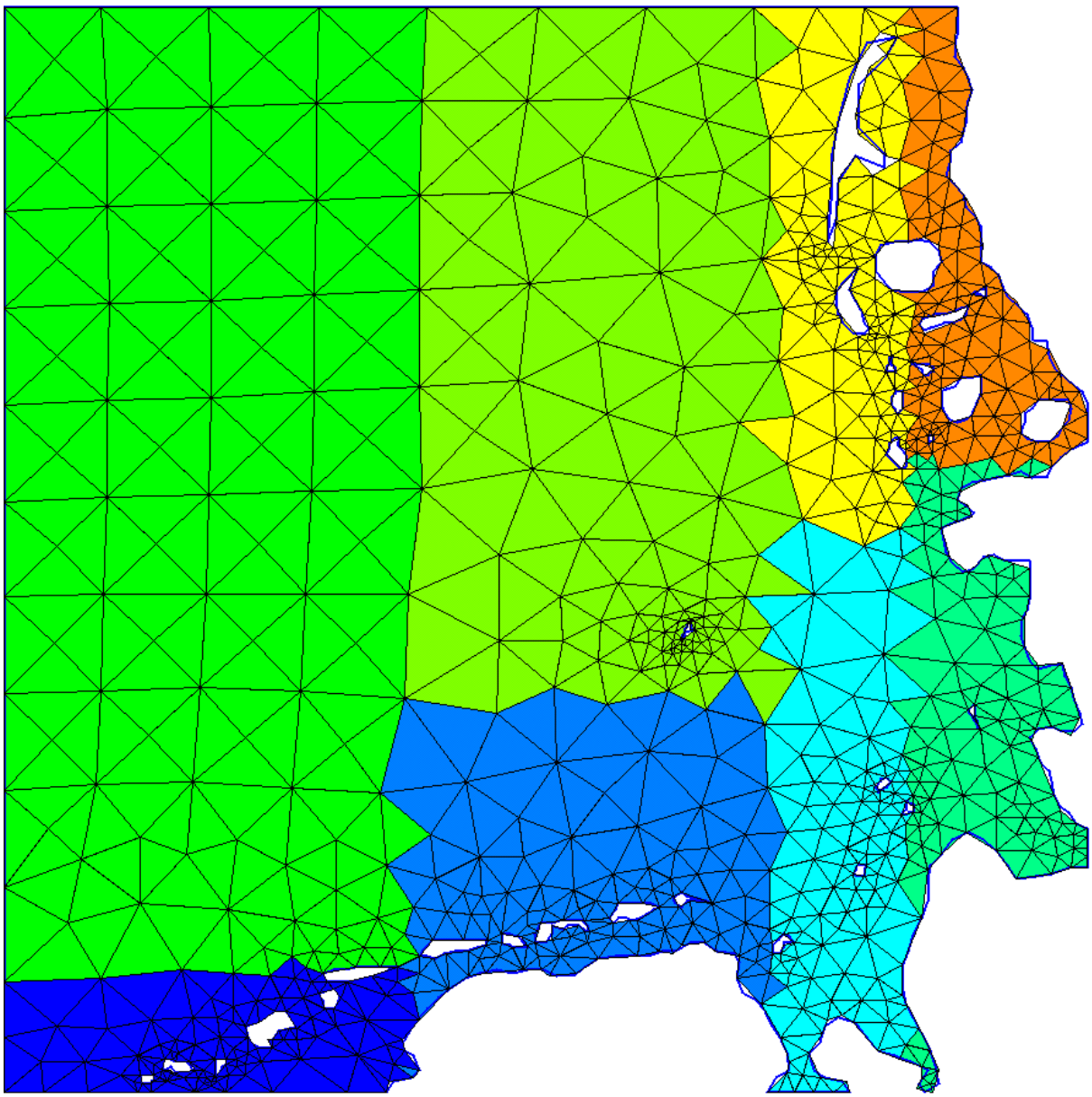


Abbildung 7.10: Nordsee mit FE-Netz

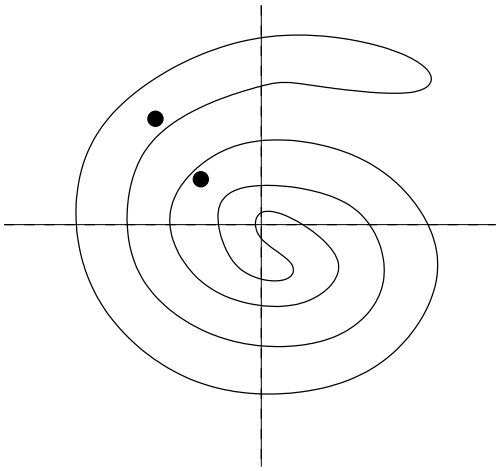
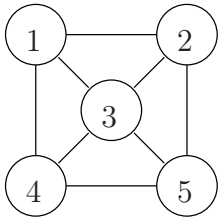


Abbildung 7.11: Gegenbeispiel zur Anwendbarkeit der RCB



	1	2	3	4	5
1	3	-1	-1	-1	0
2	-1	3	-1	0	-1
3	-1	-1	4	-1	-1
4	-1	0	-1	3	-1
5	0	-1	-1	-1	3

Abbildung 7.12: Graph und zugehörige Laplacematrix

Kerninghan/Lin

Dabei handelt es sich um ein Iterationsverfahren, daß eine vorgegebene Partition unter Berücksichtigung des Kostenfunktionals verbessert.

Wir betrachten hier nur die Bisektion (k-Wege Erweiterung möglich) und nehmen die Knotengewichte als 1 an. Außerdem sei die Anzahl der Knoten gerade.

i = 0;
$| V |$ = 2n;
// Erzeuge Π_0 so, dass Gleichverteilung erfüllt ist.
while (1) { // Iterationsschritt
    $V_0 = \{v \mid \pi(v) = 0\}$;
    $V_1 = \{v \mid \pi(v) = 1\}$;
    $V_0' = V_1' = \emptyset$;
    $\bar{V_0} = V_0$;
    $\bar{V_1} = V_1$;
    for (i = 1 ; i ≤ n ; i++)
    {
        // Wähle $v_i \in V_0 \setminus V_0'$ und $w_i \in V_1 \setminus V_1'$ so, dass
        $\sum_{e \in (\bar{V_0} \times \bar{V_1}) \cap E} w(e) - \sum_{e \in (V_0'' \times V_1'') \cap E} w(e) \rightarrow max$
        // wobei
        $V_0'' = \bar{V_0} \setminus \{v_i\} \cup \{w_i\}$
        $V_1'' = \bar{V_1} \setminus \{w_i\} \cup \{v_i\}$
        // setze
        $\bar{V_0} = \bar{V_0} \setminus \{v_i\} \cup \{w_i\}$;
        $\bar{V_1} = \bar{V_1} \setminus \{w_i\} \cup \{v_i\}$;
    } // for
    // Bem.: max kann negativ sein, d.h. Verschlechterung der Separatorkosten
    // Ergebnis an dieser Stelle: Folge von Paaren $\{(v_1, w_1), \ldots, (v_n, w_n)\}$.
    // V_0, V_1 wurden noch nicht verändert.
    // Wähle nun eine Teilfolge bis m ≤ n, die eine maximale
    // Verbesserung der Kosten bewirkt („hill climbing “)
    $V_0 = V_0 \setminus \{v_1, \ldots, v_m\} \cup \{w_1, \ldots, w_m\}$;
    $V_1 = V_1 \setminus \{w_1, \ldots, w_m\} \cup \{v_1, \ldots, v_m\}$;
    if (m == 0) break; // Ende
} // while

Das Verfahren von Kerningham/Lin wird meist in Kombination mit anderen Verfahren benutzt.

Multilevel k-Wege Partitionierung

Das Prinzip besteht darin, Knoten des Ausgangsgraphen G^0 (z.B. zufällig oder aufgrund schwerer Kantengewichte) in Clustern zusammenzufassen und diese wiederum als Knoten in einem vergröberten Graphen G^1 anzusehen. Rekursiv angewandt führt dieses Verfahren auf einen Graphen G^l.

G^l wird nun partitioniert (z.B. RSB / KL) und anschließend die Partitionsfunktion auf dem feineren Graphen G^{l-1} interpoliert. Diese interpolierte Partitionsfunktion kann jetzt wieder

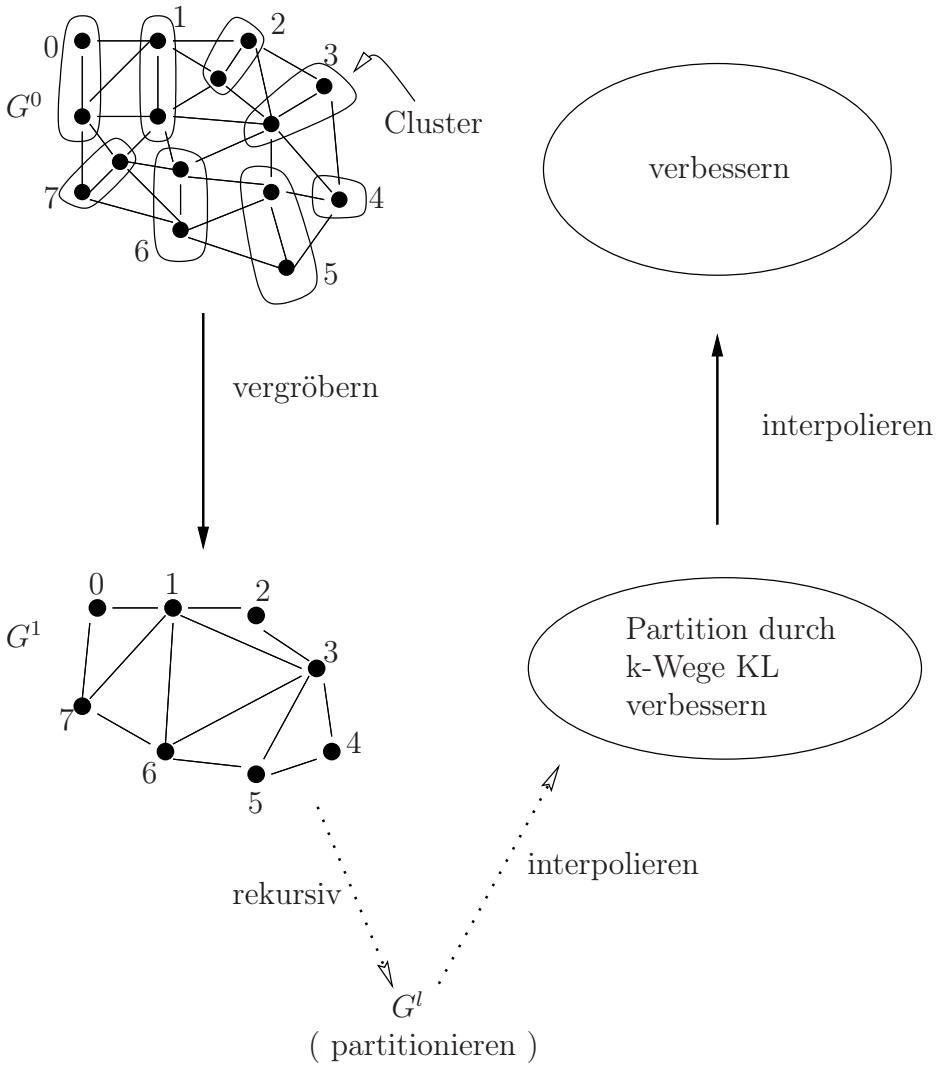


Abbildung 7.13: Multilevel-k-Wege-Partitionierung

mittels KL rekursiv verbessert und anschließend auf dem nächst feineren Graphen interpoliert werden. So verfährt man rekursiv, bis man wieder beim Ausgangsgraphen ankommt. Die Implementierung ist dabei in $\mathcal{O}$(n) Schritten möglich. Das Verfahren liefert qualitativ hochwertige Partitionierungen.

Weitere Probleme

Weitere Probleme bei der Partitionierung sind

- **Dynamische Repartitionierung:** Der Graph soll mit möglichst wenig Umverteilung lokal abgeändert werden.
- **constraint Partitionierung:** Aus anderen Algorithmenteilen sind zusätzliche Datenabhängigkeiten vorhanden.
- **Parallelisierung des Partitionsverfahrens:** Ist nötig bei großen Datenmengen (Dafür gibt es fertige Software)

7.3 Analyse paralleler Algorithmen

In diesem Abschnitt beschreiben wir wie die Leistungsfähigkeit paralleler Algorithmen untersucht werden kann.

Wir nehmen an wir wollen ein Problem Π parallel lösen. Das Problem Π ist parametrisiert durch die *Problemgröße* N, die frei gewählt werden kann.

Wir lösen das Problem Π mit einem sequentiellen bzw. einem parallelen Algorithmus, die miteinander verglichen werden sollen. Parallele Algorithmen können nicht unabhängig von der Hardware betrachtet werden auf der sie ausgeführt werden. Wir nehmen an, daß der zur Verfügung stehende Parallelrechner vom MIMD–Typ ist und aus P identischen Rechenknoten besteht. Der sequentielle Algorithmus wird immer auf einem dieser Knoten ausgeführt. Weiterhin verfügt der Parallelrechner über ein mitwachsendes Kommunikationsnetzwerk. Als Hardwareparameter sind Aufsetzzeiten und Übertragungsrate des Kommunikationsnetzwerkes sowie die Zeiten für elementare Rechenoperationen bekannt.

Eine Kombination von parallelem Algorithmus und paralleler Hardware auf der dieser ausgeführt wird bezeichnen wir als *paralleles System*. Unter der *Skalierbarkeit* eines parallelen Systems verstehen wir dessen Fähigkeit wachsende Resourcen, wie Prozessoren und Speicher, zu nutzen. Das Ziel dieses Abschnittes ist es den Begriff der Skalierbarkeit zu quantifizieren und verschiedene Skalierbarkeitsansätze vorzustellen. Skalierbarkeitsuntersuchungen ermöglichen es verschiedene parallele Systeme miteinander zu vergleichen und Leistungsvorhersagen für große Prozessorzahlen zu treffen.

7.3.1 Maße paralleler Algorithmen

Laufzeiten

Unter der Zeit $T_S(N)$ verstehen wir die Laufzeit eines bestimmten sequentiellen Algorithmus zur Lösung des Problems Π bei Eingabegröße N auf einem Knoten des Parallelrechners. So könnte das Problem Π etwa aus der Lösung eines linearen Gleichungssystemes der Größe $N \times N$ bestehen und $T_S(N)$ könnte die Laufzeit des Gaußschen Eliminationsverfahrens sein.

Unter der Zeit $T_{best}(N)$ verstehen wir die Laufzeit des *besten* (bekannten) sequentiellen Algorithmus zur Lösung des Problems Π bei Eingabegröße N. Wichtig ist, daß $T_{best}(N)$ die Laufzeit

eines sequentiellen Algorithmus darstellt der alle Probleme Π (z.B. lineare Gleichungssysteme) für fast alle Eingabegrößen N in der kürzesten Zeit löst.

Die Laufzeit $T_P(N, P)$ beschreibt die Laufzeit des zu untersuchenden parallelen Systems zur Lösung von Π in Abhängigkeit der Eingabegröße N *und* der Prozessorzahl P.

Speedup und Effizienz

Mit den Laufzeiten von oben definieren wir nun den Speedup S als

$$S(N, P) = \frac{T_{best}(N)}{T_P(N, P)}. \tag{7.1}$$

Es gilt für alle N und P, daß $S(N, P) \leq P$. Denn angenommen es sei $S(N, P) > P$, so könnten wir ein sequentielles Programm konstruieren, welches das parallele Programm simuliert (im Zeitscheibenbetrieb abarbeitet). Dieses hypothetische Programm hätte dann die Laufzeit $PT_P(N, P)$ und es wäre

$$PT_P(N, P) = P\frac{T_{best}(N)}{S(N, P)} < T_{best}(N). \tag{7.2}$$

Das neue Programm wäre also schneller als das beste sequentielle Programm was offensichtlich ein Widerspruch ist.

Die Effizienz E wird definiert als

$$E(N, P) = \frac{T_{best}(N)}{PT_P(N, P)}. \tag{7.3}$$

und es gilt $E(N, P) \leq 1$. Die Effizienz gibt den Anteil des maximalen Speedup der erreicht wird. Man kann auch sagen, daß $E \cdot P$ Prozessoren wirklich an der Lösung von Π arbeiten und der Rest $(1 - E)P$ nicht effektiv zur Problemlösung beiträgt.

Kosten

Als Kosten C definiert man das Produkt

$$C(N, P) = PT_P(N, P), \tag{7.4}$$

denn für soviel Rechenzeit muss man im Rechenzentrum bezahlen. Man bezeichnet einen Algorithmus als *kostenoptimal* falls $C(N, P) = const T_{best}(N)$. Offensichtlich gilt dann

$$E(N, P) = \frac{T_{best}(N)}{C(N, P)} = 1/const, \tag{7.5}$$

die Effizienz bleibt also konstant.

Parallelitätsgrad

Schließlich bezeichnen wir mit $\Gamma(N)$ noch den sog. *Parallelitätsgrad*, das ist die maximale Zahl gleichzeitig ausführbarer Operationen in dem besten sequentiellen Algorithmus. Offensichtlich könnten prinzipiell umso mehr Operationen parallel ausgeführt werden, je mehr Operationen überhaupt auszuführen sind, je größer also N ist. Der Parallelitätsgrad ist also von N abhängig.

Andererseits kann der Parallelitätsgrad nicht größer sein als die Zahl der insgesamt auszuführenden Operationen. Da diese Zahl proportional zu $T_S(N)$ ist, können wir sagen, dass

$$\Gamma(N) \leq O(T_S(N)) \tag{7.6}$$

gilt.

7.3.2 Skalierter Speedup

Wir wollen nun untersuchen wie sich der Speedup $S(N, P)$ eines parallelen Systems in Abhängigkeit von P verhält. Dabei stellt sich sofort die Frage wie der zweite Parameter N zu wählen ist. Dazu gibt es verschiedene Möglichkeiten.

Feste sequentielle Ausführungszeit

Hier bestimmen wir N aus der Beziehung

$$T_{best}(N) \stackrel{!}{=} T_{fix} \quad \rightarrow \quad N = N_A \tag{7.7}$$

wobei T_{fix} ein Parameter ist. Der skalierte Speedup ist dann

$$S_A(P) = S(N_A, P), \tag{7.8}$$

dabei steht A für den Namen Amdahl (siehe unten).

Wie verhält sich dieser skalierte Speedup? Dazu nehmen wir an, dass das parallele Programm aus dem besten sequentiellen Programm dadurch entsteht, dass ein Teil $0 < q < 1$ weiter sequentiell bearbeitet wird und der Rest $(1 - q)$ vollständig parallelisiert werden kann. Die parallele Laufzeit (für das feste N_A!) ist also $T_P = qT_{fix} + (1 - q)T_{fix}/P$. Für den Speedup gilt dann

$$S(P) = \frac{T_{fix}}{qT_{fix} + (1 - q)T_{fix}/P} = \frac{1}{q + \frac{1-q}{P}} \tag{7.9}$$

und somit $\lim_{P\to\infty} S(P) = 1/q$. Der maximal erreichbare Speedup wird also rein durch den sequentiellen Anteil bestimmt. Zudem sinkt die Effizienz stark ab wenn man nahe an den maximalen Speedup herankommen will. Diese Erkenntnis wird als „Gesetz von Amdahl“ bezeichnet und hat Ende der 60er Jahre zu einer sehr pessimistischen Einschätzung der Möglichkeiten des parallelen Rechnens geführt, siehe (Amdahl 1967). Dies hat sich erst geändert als man erkannt hat, dass für die meisten parallelen Algorithmen der sequentielle Anteil q mit steigendem N *abnimmt*. Der Ausweg aus dem Dilemma besteht also darin mit mehr Prozessoren immer größere Probleme zu lösen!

Wir stellen nun drei Ansätze vor wie man N mit P wachsen lassen kann.

Feste parallele Ausführungszeit

Hier bestimmen wir N aus der Gleichung

$$T_P(N, P) \stackrel{!}{=} T_{fix} \quad \rightarrow \quad N = N_G(P) \tag{7.10}$$

für gegebenes T_{fix} und betrachten dann den Speedup

$$S_G(P) = S(N_G(P), P). \tag{7.11}$$

Diese Art der Skalierung wird auch „Gustafson scaling“ genannt, siehe (Gustafson 1988). Als Motivation dienen Anwendungen wie etwa die Wettervorhersage. Hier hat man eine bestimmte Zeit T_{fix} zur Verfügung in der man ein möglichst großes Problem lösen will.

Fester Speicherbedarf pro Prozessor

Viele Simulationsanwendungen sind speicherbeschränkt, d.h. der Speicherbedarf wächst mit N wie $M(N)$ an und oft bestimmt nicht die Rechenzeit sondern der Speicherbedarf welche Probleme man auf einer Maschine noch berechnen kann. Nehmen wir an, dass der Parallelrecher aus P identischen Prozessoren besteht, die jeweils einen Speicher der Grösse M_0 besitzen, so bietet sich die Skalierung

$$M(N) \stackrel{!}{=} PM_0 \quad \rightarrow \quad N = N_M(P) \tag{7.12}$$

an und wir betrachten

$$S_M(P) = S(N_M(P), P). \tag{7.13}$$

als skalierten Speedup.

Konstante Effizienz

Als letzte Möglichkeit der Skalierung wollen wir N so wählen, dass die parallele Effizienz konstant bleibt, d.h. wir fordern

$$E(N, P) \stackrel{!}{=} E_0 \quad \rightarrow \quad N = N_I(P). \tag{7.14}$$

Dies bezeichnet man als *isoeffiziente Skalierung*. Offensichtlich ist $E(N_I(P), P) = E_0$ also

$$S_I(P) = S(N_I(P), P) = PE_0. \tag{7.15}$$

Wie wir später sehen werden ist eine isoeffiziente Skalierung nicht für alle parallelen Systeme möglich, d.h. man findet nicht unbedingt eine Funktion $N_I(P)$, die (7.14) identisch erfüllt. Somit kann man andererseits vereinbaren, dass ein System skalierbar ist genau dann wenn eine solche Funktion gefunden werden kann.

Um diese Begriffe etwas zu vertiefen betrachten wir ein

Beispiel

Es sollen N Zahlen auf einem Hypercube mit P Prozessoren addiert werden. Wir gehen folgendermassen vor:

- Jeder hat N/P Zahlen, die er im ersten Schritt addiert.
- Diese P Zwischenergebnisse werden dann im Baum addiert.

Wir erhalten somit die sequentielle Rechenzeit

$$T_{best}(N) = (N-1)t_a \tag{7.16}$$

und die parallele Rechenzeit

$$T_P(N, P) = (N/P - 1)t_a + \mathrm{ld}\, P t_m, \tag{7.17}$$

wobei t_a die Zeit für die Addition zweier Zahlen und t_m die Zeit für den Nachrichtenaustausch ist (wir nehmen an, dass $t_m \gg t_a$).

Feste sequentielle Ausführungszeit

Setzen wir $T_{best}(N) = T_{fix}$ so erhalten wir, falls $T_{fix} \gg t_a$, in guter Näherung

$$N_A = T_{fix}/t_a.$$

Da die Problemgröße auf N_A beschränkt ist hat es natürlich auch keinen Sinn mehr als $P = N_A$ Prozessoren einzusetzen. Wir erhalten also eine Beschränkung der maximal einsetzbaren Prozessorzahl. Für den Speedup erhalten wir für den Fall $N_A/P \gg 1$

$$S_A(P) = \frac{T_{fix}}{T_{fix}/P + \operatorname{ld} P t_m} = \frac{P}{1 + P \operatorname{ld} P \frac{t_m}{T_{fix}}}. \tag{7.18}$$

Feste parallele Ausführungszeit

Wir erhalten für die Gustafson–Skalierung:

$$\left(\frac{N}{P} - 1\right) t_a + \operatorname{ld} P t_m = T_{fix} \Longrightarrow N_G = P\left(1 + \frac{T_{fix} - \operatorname{ld} P t_m}{t_a}\right). \tag{7.19}$$

Auch hier erhalten wir eine Beschränkung an die maximal nutzbare Prozessorzahl: Ist $\operatorname{ld} P t_m = T_{fix}$, so würde ein Einsatz von mehr Prozessoren in jedem Fall die maximal zulässige Rechenzeit übersteigen. Es sind also maximal $2^{T_{fix}/t_m}$ Prozessoren einsetzbar. Immerhin können wir annehmen, dass $2^{T_{fix}/t_m} \gg T_{fix}/t_a$ gilt.

Für den skalierten Speedup S_G erhalten wir unter der Annahme $N_G(P)/P \gg 1$:

$$S_G(P) = \frac{N_G(P) t_a}{N_G(P) t_a/P + \operatorname{ld} P t_m} = \frac{P}{1 + \operatorname{ld} P \frac{t_m}{T_{fix}}}. \tag{7.20}$$

Hier haben wir ausgenutzt, dass $N_G(P) \approx P T_{fix}/t_a$ in guter Näherung. Im Vergleich zu $S_A(P)$ sehen wir, dass ein Faktor P im Nenner weggefallen ist, für gleiche Prozessorzahlen ist also S_G größer als S_A.

Fester Speicher pro Prozessor

Der Speicherbedarf unseres Algorithmus ist $M(N) = N$. Somit liefert $M(N) = M_0 P$ die Skalierung

$$N_M(P) = M_0 P.$$

Offensichtlich können wir nun unbegrenzt viele Prozessoren einsetzen, dafür steigt allerdings die parallele Rechenzeit auch unbegrenzt an. Für den skalierten Speedup bekommen wir:

$$S_M(P) = \frac{N_M(P) t_a}{N_M(P) t_a/P + \operatorname{ld} P t_m} = \frac{P}{1 + \operatorname{ld} P \frac{t_m}{M_0 t_a}}. \tag{7.21}$$

Für die Wahl $T_{fix} = M_0 t_a$ ist dies die selbe Formel wie S_G. In beiden Fällen sehen wir, dass die Effizienz mit P fällt.

Isoeffiziente Skalierung

Wir wählen N so, dass die Effizienz konstant bleibt, bzw. der Speedup linear wächst, d.h.:

$$S = \frac{P}{1 + \frac{P \operatorname{ld} P}{N} \frac{t_m}{t_a}} \stackrel{!}{=} \frac{P}{1 + K} \Longrightarrow N_I(P) = P \operatorname{ld} P \frac{t_m}{K t_a},$$

für ein frei wählbares $K > 0$. Da $N_I(P)$ existiert wird man den Algorithmus als skalierbar bezeichnen. Für den Speedup gilt $S_I = P/(1 + K)$.

7.3.3 Isoeffizienzanalyse

Wir werden nun das Prinzip der isoeffizienten Skalierung noch weiter formalisieren. Dies wird es uns erlauben die Skalierbarkeit verschiedenster paralleler Systeme miteinander zu vergleichen. Wir werden damit in der Lage sein z.B. eine Frage wie „Ist dieser Algorithmus zur Matrixmultiplikation auf dem Hypercube besser skalierbar als jener zur schnellen Fouriertransformation auf einem Feld“ zu beantworten.

Problemgröße

Unser Problemgrößenparameter N war bisher willkürlich gewählt. So könnte N bei der Matrixmultiplikation sowohl die Anzahl der Elemente einer Matrix als auch die Anzahl der Elemente einer Zeile bezeichnen. In diesem Fall würde sich im ersten Fall $2N^{3/2}t_f$ und im zweiten Fall $2N^3t_f$ als sequentielle Laufzeit ergeben. Für die Vergleichbarkeit von Algorithmen benötigen wir jedoch ein Aufwandsmaß das invariant bezüglich der Wahl des Problemgrößenparameters ist. Daher wählen wir als Maß für den Aufwand W eines (sequentiellen) Algorithmus dessen Ausführungszeit, wir setzen also

$$W = T_{best}(N) \tag{7.22}$$

selbst. Diese Ausführungszeit ist wiederum proportional zur Zahl der auszuführenden Operationen in dem Algorithmus. Im folgenden drücken wir nun alle Größen bezüglich dieser Arbeit W aus. So erhalten wir etwa für den Parallelitätsgrad Γ:

$$\Gamma(W) \leq O(W),$$

denn es können nicht mehr Operationen parallel ausgeführt werden als überhaupt auszuführen sind.

Mittels $N = T_{best}^{-1}(W)$ können wir schreiben

$$\tilde{T}_P(W, P) = T_P(T_{best}^{-1}(W), P),$$

wobei wir jedoch die Tilde im folgenden gleich wieder weglassen werden.

Overhead

Wir definieren den *Overhead* als

$$T_o(W, P) = PT_P(W, P) - W \geq 0. \tag{7.23}$$

$PT_P(W, P)$ ist die Zeit, die eine Simulation des sequentiellen Programms auf einem Prozessor benötigen würde. Diese ist in jedem Fall nicht kleiner als die beste sequentielle Ausführungszeit W. Der Overhead beinhaltet zusätzliche Rechenzeit aufgrund von Kommunikation, Lastungleichheit und „überflüssigen“ Berechnungen.

Isoeffizienzfunktion

Aus dem Overhead erhalten wir

$$T_P(W, P) = \frac{W + T_o(W, P)}{P}.$$

Also erhalten wir für den Speedup

$$S(W, P) = \frac{W}{T_P(W, P)} = P\frac{1}{1 + \frac{T_o(W,P)}{W}},$$

bzw. für die Effizienz

$$E(W,P) = \frac{1}{1 + \frac{T_o(W,P)}{W}}.$$

Im Sinne einer isoeffizienten Skalierung fragen wir nun: Wie muss W als Funktion von P wachsen damit die Effizienz konstant bleibt. Wegen obiger Formel ist dies dann der Fall wenn $T_o(W,P)/W = K$, mit einer beliebigen Konstanten $K \geq 0$. Die Effizienz ist dann $1/(1+K)$. Damit erhalten wir die folgende Definition:

Eine Funktion $W_K(P)$ heißt *Isoeffizienzfunktion* falls sie die Gleichung

$$T_o(W_K(P), P) = KW_K(P)$$

identisch erfüllt. Ein paralleles System heißt skalierbar genau dann wenn es eine Isoeffizienzfunktion besitzt. Das asymptotische Wachstum von W mit P ist ein Maß für die Skalierbarkeit eines Systems. Besitzt etwa ein System S_1 eine Isoeffizienzfunktion $W = O(P^{3/2})$ und ein System S_2 eine Isoeffizienzfunktion $W = O(P^2)$ so ist S_2 *schlechter* skalierbar als S_1.

Wir wenden uns nun noch zwei Fragen über die Existenz und das Mindestwachstum einer Isoeffizienzfunktion zu.

Wann gibt es eine Isoeffizienzfunktion?

Wir gehen aus von der Effizienz

$$E(W,P) = \frac{1}{1 + \frac{T_o(W,P)}{W}}.$$

Betrachten wir zunächst was mit der Effizienz bei *festem* W und wachsendem P passiert. Es gilt für jedes parallele System, dass

$$\lim_{P\to\infty} E(W,P) = 0$$

wie man aus folgender Überlegung sieht: Da W fest ist, ist auch der Parallelitätsgrad fest und damit gibt es eine untere Schranke für die parallele Rechenzeit: $T_P(W,P) \geq T_{min}(W)$, d.h. die Berechnung kann nicht schneller als T_{min} sein, egal wieviele Prozessoren eingesetzt werden. Damit gilt jedoch asymptotisch

$$\frac{T_o(W,P)}{W} \geq \frac{PT_{min}(W) - W}{W} = O(P)$$

und somit geht die Effizienz gegen 0.

Betrachten wir nun die Effizienz bei *festem* P und wachsender Arbeit W, so gilt für viele (nicht alle!) parallele Systeme, dass

$$\lim_{W\to\infty} E(W,P) = 1.$$

Offensichtlich bedeutet dies im Hinblick auf die Effizienzformel, dass

$$T_o(W,P)|_{P=const} < O(W) \tag{7.24}$$

bei festem P wächst also der Overhead weniger als linear mit W. In diesem Fall kann für jedes P ein W gefunden werden so dass eine gewünschte Effizienz erreicht wird. Gleichung (7.24) sichert also die Existenz einer Isoeffizienzfunktion. Als Beispiel für ein nicht skalierbares System sei die Matrixtransposition genannt. Wir werden später ableiten, dass der Overhead in diesem Fall $T_o(W,P) = O(W, \operatorname{ld} P)$ beträgt. Somit existiert keine Isoeffizienzfunktion.

Optimal parallelisierbare Systeme

Wir wollen nun der Frage nachgehen wie Isoeffizienzfunktionen mindestens wachsen müssen. Dazu bemerken wir zunächst, dass

$$T_P(W,P) \geq \frac{W}{\Gamma(W)},$$

denn $\Gamma(W)$ (dimensionslos) ist ja die maximale Zahl gleichzeitig ausführbarer Operationen im sequentiellen Algorithmus bei Aufwand W. Somit ist $W/\Gamma(W)$ eine untere Schranke für die parallele Rechenzeit.

Nun können sicher nicht mehr Operationen parallel ausgeführt werden als überhaupt auszuführen sind, d.h. es gilt $\Gamma(W) \leq O(W)$. Wir wollen ein System als *optimal parallelisierbar* bezeichnen falls

$$\Gamma(W) = cW$$

gilt mit einer Konstanten $c > 0$. Nun gilt

$$T_P(W,P) \geq \frac{W}{\Gamma(W)} = \frac{1}{c},$$

die minimale parallele Rechenzeit bleibt konstant. Für den Overhead erhalten wir dann in diesem Fall

$$T_o(W,P) = PT_P(W,P) - W = P/c - W$$

und somit für die Isoeffizienzfunktion

$$T_o(W,P) = P/c - W \stackrel{!}{=} KW \Longleftrightarrow W = \frac{P}{(K+1)c} = \Theta(P).$$

Optimal parallelisierbare Systeme haben somit eine Isoeffizienzfunktion $W = \Theta(P)$. Wir merken uns also, dass Isoeffizienzfunktionen mindestens linear in P wachsen.

Wir werden in den folgenden Kapiteln die Isoeffizienzfunktionen für eine Reihe von Algorithmen bestimmen, deswegen sei hier auf ein ausführliches Beispiel verzichtet.

8 Algorithmen für vollbesetzte Matrizen

In diesem Kapitel betrachten wir Algorithmen zur Transposition von Matrizen, zur Multiplikation einer Matrix mit einem Vektor bzw. einer Matrix, sowie die Auflösung von linearen Gleichungssystemen.

8.1 Datenaufteilungen

Algorithmen für vollbesetzte Matrizen sind typischerweise datenparallele Algorithmen (siehe Abschnitt 7.1.1). Wir betrachten daher vorab die gebräuchlichsten Aufteilungsstrategien für Vektoren und Matrizen.

8.1.1 Aufteilung von Vektoren

Ein Vektor $x \in \mathbb{R}^N$ entspricht einer geordneten Liste von Zahlen. Jedem Index i aus der Indexmenge $I = \{0, \ldots, N-1\}$ wird eine reelle Zahl x_i zugeordnet. Statt $\mathbb{R}^N$ wollen wir auch $\mathbb{R}(I)$ schreiben, um die Abhängigkeit von der Indexmenge deutlich zu machen. Die natürliche (und effizienteste) Datenstruktur für einen Vektor ist das Feld. Da Felder in vielen Programmiersprachen mit dem Index 0 beginnen, tut dies auch unsere Indexmenge I.

Eine Datenaufteilung entspricht nun einer Zerlegung der Indexmenge I in

$$I = \bigcup_{p \in P} I_p, \text{ mit } p \neq q \Rightarrow I_p \cap I_q = \emptyset,$$

wobei P die Prozessmenge bezeichnet. Im Sinne einer guten Lastverteilung sollten die Indexmengen I_p, $p \in P$, jeweils gleich viele Elemente enthalten.

Prozess $p \in P$ speichert somit die Komponenten x_i, $i \in I_p$, des Vektors x. In jedem Prozess möchten wir wieder mit einer zusammenhängenden Indexmenge $\tilde{I}_p$ arbeiten, die bei 0 beginnt, d.h.

$$\tilde{I}_p = \{0, \ldots, |I_p| - 1\}.$$

Die Abbildungen

$$\begin{aligned} p\colon &\quad I \to P \text{ bzw.} \\ \mu\colon &\quad I \to \mathbb{N} \end{aligned}$$

ordnen jedem Index $i \in I$ umkehrbar eindeutig einen Prozess $p(i) \in P$ und einen lokalen Index $\mu(i) \in \tilde{I}_{p(i)}$ zu:

$$I \ni i \mapsto (p(i), \mu(i)).$$

Die Umkehrabbildung

$$\mu^{-1}\colon \underbrace{\bigcup_{p \in P} \{p\} \times \tilde{I}_p}_{\subset P \times \mathbb{N}} \to I$$

liefert zu jedem lokalen Index $i \in \tilde{I}_p$ und Prozess $p \in P$ den globalen Index $\mu^{-1}(p, i)$, d.h.

$$p(\mu^{-1}(p,i)) = p \text{ und } \mu(\mu^{-1}(p,i)) = i.$$

zyklisch:

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	1	2	3	0	1	2	3	0	1	2	3	0
$\mu(i)$:	0	0	0	0	1	1	1	1	2	2	2	2	3

z.B. $I_1 = \{1, 5, 9\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

blockweise:

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	0	0	0	1	1	1	2	2	2	3	3	3
$\mu(i)$:	0	1	2	3	0	1	2	0	1	2	0	1	2

z.B. $I_1 = \{4, 5, 6\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

Abbildung 8.1: Zyklische und blockweise Aufteilung eines $\mathbb{R}^{13}$ Vektors auf vier Prozessoren.

Gebräuchliche Aufteilungen

sind vor allem die *zyklische Aufteilung* mit[1]

$$\begin{aligned} p(i) &= i \,\%\, P \\ \mu(i) &= i \div P \end{aligned}$$

und die *blockweise Aufteilung* mit

$$\begin{aligned} p(i) &= \begin{cases} i \div (B+1) & \text{falls } i < R(B+1) \\ R + (i - R(B+1)) \div B & \text{sonst} \end{cases} \\ \mu(i) &= \begin{cases} i \,\%\, (B+1) & \text{falls } i < R(B+1) \\ (i - R(B+1)) \,\%\, B & \text{sonst} \end{cases} \end{aligned}$$

mit $B = N \div P$ und $R = N \,\%\, P$. Hier ist die Idee, dass die ersten R Prozesse $B+1$ Indizes bekommen und die restlichen je B Indizes. Abb. 8.1 illustriert zyklische und blockweise Aufteilung für $N = 13$ und $P = 4$.

8.1.2 Aufteilung von Matrizen

Bei einer Matrix $A \in \mathbb{R}^{N \times M}$ wird jedem Tupel $(i, j) \in I \times J$, mit $I = \{0, \ldots, N-1\}$ und $J = \{0, \ldots, M-1\}$ eine reele Zahl a_{ij} zugeordnet.

Im Prinzip könnte man die Matrixelemente beliebig den Prozessoren zuordnen. Dann können jedoch die einem Prozessor zugeordneten Elemente im allgemeinen *nicht* wieder als Matrix dargestellt werden. Dies ist nur möglich, wenn man sich auf die separate Zerlegung der eindimensionalen Indexmengen I und J beschränkt. Dazu nehmen wir an, die Prozesse seien als zweidimensionales Feld organisiert, d.h.

$$(p, q) \in \{0, \ldots, P-1\} \times \{0, \ldots, Q-1\}.$$

[1] $\div$ bedeutet ganzzahlige Division; % die modulo-Funktion

Die Indexmengen I, J werden zerlegt in

$$I = \bigcup_{p=0}^{P-1} I_p \text{ und } J = \bigcup_{q=0}^{Q-1} I_q$$

und Prozess (p, q) ist dann für die Indizes $I_p \times I_q$ verantwortlich. Lokal speichert Prozess (p, q) seine Elemente dann als $\mathbb{R}(\tilde{I}_p \times \tilde{J}_q)$-Matrix. Die Zerlegungen von I und J werden formal durch die Abbildungen p und μ sowie q und ν beschrieben:

$$\begin{aligned} I_p &= \{i \in I \mid p(i) = p\}, \quad & \tilde{I}_p &= \{n \in \mathbb{N} \mid \exists i \in I : p(i) = p \wedge \mu(i) = n\} \\ J_q &= \{j \in J \mid q(j) = q\}, \quad & \tilde{J}_q &= \{m \in \mathbb{N} \mid \exists j \in J : q(j) = q \wedge \nu(j) = m\} \end{aligned}$$

Die Abb. 8.2 zeigt einige Beispiele für die Aufteilung einer 6×9-Matrix auf vier Prozessoren.

Welche Datenaufteilung ist nun die Beste? Dies lässt sich so nicht sagen. Generell liefert die Organisation der Prozesse als möglichst quadratisches Feld eine Aufteilung mit besserer Lastverteilung. Wichtiger ist jedoch, dass sich unterschiedliche Aufteilungen unterschiedlich gut für verschiedene Algorithmen eignen. So werden wir sehen, dass sich ein Prozessfeld mit zyklischer Aufteilung sowohl der Zeilen aus auch der Spaltenindizes recht gut für die LU-Zerlegung eignet. Diese Aufteilung ist jedoch nicht optimal für die Auflösung der entstehenden Dreieckssysteme. Muss man das Gleichungssystem für viele rechte Seiten lösen, so ist ein Kompromiss anzustreben.

Dies gilt generell für fast alle Aufgaben aus der linearen Algebra. Die Multiplikation zweier Matrizen oder die Transposition einer Matrix stellt nur einen Schritt eines größeren Algorithmus dar. Die Datenaufteilung kann somit nicht auf einen Teilschritt hin optimiert werden, sondern sollte einen guten Kompromiss darstellen. Eventuell kann auch überlegt werden, ob ein Umkopieren der Daten sinnvoll ist.

In den folgenden Abschnitten präsentieren wir deshalb entweder Algorithmen für unterschiedliche Datenaufteilungen zur Lösung einer Aufgabe (z.B. Transponieren), oder wir formulieren den Algorithmus gleich so, dass er eine Vielzahl von Datenaufteilungen zulässt (z.B. LU-Zerlegung).

8.2 Transponieren einer Matrix

Gegeben sei $A \in \mathbb{R}^{N \times M}$, verteilt auf eine Menge von Prozessen; zu „berechnen“ ist A^T mit der selben Datenaufteilung wie A. Im Prinzip ist das Problem trivial. Wir könnten die Matrix so auf die Prozessoren aufteilen, dass nur Kommunikation mit nächsten Nachbarn notwendig ist (da die Prozesse paarweise kommunizieren). Ein Beispiel zeigt Abb. 8.3). Hier wurde eine Ringtopologie zugrunde gelegt. Offensichtlich ist nur Kommunikation zwischen direkten Nachbarn notwendig $(0 \leftrightarrow 1, 2 \leftrightarrow 3, \ldots, 10 \leftrightarrow 11)$. Allerdings entspricht diese Datenaufteilung nicht dem Schema, das wir in Abschnitt 8.1 eingeführt haben und eignet sich z.B. weniger gut für die Multiplikation zweier Matrizen.

Wir besprechen nun drei Algorithmen für brauchbarere Datenaufteilungen.

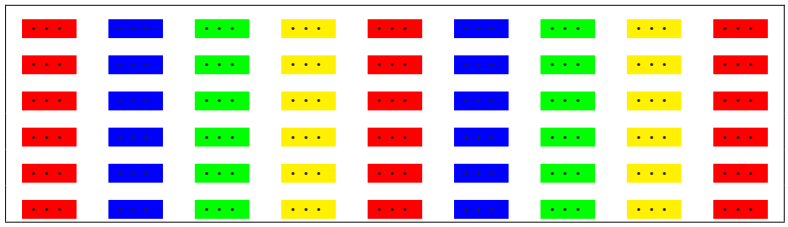
8.2.1 Eindimensionale Aufteilung

Betrachten wir o.B.d.A eine spaltenweise, geblockte Aufteilung wie in Abb. 8.4 dargestellt.

Offensichtlich hat in diesem Fall jeder Prozessor Daten an jeden anderen zu senden. Es handelt sich also um all-to-all mit persönlichen Nachrichten. Nehmen wir eine Hypercubestruktur als

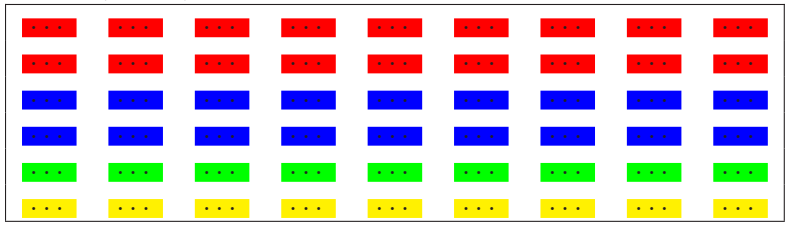
(a) $P = 1$, $Q = 4$ (Spalten), J: zyklisch:

0	1	2	3	4	5	6	7	8	J
0	1	2	3	0	1	2	3	0	q
0	0	0	0	1	1	1	1	2	ν



(b) $P = 4$,$Q = 1$ (Zeilen), I: blockweise:

0	0	0
1	0	1
2	1	0
3	1	1
4	2	0
5	3	0
I	p	μ



(c) $P = 2$, $Q = 2$ (Feld), I: zyklisch, J: blockweise:

		0	1	2	3	4	5	6	7	8	J
		0	0	0	0	0	1	1	1	1	q
		0	1	2	3	4	0	1	2	3	ν
0	0	0									
1	1	0									
2	0	1									
3	1	1									
4	0	2									
5	1	2									
I	p	μ									

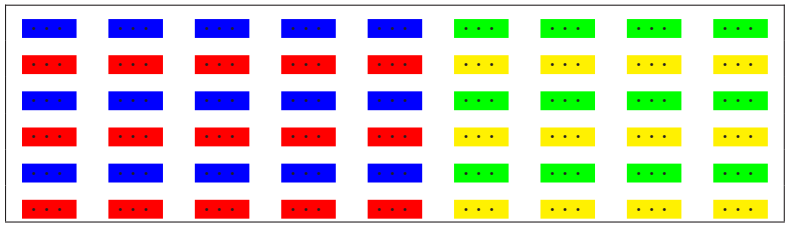


Abbildung 8.2: Aufteilung einer 6×9-Matrix auf vier Prozessoren

12	1	3	5
0	13	7	9
2	6	14	11
4	8	10	15

Abbildung 8.3: Optimale Datenaufteilung für die Matrixtransposition (die Zahlen bezeichnen die Prozessornumern).

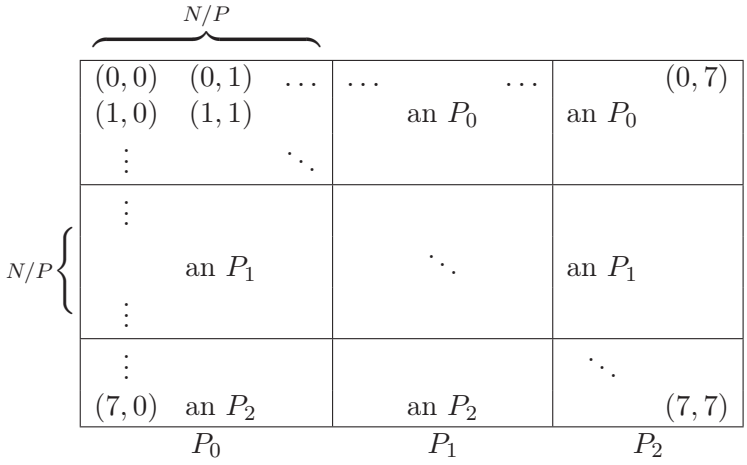


Abbildung 8.4: 8×8-Matrix auf drei Prozessoren in spaltenweiser, geblockter Aufteilung.

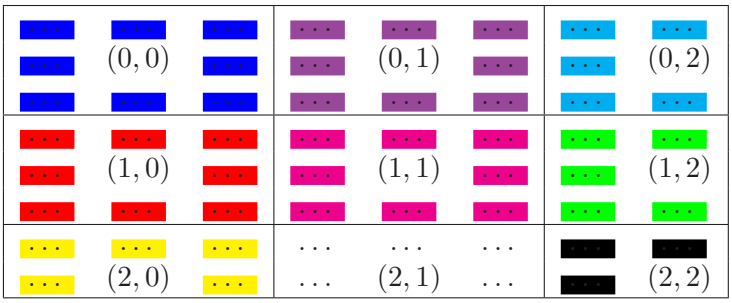


Abbildung 8.5: Beispiel für eine zweidimensionale, geblockte Aufteilung $N = 8$, $\sqrt{P} = 3$.

Verbindungstopologie an, so erhalten wir folgende parallele Laufzeit für eine $N \times N$-Matrix und P Prozessoren:

$$T_P(N,P) = \underbrace{2(t_s + t_h)\operatorname{ld} P}_{\text{Aufsetzen}} + \underbrace{t_w \frac{N^2}{P^2} P \operatorname{ld} P}_{\text{Datenübertragung}} + \underbrace{(P-1)\frac{N^2}{P^2}\frac{t_e}{2}}_{\text{Transponieren}} \approx$$

$$\approx \operatorname{ld} P(t_s + t_h)2 + \frac{N^2}{P} \operatorname{ld} P t_w + \frac{N^2}{P}\frac{t_e}{2}$$

Selbst bei festem P und wachsendem N können wir den Anteil der Kommunikation an der Gesamtlaufzeit nicht beliebig klein machen. Dies ist bei allen Algorithmen zur Transposition so (auch bei der optimalen Aufteilung oben). Matrixtransposition besitzt also keine Isoeffizienzfuktion und ist somit nicht skalierbar.

8.2.2 Zweidimensionale Aufteilung

Wir betrachten nun eine zweidimensionale, geblockte Aufteilung einer $N \times N$-Matrix auf ein $\sqrt{P} \times \sqrt{P}$-Prozessorfeld, wie in Abb. 8.5 dargestellt.

Jeder Prozessor muss seine Teilmatrix mit genau einem anderen austauschen. Der naive Transpositionsalgorithmus für diese Konfiguration arbeitet wie folgt:

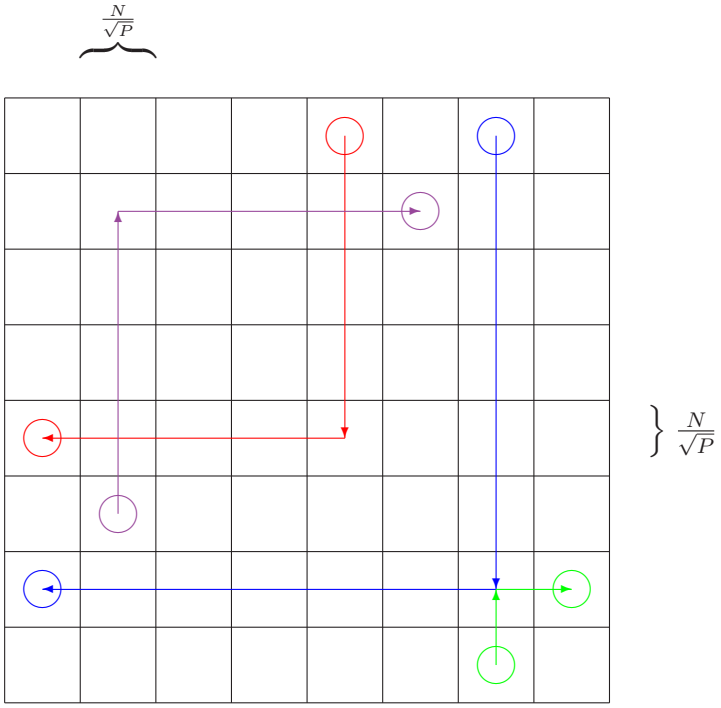


Abbildung 8.6: Diverse Wege von Teilmatrizen bei $\sqrt{P} = 8$.

Prozessoren (p, q) unterhalb der Hauptdiagonalen $(p > q)$ schicken ihre Teilmatrix in der Spalte nach oben bis zum Prozessor (q, q), dann läuft die Teilmatrix nach rechts bis in die richtige Spalte zu Prozessor (q, p).

Entsprechend laufen Daten von Prozessoren (p, q) oberhalb der Hauptdiagonalen $(q > p)$ erst in der Spalte q nach unten bis (q, q) und dann nach links bis (q, p) erreicht wird.

Abb. 8.6 zeigt verschiedene Wege von Teilmatrizen bei $\sqrt{P} = 8$.

Offensichtlich leiten Prozessoren (p, q) mit $p > q$ Daten von unten nach oben bzw. rechts nach links und Prozessoren (p, q) mit $q > p$ entsprechend Daten von oben nach unten und links nach rechts. Bei synchroner Kommunikation sind in jedem Schritt vier Sende- bzw. Empfangsoperationen notwendig, und insgesamt braucht man $2(\sqrt{P}-1)$ Schritte. Die parallele Laufzeit beträgt somit

$$\begin{aligned} T_P(N,P) &= 2(\sqrt{P}-1)\cdot 4\left(t_s + t_h + t_w\left(\frac{N}{\sqrt{P}}\right)^2\right) + \frac{1}{2}\left(\frac{N}{\sqrt{P}}\right)^2 t_e \approx \\ &\approx \sqrt{P}8(t_s + t_h) + \frac{N^2}{P}\sqrt{P}8t_w + \frac{N^2}{P}\frac{t_e}{2} \end{aligned}$$

Im Vergleich zur eindimensionalen Aufteilung mit Hypercube hat man in der Datenübertragung den Faktor $\sqrt{P}$ statt $\operatorname{ld} P$.

8.2.3 Rekursiver Transpositionsalgorithmus

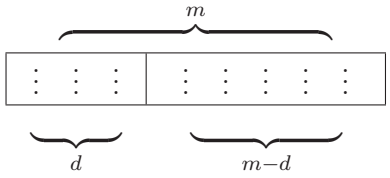
Dieser Algorithmus basiert auf folgender Beobachtung: Für eine 2×2-Blockmatrixzerlegung von A gilt

$$A^T = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}^T = \begin{pmatrix} A_{00}^T & A_{10}^T \\ A_{01}^T & A_{11}^T \end{pmatrix}$$

d.h. die Nebendiagonalblöcke tauschen die Plätze und dann muss jede Teilmatrix transponiert werden. Dies geschieht natürlich rekursiv bis eine 1×1-Matrix erreicht ist. Ist $N = 2^n$, so sind n Rekursionsschritte notwendig.

Es stellt sich heraus, dass der Hypercube die ideale Verbindungstopologie für diesen Algorithmus ist. Es sei $N = 2^n$ und $\sqrt{P} = 2^d$ mit $n \geq d$. Die Zuordnung der Indizes $I = \{0, \ldots, N-1\}$ auf die Prozessoren geschieht mittels

$$\begin{aligned} p(i) &= i \div 2^{m-d}, \\ \mu(i) &= i \,\%\, 2^{m-d} \end{aligned}$$



Es beschreiben also die oberen d Bits eines Index den Prozessor, auf den der Index abgebildet wird. Betrachten wir als Beispiel $d = 3$, d.h. $\sqrt{P} = 2^3 = 8$. Im Rekursionsschritt ist die Matrix in 2×2 Blöcke aus 4×4-Teilmatrizen zu teilen, siehe Abb. 8.7, und $2 \cdot 16$ Prozessoren müssen Daten austauschen, z.B. Prozessor $101001 = 41$ und $001101 = 13$. Dies geschieht in zwei Schritten über die Prozessoren $001001 = 9$ und $101101 = 45$. Diese sind beide *direkte* Nachbarn der Prozessoren 41 und 13 im Hypercube.

Der rekursive Transpositionsalgorithmus arbeitet nun rekursiv über die Prozessortopologie. Ist *ein* Prozessor erreicht, wird mit dem sequentiellen Algorithmus transponiert. Die parallele Laufzeit beträgt somit

$$T_P(N,P) = \operatorname{ld} P(t_s + t_h)2 + \frac{N^2}{P} \operatorname{ld} \sqrt{P} 2 t_w + \frac{N^2}{P} \frac{t_e}{2};$$

das zeigt die vollständige Formulierung:

Programm 8.1 (Rekursiver Transpositionsalgorithmus auf Hypercube)

```
parallel recursive-transpose
{
    const int d = ..., n = ...;
    const int P = 2^d, N = 2^n;

    process Π[int (p,q) ∈ {0,...,2^d − 1} × {0,...,2^d − 1}]
    {
        Matrix A, B;                    // A ist die Eingabematrix
        void rta(int r, int s, int k)
        {
            if (k == 0) { A = A^T; return; }
            int i = p − r, j = q − s, l = 2^(k−1);
            if (i < l)
            {
                if (j < l)
                {                       // links oben
```

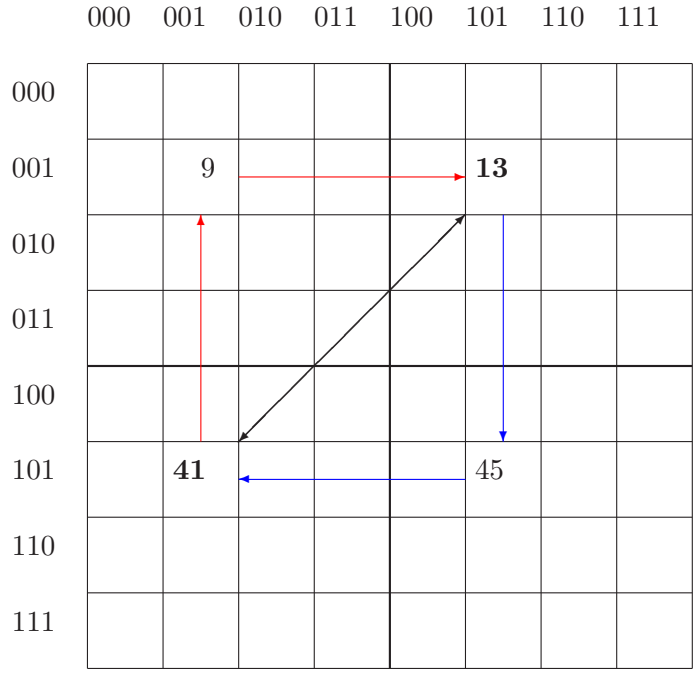


Abbildung 8.7: Kommunikation im rekursiven Transpositionsalgorithmus bei $d = 3$.

```
            recv(B,Π_{p+l,q}); send(B,Π_{p,q+l});
            rta(r,s,k − 1);
        }
        else
        {                          // rechts oben
            send(A,Π_{p+l,q}); recv(A,Π_{p,q−l});
            rta(r,s + l,k − 1);
        }
    }
    else
    {
        if (j < l) {               // links unten
            send(A,Π_{p−l,q}); recv(A,Π_{p,q+l});
            rta(r + l,s,k − 1);
        }
        else
        {                          // rechts unten
            recv(B,Π_{p−l,q}); send(B,Π_{p,q−l});
            rta(r + l,s + l,k − 1);
        }
    }
}
```

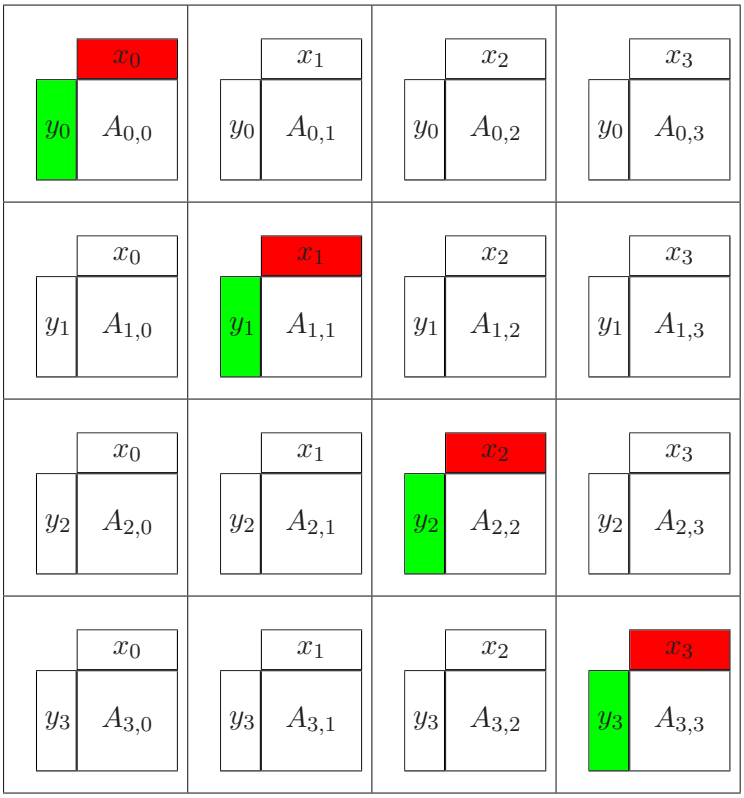


Abbildung 8.8: Aufteilung für das Matrix-Vektor-Produkt

```
        rta(0,0,d);
    }
}
```

8.3 Matrix-Vektor Multiplikation

Gegeben sei eine Matrix $A \in \mathbb{R}^{N \times M}$ und ein Vektor $x \in \mathbb{R}^M$; zu berechnen ist $y = Ax$. Auch hier gibt es wieder unterschiedliche Möglichkeiten zur Datenaufteilung. Selbstverständlich muss die Verteilung der Matrix und des Vektors aufeinander abgestimmt sein. Weiter wollen wir verlangen, dass der Ergebnisvektor $y \in \mathbb{R}^N$ genauso wie der Eingabevektor x verteilt ist.

Als Beispiel betrachten wir folgende Aufteilung (siehe Abb. 8.8): die Matrix sei blockweise auf eine Feldtopologie verteilt. Der Eingabevektor x sei entsprechend blockweise über die Diagonalprozessoren verteilt (das Prozessorfeld ist quadratisch). Das Vektorsegment x_q wird in jeder Prozessorspalte benötigt und ist somit in jeder Spalte zu kopieren (einer-an-alle). Nun berechnet jeder lokal das Produkt $y_{p,q} = A_{p,q}x_q$. Das komplette Segment y_p ergibt sich erst duch die Summation $y_p = \sum_q y_{p,q}$. Sinnvollerweise wird y_p wieder im Diagonalprozessor (p, p) durch eine alle-an-einen Kommunikation gesammelt. Das Ergebnis kann dann sofort für eine weitere Matrix-Vektor-Multiplikation als Eingabe benutzt werden.

Berechnen wir die parallele Laufzeit für eine $N \times N$-Matrix und $\sqrt{P} \times \sqrt{P}$ Prozessoren mit

einem cut-through Verbingungsnetzwerk:

$$\begin{aligned} T_P(N,P) &= \underbrace{\left(t_s + t_h + t_w \overbrace{\frac{N}{\sqrt{P}}}^{\text{Vektor}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{Austeilen von} \\ x \text{ über Spalte}}} + \underbrace{\left(\frac{N}{\sqrt{P}}\right)^2 2t_f}_{\substack{\text{lokale Matrix-} \\ \text{Vektor-Mult.}}} \\ &\quad + \underbrace{\left(t_s + t_h + t_w \frac{N}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{Reduktion} \\ (t_f \ll t_w)}} = \\ &= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{N}{\sqrt{P}} \operatorname{ld}\sqrt{P} 2t_w + \frac{N^2}{P} 2t_f \end{aligned}$$

Für festes P und $N \to \infty$ wird der Kommunikationsanteil beliebig klein, es existiert also eine Isoeffizienzfunktion, der Algorithmus ist skalierbar.

Berechnen wir die Isoeffizienzfunktion.

Umrechnen auf die Arbeit W:

$$\begin{aligned} W &= N^2 2t_f \text{ (seq. Laufzeit)} \\ &\Rightarrow N = \frac{\sqrt{W}}{\sqrt{2t_f}} \\ T_P(W,P) &= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{\sqrt{W}}{\sqrt{P}} \operatorname{ld}\sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + \frac{W}{P} \end{aligned}$$

Overhead:

$$T_O(W,P) = PT_P(W,P) - W = \sqrt{W}\sqrt{P} \operatorname{ld}\sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + P \operatorname{ld}\sqrt{P}(t_s + t_h)2$$

Isoeffizienz ($T_O(W,P) \stackrel{!}{=} KW$): T_O hat zwei Terme. Für den ersten erhalten wir

$$\begin{aligned} & \sqrt{W}\sqrt{P} \operatorname{ld}\sqrt{P} \frac{2t_w}{\sqrt{2t_f}} = KW \\ \iff \quad & W = P(\operatorname{ld}\sqrt{P})^2 \frac{4t_w^2}{2t_f K^2} \end{aligned}$$

und für den zweiten

$$\begin{aligned} & P \operatorname{ld}\sqrt{P}(t_s + t_h)2 = KW \\ \iff \quad & W = P \operatorname{ld}\sqrt{P} \frac{(t_s + t_h)2}{K}; \end{aligned}$$

somit ist $W = \Theta(P(\operatorname{ld}\sqrt{P})^2)$ die gesuchte Isoeffizienzfunktion.

8.4 Matrix-Matrix-Multiplikation

8.4.1 Algorithmus von Cannon

Die beiden $N \times N$-Matrizen A und B seien blockweise auf eine 2D-Feldtopologie ($\sqrt{P} \times \sqrt{P}$) verteilt und es ist $C = A \cdot B$ zu berechnen. Praktischerweise soll das Ergebnis C wieder in derselben Verteilung vorliegen. Prozess (p, q) muss somit

$$C_{p,q} = \sum_k A_{p,k} \cdot B_{k,q}$$

berechnen, benötigt also Blockzeile p von A und Blockspalte q von B. Der Algorithmus von Cannon besteht aus folgenden Phasen:

1. *Alignment-Phase:* Die Blöcke von A werden in jeder Zeile zyklisch nach links geschoben, bis der Diagonalblock in der ersten Spalte zu liegen kommt. Entsprechend schiebt man die Blöcke von B in den Spalten nach oben, bis alle Diagonalblöcke in der ersten Zeile liegen.

   Nach der Alignment-Phase hat Prozessor (p, q) die Blöcke

   $$A_{p,\underbrace{(q+p)\,\%\,\sqrt{P}}} \qquad \text{(Zeile } p \text{ schiebt } p \text{ mal nach links)}$$

   $$B_{\underbrace{(p+q)\,\%\,\sqrt{P}},q} \qquad \text{(Spalte } q \text{ schiebt } q \text{ mal nach oben).}$$

2. *Rechenphase:* Offensichtlich verfügt nun jeder Prozess über zwei passende Blöcke, die er multiplizieren kann. Schiebt man die Blöcke von A in jeder Zeile von A zyklisch um eine Position nach links und die von B in jeder Spalte nach oben, so erhält jeder wieder zwei passende Blöcke. Nach $\sqrt{P}$ Schritten ist man fertig.

Betrachten wir die zugehörige Isoeffizienzfunktion.

Sequentielle Laufzeit (siehe Bem. unten):

$$W = T_S(N) = N^3 2t_f$$
$$\Rightarrow N = \left(\frac{W}{2t_f}\right)^{\frac{1}{3}}$$

parallele Laufzeit:

$$\begin{aligned} T_P(N,P) &= \underbrace{\left(\sqrt{P}-1\right)\left(t_s + t_h + t_w\frac{N^2}{P}\right)}_{\text{alignment}} \quad \overbrace{4}^{\substack{\textbf{send/recv}\\ A/B}} \\ &\quad + \sqrt{P}\left(\underbrace{\left(\frac{N}{\sqrt{P}}\right)^3 2t_f}_{\substack{\text{Multiplik.}\\ \text{eines Blockes}}} + \left(t_s + t_h + t_w\frac{N^2}{P}\right)4\right) \approx \\ &\approx \sqrt{P}(t_s + t_h)8 + \frac{N^2}{\sqrt{P}}t_w 8 + \frac{N^3}{P}2t_f \\ T_P(W,P) &= \sqrt{P}(t_s + t_h)8 + \frac{W^{\frac{2}{3}}}{\sqrt{P}}\frac{8t_w}{(2t_f)^{\frac{1}{3}}} + \frac{W}{P} \end{aligned}$$

Overhead:

$$T_O(W,P) = PT_P(W,P) - W = P^{\frac{2}{3}}(t_s + t_h)8 + \sqrt{P}W^{\frac{2}{3}}\frac{8t_w}{(2t_f)^{\frac{1}{3}}}$$

Man hat also $W = \Theta(P^{3/2})$. Wegen $N = \left(\frac{W}{2t_f}\right)^{1/3}$ gilt $N/\sqrt{P} = const$, d.h. bei *fester* Grösse der Blöcke in jedem Prozessor und wachsender Prozessorzahl bleibt die Effizienz konstant.

8.4.2 Dekel-Nassimi-Salmi-Algorithmus

Beschränken wir uns beim Algorithmus von Cannon auf 1×1-Blöcke pro Prozessor, also $\sqrt{P} = N$, so können wir für die erforderlichen N^3 Multiplikationen nur N^2 Prozessoren nutzen. Dies ist der Grund für die Isoeffizienzfunktion der Ordnung $P^{3/2}$.

Nun betrachten wir einen Algorithmus der den Einsatz von bis zu N^3 Prozessoren bei einer $N \times N$-Matrix erlaubt. Gegeben seien also $N \times N$-Matrizen A und B sowie ein 3D-Feld von Prozessoren der Dimension $P^{1/3} \times P^{1/3} \times P^{1/3}$. Die Prozessoren werden über die Koordinaten (p,q,r) adressiert. Um den Block $C_{p,q}$ der Ergebnismatrix C mittels

$$C_{p,q} = \sum_{r=0}^{P^{\frac{1}{3}}-1} A_{p,r} \cdot B_{r,q} \tag{8.1}$$

zu berechnen, setzen wir $P^{1/3}$ Prozessoren ein, und zwar ist Prozessor (p,q,r) genau für das Produkt $A_{p,r} \cdot B_{r,q}$ zuständig. Nun ist noch zu klären, wie Eingabe- und Ergebnismatrizen verteilt sein sollen: Sowohl A als auch B sind in $P^{1/3} \times P^{1/3}$-Blöcke der Größe $\frac{N}{P^{1/3}} \times \frac{N}{P^{1/3}}$ zerlegt. $A_{p,q}$ und $B_{p,q}$ wird zu Beginn in Prozessor $(p,q,0)$ gespeichert, auch das Ergebnis $C_{p,q}$ soll dort liegen. Die Prozessoren (p,q,r) für $r > 0$ werden nur zwischenzeitlich zur Rechnung benutzt. Abb. 8.9 zeigt die Verteilung von A, B, C für $P^{1/3} = 4$ (P=64).

Damit nun jeder Prozessor (p,q,r) „seine“ Multiplikation $A_{p,r} \cdot B_{r,q}$ durchführen kann, sind die beteiligten Blöcke von A und B erst an ihre richtige Position zu befördern. Wie man sieht, benötigen alle Prozessoren $(p,*,r)$ den Block $A_{p,r}$ und alle Prozessoren $(*,q,r)$ den Block $B_{r,q}$. Diese Verteilung zeigt Abb. 8.9. Sie wird folgendermaßen hergestellt:

> Prozessor $(p,q,0)$ sendet $A_{p,q}$ an Prozessor (p,q,q) und dann sendet (p,q,q) das $A_{p,q}$ an alle $(p,*,q)$ mittels einer einer-an-alle Kommunikation auf $P^{1/3}$ Prozessoren. Entsprechend schickt $(p,q,0)$ das $B_{p,q}$ an Prozessor (p,q,p), und dieser verteilt dann an $(*,q,p)$.

Nach der Multiplikation in jedem (p,q,r) sind die Ergebnisse aller $(p,q,*)$ noch in $(p,q,0)$ mittels einer alle-an-einen Kommunikation auf $P^{1/3}$ Prozessoren zu sammeln.

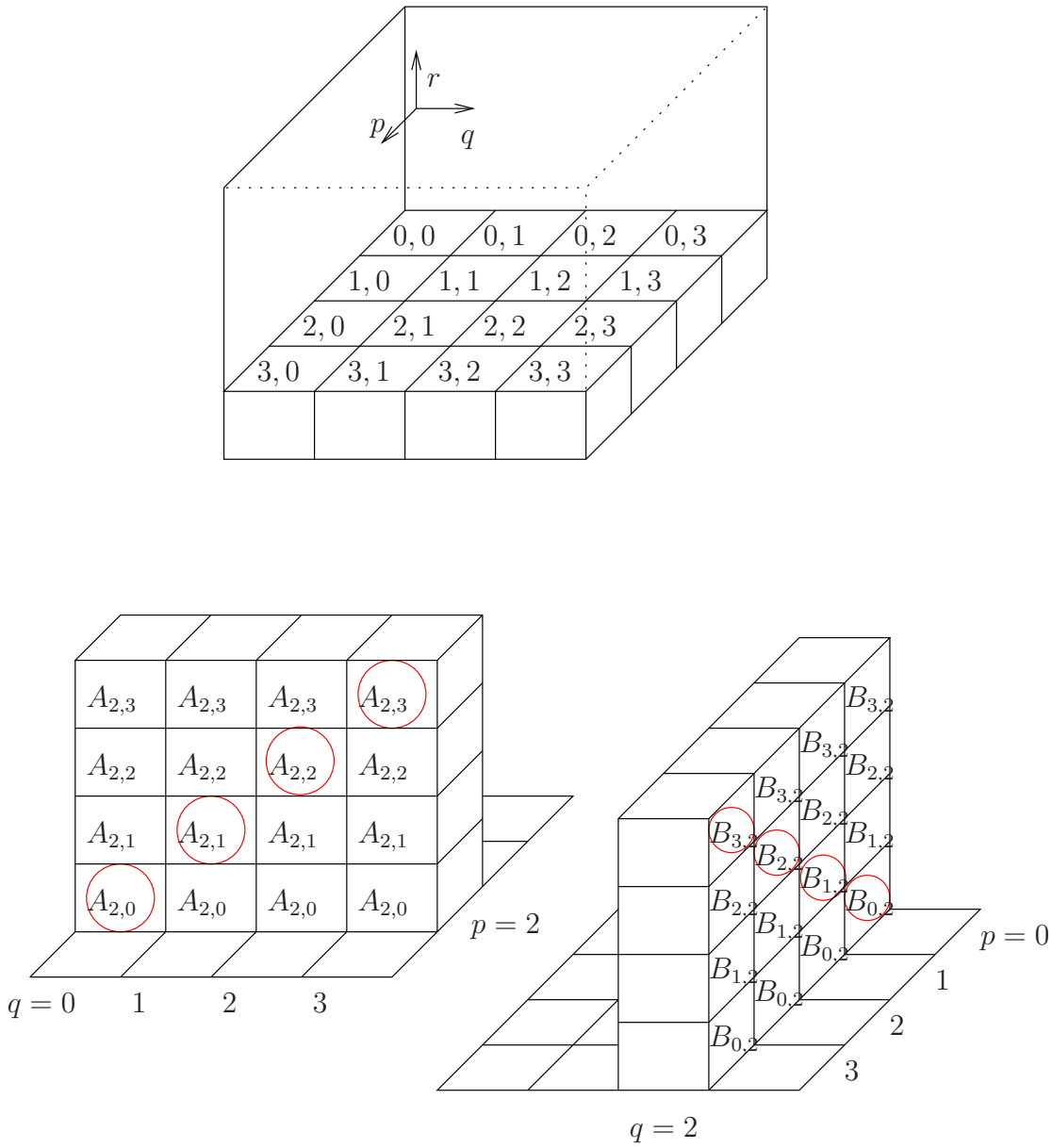


Abbildung 8.9: Oben: Verteilung der Blöcke von A und B (zu Beginn) und C (am Ende). Unten: Verteilung von A und B für die Multiplikation.

Analysieren wir das Verfahren im Detail (3D-cut-through Netzwerk):

$$
\begin{aligned}
W &= T_S(N) = N^3 2t_f \quad \Rightarrow N = \left(\frac{N}{2t_f}\right)^{\frac{1}{3}} \\
T_P(N,P) &= \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right)}_{(p,q,0)\longrightarrow(p,q,q),(p,q,p)} \overbrace{2}^{\substack{A_{p,q} \text{ u.}\\ B_{p,q}}} + \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right)\operatorname{ld} P^{\frac{1}{3}}\overbrace{2}^{A,B}}_{\text{einer-an-alle}} \\
&\quad + \underbrace{\left(\frac{N}{P^{\frac{1}{3}}}\right)^3 2t_f}_{\text{Multiplikation}} + \underbrace{\left(t_s + t_h + t_w\left(\frac{N}{P^{\frac{1}{3}}}\right)^2\right)\operatorname{ld} P^{\frac{1}{3}}}_{\substack{\text{alle-an-einen}\\ (t_f \ll t_w)}} \approx \\
&\approx 3\operatorname{ld} P^{\frac{1}{3}}(t_s + t_h) + \frac{N^2}{P^{\frac{2}{3}}} 3\operatorname{ld} P^{\frac{1}{3}} t_w + \frac{N^3}{P} 2t_f \\
T_P(W,P) &= 3\operatorname{ld} P^{\frac{1}{3}}(t_s + t_h) + \frac{W^{\frac{2}{3}}}{P^{\frac{2}{3}}} 3\operatorname{ld} P^{\frac{1}{3}} \frac{t_w}{(2t_f)^{\frac{2}{3}}} + \frac{W}{P} \\
T_O(W,P) &= P\operatorname{ld} P^{\frac{1}{3}} 3(t_s + t_h) + W^{\frac{2}{3}} P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}}
\end{aligned}
$$

Aus dem zweiten Term von $T_O(W,P)$ nähern wir die Isoeffizienzfunktion an:

$$
\begin{aligned}
& W^{\frac{2}{3}} P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}} = KW \\
\iff \quad & W^{\frac{1}{3}} = P^{\frac{1}{3}} \operatorname{ld} P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}} K} \\
\iff \quad & \boxed{W = P\left(\operatorname{ld} P^{\frac{1}{3}}\right)^3 \frac{27{t_w}^3}{4{t_f}^2 K^3}.}
\end{aligned}
$$

Also erhalten wir die Isoeffizienzfunktion $O(P(\operatorname{ld} P)^3)$ und somit eine bessere Skalierbarkeit als für den Cannon'schen Algorithmus.

BEMERKUNG 8.2 Wir haben immer angenommen, dass die optimale sequentielle Komplexität der Matrixmultiplikation N^3 ist. Der Algorithmus von Strassen hat jedoch eine Komplexität von $O(N^{2.87})$.

BEMERKUNG 8.3 Für eine effiziente Implementierung der Multiplikation zweier Matrixblöcke auf einem Prozessor sind die Ausführungen von Abschnitt 2.4 zu beachten.

8.5 *LU*-Zerlegung

Zu lösen sei das lineare Gleichungssystem

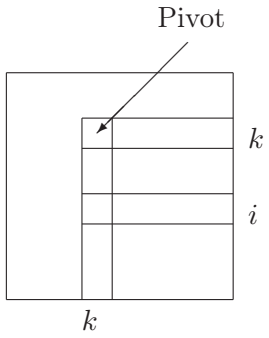
$$Ax = b \tag{8.2}$$

mit einer $N \times N$-Matrix A und entsprechenden Vektoren x, und b.

8.5.1 Sequentielles Verfahren

Das Gauß'sche Eliminationsverfahren

```
(1) for (k = 0; k < N; k++)
(2)     for (i = k + 1; i < N; i++) {
(3)         l_ik = a_ik / a_kk;
(4)         for (j = k + 1; j < N; j++)
(5)             a_ij = a_ij − l_ik · a_kj;
(6)         b_i = b_i − l_ik · b_k;
        }
```



transformiert das Gleichungssystem (8.2) in das Gleichungssystem

$$Ux = d \tag{8.3}$$

mit einer oberen Dreiecksmatrix U. Obige Formulierung hat folgende Eigenschaften:

- Die Matrixelemente a_{ij} für $j \geq i$ enthalten die entsprechenden Einträge von U, d.h. A wird überschrieben.
- Vektor b wird mit den Elementen von d überschrieben.
- Es wird angenommen, das a_{kk} in Zeile (3) immer von Null verschieden ist (keine Pivotisierung).

Die LU-Zerlegung entsteht aus der Gauß-Elimination durch folgende Überlegung:
Jeder einzelne Transformationsschritt, der für festes k und i aus den Zeilen (3) bis (5) besteht, lässt sich als Multiplikation des Gleichungssystems mit einer Matrix

$$\hat{L}_{ik} = \begin{array}{c} \\ \\ \\ \\ \mathrm{i} \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & k & & & \end{array} \\ \begin{pmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & \ddots & & & \\ & & -l_{ik} & & \ddots & & \\ & & & & & 1 & \end{pmatrix} \end{array} = I - l_{ik}E_{ik}$$

(E_{ik} ist die Matrix deren Element $e_{ik} = 1$ ist, und die sonst aus lauter Nullen besteht) von links verstehen, mit l_{ik} aus Zeile (3) des Gauß'schen Eliminationsverfahrens. Somit gilt

$$\begin{aligned} &\hat{L}_{N-1,N-2} \cdot \cdots \cdot \hat{L}_{N-1,0} \cdot \cdots \cdot \hat{L}_{2,0}\hat{L}_{1,0}A = \\ &= \hat{L}_{N-1,N-2} \cdot \cdots \cdot \hat{L}_{N-1,0} \cdot \cdots \cdot \hat{L}_{2,0}\hat{L}_{1,0}b \end{aligned} \tag{8.4}$$

und wegen (8.3) gilt

$$\hat{L}_{N-1,N-2} \cdot \cdots \cdot \hat{L}_{N-1,0} \cdot \cdots \cdot \hat{L}_{2,0}\hat{L}_{1,0}A = U. \tag{8.5}$$

Es gelten folgende Eigenschaften:

1. $\hat{L}_{ik} \cdot \hat{L}_{i',k'} = I - l_{ik}E_{ik} - l_{i'k'}E_{i'k'}$ für $k \neq i'$ ($\Rightarrow E_{ik}E_{i'k'} = 0$).

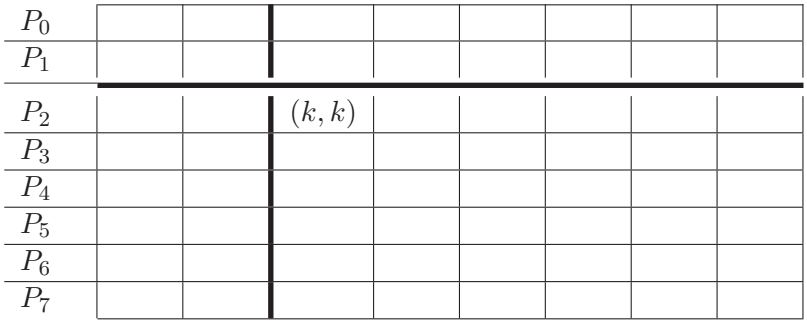


Abbildung 8.10: Zeilenweise Aufteilung einer $N \times N$-Matrix auf N Prozessoren.

2. $(I - l_{ik}E_{ik})(I + l_{ik}E_{ik}) = I$ für $k \neq i$, d.h. $\hat{L}_{ik}^{-1} = I + l_{ik}E_{ik}$.

Wegen 2 und der Beziehung (8.5)

$$A = \underbrace{\hat{L}_{1,0}^{-1} \cdot \hat{L}_{2,0}^{-1} \cdot \ldots \cdot \hat{L}_{N-1,0}^{-1} \cdot \ldots \cdot \hat{L}_{N-1,N-2}^{-1}}_{=:L} U = LU \tag{8.6}$$

Wegen 1, was sinngemäß auch für $\hat{L}_{ik}^{-1} \cdot \hat{L}_{i'k'}^{-1}$ gilt, ist L eine untere Dreiecksmatrix mit $L_{ik} = l_{ik}$ für $i > k$ und $L_{ii} = 1$.

Den Algorithmus zur LU-Zerlegung von A erhält man durch weglassen von Zeile (6) im Gauß-Algorithmus oben. Die Matrix L wird im unteren Dreieck von A gespeichert.

8.5.2 Der Fall $N = P$

Wir betrachten zunächst den einfacheren Fall $N = P$ und zeilenweise Aufteilung wie in Abb. 8.10

Im Schritt k teilt Prozessor P_k die Matrixelemente $a_{k,k}, \ldots, a_{k,N-1}$ allen Prozessoren P_j mit $j > k$ mit, und diese eliminieren in ihrer Zeile. Wir erhalten folgende parallele Laufzeit:

$$\begin{aligned} T_P(N) &= \sum_{\underbrace{m=N-1}_{\substack{\text{Anz. zu}\\ \text{eliminie-}\\ \text{render}\\ \text{Zeilen}}}}^{1} (t_s + t_h + \underbrace{t_w \cdot m}_{\substack{\text{Rest der}\\ \text{Zeile } k}}) \quad \underbrace{\operatorname{ld} N}_{\text{Broadcast}} + \underbrace{m 2t_f}_{\text{Elimination}} \\ &= \frac{(N-1)N}{2} 2t_f + \frac{(N-1)N}{2} \operatorname{ld} N t_w + N \operatorname{ld} N (t_s + t_h) \\ &\approx N^2 t_f + N^2 \operatorname{ld} N \frac{t_w}{2} + N \operatorname{ld} N (t_s + t_h) \end{aligned} \tag{8.7}$$

Für die sequentielle Laufzeit gilt:

$$\begin{aligned} T_S(N) &= \sum_{m=N-1}^{1} \underbrace{m}_{\substack{\text{Zeilen sind}\\ \text{zu elim.}}} \quad \underbrace{2mt_f}_{\substack{\text{Elim. einer}\\ \text{Zeile}}} = \\ &= 2t_f \frac{(N-1)N\,(N(N-1)+1)}{6} \approx \frac{2}{3} N^3 t_f. \end{aligned} \tag{8.8}$$

Wie man aus (8.7) sieht, wächst $N \cdot T_P = \Omega(N^3 \operatorname{ld} N)$ (beachte $P = N$!) asymptotisch schneller als $T_S = \Omega(N^3)$. Der Algorithmus ist also nicht kostenoptimal (Effizienz kann für $P = N \longrightarrow \infty$

nicht konstant gehalten werden). Dies liegt daran, dass Prozessor P_k in seinem Broadcast wartet, bis alle anderen Prozessoren die Pivotzeile erhalten haben.

Wir beschreiben nun eine *asynchrone* Variante, bei der ein Prozessor sofort losrechnet, sobald er die Pivotzeile erhalten hat.

Asynchrone Variante

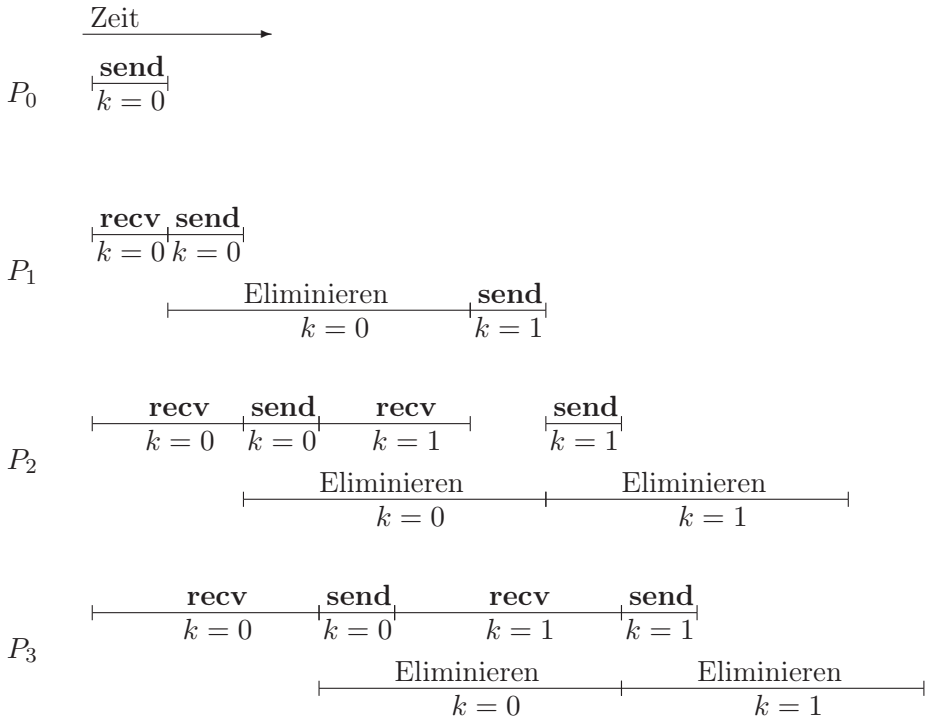
Diese Variante wollen wir erst mal genau formulieren:

PROGRAMM 8.4 (ASYNCHRONE LU-ZERLEGUNG FÜR $P = N$)

```
parallel lu-1
{
    const int N = ...;
    process Π[int p ∈ {0,...,N − 1}]
    {
        double A[N];                    // meine Zeile
        double rr[2][N];                // Puffer für Pivotzeile
        double *r;
        msgid m;
        int j, k;

        if (p > 0) m =arecv(Π_{p−1},rr[0]);
        for (k = 0; k < N − 1; k++)
        {
            if (p == k) send(Π_{p+1},A);
            if (p > k)
            {
                while (¬success(m));    // warte auf Pivotzeile
                if (p < N − 1) asend(Π_{p+1},rr[k%2]);
                if (p > k + 1) m =arecv(Π_{p−1},rr[(k + 1)%2]);
                r = rr[k%2];
                A[k] = A[k]/r[k];
                for (j = k + 1; j < N; j++)
                    A[j] = A[j] − A[k] · r[j];
            }
        }
    }
}
```

Betrachten wir einmal in dem folgenden Bild genauer, was über die Zeit passiert:



P_4 ...

Nach einer Einlaufzeit von p Nachrichtenübertragungen ist die Pipeline gefüllt, und alle Prozessoren sind ständig mit eliminieren beschäftigt. Somit erhält man folgende Laufzeit ($N = P$, immer noch!):

$$\begin{aligned} T_P(N) &= \underbrace{(N-1)(t_s+t_h+t_wN)}_{\text{Einlaufzeit}} + \sum_{m=N-1}^{1} \Big(\underbrace{2mt_f}_{\text{Elim.}} + \underbrace{t_s}_{\substack{\text{Aufsetzzeit (Rechnen+}\mathbf{send}\text{ parallel)}}} \Big) = \\ &= \frac{(N-1)N}{2} 2t_f + (N-1)(2t_s+t_h) + N(N-1)t_w \approx \\ &\approx N^2 t_f + N^2 t_w + N(2t_s+t_h). \end{aligned} \tag{8.9}$$

Den Faktor $\operatorname{ld} N$ von (8.7) sind wir somit los. Für die Effizienz erhalten wir

$$\begin{aligned} E(N,P) &= \frac{T_S(N)}{NT_P(N,P)} = \frac{\frac{2}{3}N^3 t_f}{N^3 t_f + N^3 t_w + N^2(2t_s+t_h)} = \\ &= \frac{2}{3} \frac{1}{1 + \frac{t_w}{t_f} + \frac{2t_s+t_h}{Nt_f}}. \end{aligned} \tag{8.10}$$

Die Effizienz ist also maximal $\frac{2}{3}$. Dies liegt daran, dass Prozessor k nach k Schritten idle bleibt. Verhindern lässt sich dies durch mehr Zeilen pro Prozessor (gröbere Granularität).

8.5.3 Der Fall $N \gg P$

Programm 8.4 von oben lässt sich leicht auf den Fall $N \gg P$ erweitern. Dazu werden die *Zeilen* zyklisch auf die Prozessoren $0, \ldots, P-1$ verteilt. Die aktuelle Pivotzeile erhält ein Prozessor immer vom Vorgänger im Ring.

Die parallele Laufzeit ist

$$\begin{aligned} T_P(N,P) &= \underbrace{(P-1)(t_s+t_h+t_wN)}_{\substack{\text{Einlaufzeit der}\\ \text{Pipeline}}} + \sum_{m=N-1}^{1} \Big(\underbrace{\frac{m}{P}}_{\substack{\text{Zeilen}\\ \text{pro}\\ \text{Prozes-}\\ \text{sor}}} \cdot m2t_f + t_s \Big) = \\ &= \frac{N^3}{P}\frac{2}{3}t_f + Nt_s + P(t_s+t_h) + NPt_w \end{aligned}$$

und somit hat man die Effizienz

$$E(N,P) = \frac{1}{1 + \frac{Pt_s}{N^2\frac{2}{3}t_f} + \ldots}.$$

Wegen der zeilenweisen Aufteilung gilt jedoch in der Regel, dass einige Prozessoren eine Zeile mehr haben als andere.

Eine noch bessere Lastverteilung erzielt man durch eine zweidimensionale Verteilung der Matrix. Dazu nehmen wir an, dass die Aufteilung der Zeilen- und Spaltenindexmenge

$$I = J = \{0, \ldots, N-1\}$$

durch die Abbildungen p und μ für I und q und ν für J vorgenommen wird.

Die nachfolgende Implementierung wird vereinfacht, wenn wir zusätzlich noch annehmen, dass die Datenverteilung folgende Monotoniebedingung erfüllt:

$$\begin{array}{lll} \text{Ist } i_1 < i_2 \text{ und } p(i_1) = p(i_2) & \text{so gelte} & \mu(i_1) < \mu(i_2) \\ \text{ist } j_1 < j_2 \text{ und } q(j_1) = q(j_2) & \text{so gelte} & \nu(j_1) < \nu(j_2) \end{array}$$

Damit entspricht einem Intervall von globalen Indizes $[i_{min}, N-1] \subseteq I$ einer Anzahl von Intervallen lokaler Indizes in verschiedenen Prozessoren, die wie folgt berechnet werden können:

Setze

$$\tilde{I}(p,k) = \{m \in \mathbb{N} \mid \exists i \in I, i \geq k \colon p(i) = p \wedge \mu(i) = m\}$$

und

$$\begin{aligned} \mathit{ibegin}(p,k) &= \begin{cases} \min \tilde{I}(p,k) & \text{falls } \tilde{I}(p,k) \neq \emptyset \\ N & \text{sonst} \end{cases} \\ \mathit{iend}(p,k) &= \begin{cases} \max \tilde{I}(p,k) & \text{falls } \tilde{I}(p,k) \neq \emptyset \\ 0 & \text{sonst.} \end{cases} \end{aligned}$$

Dann kann man eine Schleife

for $(i = k;\ i < N;\ i{+}{+})\ \ldots$

ersetzen durch lokale Schleifen in den Prozessoren p der Gestalt

for $(i = \mathit{ibegin}(p,k);\ i \leq \mathit{iend}(p,k);\ i{+}{+})\ \ldots$

Analog verfährt man mit den Spaltenindizes:

$$\begin{aligned} &\text{Setze} \\ \tilde{J}(q,k) &= \{n \in \mathbb{N} \mid \exists j \in j, j \geq k\colon q(j) = q \wedge \nu(j) = n\} \\ &\text{und} \\ \mathit{jbegin}(q,k) &= \begin{cases} \min \tilde{J}(q,k) & \text{falls } \tilde{J}(q,k) \neq \emptyset \\ N & \text{sonst} \end{cases} \\ \mathit{jend}(q,k) &= \begin{cases} \max \tilde{J}(q,k) & \text{falls } \tilde{J}(q,k) \neq \emptyset \\ 0 & \text{sonst.} \end{cases} \end{aligned}$$

Damit können wir zur Implementierung der LU-Zerlegung für eine allgemeine Datenaufteilung schreiten.

Programm 8.5 (Synchrone LU-Zerlegung mit allg. Datenaufteilung)

```
parallel lu-2
{
    const int N = ...;
    const int √P = ...;

    process Π[int (p,q) ∈ {0,...,√P − 1} × {0,...,√P − 1}]
    {
        double A[N/√P][N/√P];
        double r[N/√P], c[N/√P];
        int i, j, k;

        for (k = 0; k < N − 1; k++)
        {
            I = μ(k); J = ν(k);                    // lokale Indizes

            // verteile Pivotzeile:
            if (p == p(k))
            {                                      // Ich habe Pivotzeile
                for (j = jbegin(q,k); j ≤ jend(q,k); j++)
                    r[j] = A[I][j];                // kopiere Segment der Pivotzeile
                Sende r an alle Prozessoren (x,q) ∀x ≠ p
            }
            else recv(Π_{p(k),q},r);

            // verteile Pivotspalte:
            if (q == q(k))
            {                                      // Ich habe Teil von Spalte k
                for (i = ibegin(p,k+1); i ≤ iend(p,k+1); i++)
                    c[i] = A[i][J] = A[i][J]/r[J];
                Sende c an alle Prozessoren (p,y) ∀y ≠ q
            }
            else recv(Π_{p,q(k)}, c);

            // Elimination:
```

```
            for (i = ibegin(p, k + 1); i ≤ iend(p, k + 1); i + +)
                for (j = jbegin(q, k + 1); j ≤ jend(q, k + 1); j + +)
                    A[i][j] = A[i][j] − c[i] · r[j];
        }
    }
}
```

Analysieren wir diese Implementierung (synchrone Variante):

$$\begin{aligned} T_P(N,P) &= \sum_{m=N-1}^{1} \underbrace{\left(t_s + t_h + t_w \frac{m}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}\, 2}_{\substack{\text{Broadcast}\\ \text{Pivotzeile/-}\\ \text{spalte}}} + \left(\frac{m}{\sqrt{P}}\right)^2 2t_f = \\ &= \frac{N^3}{P}\frac{2}{3}t_f + \frac{N^2}{\sqrt{P}} \operatorname{ld}\sqrt{P} t_w + N \operatorname{ld}\sqrt{P}\, 2(t_s + t_h). \end{aligned}$$

Mit $W = \frac{2}{3}N^3 t_f$, d.h. $N = \left(\frac{3W}{2t_f}\right)^{\frac{1}{3}}$, gilt

$$T_P(W,P) = \frac{W}{P} + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \operatorname{ld}\sqrt{P} \frac{3^{2/3} t_w}{(2t_f)^{\frac{2}{3}}} + W^{\frac{1}{3}} \operatorname{ld}\sqrt{P} \frac{3^{1/3} 2(t_s + t_h)}{(2t_f)^{\frac{1}{3}}}.$$

Die Isoeffizienzfunktion erhalten wir aus $PT_P(W,P) - W \stackrel{!}{=} KW$:

$$\begin{aligned} & \sqrt{P} W^{\frac{2}{3}} \operatorname{ld}\sqrt{P} \frac{3^{2/3} t_w}{(2t_f)^{\frac{2}{3}}} = KW \\ \iff \quad & W = P^{\frac{3}{2}} (\operatorname{ld}\sqrt{P})^3 \frac{9t_w^3}{4t_f^2 K^3} \end{aligned}$$

also

$$W \in O(P^{3/2} (\operatorname{ld}\sqrt{P})^3).$$

Programm 8.5 kann man auch in einer asynchronen Variante realisieren. Dadurch können die Kommunikationsanteile wieder effektiv hinter der Rechnung versteckt werden.

8.5.4 Pivotisierung

Die *LU*-Faktorisierung allgemeiner, invertierbarer Matrizen erfordert Pivotisierung, bzw. ist dies auch aus Gründen der Minimierung von Rundungsfehlern sinnvoll.

Man spricht von voller Pivotisierung, wenn das Pivotelement im Schritt k aus allen $(N-k)^2$ verbleibenden Matrixelementen ausgewählt werden kann, bzw. von teilweiser Pivotisierung (*engl.* „partial pivoting“), falls das Pivotelement nur aus einem Teil der Elemente ausgewählt werden darf. Üblich ist z.B. das maximale Zeilen- oder Spaltenpivot, d.h. man wählt a_{ik}, $i \geq k$, mit $|a_{ik}| \geq |a_{mk}| \quad \forall m \geq k$.

Die Implementierung der *LU*-Zerlegung muss nun die Wahl des neuen Pivotelements bei der Elimination berücksichtigen. Dazu hat man zwei Möglichkeiten:

- Explizites Vertauschen von Zeilen und/oder Spalten: Hier läuft der Rest des Algorithmus dann unverändert (bei Zeilenvertauschungen muss auch die rechte Seite permutiert werden).

- Man bewegt die eigentlichen Daten nicht, sondern merkt sich nur die Vertauschung von Indizes (in einem Integer-Array, das alte Indizes in neue umrechnet.

Die parallelen Versionen eignen sich nun unterschiedlich gut für eine Pivotisierung.

Folgende Punkte sind für die parallele LU-Zerlegung mit teilweiser Pivotisierung zu bedenken:

- Ist der Bereich in dem das Pivotelement gesucht wird in einem einzigen Prozessor (z.B. zeilenweise Aufteilung mit maximalem Zeilenpivot) gespeichert, so ist die Suche sequentiell durchzuführen. Im anderen Fall kann auch sie parallelisiert werden.
- Allerdings erfordert diese parallele Suche nach dem Pivotelement natürlich Kommunikation (und somit Synchronisation), die das Pipelining in der asynchronen Variante unmöglich macht.
- Permutieren der Indizes ist schneller als explizites Vertauschen, insbesondere, wenn das Vertauschen den Datenaustasuch zwischen Prozessoren erfordert. Allerdings kann dadurch eine gute Lastverteilung zerstört werden, falls zufällig die Pivotelemente immer im gleichen Prozessor liegen.

Einen recht guten Kompromiss stellt die zeilenweise zyklische Aufteilung mit maximalem Zeilenpivot und explizitem Vertauschen dar, denn:

- Pivotsuche in der Zeile k ist zwar sequentiell, braucht aber nur $O(N-k)$ Operationen (gegenüber $O((N-k)^2/P)$ für die Elimination); ausserdem wird das Pipelining nicht gestört.
- explizites Vertauschen erfordert nur Kommunikation des Index der Pivotspalte, aber keinen Austausch von Matrixelementen zwischen Prozessoren. Der Pivotspaltenindex wird mit der Pivotzeile geschickt.
- Lastverteilung wird von der Pivotisierung nicht beeinflusst.

8.5.5 Lösen der Dreieckssysteme

Wir nehmen an, die Matrix A sei in $A = LU$ faktorisiert wie oben beschrieben, und wenden uns nun der Lösung eines Systems der Form

$$LUx = b \tag{8.11}$$

zu. Dies geschieht in zwei Schritten:

$$Ly = b \tag{8.12}$$
$$Ux = y. \tag{8.13}$$

Betrachten wir kurz den sequentiellen Algorithmus:

```
// Ly = b:
for (k = 0; k < N; k++) {
    y_k = b_k;   l_kk = 1
    for (i = k + 1; i < N; i++)
        b_i = b_i - a_ik y_k;
}
```

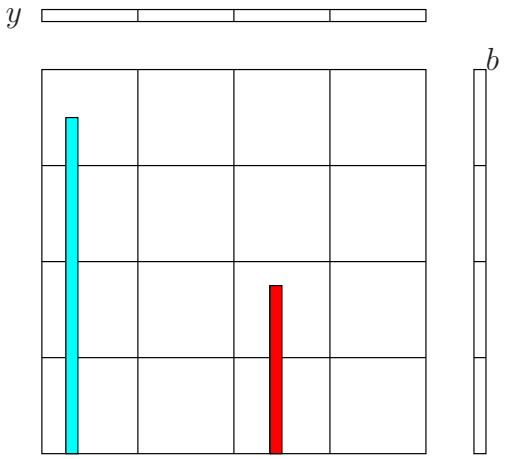


Abbildung 8.11:

```
// Ux = y:
for (k = N − 1; k ≥ 0; k − −) {
    x_k = y_k/a_kk
    for (i = 0; i < k; i + +)
        y_i = y_i − a_ik x_k;
}
```

Dies ist eine spaltenorientierte Version, denn nach Berechnung von y_k bzw. x_k wird sofort die rechte Seite für alle Indizes $i > k$ bzw. $i < k$ modifiziert.

Die Parallelisierung wird sich natürlich an der Datenverteilung der LU-Zerlegung orientieren müssen (falls man ein Umkopieren vermeiden will, was wegen $O(N^2)$ Daten und $O(N^2)$ Operationen sinnvoll erscheint). Betrachten wir hierzu eine zweidimensionale blockweise Aufteilung der Matrix (Abb. 8.11). Die Abschnitte von b sind über Prozessorzeilen kopiert und die Abschnitte von y sind über Prozessorspalten kopiert. Offensichtlich können nach Berechnung von y_k nur die Prozessoren der Spalte $q(k)$ mit der Modifikation von b beschäftigt werden. Entsprechend können bei der Auflösung von $Ux = y$ nur die Prozessoren $(*, q(k))$ zu einer Zeit beschäftigt werden. Bei einer zeilenweise Aufteilung ($Q = 1$) sind somit immer alle Prozessoren beschäftigt.

Wir bringen nun den parallelen Algorithmus für die allgemeine Datenaufteilung.

Programm 8.6 (Auflösen von $LUx = b$ bei allgemeiner Datenaufteilung)

```
parallel lu-solve
{
    const int N = ...;
    const int √P = ...;
    process Π[int (p, q) ∈ {0, ..., √P − 1} × {0, ..., √P − 1}]
    {
        double A[N/√P][N/√P];
        double b[N/√P]; x[N/√P];
        int i, j, k, I, K;

        // Löse Ly = b, speichere y in x.
```

```
// b spaltenweise verteilt auf Diagonalprozessoren.
if (p == q) sende b an alle (p, *);
for (k = 0; k < N; k++)
{
    I = μ(k); K = ν(k);
    if(q(k) == q)                          // nur die haben was zu tun
    {
        if (k > 0 ∧ q(k) ≠ q(k − 1))       // brauche aktuelle b
            recv(Π_{p,q(k−1)}, b);
        if (p(k) == p)
        {                                  // habe Diagonalelement
            x[K] = b[I];                   // speichere y in x!
            sende x[K] an alle (*, q);
        }
        else recv(Π_{p(k),q(k)}, x[k]);
        for (i = ibegin(p, k + 1); i ≤ iend(p, k + 1); i++)
            b[i] = b[i] − A[i][K] · x[K];
        if (k < N − 1 ∧ q(k + 1) ≠ q(k))
            send(Π_{p,q(k+1)}, b);
    }
}
// { y steht in x; x ist spaltenverteilt und
//   zeilenkopiert. Für Ux = y wollen wir y in b
//   speichern Es ist also x in b umzukopieren, wobei
//   b zeilenverteilt und spaltenkopiert sein soll.
for (i = 0; i < N/√P; i++)             // löschen
    b[i] = 0;
for (j = 0; j < N − 1; j++)
    if (q(j) = q ∧ p(j) = p)           // einer muss es sein
        b[μ(j)] = x[ν(j)];
summiere b über alle (p, *), Resultat in (p, p);

// Auflösen von Ux = y (y ist in b gespeichert)
if (p == q) sende b and alle (p, *);
for (k = N − 1; k ≥ 0; k − −)
{
    I = μ(k); K = ν(k);
    if (q(k) == q)
    {
        if (k < N − 1 ∧ q(k) ≠ q(k + 1))
            recv(Π_{p,q(k+1)}, b);
        if (p(k) == p)
        {
            x[K] = b[I]/A[I][K];
            sende x[K] an alle (*, q);
        }
        else recv(Π_{p(k),q(k)}, x[K]);
        for (i = ibegin(p, 0); i ≤ iend(p, 0); i++)
```

```
                    b[i] = b[i] − A[i][K] · x[K];
                if (k > 0 ∧ q(k) ≠ q(k − 1))
                    send(Π_{p,q(k−1)}, b);
            }
        }
    }
}
```

Da zu einer Zeit immer nur $\sqrt{P}$ Prozessoren beschäftigt sind, kann der Algorithmus nicht kostenoptimal sein. Das Gesamtverfahren aus LU-Zerlegung und Auflösen der Dreieckssysteme kann aber immer noch isoeffizient skaliert werden, da die sequentielle Komplexität des Auflösens nur $O(N^2)$ gegenüber $O(N^3)$ für die Faktorisierung ist.

Muss man ein Gleichungssystem für viele rechte Seiten lösen, sollte man ein rechteckiges Prozessorfeld $P \times Q$ mit $P > Q$ verwenden, oder im Extremfall $Q = 1$ wählen. Falls Pivotisierung erforderlich ist, war das ja ohnehin eine sinnvolle Konfiguration.

9 Lösen von tridiagonalen und dünnbesetzten linearen Gleichungssystemen

9.1 Tridiagonalsysteme – optimaler sequentieller Algorithmus

Als Extremfall eines dünnbesetzten Gleichungssystemes betrachten wir

$$Ax = b \qquad (9.1)$$

mit $A \in \mathbb{R}^{N\times N}$ tridiagonal.

$$\begin{pmatrix} * & * & & & & \\ * & * & * & & & \\ & * & * & * & & \\ & & * & * & * & \\ & & & * & * & * \\ & & & & * & * \end{pmatrix}$$

Der optimale Algorithmus ist die Gauß-Elimination, manchmal auch Thomas-Algorithmus genannt.

// Vorwärtselimination (diesmal lösen, nicht LU-Zerlegen):
for $(k = 0;\ k < N-1;\ k++)$ {
  $l = a_{k+1,k}/a_{k,k}$;
  $a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1}$
  $b_{k+1} = b_{k+1} - l \cdot b_k$;
} // $(N-1)\cdot 5$ Rechenoperationen
// Rückwärtseinsetzen:
$x_{N-1} = b_{N-1}/a_{N-1,N-1}$;
for $(k = N-2;\ k \geq 0;\ k--)$ {
  $x_k = (b_k - a_{k,k+1} \cdot x_k + 1)/a_{k,k}$;
} // $(N-1)3+1$ Rechenoperationen

Die sequentielle Komplexität beträgt

$$T_S = 8Nt_f$$

Offensichtlich ist der Algorithmus streng sequentiell!

9.2 Zyklische Reduktion

Betrachte eine Tridiagonalmatrix mit $N = 2M$ (N gerade). *Idee:* Eliminiere in jeder *geraden*

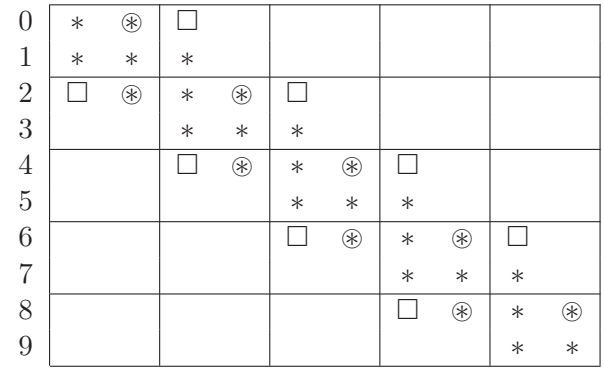


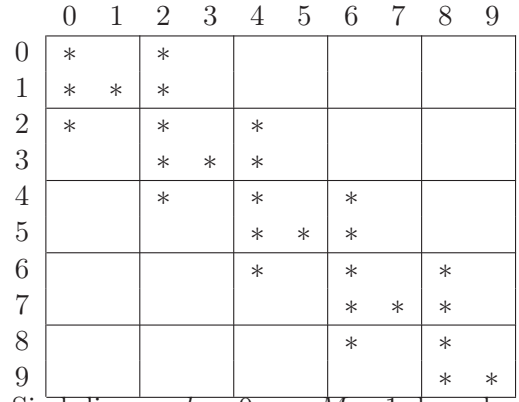
Abbildung 9.1: ⊛ werden entfernt, dabei entsteht fill-in (□).

Zeile k die Elemente $a_{k-1,k}$ und $a_{k+1,k}$ mit Hilfe der ungeraden Zeilen darüber bzw. darunter. Jede gerade Zeile ist damit nur noch mit der vorletzten und übernächsten gekoppelt; da diese gerade sind, wurde die Dimension auf $M = N/2$ reduziert. Das verbleibende System ist wieder tridiagonal, und die Idee kann rekursiv angewandt werden.

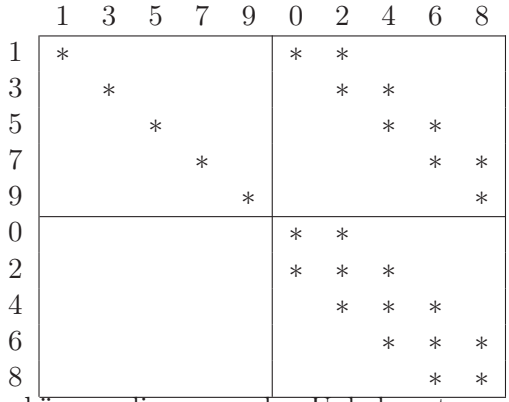
// Elimination aller ungeraden Unbekannten in geraden Zeilen:
for $(k = 1;\ k < N;\ k \mathrel{+}= 2)$
{ // Zeile k modifiziert Zeile $k-1$
  $l = a_{k-1,k}/a_{k,k}$;
  $a_{k-1,k-1} = a_{k-1,k-1} - l \cdot a_{k,k-1}$;
  $a_{k-1,k+1} = -l \cdot a_{k,k+1}$; // fill-in
  $b_{k-1} = b_{k-1} - l \cdot b_k$;
} // $\frac{N}{2} 6t_f$
for $(k = 2;\ k < N;\ k \mathrel{+}= 2)$;
{ // Zeile $k-1$ modifiziert Zeile k
  $l = a_{k,k-1}/a_{k-1,k-1}$;
  $a_{k,k-2} = l \cdot a_{k-1,k-2}$; // fill-in
  $a_{k,k} = l \cdot a_{k-1,k}$;
} // $\frac{N}{2} 6t_f$

Bemerke: Alle Durchläufe beider Schleifen können parallel bearbeitet werden (nehmen wir eine Maschine mit gemeinsamem Speicher an)!

Resultat dieser Elimination ist



bzw. nach Umordnen



Sind die x_{2k}, $k = 0, \ldots, M-1$, berechnet, so können die ungeraden Unbekannten mit

for $(k = 1;\ k < N-1;\ k \mathrel{+}= 2)$
  $x_k = (b_k - a_{k,k-1} \cdot x_{k-1} - a_{k,k+1} \cdot x_{k+1})/a_{k,k}$;
// $\frac{N}{2} 5t_f$
$x_{N-1} = (b_{N-1} - a_{N-1,N-2} \cdot x_{N-2})/a_{N-1,N-1}$;

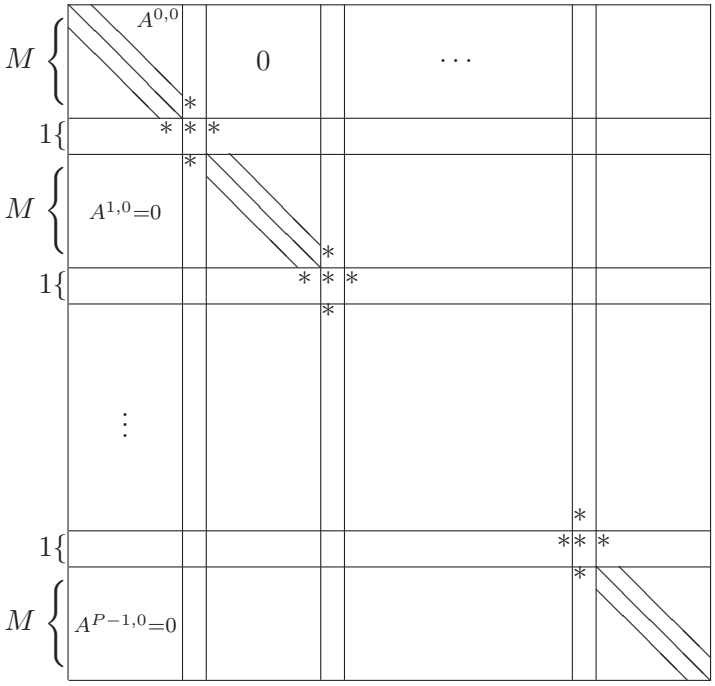


Abbildung 9.2: Matrixaufteilung für die Gebietszerlegung

parallel berechnet werden. Der *sequentielle* Aufwand für die zyklische Reduktion ist somit

$$\begin{aligned} T_S(N) &= (6+6+5)t_f\left(\frac{N}{2}+\frac{N}{4}+\frac{N}{8}+\cdots+1\right) \\ &= 17Nt_f \end{aligned}$$

Dies ist mehr als doppelt so viel wie der optimale sequentielle Algorithmus benötigt. Dafür kann die zyklische Reduktion parallelisiert werden. Die maximal erreichbare Effizienz ist jedoch

$$E_{\max} = \frac{8}{17} \approx 0.47,$$

wobei angenommen wurde, dass alle Operationen optimal parallel ausgeführt werden und Kommunikation umsonst ist (Rückwärtseinsetzten beschäftigt nur $\frac{N}{2}$ Prozessoren!). Nicht berücksichtigt wurde auch, dass die zyklische Reduktion mehr Indexrechnung benötigt!

9.3 Gebietszerlegung

Hier zeigen wir noch einen weiteren Ansatz, der den Vorteil hat, dass er sich (im Prinzip) auch auf allgemeinere Problemstellungen erweitern lässt.

Es sei P die Zahl der Prozessoren und $N = MP + P - 1$ für ein $M \in \mathbb{N}$. Wir unterteilen dann die $N \times N$-Matrix A in P Blöcke à M Zeilen mit je einer Zeile zwischen den Blöcken (Abb. 9.2). Die Unbekannten zwischen den Blöcken bilden das *Interface*. Jeder Block ist mit höchstens zwei Interface-Unbekannten gekoppelt.

Nun sortieren wir Zeilen und Spalten der Matrix so um, dass die Interfaceunbekannten am

Ende stehen. Dies ergibt folgende Gestalt:

$$\left(\begin{array}{cccc|c} A^{0,0} & & & & A^{0,I} \\ & A^{1,1} & & & A^{1,I} \\ & & \ddots & & \vdots \\ & & & A^{P-1,P-1} & A^{P-1,I} \\ \hline A^{I,0} & A^{I,1} & \dots & A^{I,P-1} & A^{I,I} \end{array}\right),$$

wobei $A^{p,p}$ die $M \times M$-Tridiagonalteilmatrixen aus Abb. 9.2 und $A^{I,I}$ eine $P-1 \times P-1$-Diagonalmatrix ist. Die $A^{p,I}$ haben die allgemeine Form

$$A^{p,I} = \left(\begin{array}{c|c|c} \dots & \begin{array}{c} * \\ \\ * \end{array} & \dots \end{array}\right).$$

Idee: Eliminiere Blöcke $A^{I,*}$ in der Blockdarstellung. Dadurch wird $A^{I,I}$ modifiziert, genauer entsteht folgende Blockdarstellung:

$$\left(\begin{array}{cccc|c} A^{0,0} & & & & A^{0,I} \\ & A^{1,1} & & & A^{1,I} \\ & & \ddots & & \vdots \\ & & & A^{P-1,P-1} & A^{P-1,I} \\ \hline 0 & 0 & \dots & 0 & S \end{array}\right),$$

mit $S = A^{I,I} - \sum_{p=0}^{P-1} A^{I,p} \left(A^{p,p}\right)^{-1} A^{p,I}$.

S wird allgemein als „Schurkomplement“ bezeichnet. Alle Eliminationen in $\sum_{p=0}^{P-1}$ können *parallel* durchgeführt werden. Nach Lösen eines Systems $Sy = d$ für die Interfaceunbekannten können die inneren Unbekannten wieder parallel berechnet werden.

S hat Dimension $P-1 \times P-1$ und ist selbst dünn besetzt, wie wir gleich sehen werden.

Ausführen des Plans

① *Transformiere $A^{p,p}$ auf Diagonalgestalt.* ($a_{i,j}$ bezeichnet $(A^{p,p})_{i,j}$, falls nicht anders angegeben):

$\forall p$ parallel
for $(k = 0; k < M-1; k++)$ // untere Diagonale
{
$l = a_{k+1,k}/a_{k,k}$;
$a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1}$;
if $(p > 0)$ $a^{p,I}_{k+1,p-1} = a^{p,I}_{k+1,p-1} - l \cdot a_{k,p-1}$;
// fill-in linker Rand
$b^p_{k+1} = b^p_{k+1} - l \cdot b^p_k$;
} // $(M-1)7t_f$
for $(k = M-1; k > 0; k--)$ // obere Diagonale
{

$l = a_{k-1,k}/a_{k,k}$;
$b^p_{k-1} = b^p_{k-1} - l \cdot b^p_k$;
if $(p > 0)$ $a^{p,I}_{k-1,p-1} = a^{p,I} - l \cdot a^{p,I}_{k,p-1}$; // linker Rand
if $(p < P-1)$ $a^{p,I}_{k-1,p} = a^{p,I}_{k-1,p} - l \cdot a^{p,I}_{k,p}$;// rechter Rand, fill-in
} // $(M-1)7t_f$

② *Eliminiere in* $A^{I,*}$.

$\forall p$ parallel:
if $(p > 0)$
{ // linker Rand $P-1$ im Interface
$l = a^{I,p}_{p-1,0}/a^{p,p}_{0,0}$;
$a^{I,I}_{p-1,p-1} = a^{I,I}_{p-1,p-1} - l \cdot a^{p,I}_{0,p-1}$; // Diagonale in S
if $(p < P-1)$ $a^{I,I}_{p-1,p} = a^{I,I}_{p-1,p} - l \cdot a^{p,I}_{0,p}$;// obere Diag. in S, fill-in
$b^I_{p-1} = b^I_{p-1} - l \cdot b^p_0$;
}
if $(p < P-1)$
{ // rechter Rand
$l = a^{I,p}_{p,M-1}/a^{p,p}_{M-1,M-1}$;
if $(p > 0)$ $a^{I,I}_{p,p-1} = a^{I,I}_{p,p-1} - l \cdot a^{p,I}_{M-1,p-1}$;
// fill-in untere Diag von S
$a^{I,I}_{p,p} = a^{I,I}_{p,p} - l \cdot a^{p,I}_{M-1,p}$;
$b^I_p = b^I_p - l \cdot b^p_{M-1}$;
}

③ *Löse Schurkomplement.* S ist *tridiagonal* mit Dimension $P-1 \times P-1$. Nehme an, dass $M \gg P$ und löse sequentiell. $\longrightarrow 8Pt_f$ Aufwand.

④ *Berechne innere Unbekannte* Hier ist nur eine Diagonalmatrix je Prozessor zu lösen.

$\forall p$ parallel:
for $(k = 0;\ k < M-1;\ k++)$
$x^p_k = (b^p_k - a^{p,I}_{k,p-1} \cdot x^I_{p-1} - a^{p,I}_{k,p} \cdot x^I_p)/a^{p,p}_{k,k}$;
// $M5t_f$

Gesamtaufwand parallel:

$$\begin{aligned} T_P(N,P) &= 14Mt_f + O(1)t_f + 8Pt_f + 5Mt_f = \\ &= 19Mt_f + 8Pt_f \end{aligned}$$

(ohne Kommunikation!)

$$\begin{aligned} E_{\max} &= \frac{8(MP+P-1)t_f}{(19Mt_f + 8Pt_f)\,P} \approx \\ &\underbrace{\approx}_{\text{für } P \ll M} \frac{1}{\frac{19}{8} + \frac{P}{M}} \leq \frac{8}{19} = 0.42 \end{aligned}$$

Bemerkung 9.1 Der Algorithmus benötigt zusätzlichen Speicher für das fill-in. Zyklische Reduktion arbeitet mit Überschreiben der alten Einträge.

9.4 LU-Zerlegung dünnbesetzter Matrizen

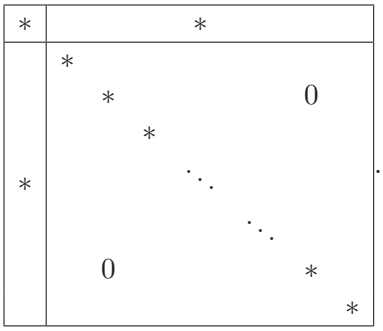
9.4.1 Sequentieller Algorithmus

Was ist eine dünnbesetzte Matrix

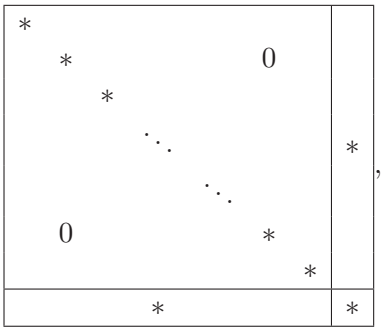
Im allgemeinen spricht man von einer dünnbesetzten Matrix, wenn sie in (fast) jeder Zeile nur eine konstante Anzahl von Nichtnullelementen besitzt. Ist $A \in \mathbb{R}^{N\times N}$, so hat A dann nur $O(N)$ statt N^2 Einträge. Für genügend großes N ist es dann bezüglich Rechenzeit und Speicher günstig, diese große Anzahl von Nullen nicht zu bearbeiten bzw. zu speichern.

Fill-in

Bei der LU-Zerlegung können im Verlauf des Eliminationsprozesses Elemente, die anfänglich Null waren, zu Nichtnullelementen werden. Man spricht dann von *„Fill-in“*. Dies hängt stark von der Struktur der Matrix ab. Als extremes Beispiel betrachte die „Pfeilmatrix“



Bei der Elimination in der natürlichen Reihenfolge (ohne Pivotisierung) läuft die gesamte Matrix voll. Bringt man die Matrix durch Zeilen- und Spaltenvertauschung auf die Form



so entsteht offensichtlich kein Fill-in. Ein wichtiger Punkt bei der LU-Zerlegung dünnbesetzter Matrizen ist das Finden einer Anordnung der Matrix, so dass das Fill-in minimiert wird. Dies hängt jedoch eng zusammen mit der

Pivotisierung

Ist die Matrix A symmetrisch positiv definit (SPD), so ist die LU-Faktorisierung immer numerisch stabil, und es ist *keine* Pivotisierung notwendig. Die Matrix kann also vorab so umgeordnet werden, dass das Fill-in klein wird. Bei einer allgemeinen, invertierbaren Matrix wird man pivotisieren müssen. Dann muss dynamisch während der Elimination ein Kompromiss zwischen numerischer Stabilität und Fill-in gefunden werden.

Deshalb beschränken sich fast alle Codes auf den symmetrisch positiven Fall und bestimmen vorab eine Eliminationsreihenfolge, die das Fill-in minimiert. Eine exakte Lösung dieses Minimierungsproblems ist $\mathcal{NP}$-Vollständig, und man verwendet daher heuristische Verfahren. Bevor wir diese besprechen, definieren wir noch den

Graph einer Matrix

Im symmetrisch positiven Fall lässt sich das Fill-in rein anhand der Null-Struktur der Matrix untersuchen. Zu einem beliebigen, nun nicht notwendigerweise symmetrischen $A \in \mathbb{R}^{N\times N}$ definieren wir einen ungerichteten Graphen $G(A) = (V_A, E_A)$ mit

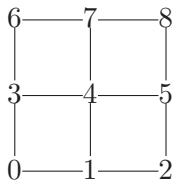
$$\begin{aligned} V_A &= \{0, \ldots, N-1\} \\ (i,j) \in E_A &\iff a_{ij} \neq 0 \vee a_{ji} \neq 0. \end{aligned}$$

Dieser Graph beschreibt sozusagen die direkten Abhängigkeiten der Unbekannten untereinander.

Beispiel:

	0	1	2	3	4	5	6	7	8
0	*	*		*					
1	*	*	*		*				
2		*	*			*			
3	*			*	*		*		
4		*		*	*	*		*	
5			*		*	*			*
6				*			*	*	
7					*		*	*	*
8						*		*	*

A



$G(A)$

Matrix-Anordnungsstrategien

Ein wichtiges Verfahren zur Anordnung von SPD-Matrizen zum Zwecke der Fill-in-Minimierung ist die *„nested disection“*. Wir erläutern das Verfahren zunächst an einem Beispiel. Der Graph $G(A)$ der Matrix A sei ein quadratisches Gitter wie in Abb. 9.3.

Nun teilen wir die Knotenmenge V_A in drei Teile: V_A^0, V_A^1 und V_A^I, so dass

- V_A^0 und V_A^1 sind (möglichst) groß,
- V_A^I ist ein Separator, d.h. bei Herausnahme von V_A^I aus dem Graphen zerfällt dieser in zwei Teile. Somit gibt es *kein* $(i,j) \in E_A$, so dass $i \in V_A^0$ und $j \in V_A^1$.
- V_A^I ist möglichst klein.

Die Abbildung zeigt eine Möglichkeit für eine solche Aufteilung.

Nun ordnet man die Zeilen und Spalten so um, dass zuerst die Indizes V_A^0, dann V_A^1 und zum Schluss V_A^I kommen. Dann wendet man das Verfahren *rekursiv* auf die Teilgraphen mit den Knotenmengen V_A^0 und V_A^1 an. Das Verfahren stoppt, wenn die Graphen eine vorgegebene Größe erreicht haben.

Für unseren Beispielgraphen würde das nach zwei Schritten so aussehen wie in Abb. 9.4.

Für das Beispiel in Abb. 9.4 führt die nested disection Nummerierung auf eine Komplexität von $O(N^{3/2})$ für die LU-Zerlegung. Zum Vergleich benötigt man bei lexikographischer Nummerierung (Bandmatrix) $O(N^2)$ Operationen.

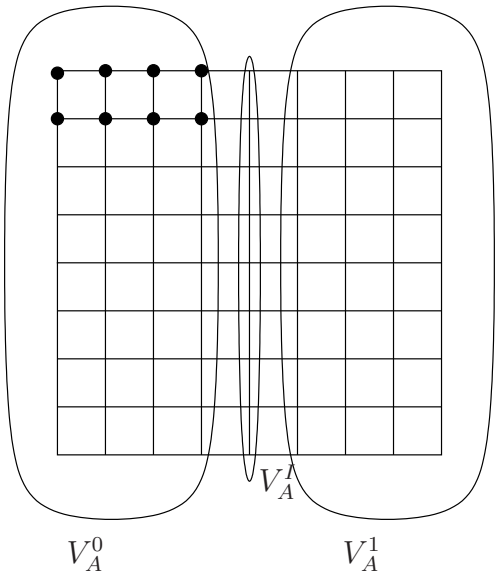


Abbildung 9.3: Graph einer Matrix für nested disection

Datenstrukturen für dünnbesetzte Matrizen

Es gibt eine ganze Reihe von Datenstrukturen zur Speicherung von dünnbesetzten Matrizen. Ziel dabei ist es, eine effiziente Implementierung von Algorithmen zu erlauben. Hierbei ist auf Datenlokalität und möglichst wenig Overhead durch zusätzliche Indexrechnung zu achten.

Eine oft verwendete Datenstruktur ist *„compressed row storage“*. Hat die $N \times N$-Matrix insgesamt M Nichtnullelemente, so speichert man die Matrixelemente Zeilenweise in einem eindimensionalen Feld

double $a[M]$;

ab. Die Verwaltung der Indexinformation geschieht mittels dreier Felder

int $s[N]$, $r[N]$, $j[M]$;

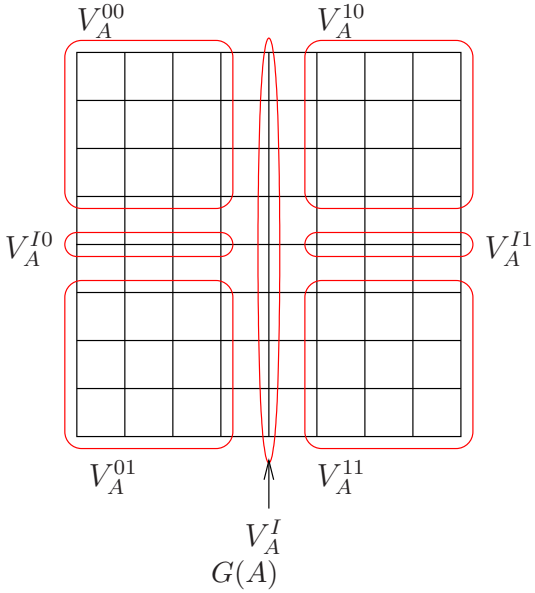
Deren Belegung zeigt die Realisierung des Matrix-Vektor-Produktes $y = Ax$:

for $(i = 0; i < N; i++)$ {
  $y[i] = 0$;
  for $(k = r[i]; k < r[i] + s[i]; k++)$
    $y[i] \mathrel{+}= a[k] \cdot x[j[k]]$;
}.

r gibt den Zeilenanfang, s die Zeilenlänge, und j den Spaltenindex.

Eliminationsformen

Bei der LU-Zerlegung in Kapitel 8 haben wir die sogenannte kij-Form der LU-Zerlegung verwendet. Dabei werden in jedem Schritt k alle a_{ik} für $i > k$ eliminiert, was eine Modifikation aller a_{ij} mit $i, j > k$ erfordert. Diese Situation ist in Abb. 9.5 links dargestellt. Bei der kji-Variante eliminiert man im Schritt k alle a_{kj} mit $j < k$. Dabei werden die a_{ki} mit $i \geq k$ modifiziert. Beachte, dass die a_{kj} von links nach rechts eliminiert werden müssen!



	V_A^{00}	V_A^{01}	V_A^{I0}	V_A^{10}	V_A^{11}	V_A^{I1}	V_A^I
V_A^{00}	$*$	0	$*$				$*$
V_A^{01}	0	$*$	$*$				$*$
V_A^{I0}	$*$	$*$	$*$				$*$
V_A^{10}				$*$		$*$	$*$
V_A^{11}					$*$	$*$	$*$
V_A^{I1}				$*$	$*$	$*$	$*$
V_A^I	$*$	$*$	$*$	$*$	$*$	$*$	$*$

A, umgeordnet

Abbildung 9.4: Matrixstruktur nach zwei Schritten nested disection

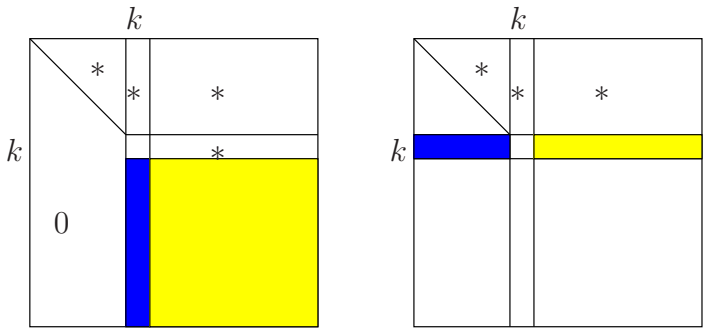


Abbildung 9.5: kij- und kji-Formen der LU-Zerlegung

Bei der folgenden sequentiellen LU-Zerlegung für dünnbesetzte Matrixen werden wir von dieser kji-Variante ausgehen.

Sequentieller Algorithmus

Im folgenden setzen wir voraus:

- Die Matrix A kann in der gegebenen Anordnung ohne Pivotisierung faktorisiert werden. Die Anordnung wurde in geeigneter Weise gewählt, um das Fill-in zu minimieren.
- Die Datenstruktur speichert alle Elemente a_{ij} mit $(i,j) \in G(A)$. Wegen der Definition von $G(A)$ gilt:
$$(i,j) \notin G(A) \;\Rightarrow\; a_{ij} = 0 \wedge a_{ji} = 0.$$
Ist $a_{ij} \neq 0$, so wird in jedem Fall auch a_{ji} gespeichert, *auch wenn dies Null ist.* Die Matrix muss nicht symmetrisch sein.

Die Erweiterung der Struktur von A geschieht rein aufgrund der Information in $G(A)$. So wird, falls $(k,j) \in G(A)$, auch ein a_{kj} formal eliminiert. Dies kann möglicherweise ein fill-in a_{ki} erzeugen, obwohl $a_{ki} = 0$ gilt.

Der nun folgende Algorithmus verwendet die Mengen $S_k \subset \{0, \ldots, k-1\}$, welche im Schritt k genau die Spaltenindizes enthalten, die eliminiert werden müssen.

```
for (k = 0; k < N; k++) S_k = ∅;
for (k = 0; k < N; k++)
{
    // ① erweitere Matrixgraph
    for (j ∈ S_k)
    {
        G(A) = G(A) ∪ {(k, j)};
        for (i = k; i < N; i++)
            if ((j, i) ∈ G(A))
                G(A) = G(A) ∪ {(k, i)};
    }

    // ② Eliminiere
    for (j ∈ S_k)
```

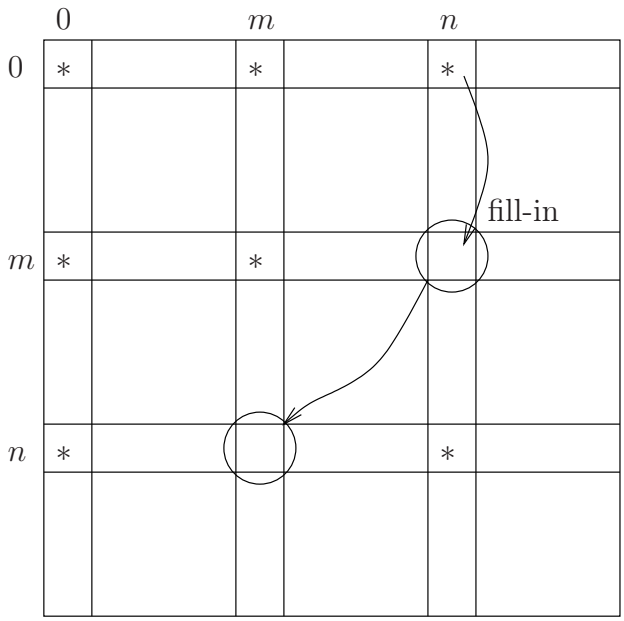


Abbildung 9.6: Beispiel

```
    {                                   // eliminiere a_{k,j}
        a_{k,j} = a_{k,j}/a_{j,j};      // L-Faktor
        for (i = j + 1; i < N; i++)
            a_{k,i} = a_{k,i} − a_{k,j} · a_{j,i};
    }

    // ③ update S_i für i > k, geht wegen Symmetrie von E_A
    for (i = k + 1; i < N; i++)
        if ((k, i) ∈ G(A))              // ⇒ (i, k) ∈ G(A)
            S_i = S_i ∪ {k};
}
```

Betrachten wir in einem Beispiel, wie die S_k gebildet werden (Abb. 9.6). Zu Beginn enthalte $G(A)$ die Elemente

$$G(A) = \{(0,0), (m,m), (n,n), (0,n), (n,0), (0,m), (m,0)\}$$

Im Schleifendurchlauf $k = 0$ wird in Schritt ③ $S_m = \{0\}$ und $S_n = \{0\}$ gesetzt. Nun wird als nächstes bei $k = m$ das $a_{m,0}$ eliminiert. Dies erzeugt das Fill-in (m, n), was wiederum im Schritt ③ bei $k = m$ die Anweisung $S_n = s_n \cup \{m\}$ zur Folge hat. Somit gilt am Anfang des Schleifendurchlaufes $k = n$ korrekt $S_n = \{0, m\}$ und in Schritt ① wird korrekt das Fill-in $a_{n,m}$ erzeugt, *bevor* die Elimination von $a_{n,\mathbf{0}}$ durchgeführt wird. Dies gelingt wegen der Symmetrie von E_A.

9.4.2 Parallelisierung

LU-Zerlegung dünnbesetzter Matrizen besitzt folgende Möglichkeiten zu einer Parallelisierung:

- *grobe Granularität:* In allen 2^d Teilmengen von Indizes, die man durch nested disection der Tiefe d gewinnt, kann man parallel mit der Elimination beginnen (siehe Abb. 9.4). Erst für die Indizes, die den Separatoren entsprechen, ist Kommunikation erforderlich.
- *mittlere Granularität:* Einzelne Zeilen können parallel bearbeitet werden, sobald die entsprechende Pivotzeile lokal verfügbar ist. Dies entspricht dem Parallelismus bei der dichten LU-Zerlegung.
- *feine Granularität:* Modifikationen einer einzelnen Zeile können parallel bearbeitet werden, sobald Pivotzeile und Multiplikator verfügbar sind. Dies nutzt man bei der zweidimensionalen Datenverteilung im dicht besetzten Fall.

Betrachten wir zunächst wieder den Fall $N = P$. Dann ist die Parallelisierung recht einfach:

PROGRAMM 9.2 (LU-ZERLEGUNG FÜR DÜNNBESETZTE MATRIZEN UND $N = P$)

```
parallel sparse-lu-1
{
    const int N = ...;
    process Π[int k ∈ {0,...,N − 1}]
    {// (nur Pseudocode!)
        // ① Belege S
        S = ∅;
        for (j = 0; j < k; j++)
            if ((k, j) ∈ G(A)) S = S ∪ {j};        // der Anfang
        // ②  Zeile k bearbeiten
        for (j = 0; j < k; j++)
            if (j ∈ S_k)
            {
                recv(Π_j, r);                      // warte, bis Π_j die Zeile j schickt
                // erweitere Muster
                for (i = j + 1; i < N; i++)
                {
                    if (i < k ∧ (j, i) ∈ G(A))     // Info ist in r
                        S = S ∪ {i};               // Prozessor i wird Zeile i schicken
                    if ((j, i) ∈ G(A) ∪ {(k, i)})  // Info ist in r
                        G(A) = G(A) ∪ {(k, i)};
                }
                // eliminiere a_{k,j} = a_{k,j}/a_{j,j};   // Info ist in r
                for (i = j + 1; i < N; i++)
                    a_{k,i} = a_{k,i} − a_{k,j} · a_{j,i};   // Info ist in r
            }
        // ③  wegschicken
        for (i = k + 1; i < N; i++)
            if ((k, i) ∈ G(A))                     // lokale Info!
                send Zeile k an Π_i;               // k weiss, dass i die Zeile k braucht.
    }
}
```

In Teil ① wird S initialisiert. Ist zu Beginn von Schritt ② das S immer noch leer, so geht dieser Prozessor sofort zu Schritt ③. Damit beginnen im Fall einer nested disection Ordnung sofort $N/2$

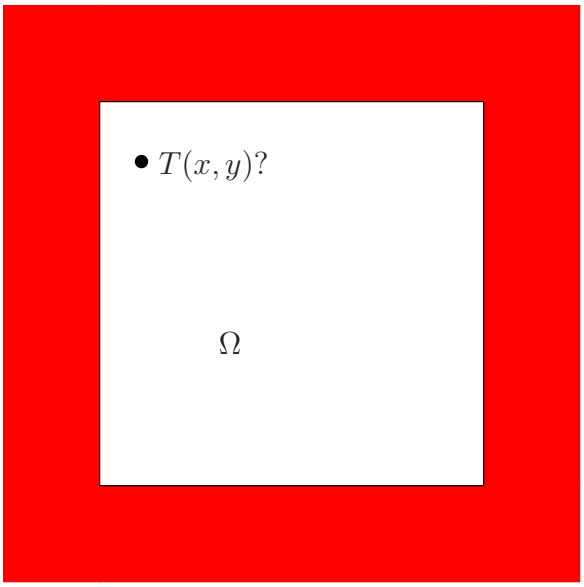


Abbildung 9.7: Problem: Temperaturverteilung in einer Metallplatte.

Prozessoren damit, ihre Zeile wegzuschicken. Im Eliminationsschritt ② erweitert Prozessor k das Muster $G(A)$ und die Menge S, sobald er Zeile j empfangen hat. Prozessor j weiss seinerseits, dass er in Schritt ③ diese Zeile dem Prozessor k schicken muss.

Der Fall $N \gg P$ (Skizze)

Jeder Prozessor hat nun einen ganzen Block von Zeilen. Drei Dinge sind zu tun:

- *Empfange Pivotzeilen* von anderen Prozessoren und speichere diese in der Menge R.
- *Sende fertige Zeilen* aus einem Sendepuffer S zu den Zielprozessoren.
- *Lokale Elimination*
  - Wähle eine Zeile j aus dem Empfangspuffer R.
  - Eliminiere damit lokal alle möglichen $a_{k,j}$.
  - Wenn eine Zeile fertig wird, stecke sie in den Sendepuffer (es kann mehrere Zielprozessoren geben).

9.5 Iterative Lösung dünnbesetzter Gleichungssysteme

9.5.1 Das kontinuierliche Problem und seine Diskretisierung

Als Beispiel betrachten wir folgendes Problem: Eine dünne, quadratische Metallplatte sei an allen Seiten eingespannt. Die zeitlich konstante Temperaturverteilung am Rand der Metallplatte sei bekannt. Welche Temperatur herrscht an jedem inneren Punkt der Metallplatte, wenn sich ein stationärer Zustand eingestellt hat?

Dieser Wärmeleitungsvorgang kann mit einem mathematischen Modell (näherungsweise) beschrieben werden. Die Metallplatte wird durch ein Gebiet $\Omega \subset \mathbb{R}^2$ beschrieben. Gesucht ist

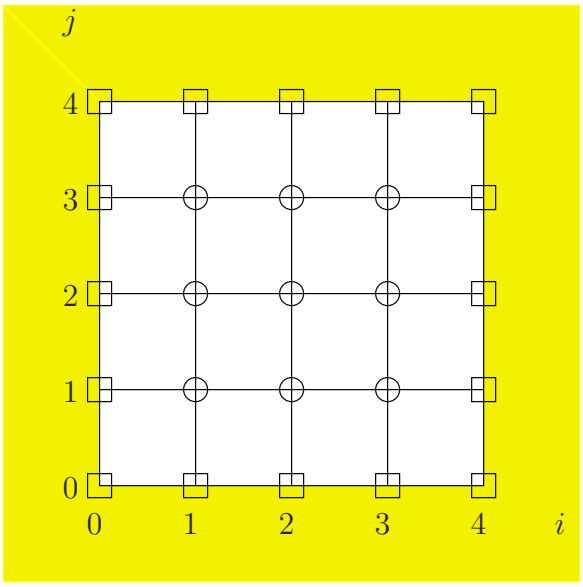


Abbildung 9.8: Äquidistantes Gitter für $N = 4$

die Temperaturverteilung $T(x, y)$ für alle $(x, y) \in \Omega$. Die Temperatur $T(x, y)$ für (x, y) auf dem Rand $\partial\Omega$ sei bekannt. Ist die Metallplatte homogen (überall gleiche Wärmeleitfähigkeit), so wird die Temperatur im Inneren durch die partielle Differentialgleichung

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0\,, \quad T(x, y) = g(x, y) \text{ auf } \partial\Omega \tag{9.2}$$

beschrieben.

Im Rechner kann man die Temperatur nicht an jedem Punkt $(x, y) \in \Omega$ (überabzählbar viele) bestimmen, sondern nur an einigen ausgewählten. Dazu sei speziell $\Omega = [0, 1]^2$ gewählt. Mittels dem Parameter $h = 1/N$, für ein $N \in \mathbb{N}$, wählen wir speziell die Punkte $(x_i, y_j) = (ih, jh)$, für alle $0 \leq i, j \leq N$. Man bezeichnet diese Menge von Punkten auch als regelmässiges, äquidistantes Gitter (siehe Abb. 9.8).

Die Punkte am Rand wurden dabei mit anderen Symbolen (Quadrate) gekennzeichnet als die im Inneren (Kreise). Nur die Temperatur an den inneren Punkten, also für $1 \leq i, j \leq N - 1$, ist unbekannt.

Wie bestimmt man nun die Temperatur T_{ij} am Punkt (x_i, y_j)? Eine gängige Methode, das Verfahren der „Finiten Differenzen“, führt dazu, dass die Temperatur am Punkt (x_i, y_j) durch die Werte an den vier Nachbarpunkten ausgedrückt wird:

$$T_{i,j} = \frac{T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}}{4} \tag{9.3}$$

$$\iff \quad T_{i-1,j} + T_{i+1,j} - 4T_{i,j} + T_{i,j-1} + T_{i,j+1} = 0 \tag{9.4}$$

für $1 \leq i, j \leq N - 1$.

An der Form (9.4) erkennt man, dass alle $(N - 1)^2$ Gleichungen für $1 \leq i, j \leq N - 1$ zusammen ein lineares Gleichungssystem ergeben, welches wir mit

$$AT = b \tag{9.5}$$

bezeichnen. Dabei entspricht $G(A)$ genau dem in Abb. 9.8 gezeichneten Gitter, wenn man die Randpunkte (Quadrate) weglässt. Die rechte Seite b von (9.4) ist nicht etwa Null, sondern enthält die Temperaturwerte am Rand!

Die so berechneten Temperaturwerte $T_{i,j}$ an den Punkten (x_i, y_j) sind *nicht* identisch mit der Lösung $T(x_i, y_i)$ der partiellen Differentialgleichung (9.2). Vielmehr gilt

$$|T_{i,j} - T(x_i, y_i)| \leq O(h^2) \tag{9.6}$$

Diesen Fehler bezeichnet man als „Diskretisierungsfehler". Eine Vergrößerung von N entspricht somit einer genaueren Temperaturberechnung.

Iterationsverfahren

Wir wollen nun das Gleichungssystem (9.5) „iterativ" lösen. Dazu geben wir uns einen beliebigen Wert der Temperatur $T^0_{i,j}$ an jedem Punkt $i \leq i, j \leq N-1$ vor (die Temperatur am Rand ist ja bekannt). Ausgehend von dieser Näherungslösung wollen wir nun eine verbesserte Lösung berechnen. Dazu benutzen wir die Vorschrift (9.3) und setzen

$$T^{n+1}_{i,j} = \frac{T^n_{i-1,j} + T^n_{i+1,j} + T^n_{i,j-1} + T^n_{i,j+1}}{4} \quad \text{für alle } 1 \leq i, j \leq N-1. \tag{9.7}$$

Offensichtlich können die verbesserten Werte $T^{n+1}_{i,j}$ für alle Indizes (i, j) gleichzeitig berechnet werden, da sie nur von den alten Werten $T^n_{i,j}$ abhängen.

Man kann tatsächlich zeigen, dass

$$\lim_{n\to\infty} T^n_{i,j} = T_{i,j} \tag{9.8}$$

gilt. Den Fehler $|T^n_{i,j} - T_{i,j}|$ in der n-ten Näherungslösung bezeichnet man als „Iterationsfehler". Wie groß ist nun dieser Iterationsfehler? Man benötigt ja ein Kriterium bis zu welchem n man rechnen muss.

Dazu betrachtet man, wie gut die Werte $T^n_{i,j}$ die Gleichung (9.4) erfüllen, d.h. wir setzen

$$E^n = \max_{1\leq i,j\leq N-1} \left|T^n_{i-1,j} + T^n_{i+1,j} - 4T^n_{i,j} + T^n_{i,j-1} + T^n_{i,j+1}\right|$$

Üblicherweise verwendet man diesen Fehler nur relativ, d.h. man iteriert so lange bis

$$E^n < \epsilon E^0$$

gilt, also der Anfangsfehler E^0 um den Reduktionsfaktor ϵ reduziert wurde. Dies führt uns zu dem sequentiellen Verfahren:

wähle N, ϵ;
wähle $T^0_{i,j}$;
$E^0 = \max_{1\leq i,j\leq N-1} \left|T^0_{i-1,j} + T^0_{i+1,j} - 4T^0_{i,j} + T^0_{i,j-1} + T^0_{i,j+1}\right|$;
$n = 0$;
while $(E^n \geq \epsilon E^0)$
{
  for $(1 \leq i, j \leq N-1)$
    $T^{n+1}_{i,j} = \frac{T^n_{i-1,j}+T^n_{i+1,j}+T^n_{i,j-1}+T^n_{i,j+1}}{4}$;
  $E^{n+1} = \max_{1\leq i,j\leq N-1} \left|T^{n+1}_{i-1,j} + T^{n+1}_{i+1,j} - 4T^{n+1}_{i,j} + T^{n+1}_{i,j-1} + T^{n+1}_{i,j+1}\right|$;
  $n = n + 1$;
}

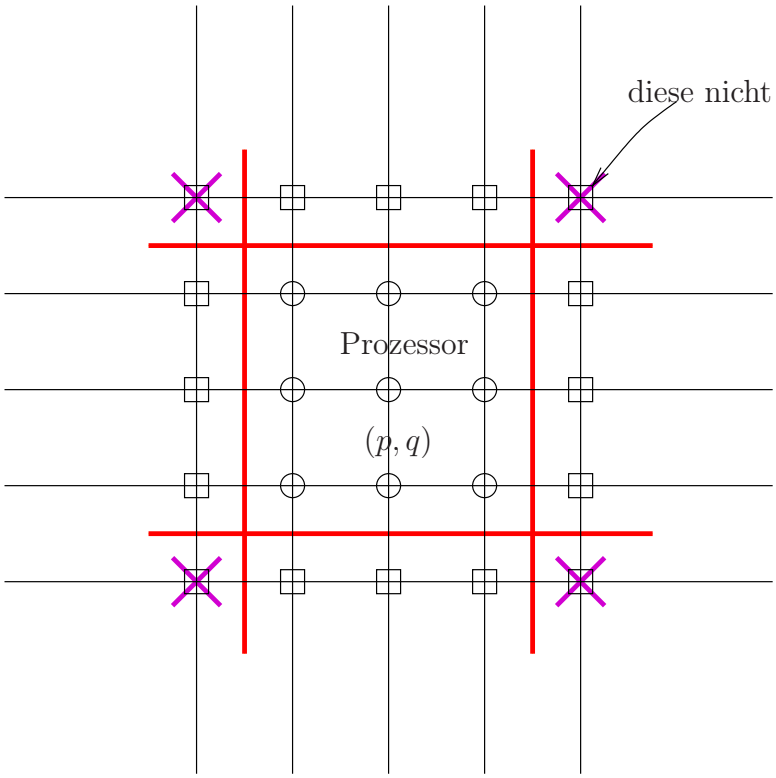


Abbildung 9.9: Matrixaufteilung für das Jacobiverfahren

Kompakter können wir dies alles mit Vektoren schreiben. Zu lösen ist das Gleichungssystem (9.5), also $AT = b$. Die Näherungswerte $T^n_{i,j}$ entsprechen jeweils Vektoren T^n. Formal berechnet sich T^{n+1} als

$$T^{n+1} = T^n + D^{-1}(b - AT^n)$$

mit der Diagonalmatrix $D = \text{diag}(A)$. Dies bezeichnet man als Jacobiverfahren. Der Fehler E^n ergibt sich aus

$$E^n = \|b - A \cdot T^n\|_\infty ,$$

wobei $\|\cdot\|_\infty$ die Maximumnorm eines Vektors ist.

9.5.2 Parallelisierung

Der Algorithmus erlaubt wieder eine datenparallele Formulierung. Die $(N+1)^2$ Gitterpunkte werden dazu durch Partitionierung der Indexmenge $I = \{0, \ldots, N\}$ auf ein $\sqrt{P} \times \sqrt{P}$-Prozessorfeld aufgeteilt. Die Partitionierung geschieht dabei *blockweise* wie in Abb. 9.9 dargestellt. Prozessor (p, q) berechnet somit die Werte $T^{n+1}_{i,j}$ mit $(i, j) \in \{start(p), \ldots, end(p)\} \times \{start(q), \ldots, end(q)\}$. Um dies zu tun, benötigt er jedoch auch Werte $T^n_{i,j}$ aus den Nachbarprozessoren mit $(i, j) \in \{start(p) - 1, \ldots, end(p) + 1\} \times \{start(q) - 1, \ldots, end(q) + 1\}$. Dies sind die Knoten, die in Abb. 9.9 mit Quadraten gekennzeichnet sind! Jeder Prozessor speichert also zusätzlich zu den ihm zugewiesenen Gitterpunkten noch eine zusätzliche Schicht von Gitterpunkten. Der parallele Algorithmus besteht somit aus folgenden Schritten:

Startwerte $T^0_{i,j}$ sind allen Prozessoren bekannt.

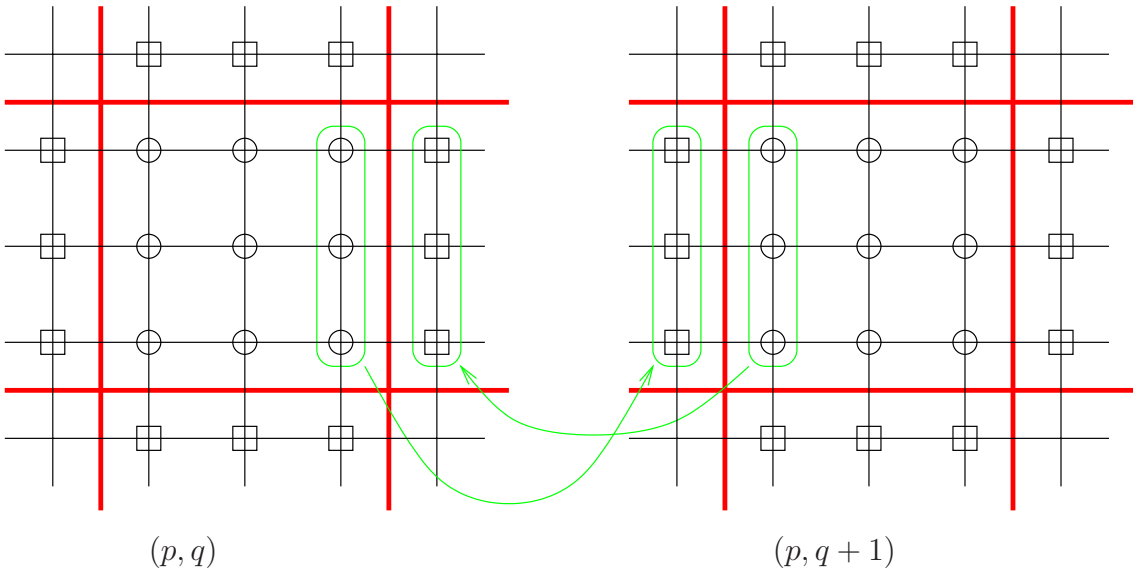


Abbildung 9.10: Austausch der Werte an den Randpunkten

while $(E^n > \epsilon E^0)$
{
berechne $T_{i,j}^{n+1}$ für $(i,j) \in \{start(p), \ldots, end(p)\} \times \{start(q), \ldots, end(q)\}$;
tausche Randwerte (Quadrate) mit Nachbarn aus;
Berechne E^{n+1} auf $(i,j) \in \{start(p), \ldots, end(p)\} \times \{start(q), \ldots, end(q)\}$;
Bilde globales Maximum;
$n = n + 1$;
}

Im Austauschschritt tauschen zwei benachbarte Prozessoren Werte aus wie in Abb. 9.10 skizziert.

Für diesen Austauschschritt verwendet man entweder asynchrone Kommunikation oder synchrone Kommunikation mit Färbung.

Wir berechnen die Skalierbarkeit *einer* Iteration:

$$
\begin{aligned}
W &= T_S(N) = N^2 t_{op} \quad \Longrightarrow N = \sqrt{\frac{W}{t_{op}}} \\
T_P(N,P) &= \underbrace{\left(\frac{N}{\sqrt{P}}\right)^2 t_{op}}_{\text{Berechnung}} + \underbrace{\left(t_s + t_h + t_w \frac{N}{\sqrt{P}}\right) 4}_{\text{Randaustausch}} + \underbrace{(t_s + t_h + t_s) \operatorname{ld} P}_{\substack{\text{globale} \\ \text{Komm.:} \\ \text{Max. für} \\ E^n}} \\
T_P(W,P) &= \frac{W}{P} + \frac{\sqrt{W}}{\sqrt{P}} \frac{4t_w}{\sqrt{t_{op}}} + (t_s + t_h + t_w) \operatorname{ld} P + 4(t_s + t_h) \\
T_O(W,P) &= PT_P - W = \\
&= \sqrt{W}\sqrt{P} \frac{4t_w}{\sqrt{t_{op}}} + P \operatorname{ld} P (t_s + t_h + t_w) + P4(t_s + t_h)
\end{aligned}
$$

Asymptotisch erhalten wir die Isoeffizienzfunktion $W = O(P \operatorname{ld} P)$ aus dem zweiten Term, obwohl der erste Term für praktische Werte von N dominant sein wird. Der Algorithmus ist praktisch optimal skalierbar.

Aufgrund der blockweisen Aufteilung hat man einen Oberfläche-zu-Volumen Effekt: $\frac{N}{\sqrt{P}} / \left(\frac{N}{\sqrt{P}}\right)^2 = \frac{\sqrt{P}}{N}$. In drei Raumdimensionen erhält man $\left(\frac{N}{P^{1/3}}\right)^2 / \left(\frac{N}{P^{1/3}}\right)^3 = \frac{P^{1/3}}{N}$. Für gleiches N und P ist also die Effizienz etwas schlechter als in zwei Dimensionen.

9.5.3 Mehrgitterverfahren

Fragen wir uns nach der Gesamteffizienz eines Verfahrens, so ist die Zahl der Operationen entscheidend. Dabei ist

$$T_S(N) = IT(N) \cdot T_{IT}(N)$$

Wieviele Iterationen nun tatsächlich auszuführen sind, hängt neben N natürlich von dem benutzen Verfahren ab. Dazu erhält man die folgenden Klassifikationen:

$$\begin{aligned}
\text{Jacobi, Gauß-Seidel}: &\quad IT(N) = \mathcal{O}(N^2) \\
\text{SOR mit } \omega_{\text{opt}}: &\quad IT(N) = \mathcal{O}(N) \\
\text{konjugierte Gradienten (CG)}: &\quad IT(N) = \mathcal{O}(N) \\
\text{hierarchische Basis d=2}: &\quad IT(N) = \mathcal{O}(\log N) \\
\text{Mehrgitterverfahren}: &\quad IT(N) = \mathcal{O}(1)
\end{aligned}$$

Die Zeit für eine Iteration $T_{IT}(N)$ ist dabei für alle Verfahren in $\mathcal{O}(N^d)$ mit vergleichbarer Konstante (liegt im Bereich von 1 bis 5).

Wir sehen also, daß z.B. das Mehrgitterverfahren sehr viel schneller ist, als das Jacobi-Verfahren. Auch die Parallelisierung des Jacobi-Verfahrens hilft da nicht, denn es gelten:

$$T_{P,\text{Jacobi}}(N,P) = \frac{\mathcal{O}(N^{d+2})}{P} \quad \text{und} \quad T_{S,\text{MG}}(N) = \mathcal{O}(N^d)$$

Eine Verdopplung von N hat also eine Vervierfachung des Aufwandes des parallelisierten Jacobi-Verfahrens im Vergleich zum `sequentiellen` Mehrgitterverfahren zur Folge!

Das führt uns auf ein grundsätzliches Gebot der parallelen Programmierung:

> Parallelisiere den besten sequentiellen Algorithmus, wenn irgend möglich!

Betrachten wir noch einmal die Diskretisierung der Laplacegleichung $\Delta T = 0$. Diese ergibt das lineare Gleichungssystem

$$Ax = b$$

Dabei wird der Vektor b durch die Dirichlet-Randwerte bestimmt. Sei nun eine Näherung der Lösung durch x^i gegeben. Setze dazu den Iterationsfehler

$$e^i = x - x^i$$

Aufgrund der Linearität von A können wir folgern:

$$Ae^i = \underbrace{Ax}_{b} - Ax^i = b - Ax^i =: d^i$$

Dabei nennen wir d^i den `Defekt` . Eine gute Näherung für e^i berechnet man durch das Lösen von

$$Mv^i = d^i \quad \text{also} \quad v^i = M^{-1}d^i$$

Dabei ist M leichter zu lösen, als A (in $\mathcal{O}$(N) Schritten, wenn $x \in \mathbb{R}^N$). Für spezielle M bekommen wir bereits bekannte Iterationsverfahren :

$$\begin{aligned} M &= I && \rightarrow \text{Richardson} \\ M &= \text{diag}(A) && \rightarrow \text{Jacobi} \\ M &= L(A) && \rightarrow \text{Gauß-Seidel} \end{aligned}$$

Wir erhalten lineare Iterationsverfahren der Form

$$x^{i+1} = x^i + \omega M^{-1}(b - Ax^i)$$

Dabei stellt das $\omega \in [0, 1]$ einen Dämpfungsfaktor dar. Für den Fehler $e^{i+1} = x - x^{i+1}$ gilt:

$$e^{i+1} = (I - \omega M^{-1}A)e^i$$

Dabei bezeichnen wir die Iterationsmatrix $I - \omega M^{-1}A$ mit S. Es ligt genau dann Konvergenz ($\lim_{i\to\infty} e^i = 0$) vor , wenn der betragsgrößte Eigenwert von S echt(!) kleiner als Eins ist.

Glättungseigenschaft

Ist die Matrix A symmetrisch und positiv definit, so hat sie nur reelle, positive Eigenwerte λ_k zu Eigenvektoren z_k. Die Richardson-Iteration

$$x^{i+1} = x^i + \omega(b - Ax^i)$$

führt wegen M = I auf die Fehlerfortpfanzung

$$e^{i+1} = (I - \omega A)e^i$$

Nun setzten wir den Dämpfungsfaktor $\omega = \frac{1}{\lambda_{\max}}$ und betrachten $e^i = z_k(\forall k)$. Dann erhalten wir

$$e^{i+1} = \left(I - \frac{1}{\lambda_{\max}}A\right) z_k = z_k - \frac{\lambda_k}{\lambda_{\max}} z_k = \left(1 - \frac{\lambda_k}{\lambda_{\max}}\right) e^i$$

$$\left(1 - \frac{\lambda_k}{\lambda_{\max}}\right) = \begin{cases} 0 & \lambda_k = \lambda_{\max} \\ \approx 1 & \lambda_k \text{ klein } (\lambda_{\min}) \end{cases}$$

Im Fall kleiner Eigenwerte haben wir also eine schlechte Dämpfung des Fehlers (sie liegt in der Größenordnung von 1 - $\mathcal{O}$ (h^2)). Dieses Verhalten ist für die Jacobi und Gauß-Seidel Iterationsverfahren qualitativ identisch.

Zu kleinen Eigenwerten gehören aber langwellige Eigenfunktionen. Diese langwelligen Fehler werden also nur sehr schlecht gedämpft. Anschaulich betrachtet bieten die Iterationsverfahren nur eine lokale Glättung des Fehlers, auf dem sie arbeiten, denn sie ermitteln neue Iterationswerte ja nur aus Werten in einer lokalen Nachbarschaft. Schnelle Oszillationen können werden schnell herausgeglättet, während langwellige Fehler die lokalen Glättungen weitgehend unverändert überstehen.

Zu Veranschaulichung betrachen wir folgendes Beispiel:

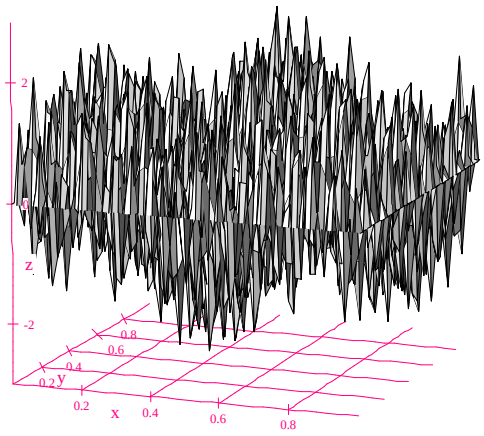


Abbildung 9.11: Initialisierungfehler

Die Laplacegleichung $-\Delta u = f$ wird mittles eines Fünf-Punkte-Sterns auf einem strukturierten Gitter diskretisiert. Die zugehörigen Eigenfunktionen sind $\sin(\nu\pi x)\sin(\mu\pi y)$, wobei $1 \leq \nu$ und $\mu \leq h^{-1}-1$ gelten. Wir setzen $h = \frac{1}{32}$ und den Initialisierungsfehler auf

$$e^0 = \sin(3\pi x)\sin(2\pi y) + \sin(12\pi x)\sin(12\pi y) + \sin(31\pi x)\sin(31\pi y).$$

Mit $\omega = \frac{1}{\lambda_{\max}}$ erhält man die Dämpfungfaktoren (pro Iteration) für die Richardson Iteration durch 0.984, 0.691 und 0 für die einzelnen Eigenfunktionen. Die untenstehenden Graphen zeigen den Initialisierungsfehler und den Fehler nach einer bzw. fünf Iterationen.

Daher kommt man auf die Idee, diese langwelligen Fehler auf gröberen Gittern darzustellen, nachdem die schnellen Ozillationen herausgeglättet worden sind. Auf diesen gröberen Gittern ist dann der Aufwand, um die Fehlerkurve zu glätten, geringer. Weil die Kurve nach der Vorglättung einigermaßen glatt ist, ist diese Restriktion auf weniger Gitterpunkte gut möglich.

Gitterhierarhie

Man erstellt also eine ganze Folge von Gittern verschiedener Feinheit und glättet zunächst auf dem feinsten die kurzwelligen Fehlerfunktionen heraus, geht dann auf das nächst gröbere Gitter über usw. .

Entsprechend erhält man eine ganze Folge linearer Systeme

$$A_l x_l = b_l,$$

da ja die Anzahl der Gitterpunkte N und damit die Länge von x auf gröberen Gittern schwindet.

Natürlich will man nach dieser Restriktion wieder auf das ursprüngliche feine Gitter zurückkehren. Dazu führt man eine Grobgitterkorrektur durch. Nehmen wir dazu an, wir seinen auf Gitterstufe l, betrachten also das LGS

$$A_l x_l = b_l$$

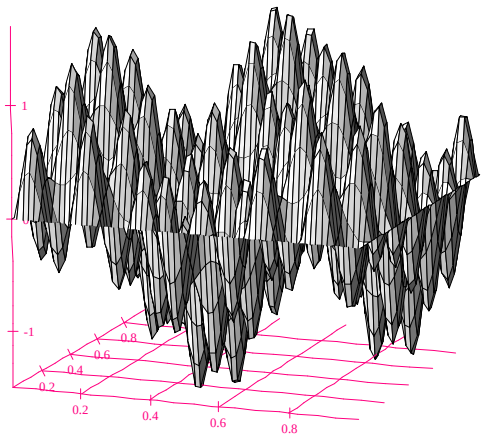


Abbildung 9.12: Fehler nach einer Iteration

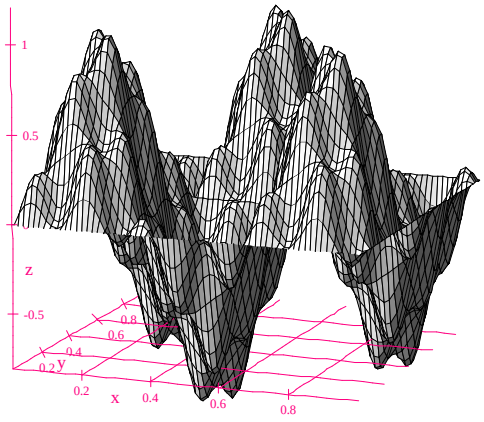


Abbildung 9.13: Fehler nach fünf Iterationen

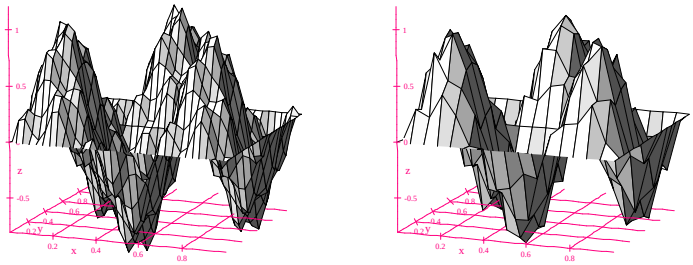


Abbildung 9.14: Der Fehler nach fünf Iterationen auf gröberen Gittern

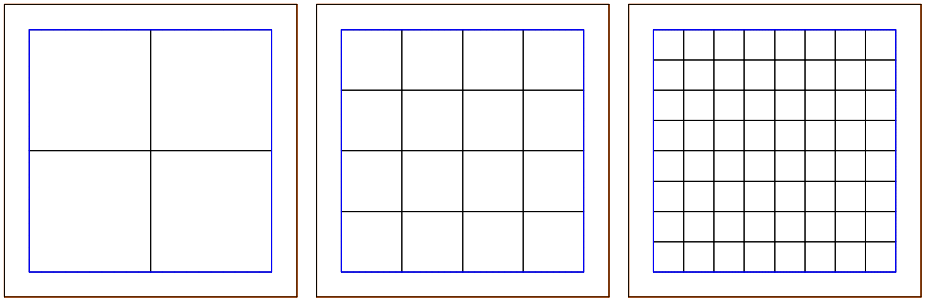


Abbildung 9.15: Strukturierte Gitterfolge

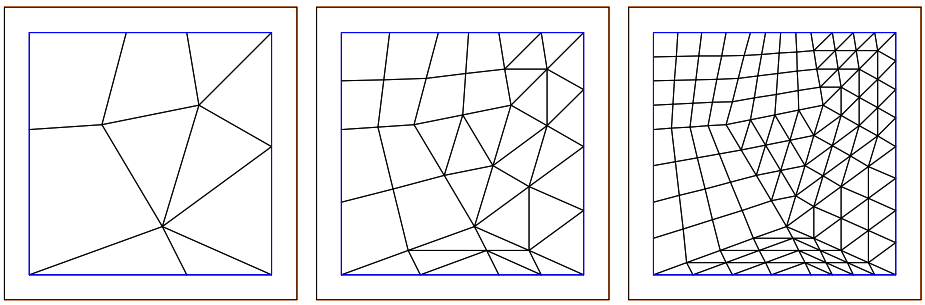


Abbildung 9.16: Unstrukturierte Gitterfolge

Auf dieser Stufe ist eine Iterierte x_l^i mit Fehler e_l^i gegeben, also die Fehlergleichung

$$Ae_l^i = b_l - A_l x_l^i$$

Nehmen wir an, x_l^i ist Ergebnis von ν_1 Jacobi, Gauß-Seidel oder ähnlichen Iterationen. Dann ist e_l^i verhältnismäßig glatt und somit auch gut auf einem gröberen Gitter darstellbar, das heißt, er kann von einem gröberen Gitter gut auf das feinere interpoliert werden. Sei dazu v_{l-1} der Fehler auf dem gröberen Gitter. Dann ist in guter Näherung

$$e_l^i \approx P_l v_{l-1}$$

Dabei ist P_l eine Interpolationsmatrix (Prolongation), die eine linerare Interpolation ausführt und so aus dem Grobgittervektor einen Feingittervektor macht.

Durch Kombination der obigen Gleichungen erhält man die Gleichung für v_{l-1} durch

$$R_l A P_l v_{l-1} = R_l(b_l - A_l x_l^i)$$

Dabei ist $R_l A P_l =: A_{l-1} \in \mathbb{R}^{N_{l-1} \times N_{l-1}}$ und R_l ist die Restriktionsmatrix, für die man z.B. $R_l = P_l^T$ nimmt.

Ein sogenanntes Zweigitterverfahren besteht nun aus den beiden Schritten:

1. ν_1 Jacobi-Iterationen (auf Stufe l)
2. Grobgitterkorrektur $x_l^{i+1} = x_l^i + P_l A_{l-1}^{-1} R_l(b_l - A_l x_l^i)$

Die rekursive Anwendung führt auf die Mehrgitterverfahren.

1	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0
0	1	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0
$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
0	$\frac{1}{2}$	0	$\frac{1}{2}$
0	0	1	0
0	0	$\frac{1}{2}$	$\frac{1}{2}$
0	0	0	1

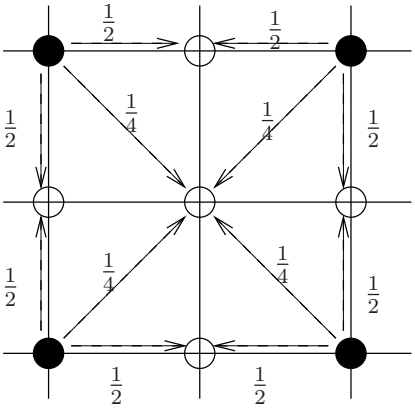


Abbildung 9.17: Prolongationsmatrix und Wirkung auf dem Gitter

mgc(l, x_l, b_l)
{
  if (l == 0) $x_0 = A_0^{-1} b_0$;
  else {
    ν_1 Iterationen Eingitterverfahren auf $A_l x_l = b_l$; //Vorglättung
    $d_{l-1} = R_l(b_l - A_l x_l)$;
    $v_{l-1} = 0$;
    for ($g = 1, \ldots, \gamma$)
      mgc($l-1, v_{l-1}, d_{l-1}$);
    $x_l = x_l + P_l v_{l-1}$;
    ν_2 Iterationen Eingitterverfahren auf $A_l x_l = b_l$; //Nachglättung
  }
}

Dabei ist es ausreichend, $\gamma = 1, \nu_1 = 1, \nu_2 = 0$ zu setzen, damit die Iterationszahl in $\mathcal{O}(1)$ liegt.

Der einmalige Durchlauf von der Stufe l hoch auf Stufe 0 und zurück wird als V-Zyklus bezeichnet.

Betrachten wir den Aufwand für den Fall eines zweidimensionalen strukturierten Gitters:

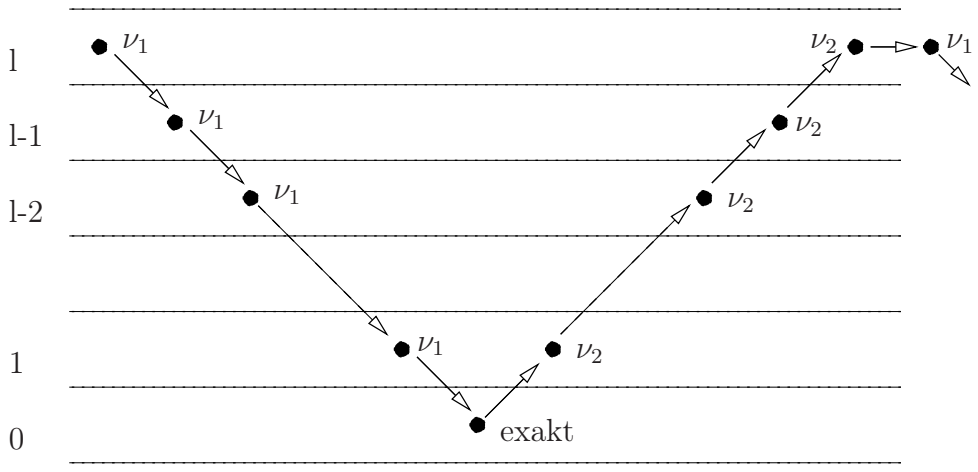


Abbildung 9.18: V-Zyklus mit $\gamma = 1$

Dabei seien N die Anzahl der Gitterpunkte in einer Zeile auf der feinsten Stufe, und $C := t_{\text{op}}$.

$$\begin{aligned} T_{\text{IT}}(N) &= \underbrace{CN^2}_{\text{Stufe l}} + \underbrace{\frac{CN^2}{4}}_{\text{Stufe l-1}} + \frac{CN^2}{16} + \ldots + \underbrace{G(N_0)}_{\text{Grobgitter}} \\ &= CN^2 \underbrace{(1 + \frac{1}{4} + \frac{1}{16} + \ldots)}_{\frac{4}{3}} + G(N_0) \\ &= \frac{4}{3} CN_l + G(N_0) \end{aligned}$$

Parallelisierung

Zur Datenverteilung in der Gitterhierarchie auf die einzelnen Prozessoren ist zu berücksichtigen:

- In der Grobgitterkorrektur muß geprüft werden, ob zur Berechnung der Knotenwerte im Grobgitter Kommunikation erforderlich ist.
- Wie verfährt man mit den gröbsten Gittern, in denen pro Dimensionsrichtung die Anzahl der Unbekannten kleiner als die Anzahl der Prozessoren wird?

Zur Illustrierung des Verfahrens betrachten wir den eindimensionalen Fall. Siehe dazu Abb. 9.19. Die Verteilung im höherdimensionalen Fall erfolgt entsprechend dem Tensorprodukt (schachbrettartig).

Die Prozessorgrenzen werden gewählt bei $p \cdot \frac{1}{P} + \epsilon$, also die Knoten, die auf dem Rand zwischen zwei Prozessoren liegen, noch dem „vorderen “zugeteilt.

Zu bemerken ist, daß der Defekt, der in der Grobgitterkorrektur restringiert wird, nur auf den eigenen Knoten berechnet werden kann, nicht aber im Overlap!

Um dem Problem der schwindenden Anzahl von Knoten in den gröbsten Gittern zu begegnen, nutzt man sukzessive weniger Prozessoren. Sei dazu $a := \text{ld}\, P$ und wieder $C := t_{\text{op}}$. Auf Stufe 0 wird nur ein Prozessor beschäftigt, erst auf Stufe a sind alle beschäftigt.

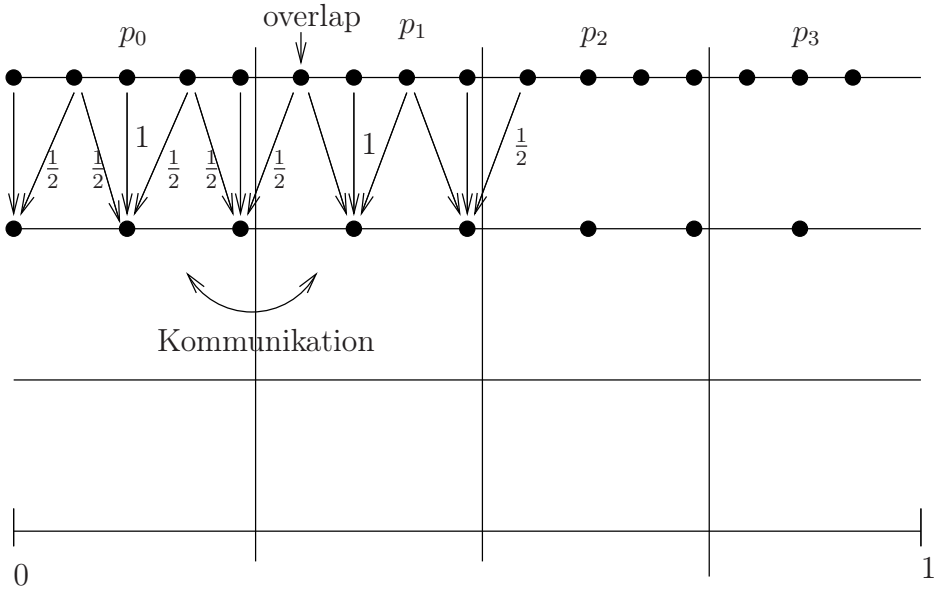


Abbildung 9.19: Eindimensionale Aufteilung bei 4 Prozessoren

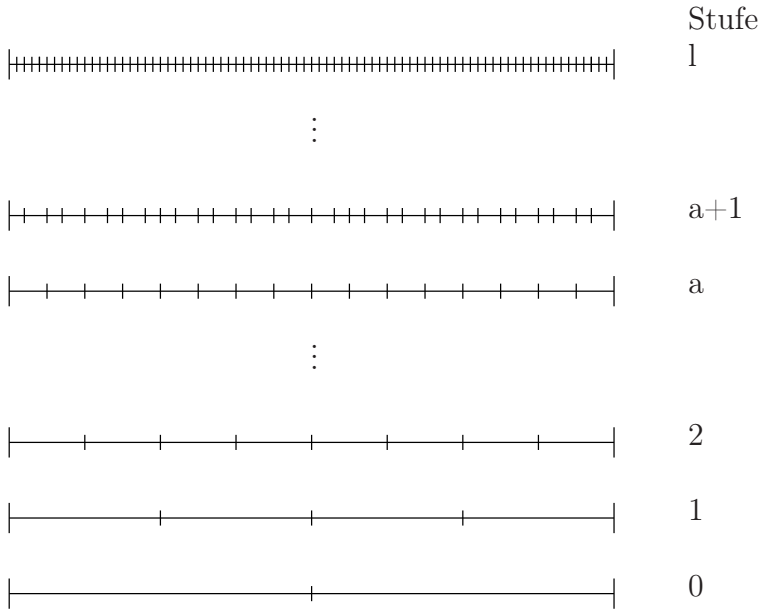


Abbildung 9.20: Eindimensionale Gitterhierarchie

Stufe	Knoten	Prozessoren	Aufwand
l	$N_l = 2^{l-a}N_a$	$P_l = P$	$T = 2^{l-a}CN_0$
a+1	$N_{a+1} = 2N_a$	$P_{a+1} = P$	$T = 2CN_0$
a	N_a	$P_a = P$	$T = CN_0$
2	$N_2 = 4N_0$	$P_2 = 4P_0$	$T = \frac{CN_2}{4} = CN_0$
1	$N_1 = 2N_0$	$P_1 = 2P_0$	$T = \frac{CN_1}{2} = \frac{C2N_0}{2} = CN_0$
0	N_0	$P_0 = 1$	$G(N_0) \overset{\text{sei}}{\approx} CN_0$

Betrachten wir den Gesamtaufwand: Von Stufe 0 bis Stufe a ist $\frac{N}{P}$ konstant, also wächst T_P wie $\operatorname{ld} P$. Daher bekommen wir

$$T_P = \operatorname{ld} P \cdot CN_0$$

In den höheren Stufen bekommen wir

$$T_P = C \cdot \frac{N_l}{P} \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) = 2C\frac{N_l}{P}$$

Den Gesamtaufwand bildet dann die Summe der beiden Teilaufwände.

Nicht berücksichtigt ist dabei die Kommunikation zwischen den Prozessoren.

Die folgende Tabelle zeigt, wie sich der Einsatz des Mehrgitterverfahrens auf die Anzahl der auszuführenden Iterationsschritte auswirkt. Aufgezeichnet wird die Anzahl der eingesetzten Prozessoren gegen die Wahl der Gitterfeinheit, die verwendet wurde. Es wurde eine Reduktion des Fehlers auf 10^{-6} durchgeführt.

P/l	5	6	7	8
1	10			
4	10	10		
16	11	12	12	
64	13	12	12	12

Die zweite Tabelle zeigt die entsprechenden Iterationzeiten in Sekunden an (gerechnet auf Cray T3D).

P/l	5	6	7	8
1	3.39			
4	0.95	3.56		
16	0.32	1.00	3.74	
64	0.15	0.34	1.03	3.85

10 Partikelmethoden

10.1 Einführung

Bei Partikelmethoden geht es darum, wie sich N Teilchen (oder Körper) unter dem Einfluss eines Kraftfeldes bewegen. Das Kraftfeld selbst hängt wieder von der Position der Teilchen ab.

Als einfachstes Beispiel betrachten wir das N-Körper-Problem: Gegeben seien N punkförmige Massen $m_1, \ldots, m_N$ an den Positionen $x_1, \ldots, x_N \in \mathbb{R}^3$. Die von Körper j auf Körper i ausgeübte Gravitationskraft (siehe Abb. 10.1) ist gegeben durch das Newton'sche Gravitationsgesetz :

$$F_{ij} = \underbrace{\frac{\gamma\, m_i m_j}{\|x_j - x_i\|^2}}_{\text{Stärke}} \cdot \underbrace{\frac{x_j - x_i}{\|x_j - x_i\|}}_{\substack{\text{Einheitsvektor von} \\ x_i \text{ in Richtung } x_j}} \tag{10.1}$$

Die Gravitationskraft lässt sich auch als Gradient eines Potential schreiben:

$$F_{ij} = m_i \frac{\gamma\, m_j (x_j - x_i)}{\|x_j - x_i\|^3} = m_i\, \nabla\left(\frac{\gamma\, m_j}{\|x_j - x_i\|}\right) = m_i\, \nabla \phi_j(x_i). \tag{10.2}$$

$\phi_y(x) = \frac{\gamma\, m}{\|y-x\|}$ heißt Gravitationspotential der Masse m an der Position $y \in \mathbb{R}^3$.

Die Bewegungsgleichungen der betrachteten N Körper erhält man aus dem Gesetz Kraft=Masse×Beschleunigu

$$\begin{aligned} &\text{für } i = 1, \ldots, N \\ m_i \tfrac{dv_i}{dt} &= m_i\, \nabla \left(\sum_{j \neq i} \frac{\gamma\, m_j}{\|x_j - x_i\|} \right) && (10.3) \\ \tfrac{dx_i}{dt} &= v_i && (10.4) \end{aligned}$$

Dabei ist $v_i(t)\colon \mathbb{R} \to \mathbb{R}^3$ die Geschwindigkeit des Körpers i und $x_i(t)\colon \mathbb{R} \to \mathbb{R}^3$ die Position in Abhängigkeit von der Zeit.

Wir erhalten also ein System gewöhnlicher Differentialgleichungen der Dimension $6N$ (drei Raumdimensionen) für dessen Lösung noch Anfangsbedingungen für Position und Geschwindigkeit erforderlich sind:

$$x_i(0) = x_i^0, \quad v_i(0) = v_i^0, \quad \text{für } i = 1, \ldots, N \tag{10.5}$$

Die Integration der Bewegungsgleichungen (10.3) und (10.4) erfolgt numerisch, da nur für $N = 2$ geschlossene Lösungen möglich sind (Kegelschnitte, Kepler'sche Gesetze). Das einfachste Verfahren ist das explizite Euler-Verfahren:

Diskretisierung in der Zeit:
$$\begin{aligned} t^k &= k \cdot \Delta t, \\ v_i^k &= v_i(t^k), \\ x_i^k &= x_i(t^k). \end{aligned}$$

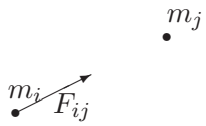


Abbildung 10.1: N-Körper Problem

$$\left.\begin{aligned} v_i^{k+1} &= v_i^k + \Delta t \cdot \nabla\left(\sum_{j\neq i} \frac{\gamma\, m_j}{\| \underbrace{x_j^k - x_i^k}_{\text{„explizit“}} \|}\right) \\ x_i^{k+1} &= x_i^k + \Delta t \cdot v_i^k \end{aligned}\right\} \text{ für } i = 1, \ldots, N \tag{10.6}$$

Sicher gibt es bessere Verfahren als das explizite Eulerverfahren, das nur von der Konvergenzordnung $O(\Delta t)$ ist, für die Parallelisierung spielt dies jedoch keine große Rolle, da die Struktur anderer Verfahren ähnlich ist. Ein anderes Problem ist schwerwiegender:

In der Kraftberechnung hängt die Kraft auf einen Körper i von der Position *aller* anderen Körper $j \neq i$ ab. Der Aufwand für eine Kraftauswertung (die mindestens einmal pro Zeitschritt notwendig ist) steigt also wie $O(N^2)$ an. In den Anwendungen kann $N = 10^6, \ldots, 10^9$ sein, was einen enormen Rechenaufwand bedeutet.

Ein Schwerpunkt dieses Kapitels ist daher die Vorstellung verbesserter sequentieller Algorithmen zur schnellen Kraftauswertung. Diese berechnen die Kräfte näherungsweise mit dem Aufwand $O(N \log N)$ (Es gibt auch Verfahren mit $O(N)$ Komplexität, die wir aus Zeitgründen weglassen). Bevor wir diese Verfahren einführen, betrachten wir noch die (simple) Parallelisierung des naiven $O(N^2)$ Algorithmus.

10.2 Parallelisierung des Standardverfahrens

Der $O(N^2)$-Algorithmus ist recht einfach zu parallelisieren. Jeder der P Prozessoren erhält $\frac{N}{P}$ Körper. Um die Kräfte für alle seine Körper zu berechnen, benötigt ein Prozessor alle anderen Körper. Dazu werden die Daten im Ring zyklisch einmal herumgeschoben.

Analyse:

$$\begin{aligned} T_S(N) &= c_1 N^2 \\ T_P(N,P) &= c_1 P \Big(\underbrace{\frac{N}{P} \cdot \frac{N}{P}}_{\substack{\text{Block } p \\ \text{mit } q}}\Big) + \underbrace{c_2 P \frac{N}{P}}_{\text{Kommunikation}} = \\ &= c_1 \frac{N^2}{P} + c_2 N \\ E(P,N) &= \frac{c_1 N}{\left(c_1 \frac{N^2}{P} + c_2 N\right) P} = \frac{1}{1 + \frac{c_2}{c_1} \cdot \frac{P}{N}} \end{aligned}$$

konstante Effizienz für $N = \Theta(P)$.

Damit ist die Isoeffizienzfunktion wegen $W = c_1 N^2$

$$W(P) = \Theta(P^2)$$

(natürlich in Bezug auf den suboptimalen Standardalgorithmus).

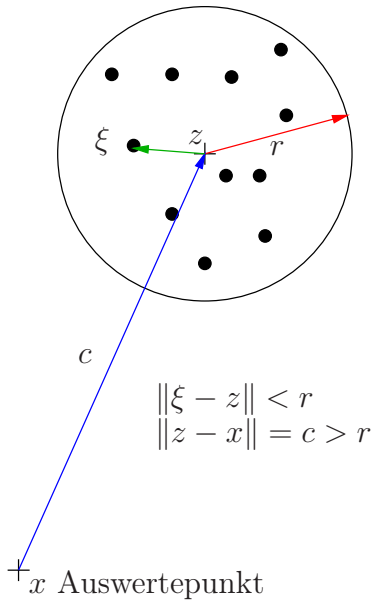


Abbildung 10.2: Ein Cluster

10.3 Schnelle Summationsmethoden

Wir leiten nun eine Methode her, um die Kraftauswertung zu beschleunigen. Dazu betrachten wir die in Abb. 10.2 dargestellte Situation: M Massenpunkte seien in einem Kreis um z mit dem Radius r enthalten. Wir wollen das Potential ϕ aller Massenpunkte im Punkt x mit $\|z - x\| = c > r$ auswerten.

Betrachten wir zunächst einen Massenpunkt an der Position ξ mit $\|\xi - z\| < r$. Das Potential der Masse in ξ sei (wir vergessen den multiplikativen Faktor γm, den kann man später hinzufügen)

$$\phi_\xi(x) = \frac{1}{\|\xi - x\|} = f(\xi - x).$$

Das Potential hängt nur vom Abstand $\xi - x$ ab. Nun fügen wir den Punkt z ein und entwickeln in eine Taylorreihe um $(z - x)$ bis zur Ordnung p (nicht mit Prozessor verwechseln):

$$\begin{aligned} f(\xi - x) &= f((z - x) + (\xi - z)) = \\ &= \sum_{|\alpha| \le p} \frac{D^\alpha f(z - x)}{|\alpha|!} (\xi - z)^{|\alpha|} + \underbrace{\sum_{|\alpha| = p} \frac{D^\alpha f(z - x + \theta(\xi - z))}{|\alpha|!} (\xi - z)^{|\alpha|}}_{\text{Restglied}} \end{aligned}$$

für ein $\theta \in [0, 1]$. Wichtig ist die Separation der Variablen x und ξ. Die Größe des Fehlers (Restglied) hängt von p, r und c ab, wie wir gleich sehen werden.

Zunächst betrachten wir aber, wie die Reihenentwicklung benutzt werden kann, um die Potentialauswertung zu beschleunigen. Dazu nehmen wir an, dass eine Auswertung des Potentials der M Massen an N Punkten zu berechnen ist, was normalerweise $O(MN)$ Operationen erforderlich macht.

Für die Auswertung des Potentials an der Stelle x berechnen wir

$$\begin{aligned}\phi(x) &= \sum_{i=1}^{M} \gamma\, m_i \phi_{\xi_i}(x) = \sum_{i=1}^{M} \gamma\, m_i f((z-x)+(\xi_i - z)) \approx \\ &\underset{\substack{\text{(Taylorreihe}\\ \text{bis}\\ \text{Ordnung}\\ p)}}{\approx} \sum_{i=1}^{M} \gamma\, m_i \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!}(\xi_i - z)^{|\alpha|} = \\ &\underset{\substack{\text{(Summe}\\ \text{vertau-}\\ \text{schen)}}}{=} \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!} \Bigg(\underbrace{\sum_{i=1}^{M} \gamma\, m_i (\xi_i - z)^{|\alpha|}}_{\substack{=: M_\alpha,\\ \text{unabhängig}\\ \text{von } x!}} \Bigg) = \\ &= \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!} M_\alpha\end{aligned}$$

Die Berechnung der Koeffizienten M_α erfordert einmalig $O(Mp^3)$ Operationen. Sind die M_α bekannt, so kostet eine Auswertung von $\phi(x)$ $O(p^5)$ Operationen. Bei Auswertung an N Punkten erhält man also die Gesamtkomplexität $O(Mp^3 + Np^5)$.

Es ist klar, dass das so berechnete Potential nicht exakt ist. Der Algorithmus macht nur Sinn, wenn der Fehler so kontrolliert werden kann, dass er vernachlässigbar ist (z.B. kleiner als der Diskretisierungsfehler). Ein Kriterium zur Fehlerkontrolle gibt die Fehlerabschätzung:

$$\frac{\phi_\xi(x) - \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!}(\xi - z)^{|\alpha|}}{\phi_\xi(x)} \le c(2h)^{p+1},$$

mit $\frac{r}{c} \le h < \frac{1}{4}$. Für den Fall $c > 4r$ reduziert sich der Fehler wie $(1/2)^{p+1}$.

Die Näherung wird also umso genauer,

- je kleiner $\frac{r}{c}$
- je größer der Entwicklungsgrad p.

Gradientenberechnung.

Im N-Körper-Problem will man nicht das Potential, sondern die Kraft, also den Gradient des Potentials, ausrechnen. Dies geht mittels

$$\frac{\partial \phi(x)}{\partial x_{(j)}} \underset{\substack{\text{Reihenentw.}\\ \uparrow\\ \text{Raumdimension}}}{\approx} \frac{\partial}{\partial x_{(j)}} \sum_{|\alpha|\le p} \frac{D^\alpha f(z-x)}{|\alpha|!} M_\alpha = \sum_{|\alpha|\le p} \frac{D^\alpha \partial_{x_{(j)}} f(z-x)}{|\alpha|!} M_\alpha$$

Man muss also nur $D^\alpha \partial_{x_{(j)}} f(z-x)$ statt $D^\alpha f(z-x)$ berechnen.

Andere Reihenentwicklungen.

Oben haben wir eine Taylorreihe verwendet. Dies ist nicht die einzige Möglichkeit einer Reihenentwicklung. In Wahrheit gibt es für die in Betracht kommenden Potentiale $\frac{1}{\log(\|\xi - x\|)}$ (2D)

und $\frac{1}{\|\xi - x\|}$ (3D) andere Entwicklungen, die sogenannten Multipolentwicklungen, die sich besser eignen. Für diese Reihen gelten bessere Fehlerabschätzungen in dem Sinne, dass sie die Form

$$\textit{Fehler} \leq \frac{1}{1 - \frac{r}{c}} \left(\frac{r}{c}\right)^{p+1}$$

haben und somit schon für $c > r$. Ausserdem ist die Komplexität in Bezug auf p besser (p^2 in 2D, p^4 in 3D). Für Details sei auf (Greengard 1987) verwiesen.

Eine von Physikern häufig verwendete Approximation des Gravitationspotentials ist eine Taylorentwicklung von

$$\phi(x) = \sum_{i=1}^{M} \frac{\gamma\, m_i}{\|(s - x) + (\xi_i - s)\|},$$

wobei s der *Massenschwerpunkt* der M Massen ist (und nicht ein fiktiver Kreismittelpunkt). Die sogenannte Monopolentwicklung lautet

$$\phi(x) \approx \frac{\sum_{i=1}^{M} \gamma\, m_i}{\|s - x\|}$$

(d.h. ein Körper der Masse $\sum m_i$ in s). Die Genauigkeit wird dann nur über das Verhältnis r/c gesteuert.

Verschieben einer Entwicklung

In den unten folgenden Algorithmen benötigen wir noch ein Werkzeug, das die Verschiebung von Entwicklungen betrifft. In Abb. 10.3 sehen wir drei Cluster von Körpern in Kreisen um z_1, z_2, z_3 mit jeweils dem Radius r. Die drei Kreise passen wiederum in einen größeren Kreis um z' mit Raidus r'. Wollen wir das Potential aller Massen in x mit $\|x - z'\| > r'$ auswerten, so könnten wir eine Reihenentwicklung um z' verwenden. Falls schon Reihenentwicklungen in den drei kleineren Kreisen berechnet wurden (d.h. die Koeffizienten M_α), so lassen sich die Entwicklungskoeffizienten der neuen Reihe aus denen der alten Reihen mit Aufwand $O(p^\alpha)$ berechnen, d.h. unabhängig von der Zahl der Massen. Dabei entsteht *kein* zusätzlicher Fehler, und es gilt auch die Fehlerabschätzung mit entsprechend größerem r'.

10.4 Gleichmäßige Punkteverteilung

Zunächst entwickeln wir einen Algorithmus, der sich für eine uniforme Verteilung der Punkte eignet. Dies hat den Vorteil einfacher Datenstrukturen und der Möglichkeit einfacher Lastverteilung. Wir stellen die Ideen für den zweidimensionalen Fall vor, da sich dies leichter zeichnen lässt. Alles kann jedoch in drei Raumdimensionen analog durchgeführt werden.

Alle Körper seien in einem Quadrat $\Omega = (0, D_{max})^2$ der Seitenlänge D_{max} enthalten. Wir überziehen Ω mit einer Hierarchie von immer feineren Gittern, die wie in Abb. 10.4 konstruiert werden. Stufe l entsteht aus Stufe $l - 1$ duch Vierteln der Elemente.

Wollen wir s Körper pro Feingitterbox, so gilt $J = \log_4\left(\frac{N}{s}\right)$. Für zwei nicht benachbarte Quadrate erhaten wir folgende Abschätzung für das r/c-Verhältnis (Massen in b, Auswertung in a) (siehe Abb. 10.5)

$$\begin{aligned} r &= \sqrt{2}\frac{k}{2} \\ c &= 2k \\ \Rightarrow \frac{r}{c} &= \frac{\sqrt{2}\,k}{4k} = \frac{\sqrt{2}}{4} \approx 0.35. \end{aligned}$$

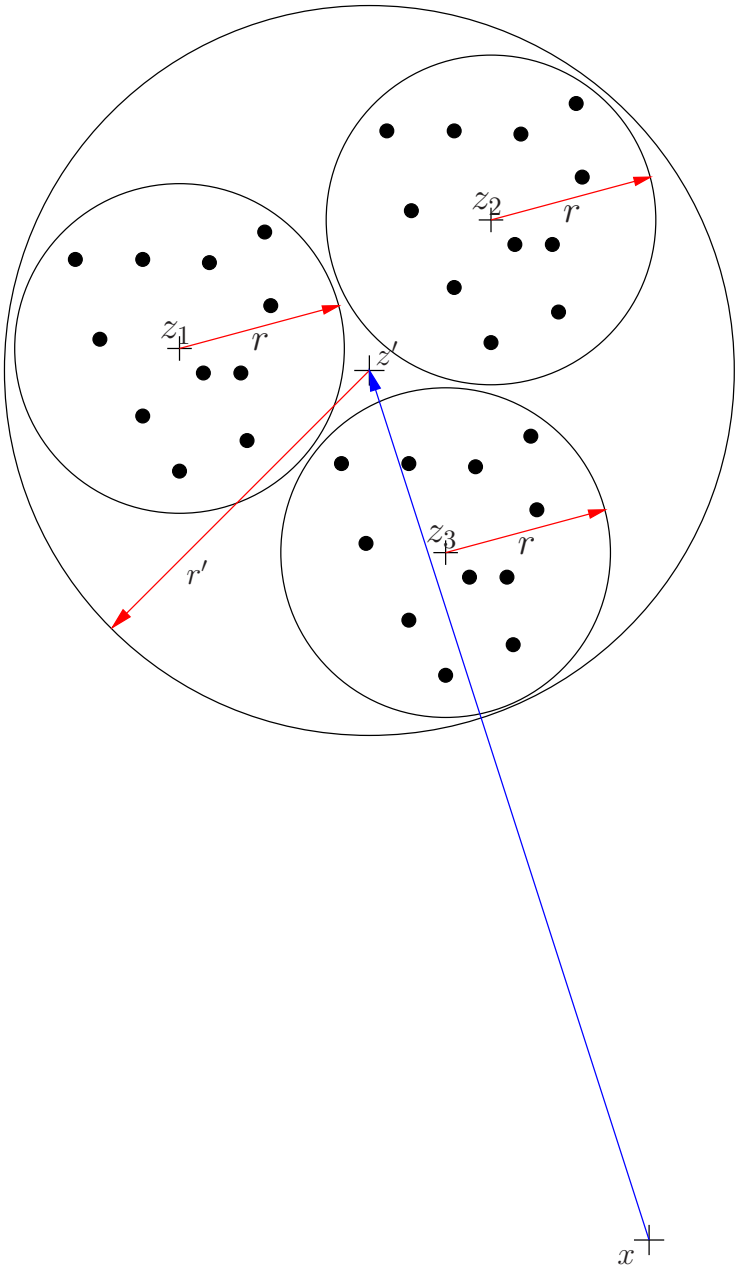


Abbildung 10.3: Verschieben einer Entwicklung

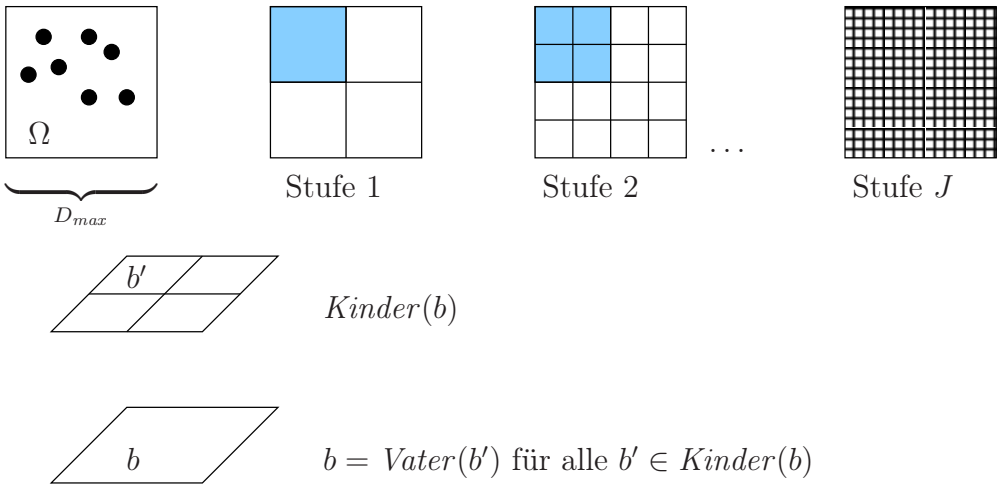


Abbildung 10.4: Konstruktion der Gitter

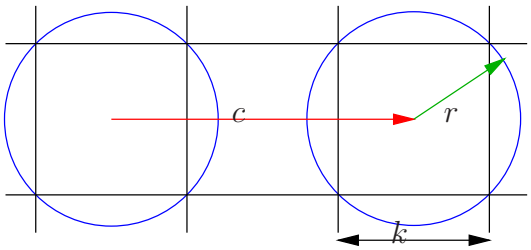


Abbildung 10.5: r/c-Abschätzung für zwei nicht benachbarte Quadrate

Für ein Element b auf Stufe l definiert man folgende Bereiche in der Nachbarschaft von b (siehe Abb. 10.6):

- $Nb(b)$ = alle *Nachbarn* b' von b auf Stufe l ($\partial b \cap \partial b' \neq \emptyset$).
- $IL(b)$ = *Interaktionsliste* von b: Kinder von Nachbarn von $Vater(b)$, die nicht Nachbar von b sind.

Folgender Algorithmus berechnet das Potential an allen Positionen $x_1, \ldots, x_N$:

	Aufwand
① *Vorbereitungsphase:*	
Für jede Feingitterbox berechne eine Fernfeldentwicklung;	$O(\frac{N}{s} sp^\alpha)$
Für alle Stufen $l = J-1, \ldots, 0$	
Für jede Box b auf Stufe l	
berechne Entwicklung in b aus Entwicklung in $Kinder(b)$;	$O(\frac{N}{s} sp^\gamma)$
② *Auswertephase:*	
Für jede Feingitterbox b	

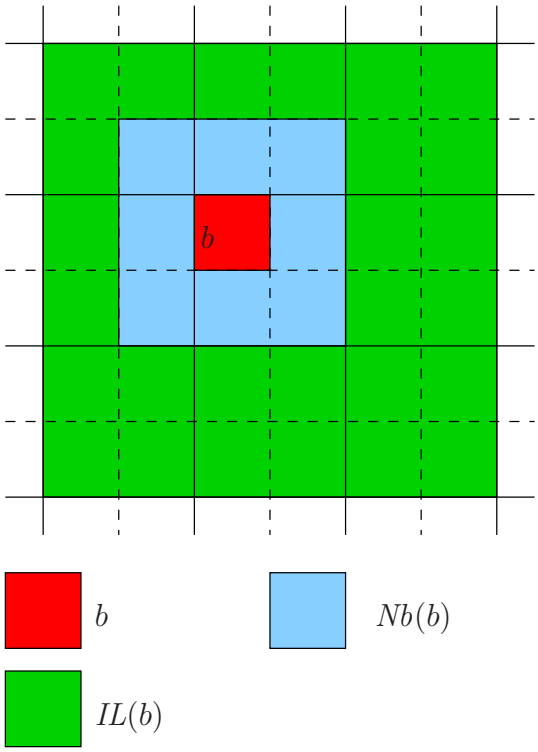


Abbildung 10.6: Nachbarn und Interaktionsliste

Für jeden Körper q in b
{
  berechne exaktes Potential aller $q' \in B$, $q' \neq q$; $O(Ns)$
  Für alle $b' \in Nb(b)$
    Für alle $q' \in b'$
      berechne Potential von q' in q; $O(N8s)$
  $\bar{b} = b$;
  Für alle Stufen $l = J, \ldots, 0$
  {
    Für alle $b' \in IL(\bar{b})$
      Werte Fernfeldentwicklung von b' in q aus; $O(\frac{N}{s} s p^\beta)$
  }

Gesamtaufwand: $O(N \log N p^\gamma + Ns + Np^\alpha + \frac{N}{s} p^\beta)$, also asymptotisch $O(N \log N)$. Dabei bezeichnet α den Exponenten für das Aufstellen der Fernfeldentwicklung und β den Exponenten für das Verschieben. Man beachte, dass man wegen der uniformen Verteilung N/s Körper pro Box auf Stufe J hat.

Bemerkung 10.1 Die Genauigkeit wird hier über den Entwicklungsgrad p gesteuert, das Verhältnis r/c ist fest.

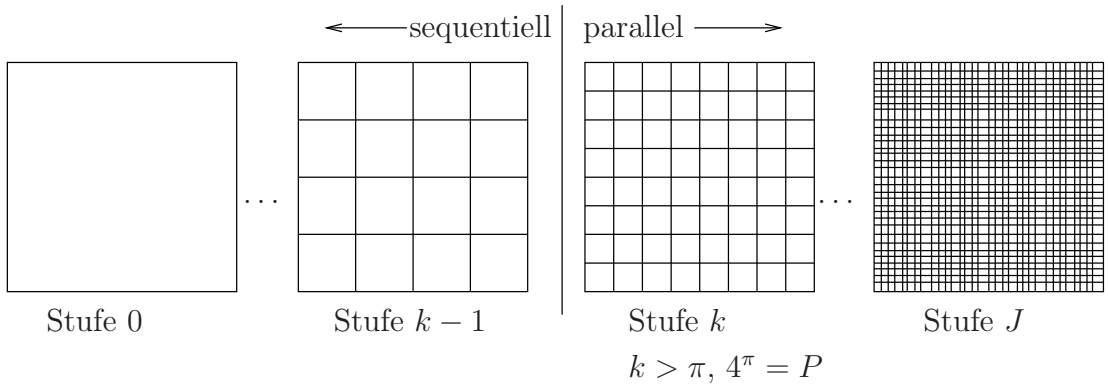


Abbildung 10.7: Aufteilung der Boxen bei der Parallelisierung

Parallelisierung

Wie in Abschnitt 9.5 (Iterationsverfahren) wird das Gitter mit den zugehörigen Körpern auf die Prozessoren aufgeteilt. Da wir nun eine Hierarchie von Gittern haben, verfahren wir wie folgt: Jeder Prozessor soll mindestens 2×2 Gitterzellen bekommen. Es sei $P = 4^\pi$, so wähle $k = \pi + 1$ und verteile das Gitter der Stufe k auf alle Prozessoren (jeder hat 2×2 Elemente). Alle Stufen $l < k$ werden auf allen Prozessoren gespeichert. Alle Stufen $l > k$ werden so aufgeteilt, dass $Kinder(b)$ im selben Prozessor wie b bearbeitet werden. Dies zeigt die Abb. 10.7 für $P = 16 = 4^2$.

Zusätzlich zu den ihm zugewiesenen Elementen speichert jeder Prozessor noch einen Überlappungsbereich von zwei Elementreihen (zu sehen in Abb. 10.8).

Da jeder mindestens 2×2 Elemente hat, liegt der Überlappungsbereich immer in direkten Nachbarprozessoren.

Die Auswertung der *IL* erfordert höchstens Kommunikation mit nächsten Nachbarn.

Zum Aufbau der Fernfeldentwicklung für die Stufen $l < k$ ist eine alle-an-alle Kommunikation erforderlich.

Skalierbarkeitsabschätzung.

Es sei $\frac{N}{sP} \gg 1$. Wegen

$$J = \log_4\left(\frac{N}{s}\right) = \log_4\left(\frac{N}{sP}P\right) = \underbrace{\log_4\left(\frac{N}{sP}\right)}_{J_p} + \underbrace{\log_4 P}_{J_s}$$

werden J_s Stufen sequentiell und $J_p = O(1)$ Stufen parallel gerechnet. Somit erhalten wir für *festen Entwicklungsgrad:*

$$T_P(N,P) \underset{(\frac{N}{P} = const.)}{=} \underbrace{c_1\frac{N}{P}}_{\substack{\text{FFE Stufe}\\ J\ldots K\\ \text{Nahfeld}\\ \text{auswerten}}} + \underbrace{c_2 \operatorname{ld} P + c_3 P}_{\substack{\text{alle-an-alle. Es}\\ \text{sind immer}\\ \text{Daten für 4}\\ \text{Zellen}}} + \underbrace{c_4 p}_{\substack{\text{berechne FFE}\\ \text{in ganz } \Omega \text{ für}\\ l = k-1\ldots 0\\ \text{in } \textit{allen}\\ \textit{Prozessoren}}} + \underbrace{c_5 J_p \frac{N}{P}}_{\substack{\text{FFE Stufen}\\ l \geq k}} + \underbrace{c_5 \frac{N}{P} J_s}_{\substack{\text{FFE Stufen}\\ l < k}}$$

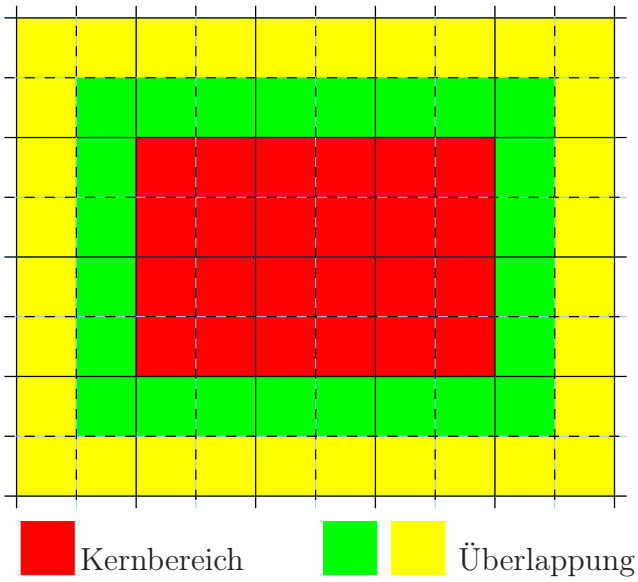


Abbildung 10.8: Überlappungsbereich eines Prozessors

Also:

$$E(N,P) = \frac{c_5 N \log N}{\big(c_4 \frac{N}{P}(\underbrace{J_s + J_p}_{\log N}) + \underbrace{(c_3 + c_4)P}_{\substack{\text{alle an alle}\\ \text{Grobgitter}\\ \text{FFE}}} + \underbrace{c_2 \operatorname{ld} P}_{\substack{\text{alle-an-}\\ \text{alle}}} + \underbrace{c_1 \frac{N}{P}}_{\substack{\text{Nahfeld lokale}\\ \text{FFE}}}\big)P} =$$

$$= \frac{1}{1 + \frac{c_4+c_4}{c_5} \cdot \frac{P^2}{N \log N} + \frac{c_2}{c_5} \cdot \frac{P \operatorname{ld} P}{N \log N} + \frac{c_1}{c_5} \cdot \frac{1}{\log N}}$$

Für eine isoeffiziente Skalierung muss N also beinahe wie P^2 wachsen!

10.5 Ungleichmäßige Verteilung

Die Annahme einer uniformen Verteilung der Körper ist in einigen Anwendungen (z.B. Astrophysik) nicht realistisch. Will man in jeder Feingitterbox genau einen Körper haben und ist D_{min} der minimale Abstand zweier Körper, so braucht man ein Gitter mit

$$\log L = \log \frac{D_{max}}{D_{min}}$$

Gitterstufen. L heisst *„separation ratio“*. Von diesen sind aber die meisten leer. Wie bei dünnbesetzten Matrizen vermeidet man nun die leeren Zellen zu speichern. In zwei Raumdimensionen heisst diese Konstruktion „adaptiver Quadtree“.

Der adaptive Quadtree wird folgendermaßen aufgebaut:

- Initialisierung: Wurzel enthält alle Körper im Quadrat $(0, D_{max})$.
- Solange es ein Blatt b mit mehr als zwei Körpern gibt:
  - Unterteile b in vier Teile

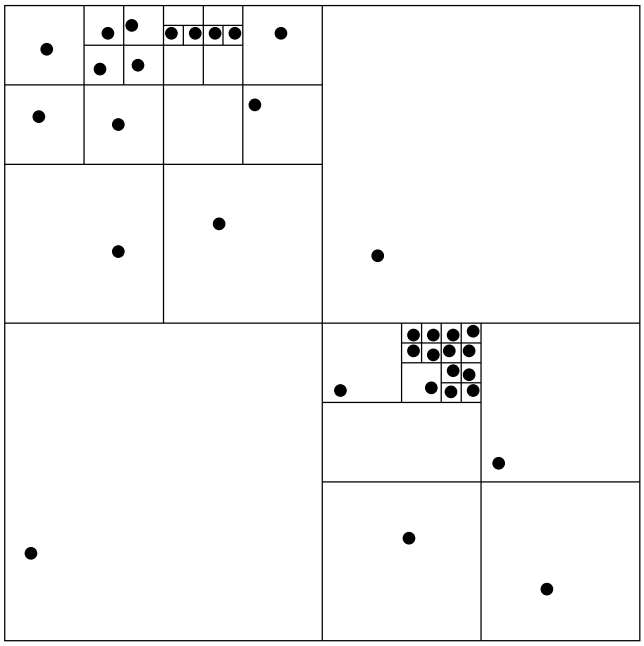


Abbildung 10.9: Ein adaptiver Quadtree

- Packe jeden Körper von b in das entsprechende Kind
- leere Kinder wieder wegwerfen.

Abb. 10.9 zeigt ein Beispiel. Der Aufwand beträgt (sequentiell) $O(N \log L)$.

Der erste (erfolgreiche) schnelle Auswertealgorithmus für ungleichmäßig verteilte Körper wurde von Barnes und Hut (1986) vorgeschlagen.

Wie im gleichmäßigen Fall wird eine Fernfeldentwicklung von den Blättern bis zur Wurzel aufgebaut (bei Barnes & Hut: Monopolentwicklung).

Für einen Körper q berechnet dann folgender rekursive Algorithmus das Potential in q:

```
Pot(Körper q, Box b)
{
    double pot = 0;
    if (b ist Blatt ∧ q = b.q) return 0;  // Ende
    if (Kinder(b) == ∅)
        return φ(b.q, q);                 // direkte Auswertung
    if (r(b)/dist(q,b) ≤ h)
        return φ_b(q);                    // FFE Auswerten
    for (b' ∈ Kinder(b))
        pot = pot + Pot(q, b');           // rekursiver Abstieg
    return pot;
}
```

Zur Berechnung wird *Pot* mit q und der Wurzel des Quadtree aufgerufen. Im Algorithmus von Barnes & Hut wird die Genauigkeit der Auswertung mittels des Parameters h gesteuert. Abb. 10.10 zeigt ein Beispiel, welche Zellen des Quadtree in Barnes & Hut Algorithmus besucht werden.

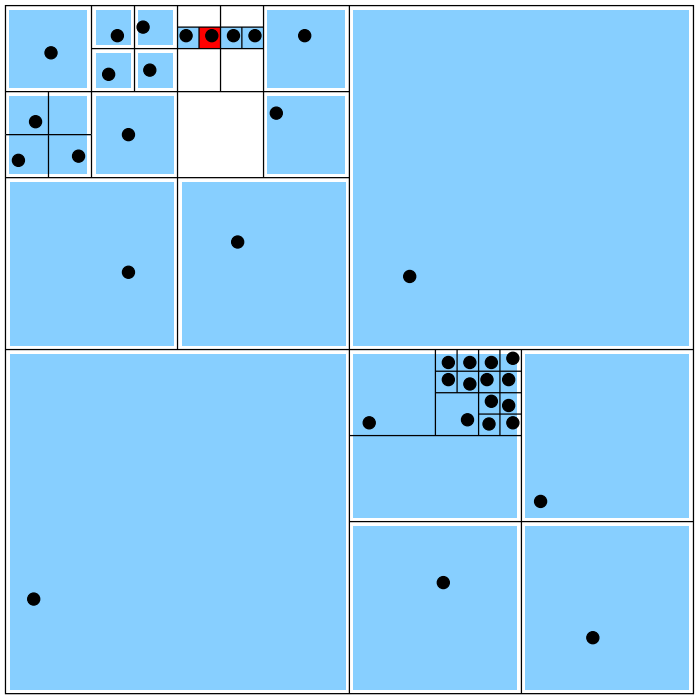


Abbildung 10.10: Beispiel zur Auswertung im Barnes & Hut Algorithmus

Parallelisierung

Die Parallelisierung dieses Algorithmus ist relativ komplex, so dass wir nur einige Hinweise geben können. Für Details sei auf (J.K. Salmon 1994) verwiesen.

Da sich die Positionen der Teilchen mit der Zeit verändern, muss der adaptive Quadtree in jedem Zeitschritt neu aufgebaut werden. Zudem muss man die Aufteilung der Körper auf die Prozessoren so vornehmen, dass nahe benachbarte Körper auch auf möglichst nahe benachbarten Prozessoren gespeichert sind. Ein sehr geschicktes Lastverteilungsverfahren arbeitet mit „raumfüllenden Kurven“. Abb. 10.11 zeigt die sogenannte Peano-Hilbert-Kurve .

Eine Hilbertkurve entsprechender Tiefe kann benutzt werden, um eine lineare Ordnung der Körper (bzw. der Blätter im Quadtree) herzustellen. Diese kann dann leicht in P Abschnitte der Länge N/P zerlegt werden. Für unser Beispiel zeigt dies Abb. 10.12. Schön ist die Datenlokalität zu erkennen.

Salmon & Warren zeigen, dass mit dieser Datenverteilung der adaptive Quadtree parallel mit sehr wenig Kommunikation aufgebaut werden kann. Ähnlich wie im uniformen Algorithmus wird dann durch eine alle-an-alle Kommunikation die Grobgitterinformation aufgebaut, die alle Prozessoren speichern. Dieses Grobgitter enthält auch Informationen darüber, welcher Prozessor welche Information besitzt.

Paralleler Aufbau des adaptiven Quadtree

- *Ausgangspunkt:* Jeder Prozessor hat eine Menge von Körpern, die *einem* zusammenhängenden Abschnitt auf der Hilbertkurve entspricht.
- *Schritt 1:* Jeder Prozessor baut lokal für *seine* Körper den Quadtree auf. Die „Nummern“ der Blätter sind dabei aufsteigend (Abb. 10.13).

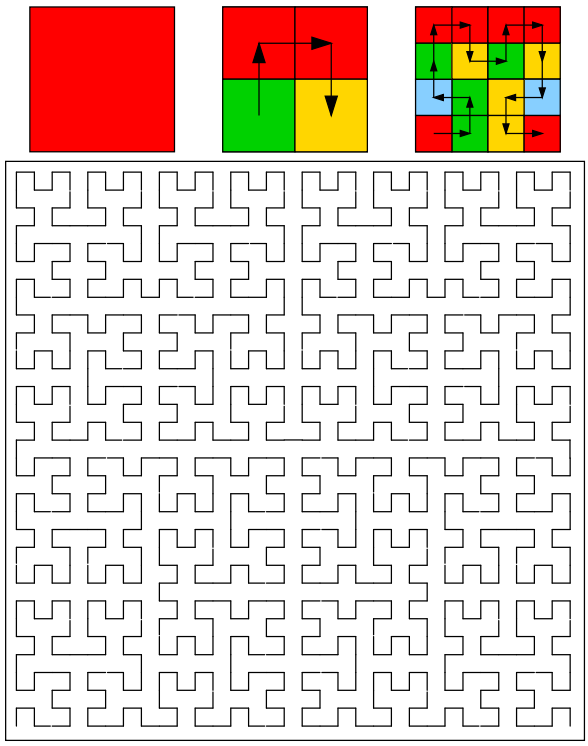


Abbildung 10.11: Peano-Hilbert-Kurve

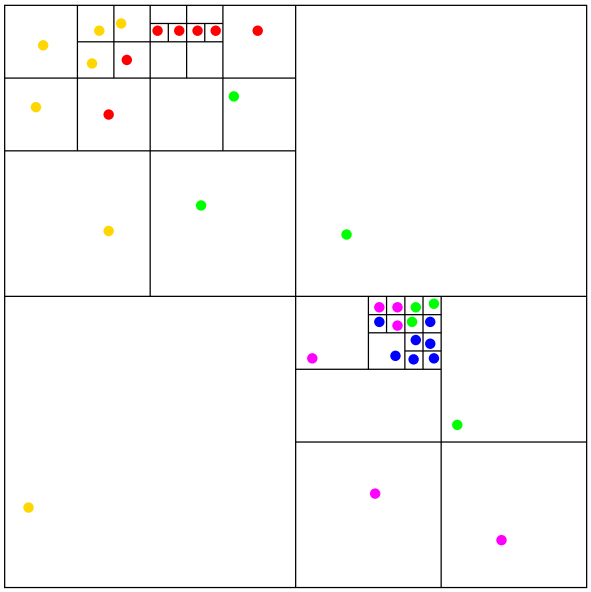


Abbildung 10.12: Verteilung der Körper auf die Prozessoren.

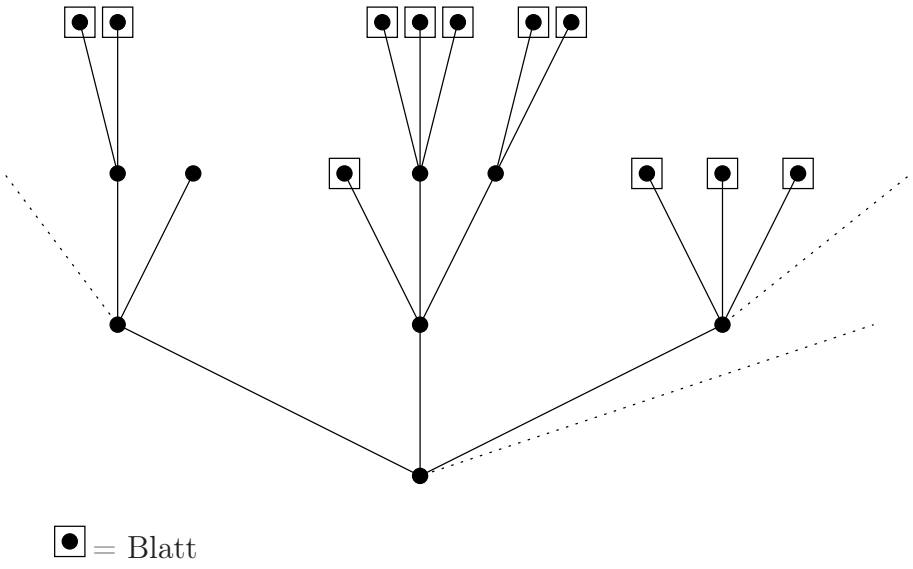


Abbildung 10.13: Lokaler Quadtree

- *Schritt 2:* Abgleich mit Nachbarprozessoren. Frage: Hätte ein sequentielles Programm für die Körper von Prozessor p dieselben Blätter erzeugt? Im allgemeinen nein, denn ein Körper von p und einer von $q \neq p$ könnten sich ja eine Box teilen.

  Was kann passieren?

  Sei b der *erste Körper* in Prozessor p und b' der *letzte* in Prozessor $p-1$. Nun gibt es zwei Möglichkeiten in Prozessor p:

  1. Körper b' liegt in der selben Box wie Körper b. $\Longrightarrow$ Zerteile Box so lange, bis beide Körper getrennt sind (siehe Abb. 10.14). Das neue Blatt ist das selbe, das auch ein sequentielles Programm berechnet hätte! Falls dem *nicht* so wäre, so müsste es einen Körper b'' in Prozessor $q \neq p$ geben, so dass $b'' \in$ neues Blatt von b. Dies ist aber unmöglich, da alle b'' vor b' oder nach dem letzten Knoten von Prozessor p kommen!
  2. Körper b' liegt nicht in derselben Box wie Körper b. $\Longrightarrow$ es ist nichts zu tun.

  Für den letzten Körper in p und den ersten in $p+1$ verfährt man ebenso!

Das Grobgitterproblem

Wie im uniformen Fall wird der Quadtree von der Wurzel bis zu einer bestimmten Tiefe in jedem Prozessor gespeichert, so dass für diese Fernfeldauswertungen keine Kommunikation notwendig ist oder, falls nötig, der Prozessor bekannt ist, der über die entsprechende Information verfügt.

Definition 10.2 Eine Box b im Quadtree heisst *Zweigknoten*, falls b nur Körper eines Prozessors p, der Vater von b jedoch Körper verschiedenener Prozessoren enthält. Diesem Prozessor p „gehört“ der Zweigknoten.

Alle Boxen des Quadtree von der Wurzel bis einschließlich der Zweigknoten werden auf allen Prozessoren gespeichert.

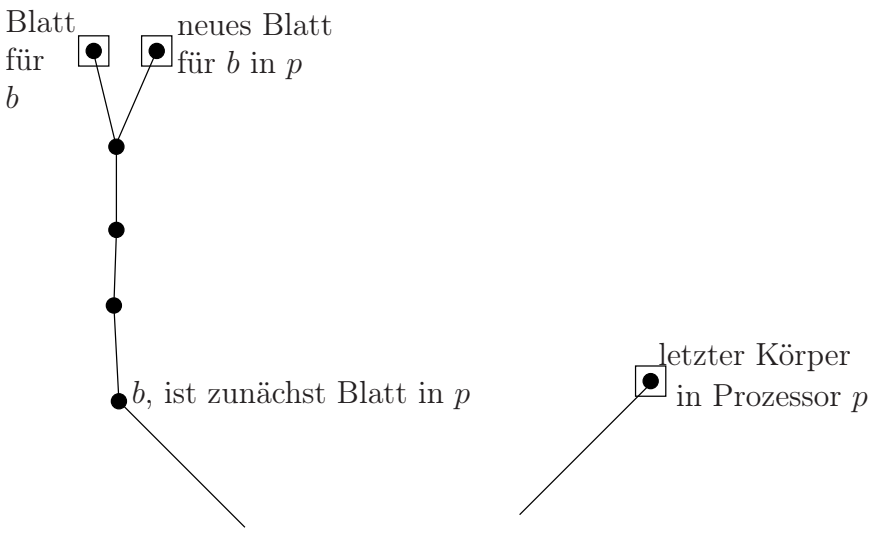


Abbildung 10.14: Austausch der Randblätter

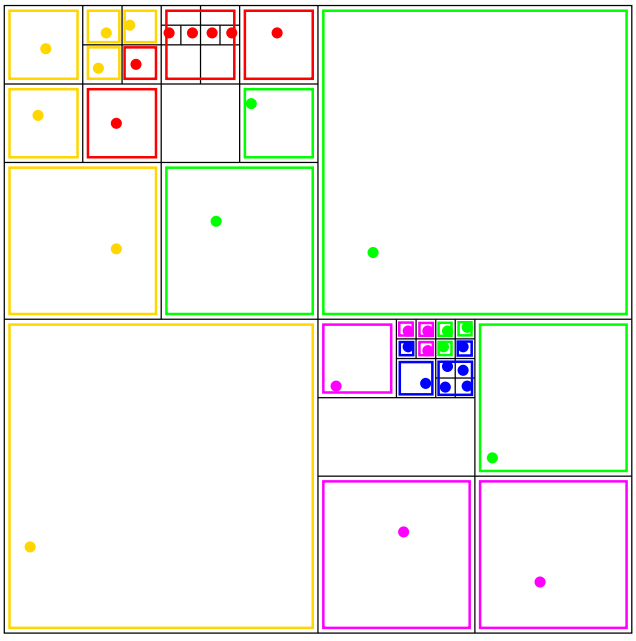


Abbildung 10.15: Die Zweigknoten in unserem Beispiel

Kraftauswertung & Kommunikation

Ist beim Auswerten ein rekursiver Aufruf in einem Zweigknoten nötig, so wird eine Nachricht an den entsprechenden Prozessor geschickt, dem der Zweigknoten gehört. Dieser bearbeitet die Anfrage asynchron und schickt das Ergebnis zurück.

Nach dem Update der Positionen berechnet man eine neue Hilbertnummerierung (geht in $\frac{N}{P} \log L$ ohne Aufbau eines Quadtree) für die lokalen Körper. Dann bekommt jeder wieder einen Abschnitt der Länge $\frac{N}{P}$. Dies geht z.B. mit einem parallelen Sortieralgorithmus!

Salmon & Warren können mit ihrer Implementierung 322 Millionen Körper (!) auf 6800 Prozessoren (!, Intel ASCI-Red) simulieren.

11 Paralleles Sortieren

11.1 Einführung

Es gibt eine ganze Reihe verschiedener Sortieralgorithmen. Wir beschränken uns hier auf

- interne Sortieralgorithmen, d.h. solche, die ein Feld von (Ganz-) Zahlen im Speicher (wahlfreier Zugriff möglich!) sortieren, und
- vergleichsbasierte Sortieralgorithmen, d.h. solche, bei denen die Grundoperationen aus Vergleich zweier Elemente und eventuellem Vertauschen besteht.

Für eine allgemeine Einführung in Sortieralgorithmen sei auf (Sedgewick 1992) verwiesen.

Die Eingabe des parallelen Sortieralgorithmus besteht aus N Zahlen. Diese sind auf P Prozessoren verteilt, d.h. jeder Prozessor besitzt ein Feld der Länge N/P. Ergebnis eines parallelen Sortieralgorithmus ist eine sortierte Folge der Eingabezahlen, die wiederum auf die Prozessoren verteilt ist, d.h. Prozessor 0 enthält den ersten Block von N/P Zahlen, Prozessor 1 den zweiten usw.

Nun besprechen wir zwei der wichtigsten sequentiellen Sortieralgorithmen, Mergesort und Quicksort.

11.1.1 Mergesort

Mergesort basiert auf folgender Idee:

> Es sei eine Folge von N Zahlen zu sortieren. Angenommen wir teilen die Folge in zwei der Länge $\frac{N}{2}$ und sortieren diese jeweils getrennt, so können wir aus den beiden Hälften leicht eine sortierte Folge von N Zahlen erstellen, indem wir jeweils das nächste kleinste Element der Teilfolgen wegnehmen.

Input: $a = \langle a_0, a_1, \ldots, a_{N-1} \rangle$;
$l = \left\langle a_0, \ldots, a_{\frac{N}{2}-1} \right\rangle$; $r = \left\langle a_{\frac{N}{2}}, \ldots, a_{N-1} \right\rangle$;
sortiere l; sortiere r;
$i = j = k = 0$;
while $(i < N/2 \ \wedge \ j < N/2)$
{
  if $(l_i \leq r_j)$ { $s_k = l_i$; $i++$; }
  else { $s_k = r_j$; $j++$; }
  $k++$;
}
while $(i < N/2)$
{
  $s_k = l_i$; $i++$; $k++$;
}
while $(j < N/2)$

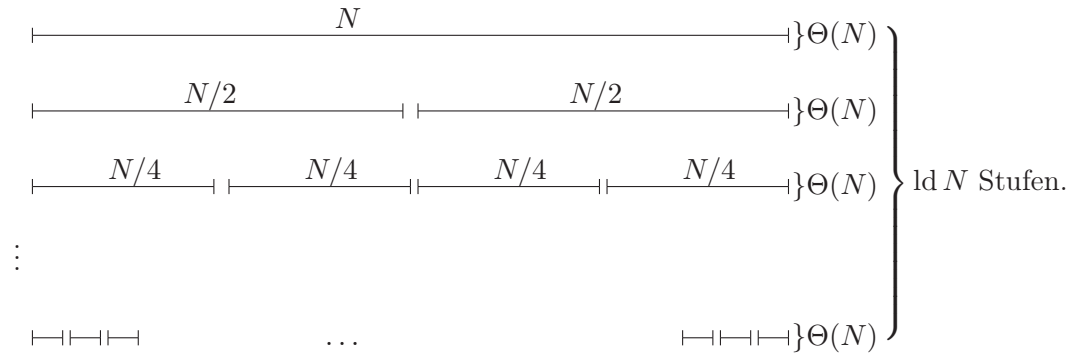


Abbildung 11.1: Mergesort

```
{
    s_k = r_j; j++; k++;
}
```

Bemerkung 11.1 Ein Problem von Mergesort ist, dass zusätzlicher Speicherplatz (zusätzlich zur Eingabe) erforderlich ist. Obige Variante kann sicher stark verbessert werden, „sortieren am Ort“, d.h. in a selbst, ist jedoch nicht möglich.

Laufzeit: Das Mischen der zwei sortierten Folgen (die drei **while**-Schleifen) braucht $\Theta(N)$ Operationen. Die Rekursion endet in Tiefe $d = \operatorname{ld} N$ (siehe Abb. 11.1), die Komplexität ist also $\Theta(N \operatorname{ld} N)$.

Es zeigt sich, dass dies die optimale asymptotische Komplexität für vergleichsbasierte Sortieralgorithmen ist.

11.1.2 Quicksort

Quicksort und Mergesort sind in gewissem Sinn komplementär zueinander: Mergesort sortiert rekursiv zwei (gleichgroße) Teilfolgen und mischt diese zusammen. Bei Quicksort zerteilt man die Eingabefolge in zwei Teilfolgen, so dass *alle* Elemente der ersten Folge kleiner sind als *alle* Elemente der zweiten Folge. Diese beiden Teilfolgen können dann getrennt (rekursiv) sortiert werden. Das Prolem dabei ist, wie man dafür sorgt, dass die beiden Teilfolgen (möglichst) gleich groß sind, d.h. je etwa $N/2$ Elemente enthalten.

Üblicherweise geht man so vor: Man wählt ein Element der Folge aus, z.B. das erste oder ein zufälliges und nennt es „Pivotelement“ . Alle Zahlen kleiner gleich dem Pivotelement kommen in die erste Folge, alle anderen in die zweite Folge. Die Komplexität hängt nun von der Wahl des Pivotelementes ab:

$\Theta(N \log N)$	falls immer in zwei gleichgroße Mengen zerlegt wird,
$\Theta(N^2)$	falls immer in eine einelementige Menge und den Rest zerlegt wird.

Bei einer zufälligen Auswahl des Pivots ist der zweite Fall extrem unwahrscheinlich. In der Regel ist Quicksort sogar schneller als Mergesort.

```
void Quicksort(int a[]; int first; int last)
{
```

*	*	*	*	*						
s				s						

Tabelle 11.1: Quicksort

```
    if (first ≥ last) return;                  // Ende der Rekursion
    // partitioniere
    wähle ein q ∈ [first, last];               // Pivotwahl
    x = a[q];
    swap(a, q, first);                         // bringe x an den Anfang
    s = first;                                 // Marke Folge 1: [first ... s]
    for (i = first + 1; i ≤ last; i++)
        if (a[i] ≤ x)
        {
            s++;
            swap(a, s, i);                     // a[i] in erste Folge
        }
    swap(a, first, s);

    //  { Nun gilt
          1. alle a[i] mit first ≤ i ≤ s sind ≤ x
          2. a[s] = x ist bereits in der richtigen Position!

    Quicksort(a, first, s − 1);                // a[s] wird nicht mitsortiert
    Quicksort(a, s + 1, last);                 // beide Folgen zusammen eins weniger
}
```

Bemerkung 11.2 Stack könnte zum Problem werden, falls N groß und worst case erreicht wird (N rekursive Aufrufe).

11.1.3 Sortiernetzwerke

Ein Sortiernetzwerk übersetzt eine Folge von N unsortierten Zahlen in eine Folge von N aufsteigend oder absteigend sortierte Zahlen, siehe Abb. 11.2. An den N Eingängen werden die unsortierten Zahlen angelegt, am Ausgang erscheint die sortierte Folge.

Intern ist das Sortiernetzwerk aus elementaren Schaltzellen, sogenannten Komparatoren, aufgebaut, die genau zwei Zahlen aufsteigend oder absteigend sortieren, siehe Abb. 11.3.

Eine Anzahl von Komparatoren wird zu einer sogenannten „Stufe“ zusammengefasst. Das ganze Netzwerk besteht aus mehreren Stufen, die ihrerseits durch ein Verbindungsnetzwerk verbunden sind, wie Abb. 11.4 zeigt. Die Anzahl der Stufen wird auch als Tiefe des Sortiernetzwerkes bezeichnet.

Alle Komparatoren einer Stufe arbeiten parallel. Sortiernetzwerke können gut in Hardware realisiert werden oder lassen sich in entsprechende Algorithmen für Parallelrechner übertragen (weshalb wir sie hier studieren wollen).

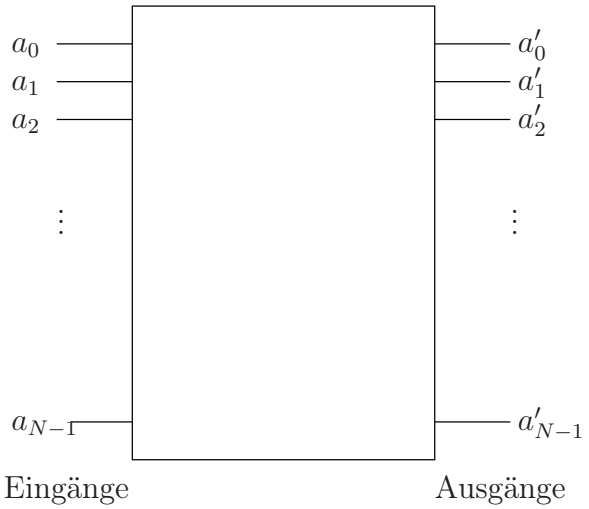


Abbildung 11.2: Ein Sortiernetzwerk

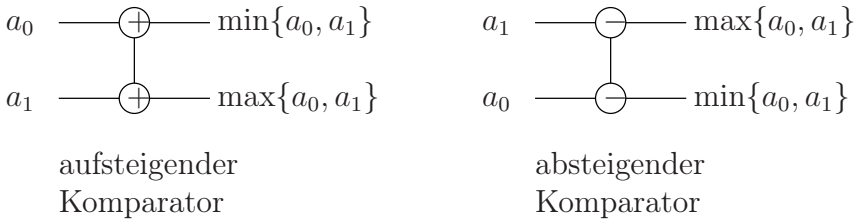


Abbildung 11.3: Komparatoren

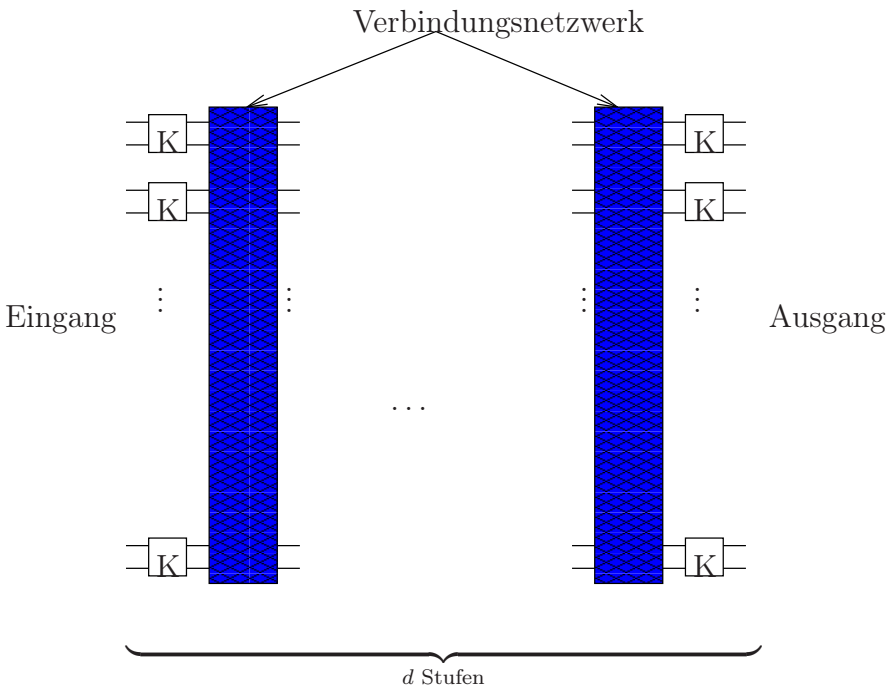


Abbildung 11.4: Ein Sortiernetzwerk der Tiefe d.

11.2 Bitonisches Sortieren

11.2.1 Vorbereitungen

Der nun folgende Algorithmus hat sowohl Eigenschaften von Quicksort als auch von Mergesort:

- Wie bei Mergesort wird die Eingabefolge der Länge N in zwei Folgen der Länge $N/2$ zerlegt. Dadurch ist die Rekursionstiefe immer $\operatorname{ld} N$ (allerdings wird die Gesamtkomplexität schlechter sein!)
- Wie bei Quicksort sind alle Elemente der einen Folge kleiner als alle Elemente der anderen Folge und können somit unabhängig voneinander sortiert werden.
- Das Zerlegen kann voll parallel mit $N/2$ Komparatoren geschehen, d.h. der Algorithmus kann mit einem Sortiernetzwerk realisiert werden.

Wir beginnen mit

DEFINITION 11.3 (BITONISCHE FOLGE) Eine Folge von N Zahlen heißt *bitonisch*, genau dann, wenn eine der beiden folgenden Bedingungen gilt

1. Es gibt einen Index $0 \leq i < N$, so dass

$$\underbrace{a_0 \leq a_1 \leq \cdots \leq a_i}_{\text{aufsteigend}} \text{ und } \underbrace{a_{i+1} \geq a_{i+2} \geq \cdots \geq a_{N-1}}_{\text{absteigend}}$$

2. Man kann die Indizes zyklisch schieben, d.h. $a'_i = a_{(i+m)\,\%\,N}$, so dass die Folge a' die Bedingung 1 erfüllt.

Normalform bitonischer Folgen

Jede bitonische Folge lässt sich auf folgende Form bringen:

$$a'_0 \leq a'_1 \leq \cdots \leq \mathbf{a'_k} > \mathbf{a'_{k+1}} \geq a'_{k+2} \geq \cdots \geq \mathbf{a'_{N-1}} < \mathbf{a'_0}$$

Wesentlich ist, dass das letzte Element des aufsteigenden Teils größer ist als der Anfang des absteigenden Teils. Ebenso ist das Ende des absteigenden Teils kleiner als der Anfang des aufsteigenden Teils.

BEWEIS: Für die Eingabefolge a gelte 1 aus der Definition der bitonischen Folgen. Es sei $a_0 \leq a_1 \leq \cdots \leq a_i$ der aufsteigende Teil. Entweder es gilt nun $a_i > a_{i+1}$ oder es gibt ein $j \geq 0$ mit

$$a_i \leq a_{i+1} = a_{i+2} = \cdots = a_{i+j} > a_{i+j+1}$$

(oder die Folge ist trivial und besteht aus lauter gleichen Elementen). Somit kann man die $a_{i+1}, \ldots, a_{i+j}$ zum aufsteigenden Teil hinzunehmen und es gilt $k = i + j$.

Ebenso verfährt man mit dem absteigenden Teil. Entweder ist schon $a_{N-1} < a_0$ oder es gibt ein l, so dass

$$a_{N-1} \geq a_0 = a_1 = \cdots = a_l < a_{l+1}$$

(oder die Folge ist trivial). In diesem Fall nimmt man $a_0, \ldots, a_l$ zum absteigenden Teil hinzu. □

Man hat also die Situation von Abb. 11.5

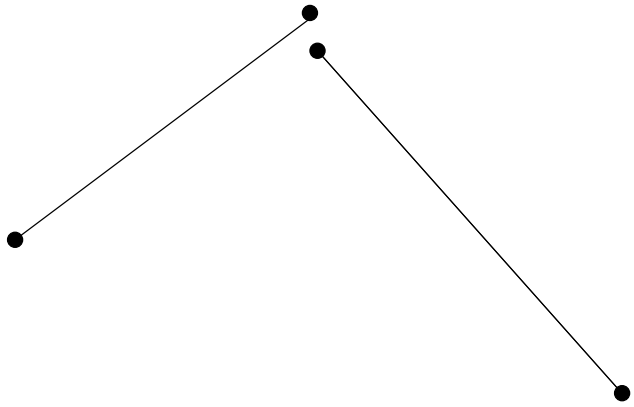


Abbildung 11.5: Normalform einer bitonischen Folge

In der folgenden Definition brauchen wir den Begriff „zyklischer Nachbar“, d.h. wir setzen

$$\begin{aligned} a_{i\oplus l} &:= a_{(i+l)\,\%\,N} \\ a_{i\ominus l} &:= a_{(i+N-l)\,\%\,N} \\ \text{für } l &\leq N \end{aligned}$$

Damit kommen wir zur

Min-Max Charakterisierung bitonischer Folgen

Definition 11.4 Ein a_k heisst (lokales) Maximum, falls es ein $l \geq 0$ gibt mit

$$a_{k\ominus(l+1)} < a_{k\ominus l} = \cdots = a_{k\ominus 1} = a_k > a_{k\oplus 1}$$

Entsprechend heisst a_k (lokales) Minimum, falls es ein $l \geq 0$ gibt mit

$$a_{k\ominus(l+1)} > a_{k\ominus l} = \cdots = a_{k\ominus 1} = a_k < a_{k\oplus 1}$$

Für nicht triviale Folgen (d.h. es sind nicht alle Elemente identisch) gilt damit:
Eine Folge $a = \langle a_0, \ldots, a_{N-1}\rangle$ ist bitonisch genau dann, wenn sie genau ein Minimum und genau ein Maximum hat (oder trivial ist).
Beweis:

- $\Rightarrow$ gilt wegen der Normalform. a'_k ist das Maximum, a'_{N-1} das Minimum.
- $\Leftarrow$ Wenn a genau ein Minimum bzw. Maximum hat, zerfällt sie in einen aufsteigenden und einen absteigenden Teil, ist also bitonisch.

□

Nach all diesen Vorbereitungen kommen wir nun zur Hauptsache, der

11.2.2 Bitonische Zerlegung

Es sei $s = \langle a_0, \ldots, a_{N-1}\rangle$ eine gegebene bitonische Folge der Länge N. Wir konstruieren zwei neue Folgen der Länge $N/2$ (N gerade) auf folgende Weise:

$$\begin{aligned} s_1 &= \left\langle \min\{a_0, a_{\frac{N}{2}}\}, \min\{a_1, a_{\frac{N}{2}+1}\}, \ldots, \min\{a_{\frac{N}{2}-1}, a_{N-1}\}\right\rangle \\ s_2 &= \left\langle \max\{a_0, a_{\frac{N}{2}}\}, \max\{a_1, a_{\frac{N}{2}+1}\}, \ldots, \max\{a_{\frac{N}{2}-1}, a_{N-1}\}\right\rangle \end{aligned} \tag{11.1}$$

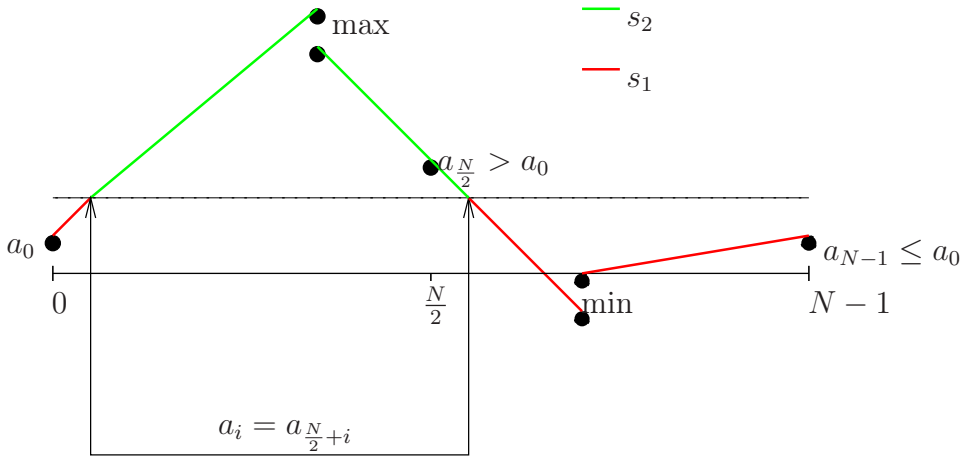


Abbildung 11.6: Maximum zwischen a_0 und $a_{\frac{N}{2}}$.

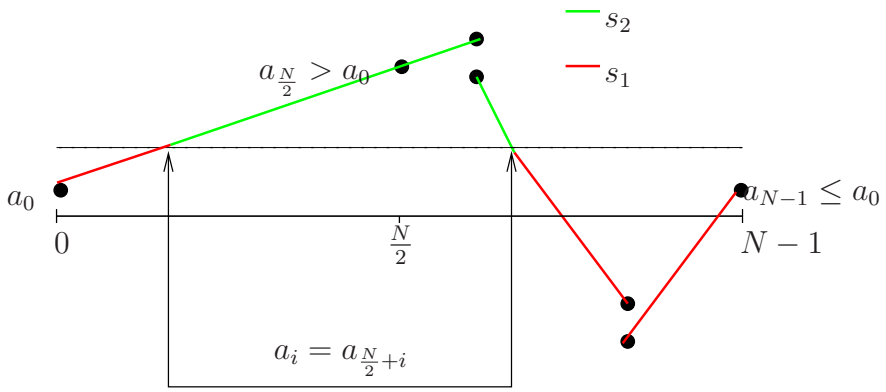


Abbildung 11.7: Maximum zwischen $a_{\frac{N}{2}}$ und a_{N-1}.

Für s_1, s_2 gilt:

1. Alle Elemente von s_1 sind kleiner oder gleich allen Elementen in s_2.
2. s_1 und s_2 sind bitonische Folgen.

Offensichtlich können s_1 und s_2 aus s mit Hilfe von $N/2$ Komparatoren konstruiert werden.
BEWEIS: Wir überlegen uns das graphisch anhand von verschiedenen Fällen:

1. Es sei $a_0 < a_{\frac{N}{2}}$: Wir wissen, dass jede bitonische Folge genau ein Maximum und Minimum hat. Da $a_0 < a_{\frac{N}{2}}$ kann das Maximum nur zwischen a_0 und $a_{\frac{N}{2}}$ oder zwischen $a_{\frac{N}{2}}$ und a_{N-1} liegen (Abb. 11.6). In diesem Fall gilt also $\min\{a_0, a_{\frac{N}{2}}\} = a_0$ und solange $a_i \leq a_{\frac{N}{2}+i}$ enthält s_1 die Elemente aus dem aufsteigenden Teil der Folge. Irgendwann gilt $a_i > a_{\frac{N}{2}+i}$, und dann enthält s_1 Elemente aus dem absteigenden Teil und anschließend wieder aus dem aufsteigenden Teil. Aus der Graphik (Abb. 11.6) ist sofort ersichtlich, dass s_1 und s_2 die Bedingungen 1 und 2 von oben erfüllen.

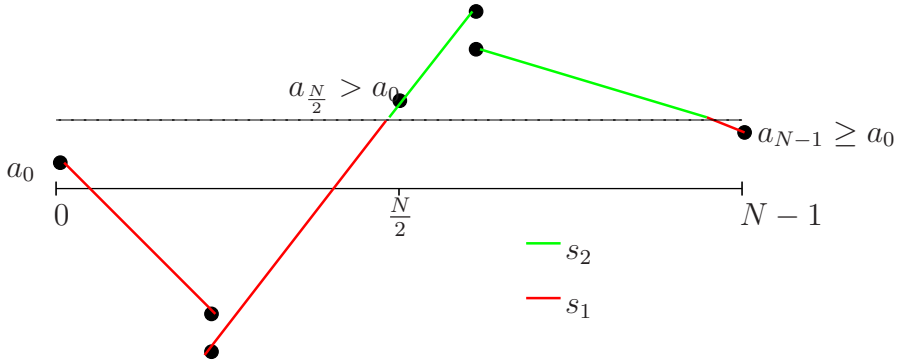


Abbildung 11.8: Minimum zwischen a_0 und $a_{\frac{N}{2}}$.

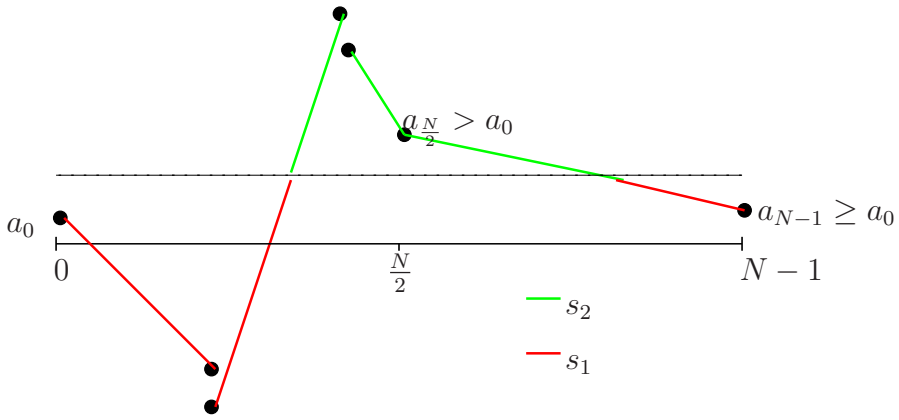


Abbildung 11.9: Minimum und Maximum zwischen a_0 und $a_{\frac{N}{2}}$.

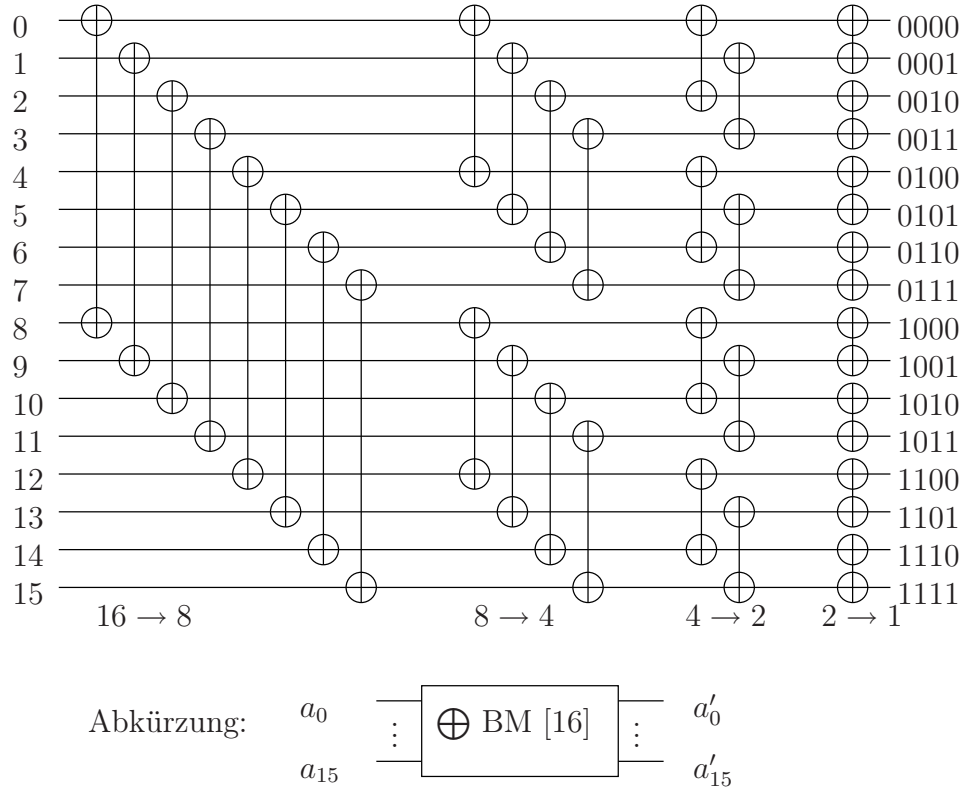


Abbildung 11.10: Bitonisches Sortiernetzwerk der Tiefe 4.

2. Die anderen Fälle: $a_0 = a_{\frac{N}{2}}$ bzw. $a_0 > a_{\frac{N}{2}}$ überlegt man sich analog.

□

Um eine *bitonische* Folge der länge $N > 2$ zu sortieren, wendet man die bitonische Zerlegung rekursiv auf beide Hälften an. Die Rekursionstiefe ist natürlich d. Abb. 11.10 zeigt ein bitonisches Sortiernetzwerk, um eine bitonische Folge von 16 Zahlen zu sortieren.

Um nun N *unsortierte* Zahlen zu sortieren, muss man diese in eine bitonische Folge verwandeln. Wir bemerken zunächst, dass man durch bitonisches Zerlegen eine bitonische Folge auch leicht in absteigender Reihenfolge sortieren kann. Dazu ist in der bitonischen Zerlegung (11.1) max mit min zu vertauschen. Entsprechend sind im Netzwerk (Abb. 11.10) die $\oplus$-Komparatoren durch $\ominus$-Komparatoren zu ersetzen. Das entsprechende Netzwerk bezeichnet man mit $\ominus\mathrm{BM}[N]$.

Bitonische Folgen erzeugt man folgendermaßen:
Eine Folge der Länge zwei ist immer bitonisch, da $a_0 \leq a_1$ oder $a_0 > a_1$. Hat man zwei bitonische Folgen der Länge N, so sortiert man die eine mit $\oplus\mathrm{BM}[N]$ aufsteigend und die andere mit $\ominus\mathrm{BM}[N]$ absteigend und erhält so eine bitonische Folge der Länge $2N$. Abb. 11.11 zeigt nun das vollständige Netzwerk zum Sortieren von 16 Zahlen.

Betrachten wir die Komplexität des bitonischen Sortierens. Für die Gesamttiefe $d(N)$ des Netzwerkes bei $N = 2^k$ erhalten wir

$$\begin{aligned} d(N) &= \underbrace{\mathrm{ld}\, N}_{\oplus\mathrm{BM}[N]} + \underbrace{\mathrm{ld}\, \tfrac{N}{2}}_{\mathrm{BM}[N/2]} + \mathrm{ld}\, \tfrac{N}{4} + \ldots + \mathrm{ld}\, 2 = \\ &= k + k-1 + k-2 + \ldots + 1 = \\ &= \Theta(k^2) = \Theta(\mathrm{ld}^2\, N). \end{aligned}$$

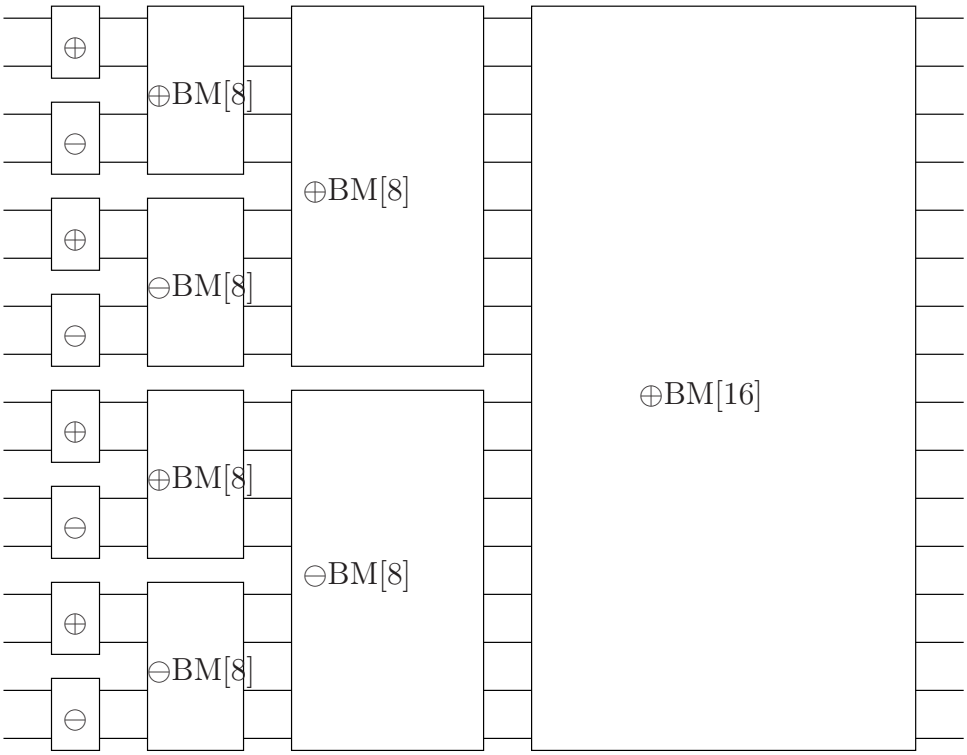


Abbildung 11.11: Sortieren von 16 unsortierten Zahlen mit bitonischem Mischen.

Damit ist die Gesamtkomplexität für bitonisches Mischen also $\Theta(N \operatorname{ld}^2 N)$.

11.2.3 Bitonisches Sortieren auf dem Hypercube

Wir überlegen nun, wie bitonisches Sortieren auf dem Hypercube realisiert werden kann. Dazu unterscheiden wir die Fälle $N = P$ (jeder hat eine Zahl) und $N \gg P$ (jeder hat einen Block von Zahlen).

$N = P$

Bitonisches Sortieren läst sich optimal auf den Hypercube abbilden! Abb. 11.10 zeigt, dass nur nächste Nachbar Kommunikation erforderlich ist! Im Schritt $i = 0, 1, \ldots, d-1$ kommunizert Prozessor p mit Prozessor $q = p \oplus 2^{d-i-1}$ (das $\oplus$ bedeutet hier wieder `XOR`). Offensichtlich erfordert aber jeder Vergleich eine Kommunikation, die Laufzeit wird also von den Aufsetzzeiten t_s der Nachrichten dominiert.

$N \gg P$

Nun erhält jeder Prozessor einen Block von $K = \frac{N}{P}$ (N durch P teilbar) Zahlen.

Nun kann man die Aufsetzzeiten amortisieren, da jeder Prozessor K Vergleiche durchführt, er „simuliert“ sozusagen K Komparatoren des Netzwerkes. Es bleibt noch das Problem, dass bitonisches Sortieren keine optimale sequentielle Komplexität besitzt, und somit keine isoeffiziente Skalierung erreicht werden kann. Dies kann man verbessern durch folgenden Ansatz:

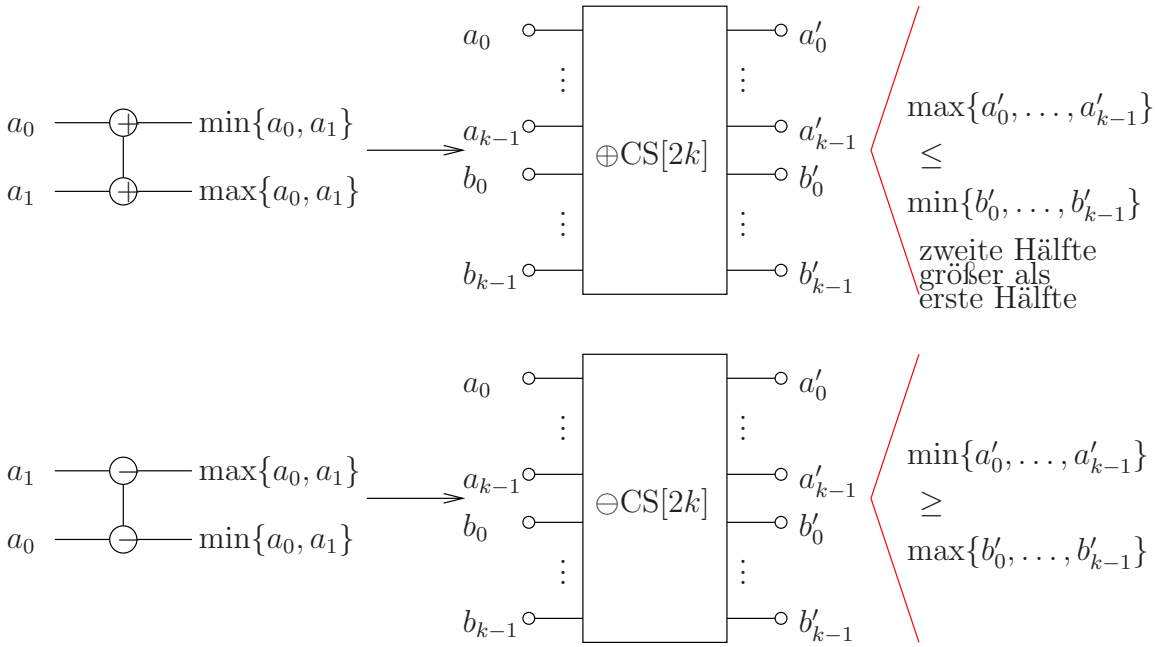


Abbildung 11.12: Komparatoren werden durch block-basierte Bausteine ersetzt.

> Es sei $P = 2^k$. Wir stellen uns bitonisches Sortieren auf P Elementen vor, nur dass jedes Element nun seinerseits eine Folge von K Zahlen ist.

Es zeigt sich, dass man $P \cdot K$ Zahlen sortieren kann, indem man die Komparatoren für zwei Zahlen durch Bausteine für $2K$ Zahlen ersetzt, wie sie in Abb. 11.12 dargestellt sind. Dabei sind die Folgen $\langle a_0, \ldots, a_{k-1}\rangle$ bzw. $\langle b_0, \ldots, b_{k-1}\rangle$ bereits aufsteigend sortiert, und die Ausgabefolgen $\langle a'_0, \ldots, a'_{k-1}\rangle$ bzw. $\langle b'_0, \ldots, b'_{k-1}\rangle$ *bleiben* aufsteigend sortiert. CS$[2k]$ kann auf zwei Prozessoren in $O(K)$ Zeit durchgeführt werden, indem jeder dem anderen seinen Block schickt, jeder die beiden Blöcke mischt und die richtige Hälfte behält.

Für die Gesamtkomplexität erhalten wir:

$$T_P(N,P) = c_1 \underbrace{\log^2 P}_{\text{Stufen}} \underbrace{\frac{N}{P}}_{\substack{\text{Aufwand}\\ \text{pro}\\ \text{Stufe}}} + \underbrace{c_2 \frac{N}{P} \log \frac{N}{P}}_{\substack{\text{einmaliges}\\ \text{Sortieren der}\\ \text{Eingabeblöcke}\\ \text{(und Ausgabe}\\ \text{bei } \ominus)}} + \underbrace{c_3 \log^2 P \frac{N}{P}}_{\text{Kommunikation}}$$

Somit gilt für die Effizienz

$$\begin{aligned} E(N,P) &= \frac{T_S}{T_P P} = \frac{c_2 N \log N}{\left(c_2 \frac{N}{P} \log \frac{N}{P} + (c_1 + c_3) \frac{N}{P} \log^2 P\right) P} = \\ &\underset{\substack{\log \frac{N}{P} = \\ \log N - \\ \log P}}{=} \frac{1}{\frac{\log N - \log P}{\log N} + \frac{c_1 + c_3}{c_2} \cdot \frac{\log^2 P}{\log N}} = \\ &= \frac{1}{1 - \frac{\log P}{\log N} + \frac{c_1 + c_3}{c_2} \cdot \frac{\log^2 P}{\log N}} \end{aligned}$$

Eine Isoeffiziente Skalierung erfordert somit

$$\frac{\log^2 P}{\log N} = K$$

und somit $N(P) = \Theta(P^{\log P})$ (da $\log N(P) = \log^2 P \iff N(P) = 2^{\log^2 P} = P^{\log P}$). Wegen $W = \Theta(N \log N)$ folgt aus $N \log N = P^{\log P} \cdot \log^2 P$ für die Isoeffizienzfunktion

$$W(P) = \Theta(P^{\log P} \log^2 P).$$

Der Algorithmus ist also *sehr* schlecht skalierbar!!

11.3 Paralleles Quicksort

Quicksort besteht aus der Partitionierungsphase und dem rekursiven Sortieren der beiden Teilmengen. Naiv könnte man versuchen, immer mehr Prozessoren zu verwenden, je mehr unabhängige Teilprobleme man hat. Dies ist nicht kostenoptimal, da die Teilprobleme immer kleiner werden.

Formal erhält man:

$$\begin{aligned} T_S(N) &= \underbrace{N + 2\frac{N}{2} + 4\frac{N}{4} + \ldots}_{\substack{\log N \\ \text{Schritte} \\ \text{(average)}}} = \\ &= N \log N \\ T_P(N) &\underset{\substack{N = P \\ \text{naive} \\ \text{Variante}}}{=} N + \frac{N}{2} + \frac{N}{4} + \cdots = \\ &= 2N \end{aligned}$$

Für die Kosten erhält man ($P = N$)

$$P \cdot T_P(N) = 2N^2 > N \log N$$

also nicht optimal.

Fazit: Man muss den Partitionierungsschritt parallelisieren, und das geht so:

Es sei $P = 2^d$, jeder hat N/P Zahlen.

Wir benutzen die Tatsache, dass ein Hypercube der Dimension d in zwei der Dimension $(d-1)$ zerfällt, und jeder im ersten Hypercube hat genau einen Nachbarn im zweiten (siehe Abb. 11.13).

Nun wird das Pivot x gewählt und an alle verteilt, dann tauschen die Partnerprozessoren ihre Blöcke aus. Die Prozessoren im ersten Hypercube behalten alle Elemente $\leq x$, die im zweiten Hypercube alle Elemente $> x$. Dies macht man rekursiv d mal und sortiert dann lokal in jedem Prozessor mit Quicksort. Fertig.

Komplexität (average case: jeder behält immer $\frac{N}{P}$ Zahlen):

$$T_P(N,P) = \underbrace{c_1 \operatorname{ld} P \cdot \frac{N}{P}}_{\text{split}} + \underbrace{c_2 \operatorname{ld} P \cdot \frac{N}{P}}_{\text{Komm.}} + \underbrace{c_3 \frac{N}{P} \operatorname{ld} \frac{N}{P}}_{\substack{\text{lokale} \\ \text{Quicksort.}}} + \underbrace{c_4 d^2}_{\substack{\text{Pivot} \\ \text{Broad-} \\ \text{cast}}} \;;$$

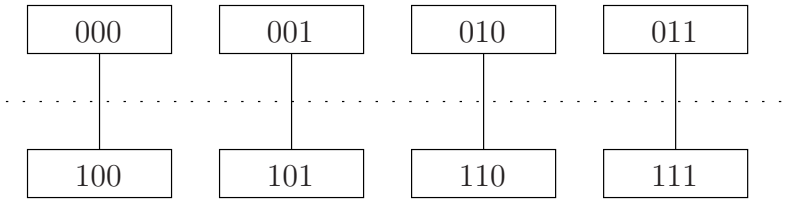


Abbildung 11.13: Zweiteilung eines Hypercube für Quicksort

der Aufwand für den Broadcast des Pivot ist $\operatorname{ld} P + \operatorname{ld} \frac{P}{2} + \cdots = d + d{-}1 + d{-}2 + \cdots = O(d^2)$. Für die Effizienz erhält man

$$\begin{aligned} E(N,P) &= \frac{c_3 N \operatorname{ld} N}{\left(c_3 \frac{N}{P} \operatorname{ld} \frac{N}{P} + (c_1+c_2)\frac{N}{P} \operatorname{ld} P + c_4 \operatorname{ld}^2 P\right) P} = \\ &= \frac{1}{\frac{\operatorname{ld} N - \operatorname{ld} P}{\operatorname{ld} N} + \frac{c_1+c_2}{c_3} \cdot \frac{\operatorname{ld} P}{\operatorname{ld} N} + \frac{c_4}{c_3} \cdot \frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N}} = \\ &= \frac{1}{1 + \left(\frac{c_1+c_2}{c_4} - 1\right) \frac{\operatorname{ld} P}{\operatorname{ld} N} + \frac{c_4}{c_3} \cdot \frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N}}. \end{aligned}$$

Für eine isoeffiziente Skalierung ist der Term aus dem Broadcast entscheidend:

$$\frac{P \operatorname{ld}^2 P}{N \operatorname{ld} N} = O(1),$$

für $N = P \operatorname{ld} P$ erhalten wir

$$\begin{aligned} N \operatorname{ld} N &= (P \operatorname{ld} P)\ \operatorname{ld}(P \operatorname{ld} P) = (P \operatorname{ld} P)\ (\operatorname{ld} P + \underbrace{\operatorname{ld} \operatorname{ld} P}_{\substack{\text{sehr} \\ \text{klein!}}}) \approx \\ &\approx P \operatorname{ld}^2 P. \end{aligned}$$

Mit einem Wachstum $N = \Theta(P \operatorname{ld} P)$ sind wir also auf der sicheren Seite (es genügt sogar etwas weniger).

Für die Isoeffizienzfunktion gilt wegen $W = N \log N$

$$W(P) = \Theta(P \operatorname{ld}^2 P),$$

also deutlich besser als bei bitonischem Sortieren.

Empfindlichkeit gegenüber Pivotwahl

Das Problem dieses Algorithmus ist, dass die Wahl eines schlechten Pivotelements zu einer schlechten Lastverteilung in *allen* nachfolgenden Schritten führt und somit zu schlechtem Speedup. Wird das allererste Pivot maximal schlecht gewählt, so landen alle Zahlen in *einer* Hälfte und der Speedup beträgt höchstens noch $P/2$. Bei gleichmäßiger Verteilung der zu sortierenden Elemente kann man den Mittelwert der Elemente eines Prozessors als Pivot wählen.

A Der Handlungsreisende und Posix Threads

In diesem Abschnitt lösen wir ein bekanntes Problem der diskreten Optimierung, das sogenannte Handlungsreisendenproblem (*engl.* travelling salesman problem), mit Hilfe einer parallelen Implementierung von branch and bound. Der Schwerpunkt liegt dabei auf der parallelen Realisierung mit Posix threads. Selbstverständlich sind zur erfolgreichen Lösung des Problems effiziente Algorithmen sehr wichtig. Diese würden aber den Rahmen dieser Übung (deutlich) sprengen. Eine Literatursammlung findet man in M. Jünger (1997).

A.1 Das Handlungsreisendenproblem

Ein Handlungsreisender (travelling salesman) muss Kunden in insgesamt n Städten besuchen. Die Abstände der Städte sind gegeben. Das Problem ist nun einen Weg minimaler Länge zu finden, der jede Stadt genau einmal berührt und am Ende wieder in die Ausgangsstadt zurückführt. Dabei ist es offensichtlich egal bei welcher Stadt man beginnt. Abb. A.1 zeigt ein Beispiel mit 12 Städten.

Das Handlungsreisendenproblem gehört zur Klasse der NP-vollständigen Problem. Dies bedeutet (unter anderem), dass exakte Algorithmen zu seiner Lösung eine Laufzeit haben, die exponentiell mit der Anzahl der Städte anwächst. Wir werden im folgenden nur exakte Algorithmen behandeln!

Wir beginnen mit der Tiefensuche (depth first search). Hierbei untersucht man ausgehend von einer Stadt systematisch alle möglichen Wege, berechnet deren Länge und merkt sich das Minimum. Nummerieren wir dazu alle Städte von 0 bis $n-1$ durch und beginnen mit der Stadt 0. So können wir als nächstes die Städte 1, 2, ..., $n-1$ besuchen, also $n-1$ Möglichkeiten. Für jede dieser Möglichkeiten gibt es als dritte Stadt dann $n-2$ Möglichkeiten und so weiter. Die Gesamtanzahl aller möglichen Pfade ist somit $(n-1)!$. Abb. A.2 zeigt die Entwicklung aller möglichen Pfade als Baum. Bei der sogenannten Tiefensuche werden die Knoten in der Reihenfolgen 0, 01, 012, 0123, 013, 0132, usw. besucht. Dies formuliert man am einfachsten mit einer rekursiven Funktion:

```
dfs (path)
{
    if (path hat Länge n)
        if (path ist bester Pfad) best = path;
    else
    {
        for (alle noch nicht benutzten Städte i)
            dfs(path∪i);
    }
}
```

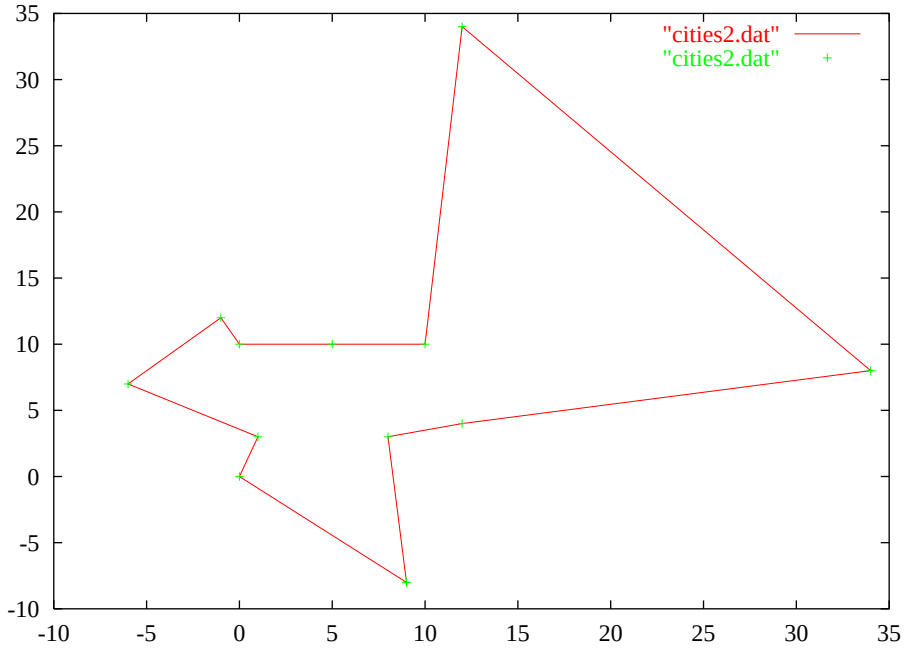


Abbildung A.1: Lösung des Handlungsreisendenproblems.

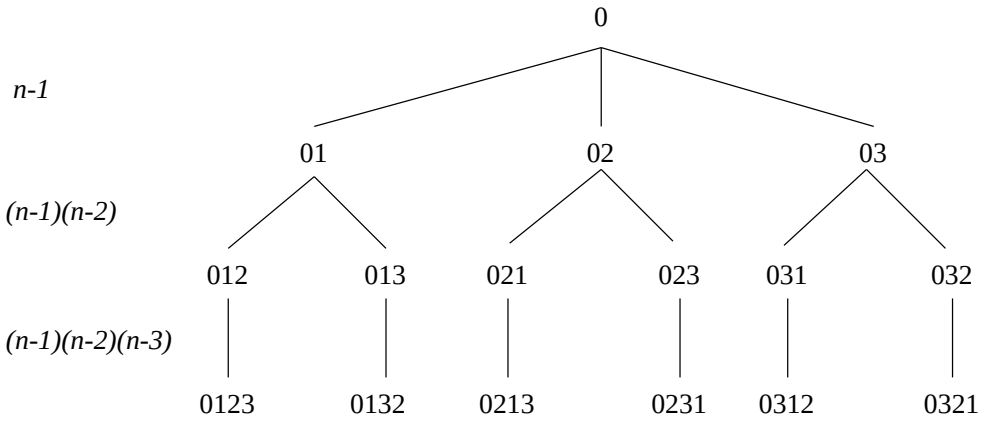


Abbildung A.2: Der vollständige Suchbaum.

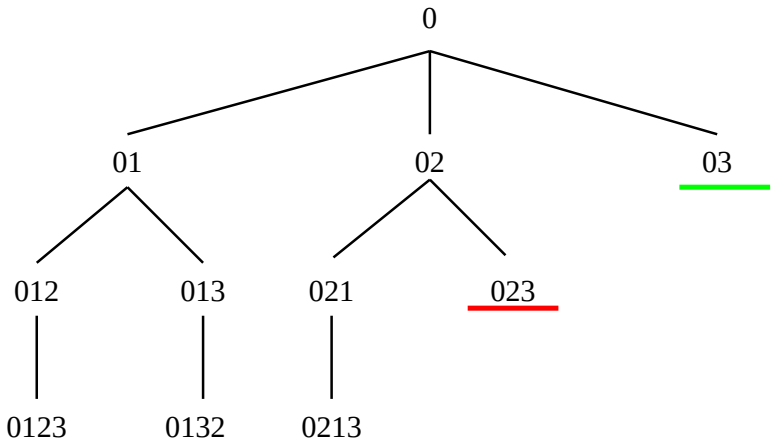


Abbildung A.3: Abgeschnittener Suchbaum in branch and bound.

Dabei ist *path* eine nicht weiter erläuterte Datenstruktur, die eine Sequenz von Städten speichert.

Eine einfache Variation der Tiefensuche ist Verzweigen und Abschneiden (branch and bound). Hier überprüft man einfach ob ein Teilpfad nicht schon die Länge des besten bis dahin bekannten Pfades erreicht hat und bricht die Suche in diesem Fall vorzeitig ab. Dies ist in folgendem Programmstück realisiert:

```
bb (path)
{
    if (path hat Länge n)
        if (path ist neuer bester Pfad) best = path;
    else
    {
        for (alle noch nicht benutzten Städte i)
            if (path∪i kürzer als bester bekannter Pfad)
                bb(path∪i);
    }
}
```

Die Abbildung A.3 zeigt den entsprechenden Suchbaum, der nun unregelmässig beschnitten ist. Dies wird die Hauptschwierigkeit bei der Parallelisierung darstellen. Bevor wir zur parallelen Implementierung kommen wollen wir das Verfahren noch in einer nichtrekursiven Variante formulieren, da sich diese besser für eine parallele Implementierung eignet:

```
path = {0};
push(path);

while (pop(path))
{
    if (path hat Länge n)
```

```
        if (path ist neuer bester Pfad) best = path;
    else
    {
        for (alle noch nicht benutzten Städte i)
            if (path∪i kürzer als bester bekannter Pfad)
                push(path∪i);
    }
}
```

Die geeignete Datenstruktur ist hierzu ein Stapel (stack). Zu Beginn wird der Stapel mit dem Pfad, der nur die Stadt 0 enthält initialisiert. In jeder Iteration wird der oberste Pfad vom Stapel entnommen. Hat der Pfad die Länge n wird er bewertet, ansonsten wird jede noch nicht besuchte Stadt angehängt und die resultierenden (um eins längeren Pfade) werden auf den Stapel gelegt falls die Länge des Pfades nicht die des besten bekannten Pfades übersteigt. Das Programm ist beendet wenn der Stapel leer ist.

A.2 Parallele Implementierung

Die Parallelisierung basiert auf der Beobachtung, dass alle Knoten des Suchbaumes unabhängig voneinander gleichzeitig bearbeitet werden können. Man muss also nur sicherstellen, dass jeder Knoten von genau einem Prozessor bearbeitet wird. Jeder Knoten im Baum (ein Teilpfad) charakterisiert auch eindeutig alle Nachfolger dieses Knotens im Baum.

In der parallelen Variante geben wir jedem Thread seinen eigenen Stapel, den nur er bearbeitet. Einen weiteren *globalen* Stapel verwenden wir um die Arbeit zwischen den Threads zu verteilen. Im Branch-and-Bound-Verfahren kann es nämlich vorkommen, dass einem Prozess die Arbeit früher ausgeht wie einem anderen, dann muss er von einem anderen Prozess Arbeit bekommen. Dies nennt man *dynamische Lastverteilung*.

Zu Beginn wird der globale Stapel mit der Wurzel initialisiert und die Arbeiterprozesse werden gestartet. Der Stapel jedes Arbeiters ist zunächst leer, er ist also arbeitslos. Ein arbeitsloser Arbeiter versucht einen Pfad aus dem globalen Stapel zu entnehmen. Ist dieser leer so erhöht er eine globale Variable, die die Anzahl der auf Arbeit wartenden Threads angibt ansonsten wird der Pfad in den lokalen Stapel geladen und das Branch-and-Bound-Verfahren gestartet. Im Branch-and-Bound überprüft ein Prozess nach einer bestimmtem Anzahl von Iterationen ob es arbeitslose Threads gibt. Ist dies der Fall (und er selbst hat offensichtlich Arbeit), so gibt der Thread Arbeit ab indem er einen Knoten aus dem lokalen Stapel holt und ihn in den globalen Stapel schreibt. Der abzugebende Knoten sollte möglichst *viel* Arbeit enthalten. Knoten weit oben im Stapel (z. B. der nächste von `pop` gelieferte Knoten) enthalten aber normalerweise wenig Arbeit, da sie weit unten im Suchbaum liegen. Man sollte also die abzugebenden Knoten möglichst vom anderen (unteren) Ende des Stapels nehmen und explizit abfragen ob der Knoten eine bestimmte Tiefe im Baum nicht überschreitet. Dazu ist der Stapel mit einer zusätzlichen Funktion zu erweitern.

Wir beginnen mit der Definition einiger Konstanten und der Datenstrukturen:

```
/* include files */
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <pthread.h>
```

```
#include <semaphore.h>

/* Programmkonstanten */
#define MAXN 24              /* sehr ambitioniert ....         */
#define MAXS 500             /* maximale Stackgroesse          */
#define MAXP 100             /* maximale Zahl von Threads      */
#define MAXC 500             /* Anz. unterbrechungsfreier
                        Durchlaeufe    */
#define MIND 7               /* minimale Baumhoehe
                    zum abgeben            */

/* Anzahl threads */
int P;

/* Problemdefinition */
int n=0;                     /* Groesse                        */
float dist[MAXN][MAXN];      /* Distanzen zwischen
                    den Staedten  */

/* Datenstruktur fuer Zustandsinformation */
typedef struct {
    char path[MAXN];         /* Pfad                           */
    int l;                   /* der Laenge l                   */
    float cost;              /* Kosten bis hierher             */
} node ;

/* Datenstruktur fuer einen Stack */
typedef struct  {
    node s[MAXS];            /* der Stack                      */
    int tos;                 /* top of stack                   */
    int bos;                 /* bottom of stack                */
} stack ;

/* Datenstruktur fuer einen shared Stack */
typedef struct  {
    node s[MAXS];            /* der Stack                      */
    int tos;                 /* top of stack                   */
    pthread_mutex_t mutex;   /* Absicherung des Stackzugriffes */
    int wait_count;          /* Anzahl wartender Prozesse      */
    sem_t wait_sem;          /* Arbeitslose blockieren hier    */
    int terminate;           /* Terminierungsbedingung         */
} shared_stack ;
```

Das Feld `dist` beschreibt die Distanzen zwischen den Städten. Diese und auch die Anzahl der Städte werden weiter unten von einer Datei eingelesen. Die Struktur `node` beschreibt einen Teilpfad bestehent aus den Städtenummern, der Anzahl der Städte und den addierten Distanzen zwischen den Städten. Die Struktur `stack` implementiert den lokalen Stapel und `shared_stack` den globalen Stapel. Der globale Stapel enthält eine Mutex-Variable um exklusiven Zugriff si-

cherzustellen, sowie eine Semaphore, die zum warten und aufwecken der arbeitslosen Threads verwendet wird.

Nun werden die globalen Variablen angelegt:

```
/* der globale Stack */
shared_stack global_stack;
    /* speichert noch zu expandierende Knoten*/
pthread_mutex_t signal_mutex = PTHREAD_MUTEX_INITIALIZER;

/* Ergebnis: */
node bestnode;                /* global bester Knoten */
pthread_mutex_t best_mutex = PTHREAD_MUTEX_INITIALIZER;
                    /* mutex f opt*/
```

Die Mutex-Variable `signal_mutex` wird benutzt um sicherzustellen, dass nur ein Thread versucht Arbeit abzugeben. Die Mutex-Variable `best_mutex` sichert den exklusiven Zugriff auf den Knoten `bestnode` ab.
Hier sind die Operationen auf dem lokalen Stapel:

```
void init_stack (stack *s)
{
    s->tos = s->bos = 0;
    return;
}

int push (stack *s, node *c)
{
    if ((s->tos+1)%MAXS == s->bos) return 0; /* stack voll */
    s->s[s->tos] = *c;
    s->tos = (s->tos+1)%MAXS;
    return 1;
}

int pop (stack *s, node *c)
{
    if (s->tos == s->bos) return 0; /* stack leer */
    s->tos = (s->tos+MAXS-1)%MAXS;
    *c = s->s[s->tos];
    return 1;
}

int pop_rear (stack *s, node *c)
{
    if (s->tos == s->bos) return 0; /* stack leer */
    *c = s->s[s->bos];
    s->bos = (s->bos+1)%MAXS;
    return 1;
}
```

```
int peek_rear (stack *s, node *c)
{
    if (s->tos == s->bos) return 0; /* stack leer */
    *c = s->s[s->bos];
    return 1;
}

int empty (stack *s)
{
    if (s->tos == s->bos) return 1; /* stack leer */
    return 0;
}
```

Hier ist vor allem auf die zusätzlichen Funktionen `peek_rear` und `pop_rear` hinzuweisen, die den Zugriff auf das unterste Element im Stapel realisieren.
Und hier die Operationen auf dem globalen Stapel:

```
/*****************************************************
 * Operationen auf dem shared Stack
 *****************************************************/

void init_shared_stack (shared_stack *s)
{
    s->tos = 0;
    pthread_mutex_init(&(s->mutex),NULL);
    sem_init(&(s->wait_sem),0,0); /* S=0 am Anfang ! */
    s->wait_count = 0;
    s->terminate = 0;
    return;
}

int shared_waiting (shared_stack *s)
{
    return s->wait_count;
}

int shared_terminate (shared_stack *s)
{
    return s->terminate;
}

void shared_signal (shared_stack *s)
{
    if (s->wait_count<=0) return;
    pthread_mutex_lock(&s->mutex);
    (s->wait_count)-;
    sem_post(&s->wait_sem); /* wecke einen auf */
    pthread_mutex_unlock(&s->mutex);
```

```
    return;
}

int shared_wait (shared_stack *s)
{
    pthread_mutex_lock(&s->mutex);
    if (s->wait_count==P-1) /* alle sind fertig */
    {
        s->terminate = 1;
        pthread_mutex_unlock(&s->mutex);
        return -1;          /* Terminierung !   */
    }
    (s->wait_count)++;
    pthread_mutex_unlock(&s->mutex);
    sem_wait(&s->wait_sem); /* erwache ...  */
    return 1;
}

int shared_push (shared_stack *s, node *c)
{
    pthread_mutex_lock(&(s->mutex));
    if ((s->tos+1)%MAXS == 0)
    {
        pthread_mutex_unlock(&(s->mutex));
        return 0; /* stack voll */
    }
    s->s[s->tos] = *c;
    s->tos = (s->tos+1)%MAXS;
    pthread_mutex_unlock(&(s->mutex));
    return 1;
}

int shared_pop (shared_stack *s, node *c)
{
    pthread_mutex_lock(&(s->mutex));
    if (s->tos == 0)
    {
        pthread_mutex_unlock(&(s->mutex));
        return 0; /* stack leer */
    }
    s->tos = (s->tos+MAXS-1)%MAXS;
    *c = s->s[s->tos];
    pthread_mutex_unlock(&(s->mutex));
    return 1;
}
```

Exklusive Zugriffe auf den globalen Stapel sind mit einer Mutex-Variable abgesichert. Mit der Funktion `shared_wait` erhöht ein Thread die Anzahl der Wartenden und blockiert auf einer

Semaphore. Die Funktion `shared_signal` ist das Gegenstück dazu. Mit `shared_waiting` kann man die Anzahl der wartenden Threads erfragen ohne zu blockieren.

Auf den Knoten gibt es die folgenden Operationen:

```
void init_root (node *c)
{
    c->path[0] = 0;
    c->l = 1;
    c->cost=0;
    return;
}

void init_best (node *c)
{
    c->l = 0;
    c->cost=1E100;
    return;
}

void update_best (node *c, int p)
{
    if (c->cost>=bestnode.cost) return;
    pthread_mutex_lock(&best_mutex);
    if (c->cost<bestnode.cost)
    {
        bestnode = *c;
        printf("%2d: %g\n",p,c->cost);
    }
    pthread_mutex_unlock(&best_mutex);
    return;
}

void expand_node (stack *s, node *c, float bound)
            /* auf lokalem Stack ! */
{
    char used[MAXN];
    node cc;
    int i;
    float newcost;
    int lastcity;

    /* erzeuge used flags */
    for (i=0; i<n; i++) used[i]=0;
    for (i=0; i<c->l; i++) used[c->path[i]]=1;

    /* generiere Soehne mit bound */
    lastcity = c->path[c->l-1];
    for (i=n-1; i>=0; i-)
```

```
    {
        if (used[i]) continue;
        newcost = c->cost + dist[lastcity][i];
        if (newcost<bound)
        {
            cc = *c;
            cc.path[c->l] = (char) i;
            cc.l = c->l+1;
            cc.cost = newcost;
            if (!push(s,&cc))
            {
                printf("stack full\n");
                exit(1);
            }
        }
    }
    return;
}
```

Die Funktion `expand_node` nimmt einen Knoten vom gegebenen (lokalen) Stapel und stapelt die Nachfolger falls die Pfadlänge `bound` nicht überschritten wird.

Nun kommt die Hauptsache:

```
void branch_and_bound (stack *s, int p)
{
    node c,cc;
    int cnt=0;

    while (pop(s,&c))
    {
        if (c.l==n)
        {
            c.cost += dist[c.path[n-1]][0];
            update_best(&c,p);
        }
        else
            expand_node(s,&c,bestnode.cost);

        if (cnt<MAXC) cnt++;
        else {
            cnt = 0;
            if (shared_waiting(&global_stack)==0) continue;
            if (empty(s)) continue;
            if (peek_rear(s,&cc))
                if (n-cc.l<=MIND) continue;

            pthread_mutex_lock(&signal_mutex);
                    /* nur einer gibt ab ! */
```

```
            while (shared_waiting(&global_stack)>0)
            {
                if (pop_rear(s,&cc))
                {
                    printf("%2d: giving off work\n",p);
                    shared_push(&global_stack,&cc);
                    shared_signal(&global_stack);
                }
                else break;
            }
            pthread_mutex_unlock(&signal_mutex);
        }
    }

    return;
}
```

Vom lokalen Stapel werden solange Knoten expandiert bis dieser leer ist. Alle `MAXC` Iterationen wird überprüft ob ein anderer Thread wartet. Ist dies der Fall gibt ein Thread Arbeit ab, dazu muss er exklusiven Zugriff auf `signal_mutex` und einen Knoten genüegender Tiefe auf seinem lokalen Stapel haben.

Die Arbeiter versuchen endlos Arbeit vom globalen Stapel zu holen und lokal zu bearbeiten. Die Funktion `shared_wait` liefert den Wert -1 wenn bereits $P-1$ Threads auf Arbeit warten. In diesem Fall darf der letzte Prozess nicht blockieren sondern muss die anderen aufwecken. Wird ein Prozess geweckt, d. h. er kehrt aus `shared_wait` zurück muss er noch mal mittels `shared_terminate` prüfen ob die Terminierungsphase eingeleitet ist. Hier der Code für die Arbeiter:

```
void Worker (int *p) /* kriegt die Prozessnummer als Parameter */
{
    stack *local_stack;
    node cc;
    int i;

    /* allokiere lokalen leeren stack */
    local_stack = malloc(sizeof(stack));
    init_stack(local_stack);

    while (1)
    {
        /* versuche was vom globalen stack zu bekommen */
        if (shared_pop(&global_stack,&cc))
        {
            printf("%2d: got work\n",*p);
            push(local_stack,&cc);         /* auf lokalen stack */
            branch_and_bound(local_stack,*p); /* und abarbeiten */
        }
        else /* keine Arbeit da */
```

```
        {
            printf("%2d: out of work\n",*p);
            if (shared_wait(&global_stack)<0)
            {
                printf("%2d: terminating\n",*p);
                /* Terminierung, wecke alle anderen ! */
                for (i=0; i<P-1; i++)
                    shared_signal(&global_stack);
                break; /* ende */
            }
            else
            {
                if (shared_terminate(&global_stack))
                {
                    printf("%2d: terminating\n",*p);
                    break; /* ende */
                }
            }
        }
    }

    free(local_stack);

    return;
}
```

Das Einlesen der Städtedatei geben wir der Vollständigkeit halber auch noch an:

```
void read_cities (char *fname)
{
    float x[MAXN], y[MAXN];
    FILE *stream;
    int i,j;

    /* oeffne Datei */
    if ( (stream=fopen(fname,"r"))==NULL )
    {
        printf("Cannot open %s\n",fname);
        exit(1);
    }

    /* Lese Anzahl Staedte */
    fscanf(stream," %d", &n);
    if (n<2 || n>MAXN)
    {
        printf("Wrong n=%d\n",n);
        exit(1);
    }
```

```
    /* Lese Positionen */
    for (i=0; i<n; i++)
    {
        if (fscanf(stream," %f %f",x+i,y+i)!=2)
        {
            printf("Cannot read city i=%d\n",i);
            exit(1);
        }
    }
    fclose(stream);
    printf("%d Staedte gelesen.\n",n);

    /* Berechne Distanzen */
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            dist[i][j] = sqrt( (x[i]-x[j])*(x[i]-x[j])+
                        (y[i]-y[j])*(y[i]-y[j]) );

    return;
}
```

Schließlich fehlt nur noch das Hauptprogramm:

```
int main (int argc, char **argv)
{
    int i;
    node root;
    pthread_t workers[MAXP];
    int p[MAXP];

    /* Argumente richtig ? */
    if (argc<3)
    {
        printf("<progname> <numthreads> <infile>\n");
        exit(1);
    }

    /* Anzahl Threads ist das erste Argument */
    P = atoi(argv[1]);
    if (P<=0 || P>MAXP)
    {
        printf("Anzahl Threads nicht richtig, P=%d\n",P);
        exit(1);
    }

    /* lese Daten */
    read_cities(argv[2]);
```

```
    /* besten Weg */
    init_best(&bestnode);

    /* initialisiere stack mit Wurzel */
    init_root(&root);
    init_shared_stack(&global_stack);
    shared_push(&global_stack,&root);

    /* starte Threads */
    for (i=0; i<P; i++)
    {
        p[i] = i;
        pthread_create(&(workers[i]),NULL,(void *)Worker,
                       (void *)&(p[i]));
    }

    /* warte auf alle Threads */
    for (i=0; i<P; i++)
    {
        pthread_join(workers[i],NULL);
    }

    /* Drucke Ergebnis */
    printf("%g\n",bestnode.cost);
    for (i=0; i<n; i++)
        printf("%d ",(int)bestnode.path[i]);
    printf("\n");

    return 0;
}
```

Es enthält die Initialisierung der globalen Datenstrukturen, das Starten und Stoppen der Arbeiter sowie die Ausgabe des Ergebnisses.

Das Programm liefert auf einer 2-Prozessormaschine unter dem Betriebssystem LINUX die folgenden Ergebnisse (Rechenzeit in Sekunden):

N	10	11	12
$P = 1$	0.268	1.869	15.987
$P = 2$	0.187	1.052	8.397

Für das größte Problem mit 12 Städten wird eine Beschleunigung von 1.9 erreicht.

B Dichtegetriebene Grundwasserströmung

Wir betrachten im folgenden ein Beispiel für eine dreidimensionale gitterbasierte Berechnung eines instationären (also von der Zeit abhängigen), nichtlinearen Problems.

B.1 Problemstellung

Wir wollen den Grundwasserfluß innerhalb des porösen Mediums Sand berechnen. Dazu wählen wir einen Würfel der Kantenlänge $300m^3$. In der oberen Hälfte des Würfels befinde sich Salzwasser (*engl.* brine), in der unteren Süßwasser.

An der Grenzschicht zwischen den beiden Wasserschichten herrscht ein instabiles Gleichgewicht. Das bedeutet, daß kleine Inhomogenitäten innerhalb der Grenzschicht ein Absinken des Salzwassers an diesen Stellen bewirkt. Gleichzeitig wird Süßwasser noch oben verdrängt. Das führt zur Ausbildung von sogenannten Konvektionswalzen oder Fingern und so zur Durchmischung der beiden Wasserschichten.

Natürlicherweise entstehen solche Inhomogenitäten durch die irreguläre und ungleichmäßige Verteilung der Sandkörner. In der perfekten Rechnerwelt[1] muß eine solche Inhomogenität künstlich erzeugt werden, indem man eine Delle in die Grenzschicht einbaut (z.B. eine kleine Absenkung des Salzwassers auf einem rechteckigen Teil der Grenzschicht). Iterationsfehler und Diskretisierungsfehler bewirken dann die Entstehung weiterer Finger. Abbildung B.3 zeigt die Isoflächen der Konzentration $C = 0.5$ zu zwei Zeitpunkten am Anfang und Ende der Simulation.

[1] nicht mit der Aussage zu verwechseln, die Rechner seien perfekt

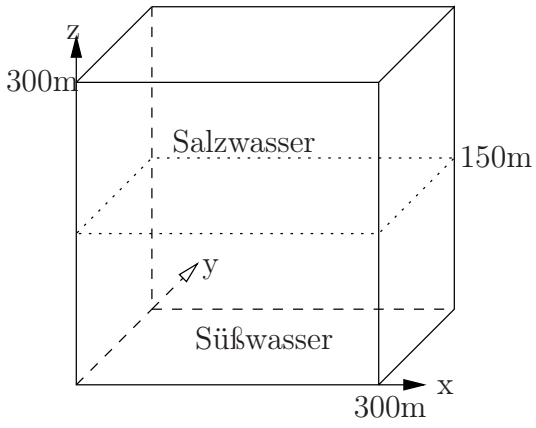


Abbildung B.1: Ausgangssituation

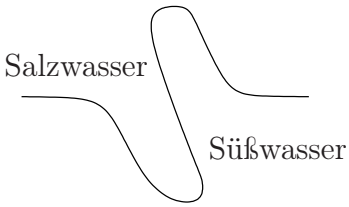


Abbildung B.2: Ausbildung eines Fingers

B.2 Mathematisches Modell

Zunächst geben wir Begriffe an und bestimmen die Unbekannten, die wir berechnen wollen. Betrachen wir einen Ausschnitt unseres Mediums, ein sogenanntes REV (repesentative elementary volume).

Den Raum, in dem das Wasser fließen kann, nennen wir Porenraum. Als charakteristische Größe eines Porenraumes definieren wir die Porosität Φ als

$$\Phi = \frac{\text{Volumen des (erreichbaren) Porenraumes im REV}}{\text{Volumen des REV}}$$

Nicht erreichbarer Porenraum könnten Hohlräume innerhalb der Sandkörner sein. Es ist $\Phi \in [0,1]$. Bei uns ist $\Phi = 0.1$ im ganzen Würfel.

Ein REV erhält man dadurch, daß man einen sehr kleinen Würfel nimmt, ihn vergrößert und die Veränderung von Φ betrachtet. Wenn diese bei der Vergrößerung kaum noch Schwankungen aufweist, hat man die Größe eines REV erhalten.

Die Dichte des Salzwassers ϱ_B betrage $1200\frac{kg}{m^3}$, die des Süßwassers ϱ_0 betrage $1000\frac{kg}{m^3}$. Die Mischung besteht immer aus einem Anteil Salzwasser und einem Anteil Süßwasser. Wir definieren die

$$\text{Konzentration } C = \frac{\text{Masse Salzwasser im REV}}{\text{Masse der Mischung im REV}}$$

und die

$$\text{Dichte } \varrho = \frac{\text{Masse der Mischung im REV}}{\text{Volumen der Mischung im REV}}$$

Zu bestimmen ist die Konzentration $\mathbf{C(x,y,z,t)}$. Dabei bedeutet C = 0 reines Süßwasser und C = 1 reines Salzwasser.

Die Dichte der Mischung ϱ ist abhängig von der Konzentration: $\varrho = \varrho(C(x,y,z,t))$. Wir verwenden dazu die lineare Interpolation $\varrho(C) = C \cdot \varrho_B + (1-C)\varrho_0$. Dann definieren wir noch die intrinsische Dichte

$$\begin{aligned}\varrho(C) \cdot C &= \frac{\text{Masse der Mischung im REV}}{\text{Volumen der Mischung im REV}} \cdot \frac{\text{Masse Salzwasser im REV}}{\text{Masse Mischung im REV}} \\ &= \frac{\text{Masse Salzwasser im REV}}{\text{Volumen des Porenraums im REV}}\end{aligned}$$

Das Volumen der Mischung im REV ist ja gleich dem Volumen des Porenraumes im REV.

Bezeichne Ω den Gesamtwürfel und sei $\Omega' \subseteq \Omega$. Wir bezeichnen die Gesamtmasse der Mischung

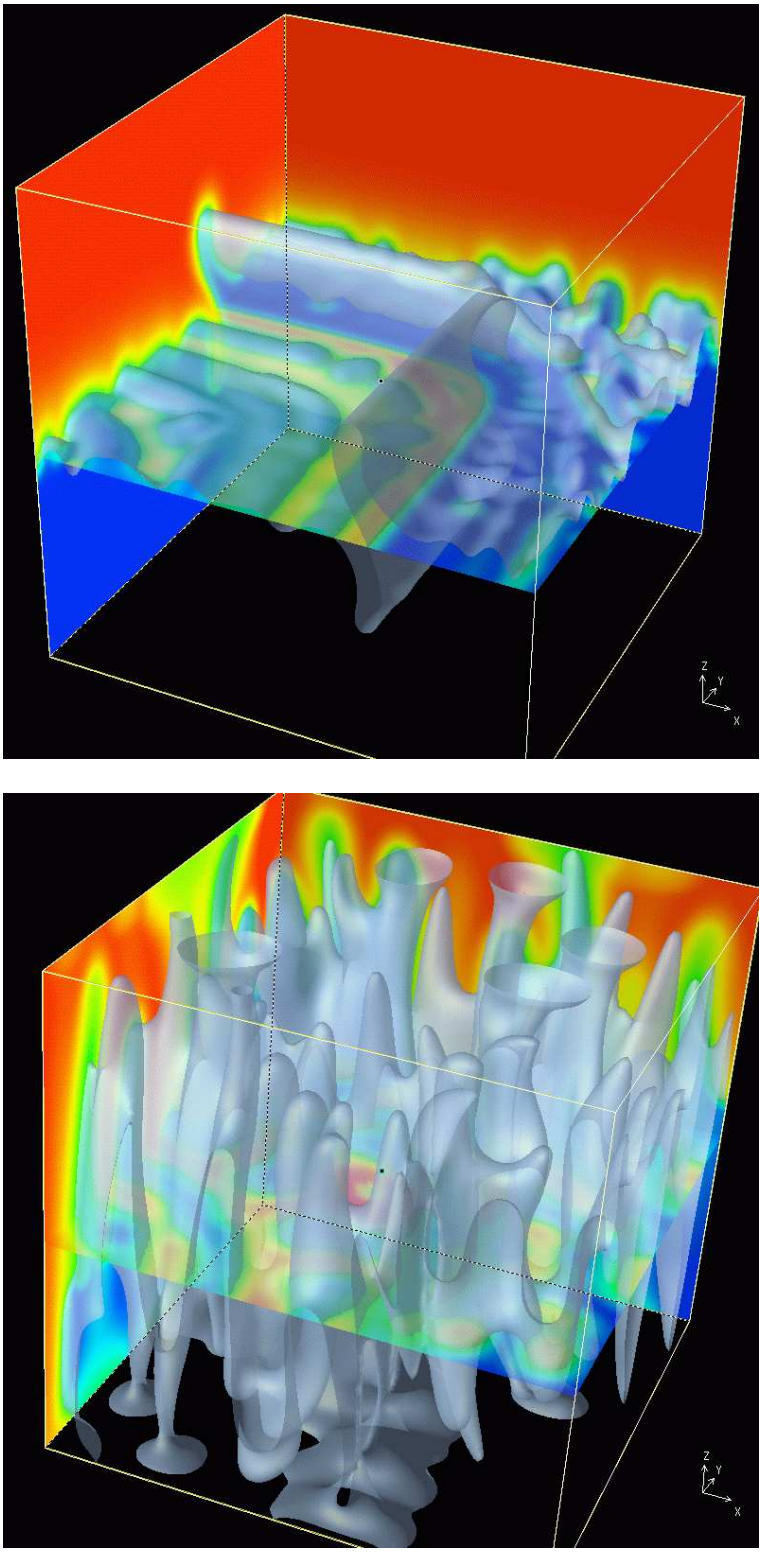


Abbildung B.3: Oben: Ausbildung des ersten Fingers, unten: ein später Zeitpunkt.

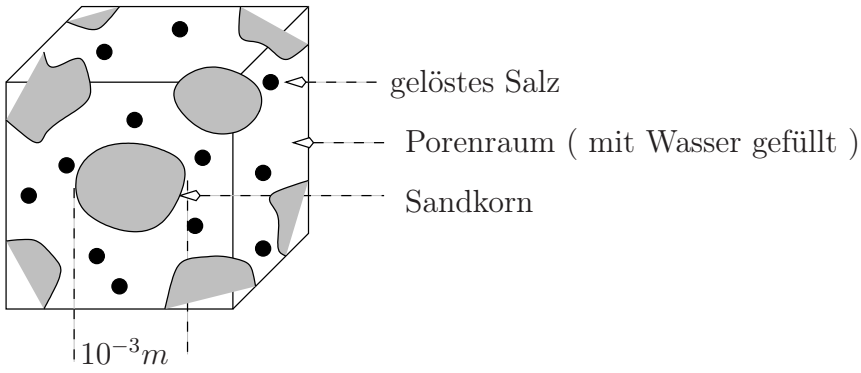


Abbildung B.4: REV

in Ω' mit $M_{\Omega'}(t)$ und die Gesamtmasse des Salzwassers in Ω' mit $M^B_{\Omega'}(t)$ und erhalten

$$M_{\Omega'}(t) = \int_{\Omega'} \Phi\varrho(C(x,y,z,t))dxdydz$$
$$M^B_{\Omega'}(t) = \int_{\Omega'} \Phi\varrho(C(x,y,z,t)) \cdot C(x,y,z,t)dxdzdz$$

Wir stellen nun Gleichungen für den Massenerhalt auf.

I. Massenerhaltung für die Mischung in Ω

Es gilt

$$\frac{\partial(\Phi\varrho(C))}{\partial t} + \nabla \cdot j = 0$$

Dabei bedeutet $M_{\Omega'}(t+\Delta t) - M_{\Omega'}(t)$ die Zu- und Abflüsse über $\partial\Omega'$ im Zeitintervall $[t, t+\Delta t]$. Das j ist der sogenannte Massenfluß:

$$j = \varrho(C) \cdot v\ \frac{kg}{m^3}\frac{m}{s} = \varrho(C) \cdot v\ \frac{kg}{m^2 s}$$

j gibt also an, wieviel Kilogramm in einer Sekunde über einen Quadratmeter fließen. Weiter ist

$$v = -\frac{k}{\mu}(\nabla p - \varrho(C) \cdot \vec{g}) \quad \text{(Darcy-Gesetz)}$$

Dabei sind k die absolute Permeabilität des Mediums, μ die dynamische Viskosität der Flüssigkeit, $\vec{g} = (0,0,g)^T$ die Erdanziehungskraft mit der Gravitationskonstanten g und p der Druck der Flüssigkeit. Die Geschwindigkeit v ist proportional zu $-\nabla p$, es findet also ein Fluß in Richtung des größten Druckgefälles statt. Der Druck **p(x,y,z,t)** ist eine weitere Unbekannte.

Als Randbedingung haben wir

$$j \cdot n = 0 \text{ auf } \partial\Omega$$

Dabei ist n die äußere Normale an Ω. Der Rand unsers Würfels ist also undurchlässig nach außen und innen.

Eine Anfangsbedingung haben wir nicht, da keine zeitliche Ableitung nach p vorkommt.

II. Massenerhaltung für das Salzwasser

Wir haben

$$\frac{\partial(\Phi\varrho(C)\cdot C)}{\partial t}+\nabla\cdot j_B=0$$

Dabei ist

$$j_B=j\cdot C-\varrho(C)D_e(v)\nabla C$$

∇C zeigt einen Fluß in Richtung des höchsten Konzentrationsgefälles an. In $D_e(v)$ sind molekulare Diffusion und hydrodynamische Dispersion enthalten.

Wir haben die Randbedingung

$$j_b\cdot n=0 \text{ auf } \partial\Omega$$

Der Würfel ist also wieder nach außen isoliert. Zusätzlich haben wir die Anfangsbedingung

$$C(x,y,z,o)=C_0(x,y,z)$$

Die ist uns ja gegeben (mit Berücksichtigung der eingebauten Delle, um die Inhomogenitäten in der Grenzschicht zu erhalten).

Zusammenfassung

Zusammengefasst erhalten wir das folgende dreidimensionale, instationäre, gekoppelte System zweier nichtlinearer partieller Differentialgleichungen:

$$\frac{\partial(\Phi\varrho(C))}{\partial t}-\nabla\cdot\left\{\varrho(C)\frac{k}{\mu}(\nabla p-\varrho(C)\vec{g})\right\}=0$$

$$\frac{\partial(\Phi\varrho(C)C)}{\partial t}-\nabla\cdot\left\{C\varrho(C)\frac{k}{\mu}(\nabla p-\varrho(C)\vec{G})+\varrho(C)D_e(v)\nabla C\right\}=0$$

Vereinfachung

Zur Vereinfachung wählen wir die sogenannte Bussinesq Approximation: Wir behalten die Abhängigkeit der Dichte von der Konzentration nur im Darcy-Gesetz bei. In den anderen Fällen ersetzen wir $\varrho(C)$ durch ϱ_0.

Zusätzlich setzen wir $D_e(v)=D\cdot I$ (unabhängig von v).

Das führt auf eine elliptische partielle Differentialgleichung für p und eine parabolische partielle Differentialgleichung für C. Letztere ist allerdings singulär gestört, da der konvektive Fluß $j\cdot C$ dominiert. Daher verhält sie sich wie eine hyperbolische PDGL.

$$\nabla\cdot j=0 \quad \text{in } \Omega \tag{I'}$$

$$j=-\varrho_0\frac{k}{\mu}(\nabla p-\varrho(C)\vec{g})$$

$$j\cdot n=0 \quad \text{auf } \partial\Omega$$

$$\frac{\partial(\Phi\varrho_0 C)}{\partial t}+\nabla\cdot j_B=0 \quad \text{in } \Omega \tag{II'}$$

$$j_B=\underbrace{j\cdot C}_{\substack{\text{nichtlinear}\\ \text{in C}}}-\underbrace{\varrho_0 D\nabla C}_{\substack{\text{linear}\\ \text{in C}}}$$

$$j_B\cdot n=0 \quad \text{auf } \partial\Omega$$

$$C(x,y,z,0)=C_0(x,y,z)$$

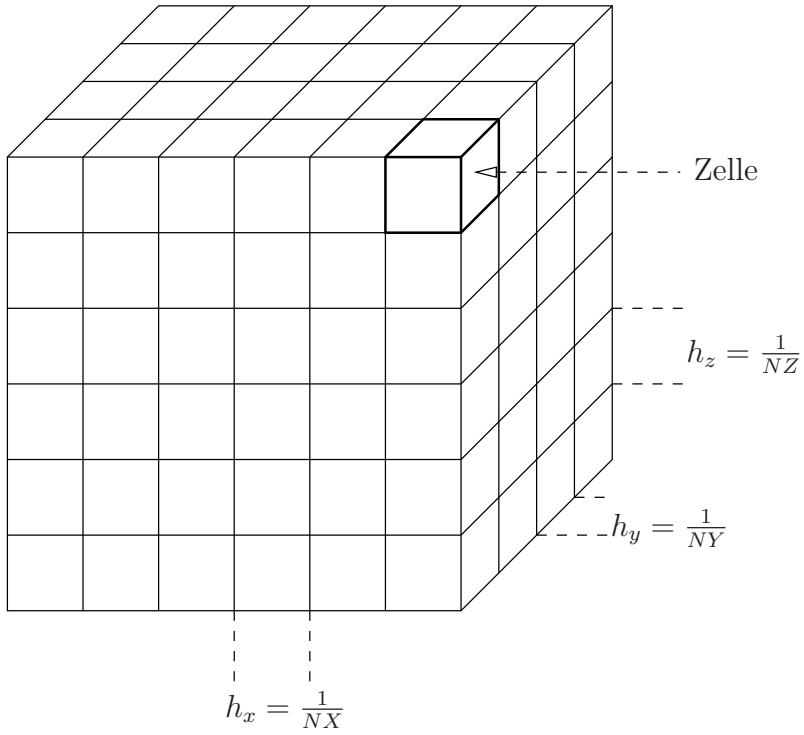


Abbildung B.5: In Zellen aufgeteilter Würfel

B.3 Diskretisierung

Wir verwenden ein zellzentriertes Finite-Volumen-Verfahren. Wir teilen den Würfel in Zellen auf, die wir durch den (C,p)-Wert in der Zellmitte repräsentieren.

Eine Zelle hat die Kantenlängen $h_x = \frac{1}{NX}$, $h_y = \frac{1}{NY}$ und $h_z = \frac{1}{NZ}$, wobei NX, NY und NZ die Anzahl der Zellen im Würfel in Richtung der entsprechenden Koordinate sind. Der Zellmittelpunkt einer Zelle liegt demnach bei $((i + \frac{1}{2})h_x, (j + \frac{1}{2})h_y, (k + \frac{1}{2})h_z)$. Die Wände der Zelle werden mit W, E („vorderer "und „hinterer "Nachbar in x-Richtung), S, N („vorderer "und „hinterer "Nachbar in y-Richtung) und B, T („vorderer "und „hinterer "Nachbar in z-Richtung) bezeichnet.

Im folgenden bezeichnen wir mit $\Omega_{i,j,k}$ die Zelle, die den Gitterpunkt (i, j, k) enthält.

I. Strömungsgleichung

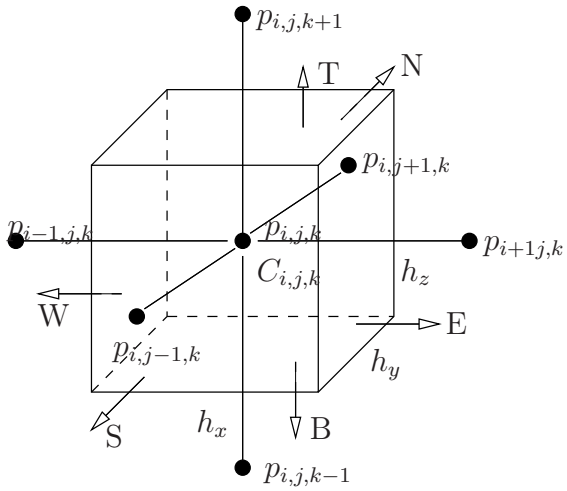


Abbildung B.6: Einzelne Zelle

$$\int_{\Omega_{i,j,k}} \nabla \cdot j \; dx \overset{\text{Satz von Gauß}}{=} \int_{\partial\Omega_{i,j,k}} j \cdot n \; ds = \int_{\partial\Omega_{i,j,k}} \left\{ -\varrho_0 \frac{k}{\mu} (\nabla p - \varrho(C)\vec{g}) \right\} \cdot n \; ds$$

$$\approx \left[-\varrho_0 \frac{k}{\mu} \overbrace{\left(\frac{p_{i+1,j,k} - p_{i,j,k}}{h_x} \right)}^{E} + \varrho_0 \frac{k}{\mu} \overbrace{\left(\frac{p_{i,j,k} - p_{i-1,j,k}}{h_x} \right)}^{W} \right] h_y h_z$$

$$+ \left[-\varrho_0 \frac{k}{\mu} \overbrace{\left(\frac{p_{i,j+1,k} - p_{i,j,k}}{h_y} \right)}^{N} + \varrho_0 \frac{k}{\mu} \overbrace{\left(\frac{p_{i,j,k} - p_{i,j-1,k}}{h_y} \right)}^{S} \right] h_x h_z$$

$$+ \left[-\varrho_0 \frac{k}{\mu} \overbrace{\left(\frac{p_{i,j,k+1} - p_{i,j,k}}{h_z} + \frac{\varrho(C_{i,j,k+1}) + \varrho(C_{i,j,k})}{2} g \right)}^{T} \right.$$

$$\left. + \varrho_0 \frac{k}{\mu} \overbrace{\left(\frac{p_{i,j,k} - p_{i,j,k-1}}{h_z} + \frac{\varrho(C_{i,j,k}) + \varrho(C_{i,j,k-1})}{2} \right)}^{B} \right] h_x h_y$$

$$= 0$$

Für die Randzellen fällt der entsprechende Term für die jeweilige Wand weg (Randbedingung).

Die obige Formel wird noch mit $\frac{1}{\varrho_0 \frac{k}{\mu}}$ und $\frac{1}{h_x h_y h_z}$ multipliziert. Daraus erhalten wir unter

Verwendung des SOR-Verfahrens die folgende Differenzenformel (für eine innere Zelle):

$$p_{i,j,k}^m = \omega \cdot \left(\frac{1}{2} \left(\frac{1}{h_x^2} + \frac{1}{h_y^2} + \frac{1}{h_z^2} \right) \left[\frac{p_{i+1,j,k}^{m-1} + p_{i-1,j,k}^m}{h_x^2} + \frac{p_{i,j+1,k}^{m-1} + p_{i,j-1,k}^m}{h_y^2} \right. \right.$$
$$\left. \left. + \frac{p_{i,j,k+1}^{m-1} + p_{i,j,k-1}^m}{h_z^2} + g \frac{\varrho(C_{i,j,k+1}) - \varrho(C_{i,j,k-1})}{2h_z} \right] \right)$$
$$+ (1-\omega) \cdot p_{i,j,k}^{m-1}$$

Da die Matrix den Eigenwert Null zum Eigenvektor $(1, \ldots, 1)^T$ hat, muß zusätzlich noch die Bedingung

$$\sum_{i,j,k} p_{i,j,k} = 0$$

gestellt werden.

Lokale Massenerhaltung: Die Flüsse werden von beiden Seiten identisch berechnet.

II. Transportgleichung

Wir teilen die Zeitachse in gleichlange Zeitintervalle $\mathcal{T}^n := [t^{n-1}, t^n]$ auf.

Bemerkungen zur Rechnung: Wir definieren die konvektiven Flüsse $F(C) := Cj \cdot n^*$, wobei n^* die Normale in positiver x,y,z-Richtung bedeutet, also nur für die „hinteren "Begrenzungsflächen der Zelle die äußere Normale ist. Die F^n werden direkt aus den C^n berechnet. Alle hochgestellten Indizees bedeuten hier einen Zeitindizee und *keinen* Iterationsschritt!

$$\int_{\Omega_{i,j,k}} \int_{\mathcal{T}^n} \frac{\partial(\Phi \varrho_0 C)}{\partial t} \, dt \, dx + \int_{\mathcal{T}^n} \int_{\Omega_{i,j,k}} \nabla \cdot \{ jC - \varrho_0 D \nabla C \} \, dx \, dt$$
$$= \int_{\Omega_{i,j,k}} [\Phi \varrho_0 C]_{t^{n-1}}^{t^n} \, dx + \int_{\mathcal{T}^n} \int_{\partial\Omega_{i,j,k}} Cj \cdot n - \varrho_0 (D \nabla C) \cdot n \, ds \, dt$$
$$\approx \overbrace{\Phi \varrho_0 (C_{i,j,k}^n - C_{i,j,k}^{n-1}) h_x h_y h_z}^{\text{Mittelpunktsregel}}$$
$$+ \Delta t \cdot \left\{ \left(F_{i+\frac{1}{2},j,k}^{n-1} - F_{i-\frac{1}{2},j,k}^{n-1} \right) h_y h_z + \left(F_{i,j+\frac{1}{2},k}^{n-1} - F_{i,j-\frac{1}{2},k}^{n-1} \right) h_x h_z \right.$$
$$\left. + \left(F_{i,j,k+\frac{1}{2}}^{n-1} - F_{i,j,k-\frac{1}{2}}^{n-1} \right) h_x h_y \right\}$$
$$- \varrho_0 \Delta t D \cdot \left\{ \left(\frac{C_{i+1,j,k}^n - C_{i,j,k}^n}{h_x} - \frac{C_{i,j,k}^n - C_{i-1,j,k}^n}{h_x} \right) h_y h_z \right.$$
$$+ \left(\frac{C_{i,j+1,k}^n - C_{i,j,k}^n}{h_y} - \frac{C_{i,j,k}^n - C_{i,j-1,k}^n}{h_y} \right) h_x h_z$$
$$\left. + \left(\frac{C_{i,j,k+1}^n - C_{i,j,k}^n}{h_z} - \frac{C_{i,j,k}^n - C_{i,j,k-1}^n}{h_z} \right) h_x h_y \right\} = 0$$

Durch das Auflösen nach $C_{i,j,k}^n$ erhält man die Differenzenformel. Die $C_{i,j,k}^n$ müssen aber alle auf derselben Zeitebene berechnet werden. Das macht jeweils das Lösen eines Gleichungssystems erforderlich. Die Werte sind also nur implizit berechenbar.

Berechnung der konvektiven Flüsse

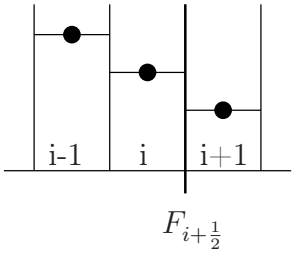


Abbildung B.7: Godunov-Verfahren 1.Ordnung

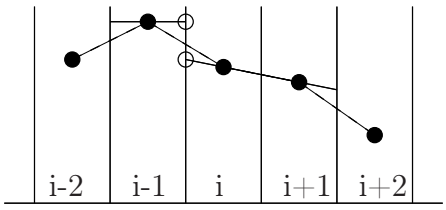


Abbildung B.8: Godunov-Verfahren 2.Ordnung

Wir argumentieren im folgenden eindimensional. Die C_i werden ja als Zellmittelwert, der in der ganzen Zelle konstant ist, interpretiert (zellzentriertes Finite-Volumen-Verfahren). Der Fluß wird immer von der Zelle mit höherer Konzentration in die mit geringerer Konzentration stattfinden. Daher wollen wir als Konzentrationswert an der Zellgrenze den der Zelle mit der höheren Konzentration wählen. Das wird auch als *upwind* bezeichnet.

In der Berechnung von $F_{i+\frac{1}{2}}$ benutze also

$$\begin{cases} C_i & \text{falls} j \cdot n \geq 0 \text{ (Fluss in positiver x-Richtung)} \\ C_{i+1} & \text{falls} j \cdot n < 0 \text{ (Fluss in negativer x-Richtung)} \end{cases}$$

Analog wird die Berechnung für die linke Zellwand durchgeführt. Da j aber in unserem Fall von der Konzentration abhängig ist, die an den Zellwänden Unstetigkeiten aufweist, ist die Berechnung von $s := j \cdot n$ nicht so einfach. Wir benötigen die sogenannte Rankine-Hugoniot-Bedingung , die uns die Schockgeschwindigkeit

$$s = \frac{C^R j(C^R) \cdot n - C^L j(C^L) \cdot n}{C^R - C^L} \geq 0$$

liefert. Dieses Verfahren heißt Godunov-Verfahren erster Ordnung . Es bietet eine Genauigkeit von $\mathcal{O}(h)$. Das Verfahren hat allerdings ein starkes Verschmieren der Konzentration zur Folge, die auch als numerische Diffusion bezeichnet wird. Diese ist unerwünscht, daher geht man über auf die Godunov-Verfahren zweiter Ordnung :

Man verwendet eine stückweise lineare Rekonstruktion, d.h. man nimmt die Konzentration nicht mehr als konstant in der ganzen Zelle an, sondern als linear. Dazu bestimmt man die Steigungen zwischen den Konzentrationswerten benachbarter Zellen. Für jede Zelle hat man dann zwei Möglichkeiten, die Steigung innerhalb der Zelle zu wählen: Entweder als Steigung zu linken oder zur rechten Nachbarzelle. Man wählt die (betragsmäßig) kleinere aus. Falls man in einer Zelle ein lokales Minimum oder Maximum vorliegen hat, wählt man in dieser Zelle die Steigung Null. An den Zellgrenzen kann es dann wieder zu Unstetigkeiten kommen, die

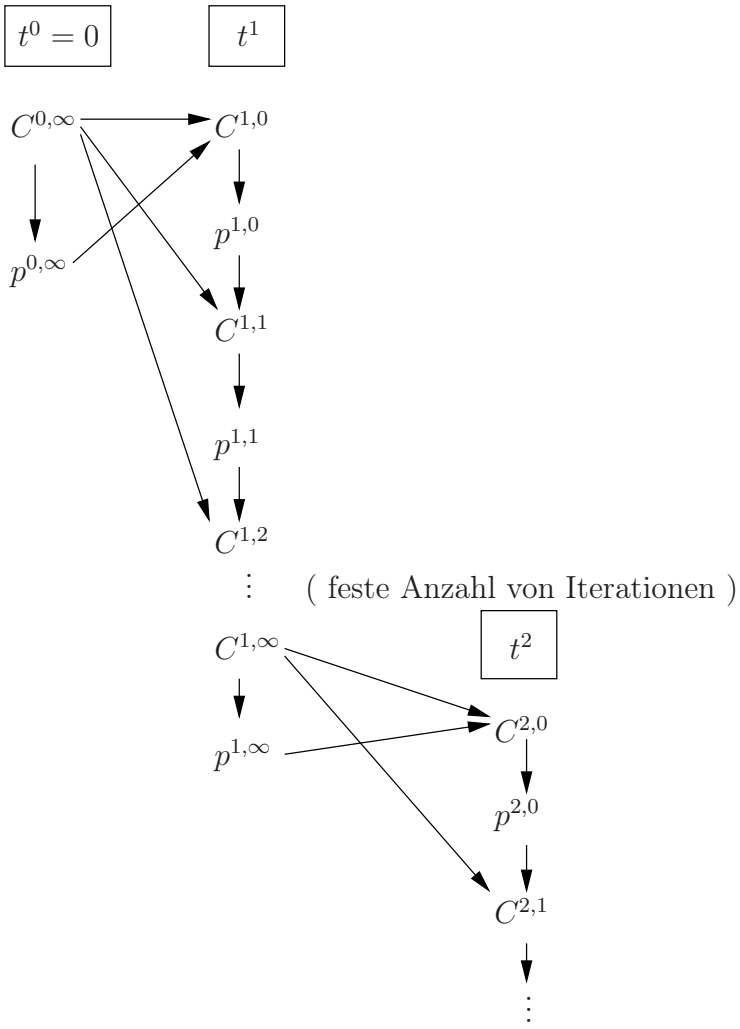


Abbildung B.9: Gesamtalgorithmus

wie oben beschrieben gehandhabt werden. Wichtig ist zu beachten, daß zur Berechnung der konvektiven Flüsse sogar die zweiten Nachbarn der eigenen Zelle benötigt werden, die ja in den Nachbarzellen jeweils zur linearen Rekonstruktion benötigt werden. Die Flüsse $F_{i\pm\frac{1}{2}}$ hängen also von $C_{i-2}, \ldots, C_{i+2}$ ab!

Gesamtalgorithmus

Das Zeitschrittverfahren und die Fixpunktiteration werden gekoppelt. Der Druck p (. , t) hängt nur von C (. , t) ab !

Auf Zeitstufe Null wird mittels der gegebenen Anfangskonzentration der Anfangsdruck berechnet (iteratives Auflösen des entsprechenden Gleichungssystems). Mit Hilfe dieser werden die Konzentrationswerte auf Zeitstufe Eins berechnet und damit die Druckwerte auf Zeitstufe Eins. Diese sind allerdings noch in der nullten Iteration. Also wird nun iteriert. Nach einer gewissen Iterationstiefe wird dann auf die zweite Zeitstufe gewechselt. So fährt man fort.

Jede Berechnung von $C^{n,r}$ bzw. $p^{n,r}$ erfordert das Lösen eines Gleichungssystems. Die werden allerdings nicht sehr genau gelöst.

B.4 Parallelisierung

Datenaufteilung

Große Buchstaben kennzeichnen grundsätzlich globale, kleine Buchstaben lokale Koordinaten.

Wir benutzen ein Feld von

$$PX \times PY \times PZ \text{ Prozessoren}$$

für ein Feld von

$$NX \times NY \times NZ \text{ Zellen.}$$

Das bedeutet, daß jeder Prozessor auf

$$\underbrace{\frac{NX}{PX}}_{=lx} \times \underbrace{\frac{NY}{PY}}_{=ly} \times \underbrace{\frac{NZ}{PZ}}_{=lz}$$

Zellen arbeitet. Jeder speichert (allokiert) aber

$$\left(\frac{NX}{PX}+4\right) \times \left(\frac{NY}{PY}+4\right) \times \left(\frac{NZ}{PZ}+4\right)$$

Zellen, da maximal Informationen über den zweiten Nachbarn zur Berechnung notwendig sind (bei der Berechnung der konvektiven Flüsse).

Betrachte den Prozessor

$$px \in 0 \ldots PX-1,\ py \in 0 \ldots PY-1,\ pz \in 0 \ldots PZ-1$$

Dieser ist verantwortlich für die Zellen

$$[\underbrace{px \cdot lx}_{OX}, (px+1) \cdot lx - 1] \times [\underbrace{py \cdot ly}_{OY}, (py+1) \cdot ly - 1] \times [\underbrace{pz \cdot lz}_{OZ}, (pz+1) \cdot lz - 1]$$

Betrachten wir dazu ein Beispiel in einer Dimension: Sei $NX = 12$, $PX = 3$, also $lx = \frac{NX}{PX} = 4$.Siehe Abb. B.10.

Bei der Modifikation eines sequentiellen Programmes erhalten alle Funktionen die Werte (ox, oy, oz), (OX, OY, OZ), (lx, ly, lz) als Parameter und können dann in der parallelen Version verwendet werden. Zusätzlich sind nur Kommunikationen einzufügen.

Datenaustausch

Die einzelnen Prozessoren müssen im Überhang die Werte der Nachbarzellen speichern, um ihre eigenen Werte berechnen zu können. Die Werte der Nachbarzellen müssen also gegenseitig ausgetauscht werden. Für die Druckwerte und die Konzentrationswerte wird ein Überlapp von einer Zelle, für die alten Konzentrationswerte ein Überlapp von zwei Zellen benötigt.

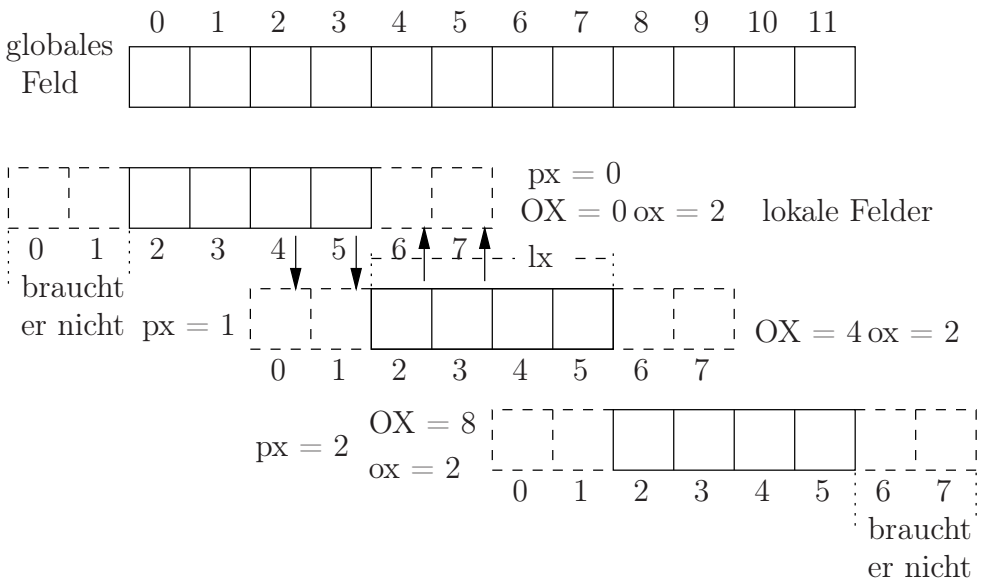


Abbildung B.10: Datenaufteilung und Austausch in einer Dimension

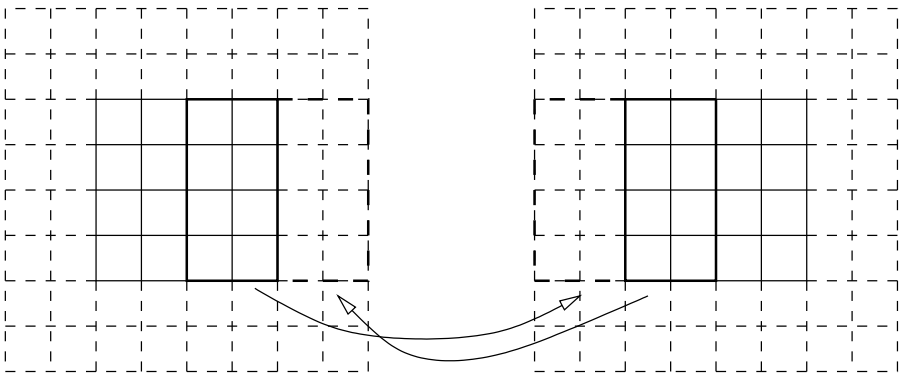


Abbildung B.11: Datenaustausch an einer Grenze in zwei Dimensionen

Literatur

J. Goodman A. S. Tanenbaum (1999). *Computerarchitektur*. Prentice Hall.

G. M. Amdahl (1967). Validity of the single-processor approach to achievieng large scale computing capabilities. In: *AFIPS Conference Proceedings*, Band 30, S. 483–485.

Gregory R. Andrews (1991). *Concurrent programming: principles and practice*. The Benjamin/Cummings Publishing Company. ISBN 0-8053-0086-4.

Barnes und Hut (1986). A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, **324**: 446–449.

R. Buyya (1999). *High Performance Cluster Computing*. Prentice Hall.

D. E. Culler, J. P. Singh und A. Gupta (1999). *Parallel Computer Architecture*. Morgan Kaufmann.

M. J. Flynn (1972). Some computer organizations and their effectiveness. *IEEE Trans. Computers*, **21(9)**: 948–960.

L.F. Greengard (1987). *The Rapid Evaluation of Potential Fields in Particle Systems*. Dissertation, Yale University.

J. L. Gustafson (1988). Reevaluating amdahl's law. *CACM*, **31**(5): 532–533.

J. L. Hennessy und D. A. Patterson (1996). *Computer Architecture – A quantitative approach*. Morgan Kaufmann.

G.S. Winckelmans J.K. Salmon, M.S. Warren (1994). Fast parallel treecodes for gravitational and fluid dynamical n-body systems. *Int. J. Supercomputer Application*, **8**: 129–142.

V. Kumar, A. Grama, A. Gupta und G. Karypis (1994). *Introduction to Parallel Computing*. Benjamin/Cummings.

Leslie Lamport (1978). Laber süelz blubb. *CACM*, **21(7)**: 558–564.

G. Rinaldi M. Jünger, G. Reinelt (1997). The Travelling Salesman Problem: A Bibliography. In: F. Maffioli M. Dell'Amico (Herausgeber), *Annotated Bibliography in Combinatorial Optimization*, S. 199–221. Wiley.

B. Nichols, D. Buttlar und J. Farell (1996). *Pthreads Programming*. O'Reilly.

Sedgewick (1992). *Algorithmen in C++*. Addison–Wesley.

W. R. Stevens (1999). *UNIX Network Programming*. Prentice Hall.

Eric F. van de Velde (1993). *Concurrent Scientific Computing*. Springer Verlag. ISBN 3-540-94195-9.