Vorlesungsskript

Technik des Wissenschaftlichen Rechnens

TU München
TU Chemnitz

Wintersemester 93/94
Überarbeitung: WS 94/95

Ulrich Rüde

2. Juni 1995

Kapitel 1

Einführung

1.1 Was ist Scientific Computing (Versuch einer Definition)

In den letzen Jahren hat sich eine neue, eigenständige Disziplin, das *Scientific Computing* etabliert. Im Deutschen wird für das Gebiet der — leider etwas unglückliche — Begriff *Wisssenschaftliches Rechnen* verwendet.

Scientifc Computing läßt als Synthese seiner Bestandteile definieren:

$$\boxed{\text{Scientific Computing} = \left\{ \begin{array}{c} \text{Mathematik} \\ + \\ \text{Informatik} \\ + \\ \text{Anwendung} \end{array} \right.}$$

Eine alternative, zweckorientierte Definition könnte etwa sein:

Scientific Computing = Rechnergestützte Simulation technischer, naturwissenschaftlicher oder sozialwissenschaftlicher Systeme mit Hilfe mathematischer Methoden

1.2 Zur Bedeutung des Scientific Computing

Für viele wissenschaftliche und technische Fachrichtungen entwickeln sich *rechnergestützte Simulationen* zu einem wichtigen neuen Hilfsmittel. Sie sind Alternativen zum Bau teurer Prototypen und Versuchsanlagen, die nicht nur Kosten sparen, sondern auch den Entwicklungszyklus verkürzen, und damit zur Konkurrenzfähigkeit der Wirtschaft beitragen. In anderen Fällen müssen Simulationen das Experiment ganz ersetzen, wenn man man z.B. an die Simulation des Zusammenstoßes zweier Galaxien oder die Veränderungen des Weltklimas denkt. Aktuelle und zukünftige Superrechner ermöglichen vielleicht sogar die Lösung der sogenannten *grand challenge* Probleme, von denen jedes einen entscheidenden wissenschaftlichen Durchbruch markiert. Diese Simulationen sind die Grundaufgabe des *Scientific Computing.*

Neben den Herausforderungen an der Grenze des überhaupt Machbaren, sollte man auch an die Bedeutung des Scientific Computing für die Lösung von Alltagsaufgaben denken. Methoden des Scientific Computing können auch helfen, strom- und wassersparende Waschmaschinen zu konstruieren, die Heizanlagen von Wohnungen zu steueren, die Energiesparwirkungen von Dämmmaßnahmen abzuschätzen, den optimalen Düngemitteleinsatz zu finden, Schadstoffausbreitungen vorherzusagen, und vieles mehr. Das Ziel solcher Aufgabenstellungen ist es, die *Qualität* eines Produktes oder Prozesses zu verbessern, so daß die Wirtschaftlichkeit erhöht wird und gleichzeitig oft auch der Umwelt entlastet wird. Die Bedeutung dieser breiten Einsatzmöglichkeiten des Scientific Computing kann heute noch gar nicht abgeschätzt werden. Rechnergestützte Simulationen sollten deshalb nicht nur als Werkzeug für wenige Spezialisten gesehen werden, sondern sollten in allen wissenschaftlichen Disziplinen als Hilfsmittel verwendet werden.

Die zentrale ökonomische (und ökologische) Bedeutung der numerischen Simulation liegt darin, daß sie potentiell die Effizienz fast aller technischer Geräte oder Prozesse verbessern kann, den Luftwiderstand eines Verkehrsflugzeuges genauso, wie den Materialaufwand bei der Herstellung eines Joghurtbechers.

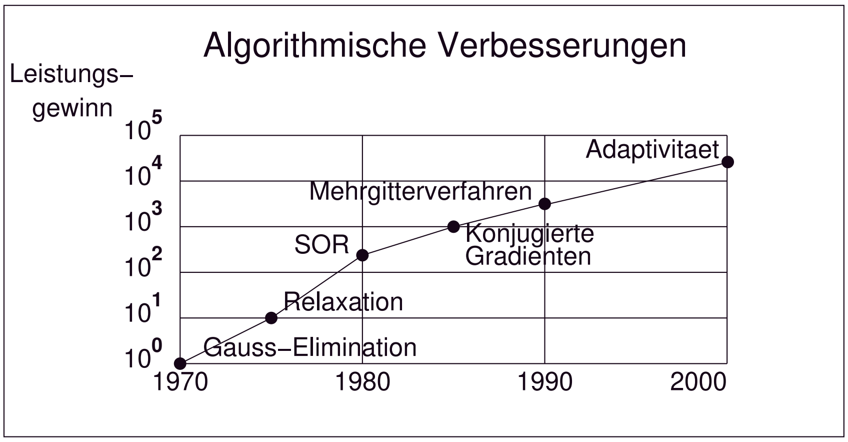
Der rasante Fortschritt der Computertechnik (Abb. 1.1) hat dazu geführt, daß eine vor zwanzig Jahren noch fast unvorstellbare Rechenleistung heute bereits in normalen Workstations verfügbar ist. Genauso spektakulär wie die Entwicklung der Computertechnik, wenn auch weniger öffentlichkeitswirksam, sind Fortschritte, die aus verbesserten mathematischen Methoden resultieren (Abb. 1.1). Wenn es nur nach den Bedeutungen der wissenschaftlichen Beiträge ginge, könnte der Computertomograph auch Mathematomograph heißen, da er ohne leistungsfähige mathematischen Algorithmen undenkbar wäre.

Bausteine des Scientific Computing

Der erfolgreiche Einsatz von Simulationen bedingt die eng verflochtene Synthese dreier Gebiete, Anwendung, Informatik und Mathematik, und führt zu einer neuen, interdisziplinäre Fachrichtung, dem *Wissenschaftlichen Rechnen* oder *Scientific Computing.* Sie besteht aus mehr als nur einem unkoordinierten Nebeneinander von Methoden aus den Teilgebieten, und führt zu einer substantiell neuen Arbeitsweise (siehe Abb. 1.2).

Häufig müssen die vorhandenen mathematischen Modelle erst gründlich überarbeitet werden, bevor sie für die numerische Simulation geeignet sind. Deshalb ist in Scientific Computing Projekten fast immer eine Analysephase erforderlich, in der die Anwender, Mathematiker und Informatiker intensiv zusammenarbeiten. Oft ist bereits diese Analyse und das daraus resultierende verbesserte Verständnis des Systems der erste wesentliche Fortschritt.

Konventionelle Methoden der mathematischen Analysis müssen oft signifikant modifiziert werden, wenn sie in leistungsfähige Simulationsprogramme umgesetzt werden sollen. In diesem Bereich ist von der Numerik zwar bereits viel Vorarbeit geleistet worden, aber trotzdem ist die Lücke zwischen den Anforderungen komplexer Anwendungen und den verfügbaren Algorithmen und Programmpaketen noch riesig. Neben der Neuentwicklung von Algorithmen wird



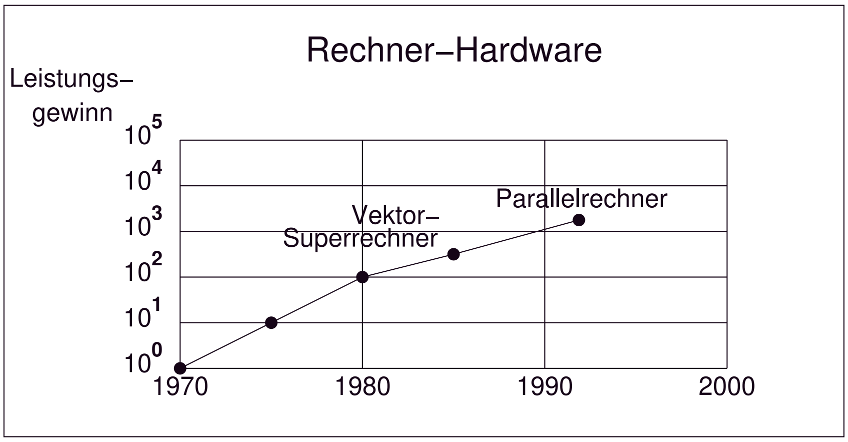


Abbildung 1.1: Fortschritte im Scientific Computing (aus einer Broschüre der US Nat'l Science Foundation zum Förderprogramm im Bereich *High Performance Computing and Networking*)

es immer wichtiger, vorhandene Einzelbausteine zu nutzen und effizient zusammenzusetzen. Der Trend zu Parallelrechnern steigert die Komplexität zusätzlich.

Die Numerik ihrerseits baut auf der Analysis, (linearen) Algebra, Funktionalanalysis und Approximationstheorie auf, aber je nach Anwendung kann aber auch fast jede andere mathematische Teildisziplin relevant werden. Als Beispiele seien die Modelle der statistischen Physik aufgeführt, die natürlich auf der Kenntnis der Wahrscheinlichkeitstheorie beruhen, oder Probleme der diskreten Optimierung, die Verbindungen zur Graphentheorie oder Topologie haben können.

Die Gebiete *Rechnerarchitektur* und *Programmiertechniken* sind naheliegende Beiträge der Informatik zum Scientific Computing. Da die Komplexität der (Simulations-) Programme wächst, bekommt auch das systematische *Software Engineering* eine große Bedeutung. Das gilt natürlich besonders, wenn die Programme nicht reine Forschungswerkzeuge sein sollen, sondern auch im industriellen Alltag angewendet werden sollen, sei es als Werkzeug des Entwick-

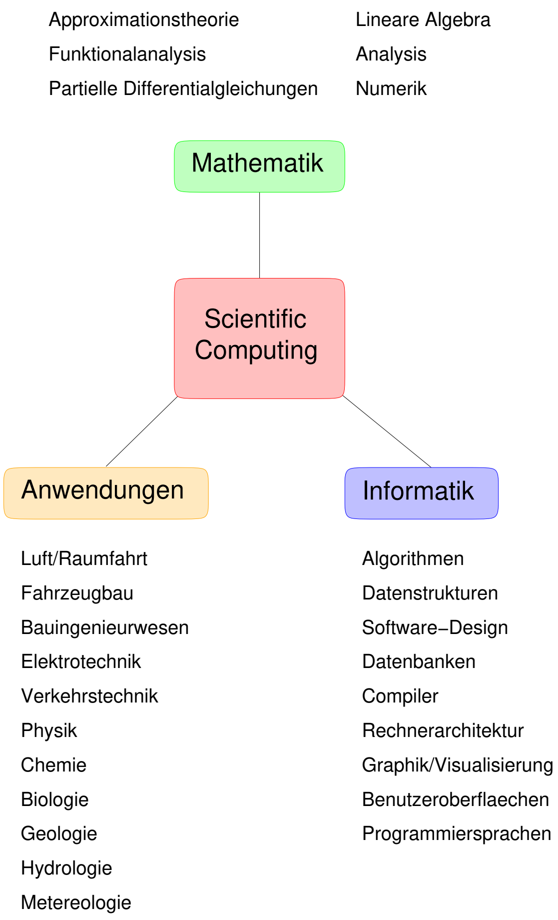


Abbildung 1.2: Scientific Computing als Querschnittswissenschaft

lungsingenieurs oder sogar als integrierte Steuerung z.B. eines Kernreaktors. Komplexe Anwendungen auf Parallelrechnern erzwingen auch den Einsatz moderner Programmierhilfsmittel. Das Scientfic Computing erfordert besondere Programmiersprachen und dazugehörige Übersetzer. Dabei sind Optimierungstechniken von großer Bedeutung, einschließlich Methoden zur automatischen oder semiautomatischen Parallelisierung durch Programmtransformationen.

Im folgenden seien noch einige Gebiete aufgeführt, die eine allgemeine Bedeutung für das Scientific Computing haben. Dies ist zuerst die *Visualisierung*, die für das Scientific Computing zu einem unentbehrlichen Hilfsmittel geworden ist, um die Ergebnisse von (Simulations-) Experimenten zu veranschaulichen und zu bewerten. Darüber hinaus ist auch die Frage nach *Benutzerschnittstellen* bedeutsam.

Von wachsender Bedeutung im Scientific Computing ist auch die *Computeralgebra*, mit deren Hilfe auch symbolische Rechnungen durchgeführt werden können. Es gibt erste Ansätze zu kombiniert symbolisch-numerischen Algorithmen, die eine ganz neue Qualität haben, und deren langfristige Bedeutung noch nicht absehbar ist.

Rechnernetze ermöglichen den Zugang zu entfernten Hochleistungsrechenern und Programmbibliotheken, sie ermöglichen auch die Verwendung verteilter Daten z.B. für die Wettervorhersage oder Simulation von umweltrelevanten Vorgängen. Bei entsprechechendem Ausbau der Netzkapazität können die Simulationen selbst global verteilt werden. Bei der Wettervorhersage ist es z.B. denkbar, daß jede Wetterstation die Simulation für ihr Einzugsgebiet selbst rechnet, und dazu natürlich in kontinuierlichem Datenaustausch mit benachbarten Stationen stehen muß. Die Gesamtwettervorhersage ergäbe sich dann aus dem Zusammenwirken aller Einzelstationen.

1.3 Weitere Bemerkungen zum Thema Scientific Computing

Schlagwörter im Zusammenhang mit Scientific Computing

- HPSC = High Performance Super Computing
- HPCN = High Performance Computing and Networking
- *Grand Challenges* = Liste von Problemen, die man hofft, durch weiteren Fortschritt im SC lösen zu können, z.B.:
  - Klimamodellierung
  - Turbulenz
  - Halbleiter
  - Analyse des menschlichen Genoms

Zusammenfassung

Scientific Computing ist einer der Schlüssel zum Fortschritt in immer mehr Bereichen der Wissenschaft und hat ein noch gar nicht abschätzbares Potential für die Lösung von Problemen im größten — aber auch im kleinen Maßstab.

1.4 Inhalt der Vorlesung (vorläufige Planung)

1. Beispiele und typische Algorithmen
   - *Eigen-Spiele* (Biologie)
   - DIfferentialgleichungen und Differenzengleichungen
   - Partielle Differentialgleichungen (Diffusion, Wärmeleitung)
     - Relaxationsverfahren
     - Direkte Verfahren, Bandlöser, Nested Dissection
     - Mehrgitterverfahren
   - Kombinatorische Optimierung (*Travelling Salesman*)
   - Chip-Layoutoptimierung

2. Rechnerarchitektur

    - Stand der Technik
    - VHSIC vs. VLSI
    - Mips, Flops, Speicher (Zugriffszeiten),
    - CISC/RISC, Von-Neumann-Rechner
    - VLIW-superskalar, Pipelining, Vektorrechner (Cray)
    - Speicherhierarchie, Register, Caches, Virtueller Speicher
    - Parallelrechner, SIMD/MIMD
    - gemeinsamer/verteilter Speicher, virtuell gemeinsamer Speicher, Verbindungstopologie
    - Spezielle Architekturen (geplant DEC-Alpha und CRAY T3D)

3. Datenstrukturen und Algorithmen der linearen Algebra

    - Voll besetzte Matrizen
      Datenstruktur — dynamische Speicherallokation
      Blas (Basic Linear Algebra Subprograms)
      Bibliotheken LAPACK, LINPACK, Numerical Recipes
      Programmoptimierung am Beispiel der Matrix-Multiplikation
    - Dünn besetzte Matrizen
      Datenstrukturen und Algorithmen
    - Vektorsprachen
      Fortran 90, Clam, etc.

4. Programmiertechnik

    - Programmierparadigmen
      Funktionale, prozedurale, objektorientierte Programmierung
    - Programmiersprachen (Fortran 77, Fortran 90, C, C++)
    - Programmierwerkzeuge
    - Parallele Programmierung
        - Datenparallele Programmierung mit High Performance Fortran (HPF)
        - Communicating Sequential Processes CSP, Message Passing Systeme (PVM, MPI)
        - Linda
        - Datenfluß-Parallelität

5. Visualisierung

    - 2-D
    - 3-D
    - Animation
    - Datenaufbereitung, Datenanlyse

Kapitel 2

Beispiele für Scientific Computing

2.1 Eigen-Spiele

Der erste Teil dieses Abschnittes basiert auf dem Buch *Das Spiel* von Manfred Eigen und Ruthild Winkler [8]. Die Ideen sind auch verwandt zu den sogenannten zellulären Automaten[1].

2.1.1 Modellbildung

Wir untersuchen ein einfaches Ökosystem:

1. Es wächst Gras
2. Hasen fressen Gras, die Zahl der Hasen nimmt zu
3. Füchse fressen Hasen, Zahl der Füchse nimmt zu
4. Füchse werden gejagt

Als erstes modellieren wir einen ökologisch abgeschlossenen Raum durch ein quadratisches *Spielfeld* mit $N \times N$ Feldern.

In jedem *Spielzug* wird eines der Felder zufällig (durch Würfeln) bestimmt. Felder können leer sein oder durch einen Spielstein besetzt. Dabei sind die Steine

- Grün = Gras
- Gelb = Hase
- Rot = Fuchs.

Weiter gibt es eine Umwandlungstabelle für erwürfelte Felder:

Nachbarn – Feld	leer	grün	gelb	rot
alle leer	→ grün	—	—	rot → leer
grün	→ grün	—	grün → gelb	rot → leer
gelb	→ grün	grün → gelb	—	rot → leer
rot	→ grün	—	gelb → rot	rot → leer

[1] http://alife.santafe.edu/alife/topics/cas/ca-faq/ca-faq.html

Das Wachstumsverhalten ist also folgendermaßen charakterisiert.

- Gras: nur abhängig von der Zahl der Leerfelder, wir nehmen also an, daß jederzeit genügend Grassamen im Boden vorhanden ist.
- Hasen: abhängig von Gras und Hasen, das Wachsttumsverhalten der Hasenpopulation wird deshalb als *autokatalytisch* bezeichnet.
- Füchse: abhängig von Hasen und Füchsen, ebenfalls autokatalytisch
- Jäger: gehören bereits nicht mehr zum System und werden nicht modelliert.

Hasen und Füchse können aussterben. Die Wahrscheinlichkeit ist zwar umso kleiner, je größer N ist, andererseits wächst die Wahrscheinlichkeit mit der Länge des Spieles beliebig an, so daß das Aussterben der Hasen und Füchse bei sehr langer Spieldauer irgendwann einmal eintreten muß.

Das hier beschriebene Modell kann natürlich "von Hand" gespielt werden. Für größeres N und deshalb auch große Zahl von Spielzügen ist es aber angebracht das System auf dem Rechner zu *simulieren.*

2.1.2 Computersimulation

In diesem Abschnitt wird ein Computerprogramm diskutiert, mit dem das obige einfache Modellökosystem simuliert werden kann.

Wie auch sonst im Text wird zur Formulierung von Algorithmen hauptsächlich C++ verwendet. Zum Lernen von für C++ sei (unter vielen anderen) auf die Bücher von Stroustrup [24] oder F. Bause und W. Tölle [5] hingewiesen.

Das erste Programmfragment zeigt den Kern der Simulation, also die Ausführung eines Spielzuges in dem Unterprogramm `spielzug`. Die Grunddatenstrukturen sind einerseits der *Aufzählungstyp* `enum Spezies` und das Feld `umwelt`.

```
enum Spezies { Leer, Gras, Hase, Fuchs };
Spezies umwelt[N][N];   // muss mit Leer initialisiert werden

spielzug() {
    int i = rand() % N; j = rand() % N;  // '%' ist modulo
    switch(umwelt[i][j]) {
        case Leer:
            umwelt[i][j] = Gras;
            break;
        case Gras:
            if (neigh(i, j, Hase)) umwelt[i][j] = Hase;
            break;
        case Hase:
            if (neigh(i, j, Gras)) "Nachbar wird zu Hase";
            else if (neigh(i, j, Fuchs)) umwelt[i][j] = Fuchs;
            break;
        case Fuchs:
```

```
            if (treffer) umwelt[i][j] = Leer;
            break;
      }
}
```

Als zweites Programmfragment stellen wir noch die Auswertung der momentanen Simulationssituation in einer Routine `auswertung` vor.

```
int zahl[Fuchs];  // muss mit "0" initialisiert werden

auswertung() {
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            zahl[umwelt[i][j]]++;
    print zahl ;  // von Leer..Fuchs
}
```

Interessierte können z.B. versuchen, das Modell um *Heuschrecken* zu erweitern, und dazu Bedingungen zu entwickeln, die zum Auftreten der typischen Heuschreckenschwärme führen.

2.1.3 Auswertung der Simulation

Abb. 2.1 zeigt einen typischen Simulationsverlauf. Die Hasenpopulation (durchgezogene Linie) steigt anfangs wegen der hohen Verfügbarkeit von Nahrung (Gras, gepunktete Linie) stark an. Mit gewisser Verzögerung steigt dann aber auch die Fuchspopulation an, und zusammen mit der reduzierten Menge Gras, führt dies zu einer Trendumkehr bei der Hasenpopulation. Ist die Hasenpopulation dann weit genug reduziert, fängt auch die Fuchspopulation an zu fallen, und da sich nun auch die Grasmenge wieder erholt hat, kann auch die Zahl der Hasen wieder steigen.

Zusammengenommen zeigt das System also ein oszillierendes Verhalten, wobei die drei Populationen phasenversetzt sind. Um in der Visulisierung hochfrequente Oszillationen auszufiltern, zeigt der Plot die *gleitenden Durchschnitte* der Population von jeweils 63 aufeinander folgenden Zeitschritten.

In der derzeitigen Form ist die Simulation *qualitativ*. Das Modell ist viel zu grob, um quantitative Vorhersagen für ein reales Ökosystem zu machen. Dies ist aber auch nicht der Zweck des Modells. Es dient primär dazu, um prinizpielle Zusammenhänge in einem Ökosystemen zu untersuchen. Das Hauptanliegen von Eigen und Winkler [8] war es dabei, zu demonstrieren, daß ein System trotz der Zufälligkeit jeder einzelnen Aktion, doch in der Gesamtheit zu einem deterministischen Verhalten führen kann. Das Oszillieren der Populationen ist noch dazu den einfachen *Spielregeln* nicht von vornherein anzusehen, was natürlich das Modell noch zusätzlich interessant macht.

In unserer Auswertung des Modells wurden bisher nur die Gesamtzahlen der Populationen in jedem Zeitschritt ausgewertet. Derartige Systeme können aber auch dazu dienen, die *räumliche Verteilung* der Spezies mit zu untersuchen. Geeignete Erweiterungen können also zum Besipiel dazu dienen, die Ausbreitung

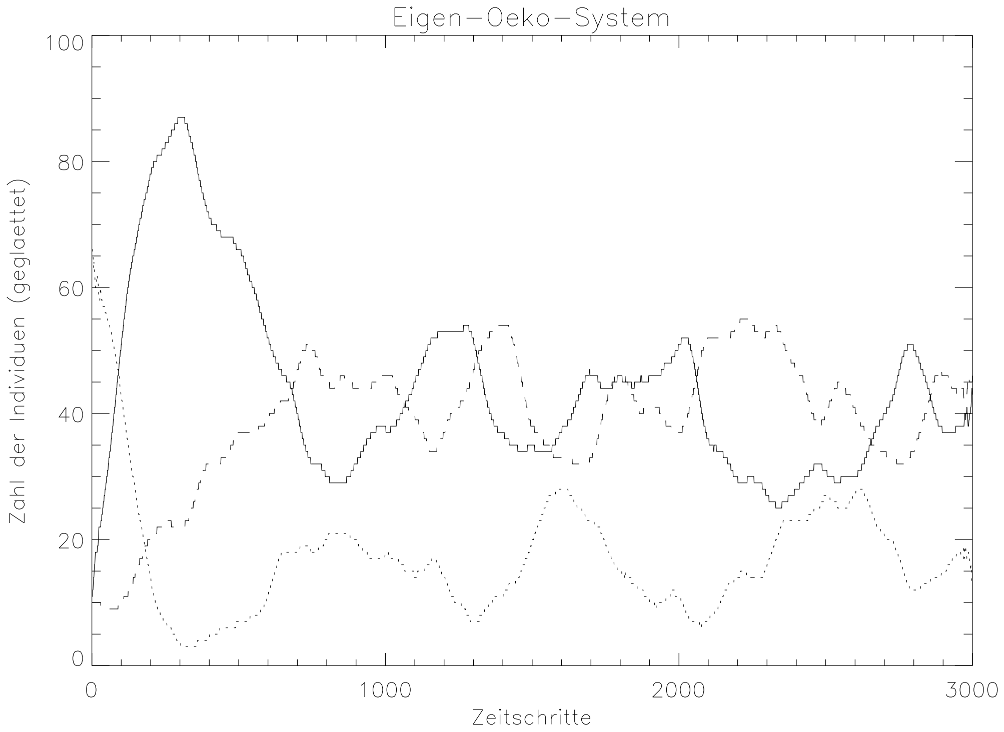


Abbildung 2.1: Typischer Simulationslauf: Die Populationsstärken tendieren zum Oszillieren. Gras (gepunktet), Hasen (durchgezogen), Füchse (gestrichelt)

von Schädlinen zu untersuchen. In Ökosystemen ist oft auch die Koexistenz von konkurrierenden Arten nur möglich, wenn sie räumlich getrennt auftreten können.

Effiziente Programmierung

Da die Funktion f (d.h. das Unterprogramm `spielzug`) sehr häufig aufgerufen wird, ist es entscheidend hier Techniken anzuwenden, die zu laufzeiteffizienten Programmen führen Das erfordert hier insbesondere gute *Zufallszahlgeneratoren*.

Die Programmierung der Auswertung als separate Routine `auswertung` führt leider zu einem erheblichen Rechenmehraufwand. Wird die Auswertung häufig (relativ zur Zahl der Spielzüge) aufgerufen, dann muß die Auswertungsstatistik unbedingt *inkrementell*, innerhalb der Routine `spielzug` erledigt werden. Leider zerstört dies die modulare Programmarchitektur.

Der Aufruf der Unterprogramme und eventueller Hilfsroutinen selbst sollte durch offene Unterprogrammtechnik (*inline*-Technik) beschleunigt werden. In Zukunft wird dies allerdings durch intelligente automatisch optimierende Compiler weitgehend selbständig erfolgen.

Auch die geschickte Implementierung des zweidimensionalen Feldes ist von Bedeutung. Zur Darstellung der Zustände jedes Feldelementes in unserem Beispiel reichen nur 4 Bit. Deshalb kann man überlegen, ob eine kompakte Speicherung sinnvoll ist. Da in Zukunft sich wohl Rechner mit 64-Bit Worten durchsetzen werden, könnten in einem Wort bis zu 16 Feldelemente gespeichert werden. Deshalb muß überlegt werden, ob eine Speicherung als Feld von `short int` mit 32 oder 16 Bit Worten oder als *char* mit 8 Bit nicht besser sind. Im Extremfall kann auch mit Hilfe von C/C++ Bitfeldern eine völlig redundanzfreie Speicherung erreicht werden.

Ob dies sinnvoll ist, hängt natürlich sehr von der Rechnerarchitektur ab. Bei moderat großen *Spielwelten*, kann die platzverschwenderische Speicherung vorzuziehen sein, wenn die Zugriffe auf ein volles Wort entsprechend schnell sind. Andererseits kann die kompakte Speicherung dazu helfen, daß das Arbeitsfeld in einem schnellern Cache-Speicher oder sogar in den Registern Platz findet. Ist das möglich, dann kann der langsamere Zugriff wieder mehr als aufgewogen werden.

Typisch für den hier vorgestellten Algorithmus ist der *random*-Zugriff auf das Feld `umwelt`, der eine Vektorisierung oder Parallelisierung des Programmes erschweren würde.

Die hier aufgeführten Aspekte werden später, vor allem im Kapitel **??** über Rechnerarchitektur und im Kapitel **??** über Programmiersprachen, im Detail besprochen.

Als letzter Aspekt sei noch die leichte Erweiterbarkeit des Programmes angesprochen. Beim jetzigen Programmaufbau ist der ganze Simulationsablauf in der Routine Spielzug enthalten. Bei einem so kleinen Modell ist dies zwar angemessen, bei großen, komplexen Modellen kann es jedoch einen modulareren Progrogammaufbau von Vorteil sein. Eine Möglichkeit wäre insbesondere die *objektorientierte* Programmierung, bei der jede Spezies als eigene Klasse implementiert wird. Diese Art der Programmierung ist besonders änderungsfreundlich, allerdings bei der Verwendung von *virtuellen Funktionen* auch schlechter von der Laufzeiteffizienz.

Zusammenfassung

- Qualitative Aussagen (in der derzeitigen Form für quantitative Untersuchungen ungeeignet)
- Einfache Regeln verursachen komplexes Verhalten
- Zusammenspiel von Zufall und Determinismus (Schicksal eines Feldes oder einer Kugel zufällig, Gesamtsystem deterministisch?)
- In der angegebenen Form nur Mengensimulation, es wäre aber auch räumliche Verteilung interessant (Wanderungsverhalten, Schädlingsausbreitung)

Aus Sicht des Scientifc Computing halten wir fest: Die *Simulationswelt* besteht aus einem $N \times N$ Feld, von denen jedes einen von endlich vielen Zustanden aus der Menge $K = \{\text{Leer}, \text{Gras}, \text{Hase}, \text{Fuchs}\}$ annehmen kann. Mathematisch gesehen ist der Zustand der Welt also beschrieben durch die Elemente des $K^{N\times N}$. Typisch für N sind Größen von etwa 5-50. Die Dynamik des Systems wird beschrieben duch eine *Übergangsfunktion* mit Zufallseinflüssen

$$f : K^{3\times 3} \longrightarrow K^{3\times 3},$$

die die *lokale* Entwicklung eines Feldelementes beschreibt. Während damit die *lokalen Eigenschaften* direkt gegeben sind, ist es Aufgabe der Simulation, eine Einblick in das *globale* Verhalten zu geben.

2.2 Modellierung mit Differentialgleichungen und Differenzengleichungen.

2.2.1 Modellierung eines Ökosystems mit einem System gewöhnlicher Differentialgleichungen

Spielt die räumliche Verteilung der Spezies keine Rolle, so kann das typische Räuber-Beute-Modell aus Abschnitt 2.1 auch mit Hilfe von Differentialgleichungen beschrieben werden. Dazu nimmt man an, daß die Zahl der Individuen so groß ist, daß die Modellierung mit reellen Zahlen (und auf dem Rechner natürlich mit Gleitpunktzahlen) sinnvoll ist. Bezeichent man die Menge Gras, die Zahl der Hasen und die Zahl der Füchse mit g, h, und f jeweils so, daß die Größen auf 1 normiert sind, so kann man folgendes Modell aufstellen

$$\begin{aligned}
\frac{d}{dt}g(t) &= 1 - h(t)g(t), \\
\frac{d}{dt}h(t) &= h(t)\left(g(t) - f(t)\right), \\
\frac{d}{dt}f(t) &= f(t)h(t) - 0.8f(t).
\end{aligned}$$

Die erste Gleichung hier beschreibt das Wachstum der Grasmenge und das "Gefressenwerden von Gras" mit einer Rate, die proportional zum Produkt $g(t)h(t)$ aus Gras und Hasen ist. In der zweiten Gleichung für die Hasen ist die Wachstumsrate ebenfalls mit $g(t)h(t)$ gegeben. Zusätzlich geht hier die Zahl der Füchse mit $-f(t)h(t)$ mit ein. Bei der Gleichung für die Füchse ist dies wieder der Beitrag zum Wachstum. Der negative Term in dieser Gleichung ist $-0.8 * f(t)$ und simuliert ein Jägerverhalten, bei dem Füchse proportional zu Ihrem Anzahl geschossen werden. Ein typischer Simulationslauf ist in Abb. 2.2 zu sehen.[2] Das System tendiert zu Schwingungen, die werden jedoch mit der Zeit herausgedämpft und es stellt sich ein stationäres Gleichgewicht ein.

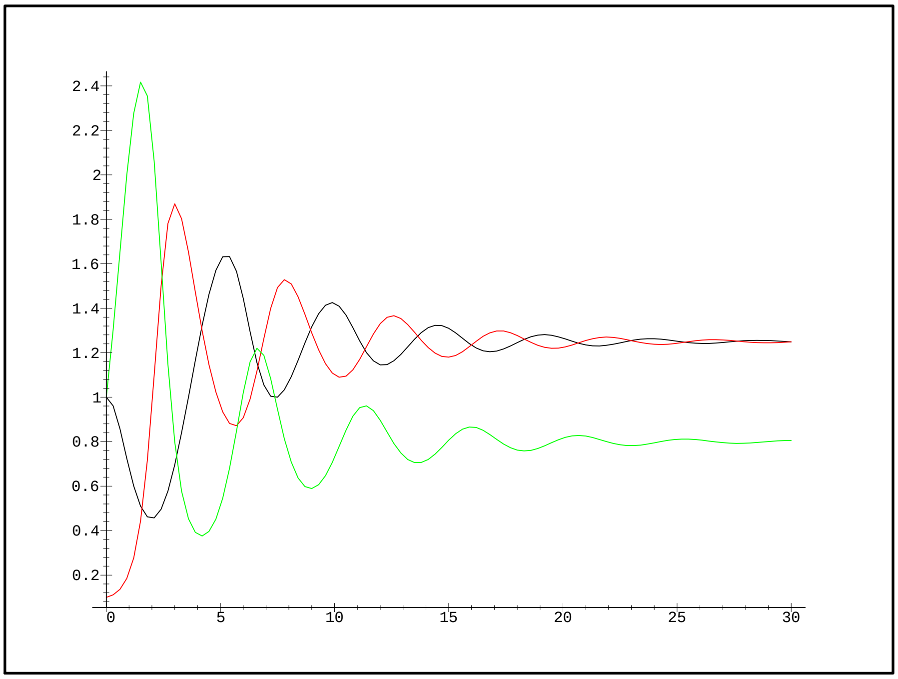


Abbildung 2.2: Simulationslauf: Gedämpfte Oszillationen

Schließlich ersetzen wir die dritte Gleichung durch

$$\frac{d}{dt}f(t) = f(t)h(t) - 0.5\sqrt{f(t)} - 0.3f(t).$$

In diesem Modell unternehmen die Jäger Anstrengungen, die Zahl der geschossenen Füchse auch bei geringer Populationszahl zu halten. Für kleine $f(t)$ gilt nämlich $\sqrt{f(t)} > f(t)$. Das dazughörige Systemverhalten ist in Abb. 2.3 dargestellt. Wieder ist eine gedämpfte Schwingung zu erkennen, jedoch sind die Ausschläge gerade am Anfang wesentlich größer. Am Anfang wird durch die hohe Abschußquote die Zahl der Füchse so stark dezmiert, daß es zu einer richtiggehenden Hasenplage kommt. Die Folge davon ist wieder ein starker Rückgang der

[2] Die Berechnungen und Graphen wurden mit dem Programm *maple* durchgeführt.

Grasmenge, die sich z.B. als erhebliche Schäden für die Landwirtschaft interpretieren ließen. Aus dem Simulationsverlauf ist auch ersichtlich, daß steuernde Gegenmaßnahmen zum Zeitpunkt, wenn die Grasmenge am stärksten reduziert ist zu spät kämen, da zu diesem Zeitpunkt die Zahl der Füchse bereits stark im Anwachsen begriffen ist und deshalb die Zahl der Hasen wieder rückläufig ist.

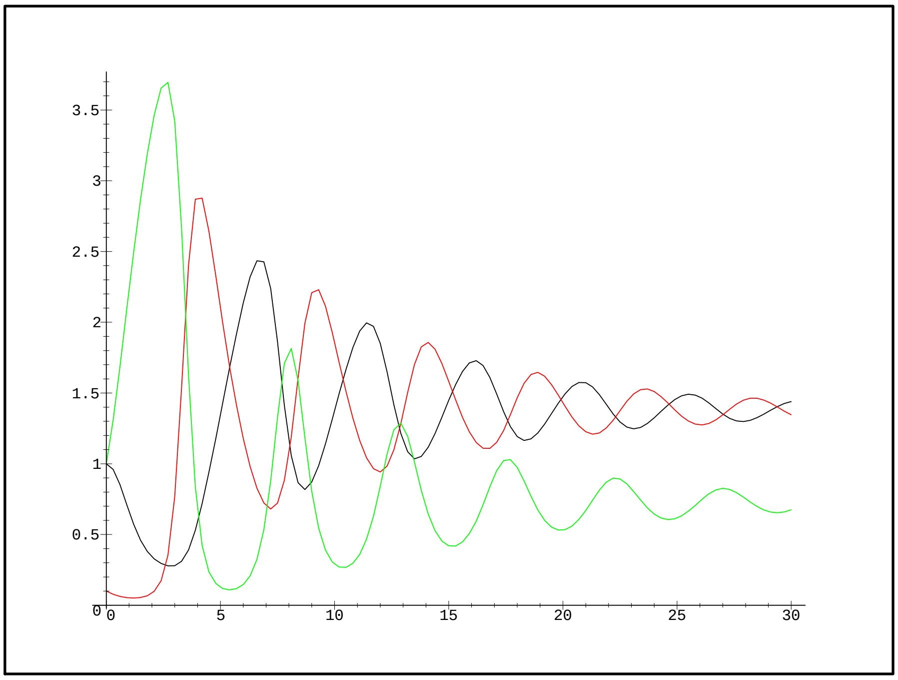


Abbildung 2.3: Simulationslauf: Gedämpfte Oszillationen mit starken Ausschlägen am Anfang

2.2.2 Modellierung mit Differenzengleichungen

Die numerische Lösung von Differentialgleichungen erfolgt im einfachsten Fall mit dem *Eulerverfahren* (siehe z.B. Stoer und Bulirsch [22]). Dabei wird ein System gewöhnlicher Differentialgleichungen erster Ordnung für die vektorwertige Funktion $y(t) : \mathbb{R} \longrightarrow \mathbb{R}^d$ in der Form

$$y'(t) = f(t, y(t))$$

durch Einführung von Zeitschritten τ diskretisiert. Dazu wird die Ableitung $f'(t)$ durch den Differenzenquotienten

$$y' \approx \frac{y(t+\tau) - y(t)}{\tau}$$

ersetzt. Dies führt auf eine *Differenzengleichung* der Form

$$y(t+\tau) = y(t) + \tau f(t, y(t)).$$

Bei geringen Genauigkeitsanforderungen liefert die Folge $y_n = y(t + n\tau)$ Näherungen für das System von Differentialgleichungen. Genauere Verfahren zum Lösen von gewöhnlichen Differentialgleichungen werden z.B. in Stoer, Bulirsch [22] beschrieben.

Bei der Modellierung zeitabhängiger Abläufe kann man entweder zunächst das System mit Differentialgleichungen beschreiben und hat dann eine Vielzahl numerischer Verfahren zur Auswahl, die das System (mehr oder weniger direkt) in eine Differenzengleichung überführen. Moderne numerische Verfahren zeichen sich nicht nur durch eine hohe Genauigkeit aus, sonderen sie besitzen zusätzliche eingebaute Intelligenz bei der sogenannten *Schrittweitensteuerung*. Bei diesen Verfahren wird die Schrittweite τ automatisch problemangepaßt gewählt, so daß immer eine vom Benutzer vorgegebene Genauigkeitschranke eingehalten wird. Die Verwendung solcher Software ist bei allen Problemen, die mehr als nur minimale Genauigkeit erforderen, dringend angeraten. Typische Softwarepakete findet man z.B. in netlib[3] in den Kapiteln ode[4] und odepack.[5] Diese und weitere Programme können auch von der elib[6] in Berlin bezogen werden.

Bei manchen Simulationen, z.B. in der Ökologie oder den Wirtschaftswissenschaften ist jedeoch die direkte Modellierung mit Differenzengleichungen weit verbreitet.

2.2.3 Ausflug ins Chaos

Differentialgleichungen, wie in Abschnitt 2.2.1 beschrieben, können zu *chaotoischem Verhalten* führen. Bei Systemen von Differentialgleichungen muß dazu aber die Zahl der Gleichungen größer gleich 3 sein, und mindestens eine Gleichung muß nichtlinear sein.

Bei Differenzengleichungen kann chaotisches Verhalten bereits bei einer einzelnen nichtlinearen Gleichung auftreten. Deshalb wollen wir für diesen kurzen Ausflug ins Chaos exemplarisch die logistische Abbildung

$$x_{n+1} = a x_n (1 - x_n)$$

betrachten, die das Wachstum einer Population in einem beschränktem Lebensraum beschreibt. a ist dabei ein positiver Parameter. Das autokatalytische Wachstum wird durch den Faktor x_n, die Wachstumsbeschränkung durch den Faktor $1 - x_n$ modelliert. Die entstehende Folge ist in dieser Gleichung auf den Bereich $0 \leq x_n \leq 1$ normiert. In Abhängigkeit von dem Wert für a zeigt sich jetzt drastisch unterschiedliches Verhalten. Für gewisse Werte von a

- existiert ein eindeutiger Grenzwert
- existieren mehrere Häufungspunkte zwischen denen die Folge hin- und her oszilliert

[3] http://www.netlib.org/index.html
[4] http://www.netlib.org/ode/index.html
[5] http://www.netlib.org/odepack/index.html
[6] http://elib.zib-berlin.de/

- ist das Verhalten chaotisch, d.h. es existieren kein Grenzwert oder endliche Zahl von Häufungspunkten.

Das typische Iterationsverhalten ist in Abb. 2.4 und das Bifurkationsdiagramm in Abb. 2.5 angegaben. Die kritischen Werte für a sind in 2.6 enthalten. Diese

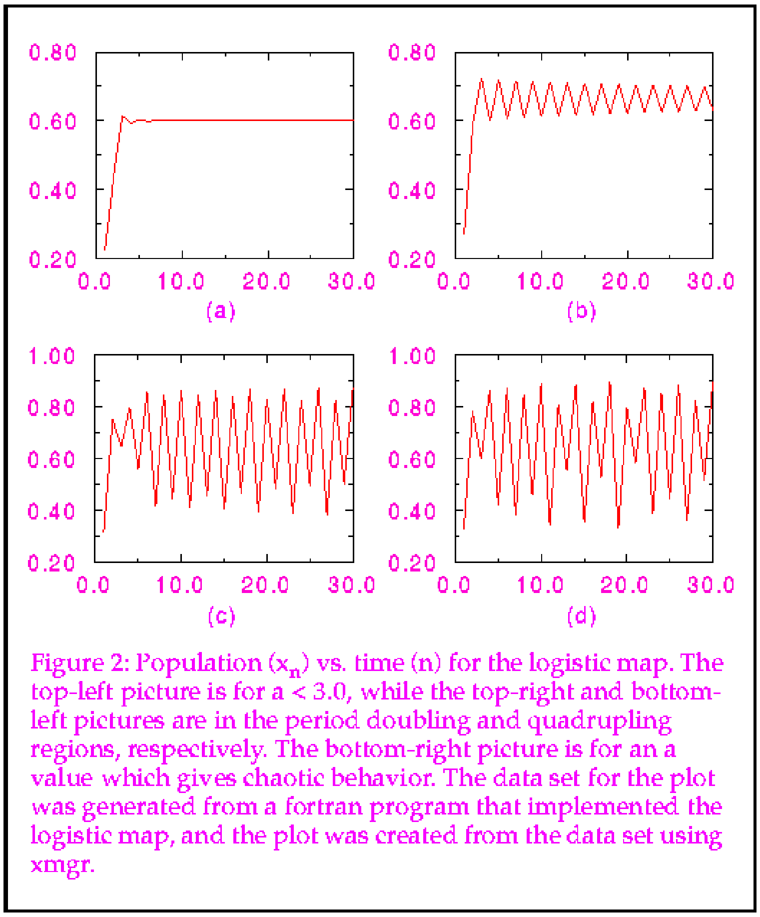


Abbildung 2.4: Typischer Iterationsverlauf

Abbildungen sind dem Kapitel Chaos from Nonlinear Mappings[7] aus dem Computational Science Education Project[8] entnommen, in dem man auch noch weitere Einzelheiten zum Thema Chaos findet.

[7] http://csep1.phy.ornl.gov/ch/ch.html
[8] http://csep1.phy.ornl.gov/csep.html

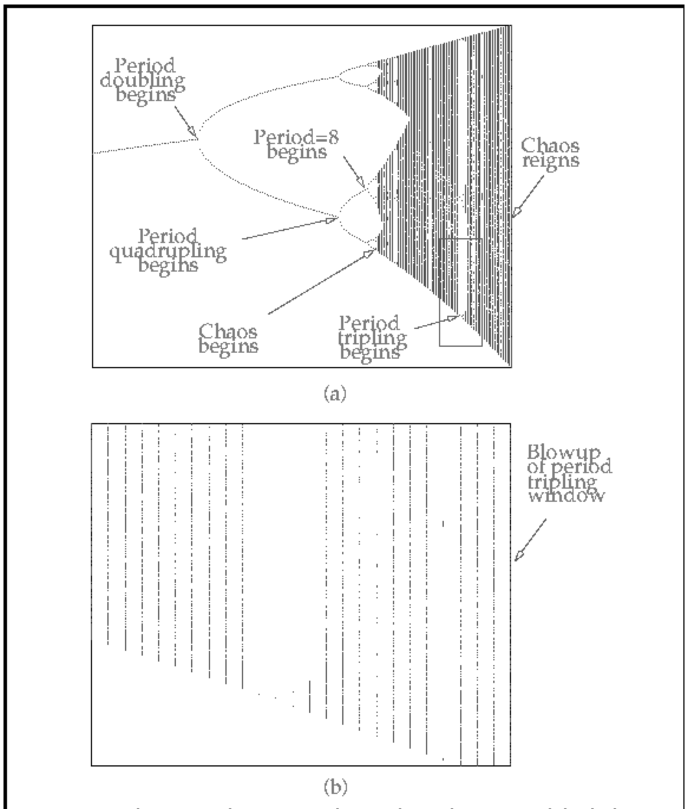


Figure 3: Bifurcation diagram and an enlarged section of the bifurcation diagram for the logistic map. The data set for the plot was generated from a fortran program that implemented the logistic map, and the plot was created from the data set using xmgr.

Abbildung 2.5: Bifurkationsdiagramm der logistischen Gleichung

The Logistic Map

$$x_{n+1} = ax_n(1 - x_n)$$

There are major transitions at the following values of a:

1.0—below this value the population cannot survive
2.0—oscilllatory approach to the asymptotic value
3.0—“period” of the population doubles (beginning of the period-doubling)
3.449499—period quadruples
3.544090—period of 8
3.564407—period of 16
3.568759—period of 32
3.569692—period of 64
...
...
3.569946—period doubling region ends and chaos begins
$\sim$ 3.74—very small period = 5 window opens up
3.828427—small period tripling window opens up
$\sim$ 3.855—period tripling cascade ends and chaos resumes
$\sim$ 4.0—chaos reigns!!

Abbildung 2.6: Kritische Werte

2.3 Partielle Differentialgleichungen

2.3.1 Das Relaxationsverfahren am Beispiel der Wärmeverteilung in einer Platte

Physikalisches Grundprinzip

In diesem Abschnitt diskutieren wir als typisches Problem die Simulation der Wärmeverteilung in einem Werkstück. Als Beispiel edient eine quadratische Platte, die in Abb. 2.7 schematisch dargestellt ist.

Wir nehmen an, die Temperatur ist auf den Rändern bekannt und im Inneren der Platte soll sie durch Simulation bestimmt werden. In diesem Beispiel betrachten wir ein *stationäres Problem*, also eine Situation in der sich das System in einem zeitlichen Gleichgewichtszustand befindet. Das physikalische Grundprinzip ist in diesem Fall:

Temperatur an jedem Punkt = Durchschnitt der benachbarten Temperaturen

Diese Aussage wird direkt als Grundlage für die numerische Simulation verwendet. Mathematisch wird das Verhalten der Wäremeverteilung durch die Lösung einer *elliptischen partiellen Differentialgleichung* beschrieben.

$$\Delta u = 0 \quad \text{in } \Omega \tag{2.1}$$

$$u = g \quad \text{auf} \quad \partial\Omega \quad = \text{Rand von} \quad \Omega \tag{2.2}$$

$$\Omega = [0,1]^2, \tag{2.3}$$

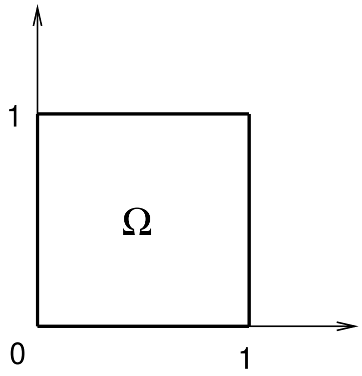


Abbildung 2.7: Wärmeverteilung in einer Platte

wobei der Laplace Operator definiert ist als

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}.$$

(2.1) ist die *Laplacegleichung*, (2.2) bezeichnet man als *Dirichletrandbedingungen*.

Die Grundsituation des Wärmeverteilungsproblems kann auf viele andere physikalische Probleme verallgemeinert werden. Besonders interessante allgemeinere und verwandte Probleme entstehen durch

- zusätzliche (Wärme)quellen
- inhomogenes Material
- nichtlineare (Temperatur)leitfähigkeit
- Schmelzvorgänge (Stefan-Probleme)
- Aufgaben in kompizierter Geometrie
- drei (anstelle von zwei) Raumdimensionen

Gleichungen des gleichen Typs beschreiben aber auch andere physikalische Vorgänge, z.B.

- Strömung in porösen Medien, Schadstoffausbreitungen
- gepanntes Tuch (Zelt oder Trommel)
- (elektrostatische) Felder

Für zeitabhängige Probleme (mit mit eins, zwei oder drei Raumdimensionen) ergibt sich als Prototyp die *parabolische Differentialgleichung*

$$\frac{\partial u}{\partial t} = \nabla(D(x,u) \cdot \nabla u) \quad \text{in} \quad [0,T] \times \Omega \subset \mathbb{R}^4$$
$$u(t=0,x) = u_0(x) \quad \text{Anfangswert}$$
$$u(t, x \in \partial\Omega) = r(t,x) \quad \text{Randwert}$$

Diese *parabolische Gleichung* beschreibt z.B. instationäre Diffusionsprozesse wie die zeitabhängige Strömung in porösen Medien.

Diskretisierung

Zur numerischen Behandlung ersetzen wir jetzt die Platte durch ein *Drahtgitter*, siehe Abb. 2.8 Natürlich stellt sich die Frage, was der Fehler beim Übergang auf dieses Ersatzmodell ist.

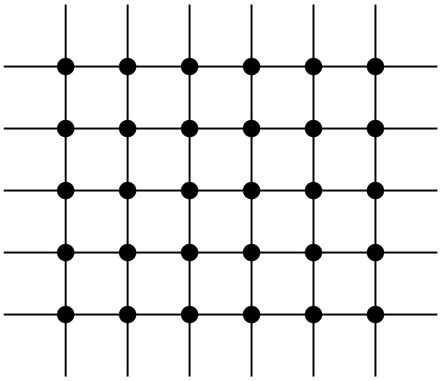


Abbildung 2.8: Drahtgitter als Ersatzmodell für die Platte

Für die Temperatur an Kreuzungspunkten $u_{i,j}$ des Gitters erhalten wir die Gleichungen

$$u_{ij} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j+1} + u_{i,j-1}) \text{ für } i, j = 1, \ldots, N-1. \tag{2.4}$$

Die *Dirichletrandbedingung* gibt an den Rändern feste Werte vor. In den Differenzengleichungen treten diese Werte in den Randgitterpunkten mit Index $i = 0, N$ und $j = 0, N$ auf. Entsprechend der Aufgabenstellung betrachten wir diese Werte als gegeben.

In der Praxis kommen häufig auch *Neumannrandbedingungen* vor, bei denen die *Normalenableitung*, also die Steigung der Lösungsfunktion senkrecht zum Rand vorgegeben ist. Physikalisch entspricht dies der Vorgabe eines Wärmeflusses in den Randpunkten. Der häufigste Fall ist der von Rändern mit Isolierung, die also keine Wärmezu- oder Abfuhr am Rand erlauben. In unserer Simulation würde dies bedeuten, daß die Randpunkte gleiche Temperatur haben, wie die mit ihnen verbunden inneren Punkte des Gitters. Daraus entstehen Gleichungen der Form

$$\begin{aligned} u_{0,i} &= u_{1,i},\ u_{N,i} = u_{N-1,i}, \quad \text{für } i = 1, \ldots, N-1, \\ u_{j,0} &= u_{j,1},\ u_{j,N} = u_{j,N-1}, \quad \text{für } j = 1, \ldots, N-1 \end{aligned} \tag{2.5}$$

für die Randpunkte.

Die Gleichungen vereinfachen sich noch, wenn die Randgleichungen nicht explizit verwendet werden, sondern die dadurch vorgeschriebenen Werte direkt in den inneren Gleichungen (2.4) substituiert werden.

Der Diskretisierungsprozess und die damit verbundenen Fehler sind eine wichtige Fragestellung der numerischen Analysis. Letzendlich wird hier wie in Kapitel 2.2 eine Differentialgleichung durch Differenzengleichungen ersetzt. Im

Unterschied zu dort werden aber die räumlichen Ableitungen durch Differenzenausdrücke angenähert. Während man so bei gewöhnlichen Differentialgleichungen Rekursionsformeln erhält, mit denen man von Zeitschritt zu Zeitschritt weiterrechnen kann, erhält man aus elliptischen Differentialgleichungen ein System von *simultanen* Gleichungen, die gelöst werden müssen.

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} &\approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \\ \frac{\partial^2 u}{\partial y^2} &\approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} \end{aligned}$$

Unser Vorgehen wird als *Differenzenverfahren* bezeichnet. Alternative Techniken zur Diskretisierung von partiellen Differentialgleichungne sind

- Finite Differenzen Verfahren
- Finite Elemente Verfahren
- Finite Volumen Verfahren

Simulation

Das Programm in Abb. 2.9 simuliert das stationäre Wärmeleitungsproblem.

```
real u[N+1][N+1];    /* initialisieren mit Randwerten, im Inneren
                        "0" oder Mittelwert der Randwerte */

for (int it=0; it<MAXIT; it++) {
    real udiff=0;
    for (int i=1; i<N; i++)
        for (int j=1; j<N; j++) {
            real un = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j+1]+u[i][j-1]);
            udiff += fabs(u[i][j]-un);
            u[i][j] = w*un + (1-w) * u[i][j];
        }
        if (udiff / N*N) > TOL) break;  (*)
}
```

Abbildung 2.9: Gauß-Seidel Realxation

Dies ist das sogenannte *lexikographische Gauß-Seidel-Verfahren*. Das Gauß-Seidel-Verfahren wurde in seinen Grundzügen von Gauß 1823 beschrieben. Der im Programm auftretende Parameter `w` ist zunächst 1. Andere Werte für `w` werden im Abschnitt 2.3.2 besprochen.

Die beiden inneren Schleifen verändern sukzessive die Werte jeder Unbekannten so, daß die dazu gehörige Gleichung erfüllt ist. Nach einem Durchlauf durch diese zwei Schleifen erhält man aber im allgemeinen noch keine Lösung des Gleicungsystems da spätere Rechenschritte die Korrektheit der früheren Gleichungen immer wieder zerstören. Da die Rechnung aber physikalisch korrekt den Austausch der Wärmeenergie zwischen benachbarten Punkten beschreibt, ist das Gesamtverfahren, in dem die inneren zwei Scheifen iterativ wiederholt werden *konvergent*, das heißt sie streben, wenn man lange genug rechnet, zur exakten Lösung. Für Gleichungen der vorliegenden Bauart läßt sich dies auch streng mathematisch beweisen und die Konvergenzgeschwindigkeit abschätzen, siehe z.B. Stoer und Bulirsch [22].

Dies wirft natürlich die Frage auf, wie oft man die äußere `it`-Schleife durchlaufen muß. Für ein vernünftiges Programm wird eine *Abbruchbedingung* benötigt. Dazu kan man in den zwei inneren Schleifen die Änderung mitberechnen. rDder Abbruch der `it`-Schleife erfolgt, wenn die durchschnittliche (alternativ die maximale) Änderung klein ist. Die Abbruchbedingung `(*)` ist aber gefährlich, da die dort gemessene *Iterationsgeschwindigkeit* eigentlich keine direkte Aussage über den verbleibenden Fehler enthält.

Beispiel

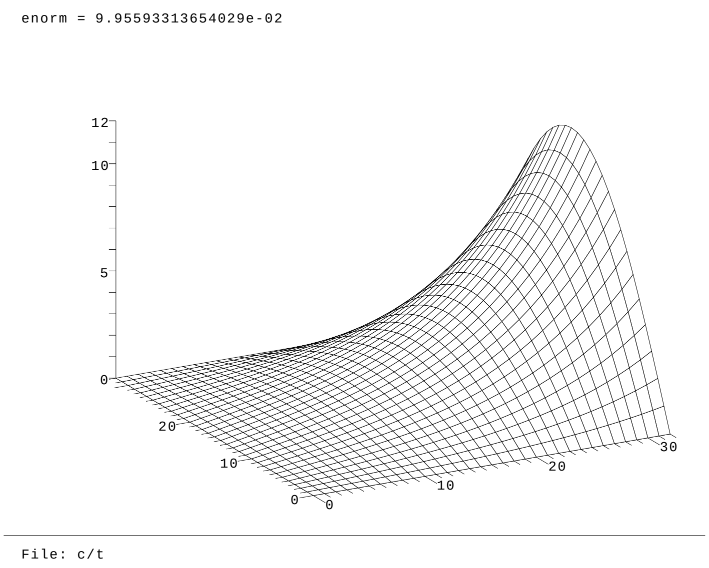


Abbildung 2.10: Exakte Lösung des Beispielproblems

In Abb. 2.10 wird die exakte Lösung einer Beispielrechnung gezeigt. Die gezeigte Funktion

$$u = \sinh(x)\sin(y) \tag{2.6}$$

ist eine exakte Lösung der Laplacegleichung (2.1), wie man einfach nachrechen kann. Obwohl hier also keine besonders interessanten Probleme berechnet werden, kann man an diesem Beispiel sehr schön die Konvergenzeigenschaften des Gauß-Seidel Verfahrens studieren. Die Werte dieser Funktion werden am Rand vorgeschrieben und der Rest der Funktionswerte mit 0 initialisiert. Die Rechnung startet also mit der in Abb. 2.11 gezeigten Funktion. Das Gauß-Seidel-Verfahren durchläuft nun das Gitter von links-oben zeilenweise nach rechts-unten. Das Ergebnis nach einem Durchlauf durch das ganze Gitter ist in Abb.

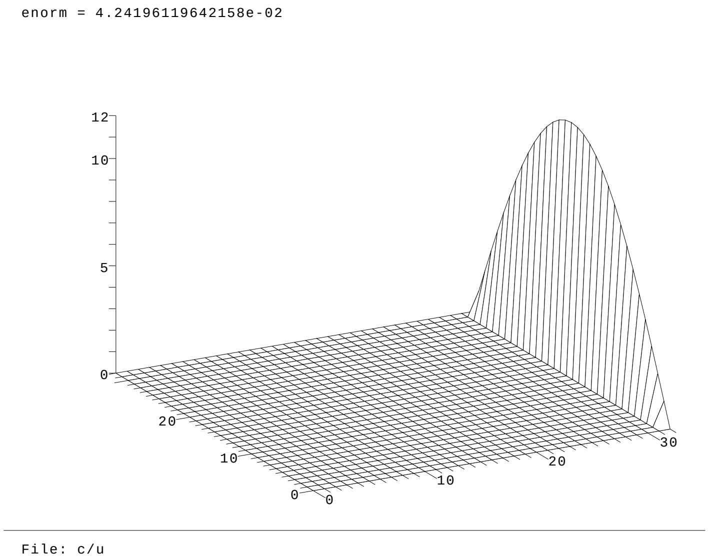


Abbildung 2.11: Startnäherung

2.12 dargestellt. Man erkennt, daß sich überhaupt nur nur eine der Gitterlinien, nämlich die ganz am rechten Rand verändert hat. Entsprechend sind nach zwei Durchläufen nur zwei Gitterlinein im Inneren des Gebietes von 0 verschieden, siehe Abb. 2.13. Die nächsten Abbildungen 2.14, 2.15 und 2.16 zeigen das Resultat der Rechnung nach 10, 100 und 1000 Durchläufen durch die `it`-Schleife. Man erkennt, daß erst die nach 1000 Durchläufen erzielte Näherung befriedigend genau das gewünschte Lösungsverhalten wiedergibt.

2.3.2 Andere Verfahren

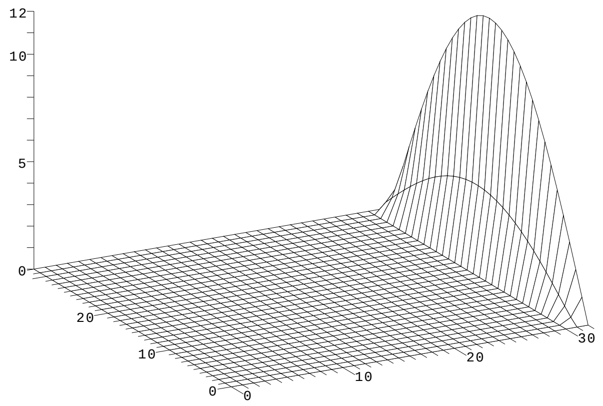
Das Gauß-Seidel Verfahren ist

- einfach und speicherplatzsparend
- aber bei N groß ($N > 10$) langsam. Die erforderliche Zahl der Durchläufe durch `it`-Schleife wächst wie N^2, damit ist der Gesamtaufwand von der Ordnung $O(N^4)$

Es gibt eine Vielzahl von Möglichkeiten, die Einzelschritt-Relaxationen zu steuern, z.B.

- Fixe Reihenfolge
- Lexikographisch
- $\left.\begin{array}{l}\text{Red-Black}\\ \text{Four-Color}\end{array}\right\}$ für Parallelisierung $\left\{\begin{array}{l}\text{bei 5 pkt/Sternen}\\ \text{bei 9 pkt-Sternen}\end{array}\right.$
- Lexikographisch vorwärts-rückwärts (SGS)

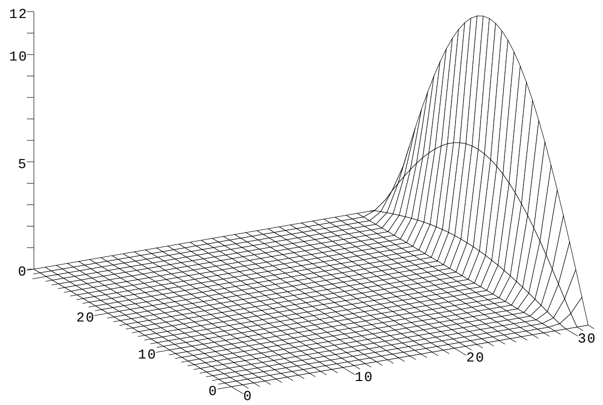
enorm = 4.47048252627281e-02



File: c/u.2

Abbildung 2.12: Nach einem Gauß-Seidel Durchlauf

enorm = 4.72827595615452e-02



File: c/u.7

Abbildung 2.13: Nach 2 Gauß-Seidel Durchläufen

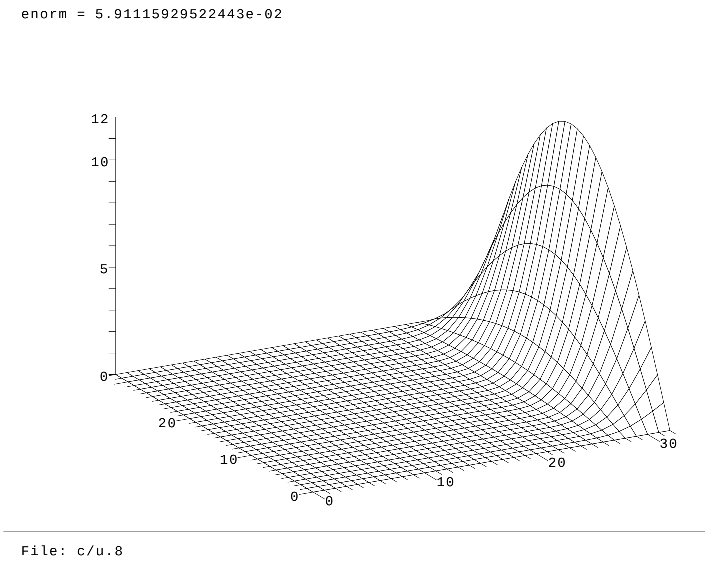


Abbildung 2.14: Nach 10 Gauß-Seidel Durchläufen

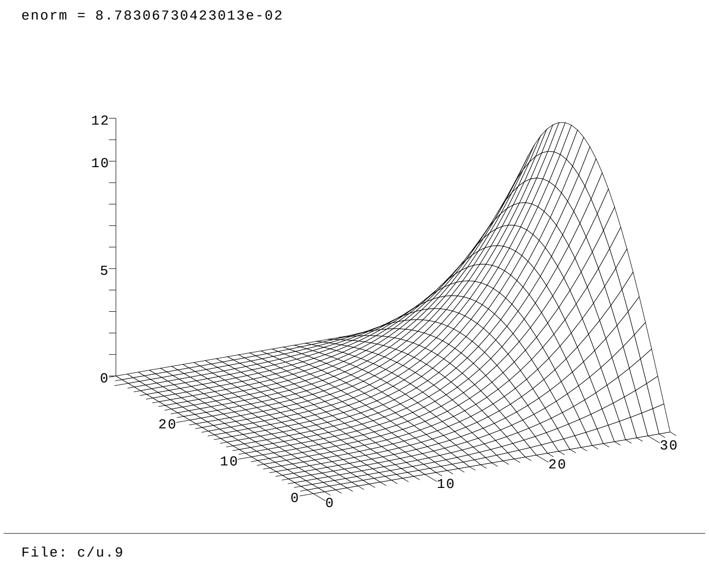


Abbildung 2.15: Nach 100 Gauß-Seidel Durchläufen

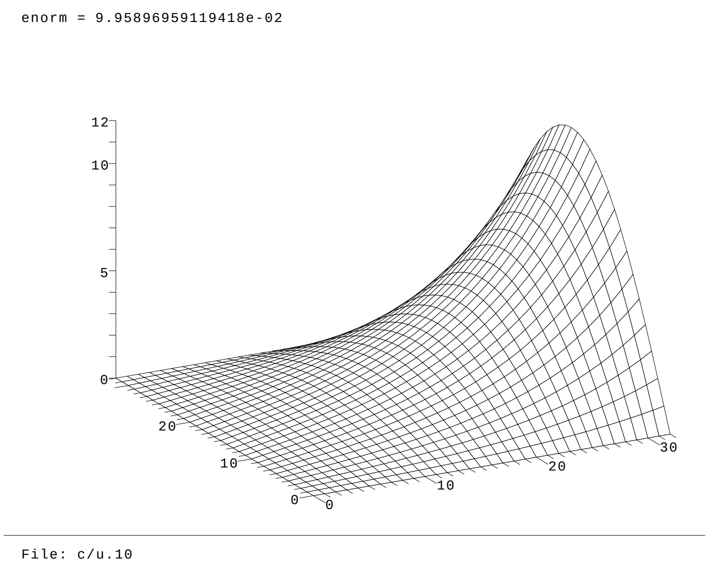


Abbildung 2.16: Nach 1000 Gauß-Seidel Durchläufen

- Lexikographisch vorwärts-rückwärts-aufwärts-abwärts (analog in 3-D)
- Statistisch (zufällig wählen)
- Adaptiv (z.B. maximales Residuum wählen)

Eine substantielle Beschleunigung allein damit läßt sich aber für unsere Anwendungsbeispiele nicht erreichen.

Überrelaxation

Sukzessive Relaxationsverfahren lassen sich beschleunigen, wenn man den Parameter `w` geschickt wählt. Bei optimal gewähltem $\mathtt{w} \in (1, 2)$ ist die erforderliche Zahl der Durchläufe nur von Ordnung $O(N)$ und damit der Gesamtaufwand $O(N^3)$. Schneller geht es nicht, da es im Gitter eine *Lichtgeschwindigkeit* der Informationsausbreitung gibt. In jedem Durchlauf kann sich (zumidestens entgegen der Relaxationsrichtung) die Information höchstens um eine Masche weiter ausbreiten. Das Verfahren erfordert deshalb mindestens N Durchläufe. Für Modellprobleme ist das optimale `w` bekannt, aber für die meisten praktischen Anwendungen muß es experimentell bestimmt werden.

Die Überrelaxation kann das Gauß-Seidelverfahren drastisch beschleunigen und sollte bei allen Problemen (abgesehen von den allereinfachsten) verwendet werden. Im Abschnitt 2.3.4 werden wir allerdings Mehrgitterverfahren vorstellen, die nochmals substantiell schneller als Überrelaxation sind.

Weitere Verbesserungen

Hier werden noch einige weitere Verfahren und Varianten aufgeführt, die typisch zur Lösung von elliptischen Gleichungen verwendet werden. Als weiterführende

Literatur zu iterativen Verfahren empfehlen wir Hackbusch [14], Axelson [1] und das Kapitel über *Iterative Methods for Linear Systems*[9] im *Computational Science Education Project*[10].

Typische Algorithmen und ihre Varianten sind

- Nested iteration – Kaskadenalgorithmus: Startlösung wird durch Interpolation von einem $N/2 \times N/2$-Gitter erzeugt. Die Lösung des gröberen Problems erfolgt rekursiv genau so (Start auf noch gröberem Gitter). Dies führt zu einer weiteren Beschleunigung des Lösungsprozesses.
- ADI-Verfahren
- CG-Verfahren (mit Präkonditionierung). Hierzu gibt es viel Information und eine reiche Literatursammlung im Abschnitt Iterative Methods for Linear Systems[11] aus dem Computational Science Education Project[12]
- Direkte Verfahren, z.B. diskrete Fouriertransformationen oder der Bunemann Algorithmus. Zur direkten Lösung des Gleichungssystemes siehe auch Abschnitt 2.3.3
- Mehrgitterverfahren, siehe 2.3.4

Parallelisierung

Das Einzelschrittverfahren (Gauß-Seidel) ist (beim ersten Hinsehen) inhärent sequentiell. Alternativ kann man deshalb das Gesamtschrittverfahren von Gauß-Jakobi verwenden. Der Algorithmus unterscheidet sich vom Gauß-Seidelverfahren dadurch, daß ein zweites Feld von Werten verwendet wird, und alle Berechnungen die alten, noch nicht modifizierten Werte verwenden. Nach dem Durchlauf muß das alte Feld zurückkopiert werden. (Alternativ und besser wird natürlich abwechselnd zwischen den Feldern hin- und hergeschaltet).

```
for (int i=1; i<N; i++)
  for (int j=1; j<N; j++)
    uz[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]);

for (int i=1; i<N; i++)
  for (int j=1; j<N; j++)
    u[i][j] = uz[i][j];
```

Der Gauß-Jacobi Algorithmus ist trivial parallelisierbar, da alle Berechnungen der inneren Schleifen völlig unabhängig voneinendar sind. Leider kann beim Gauß-Jacobi Verfahren keine Überrelaxation verwendet werden. Das Verfahren ist noch etwas langsamer als das normale Gauß-Seidel Verfahren. Mögliche Parallisierungsgewinne müssen deshalb im Vergleich zu Ergebnissen für Verfahren

[9] http://csep1.phy.ornl.gov/la/section3_9.html
[10] http://csep1.phy.ornl.gov/csep.html
[11] http://csep1.phy.ornl.gov/la/section3_9.html
[12] http://csep1.phy.ornl.gov/csep.html

mit Überrelaxation gesehen werden, die sich zwar nicht so gut parallelisiererm lassen, aber wesentlich bessere Konvergenzeigenschaften haben.

Auch das Gauß-Seidel-Verfahren (und damit die Überrelaxation) ist nämlich bei näherem Hinsehen nach dem Schema in Abb. 2.17 parallelisierbar. In dieser

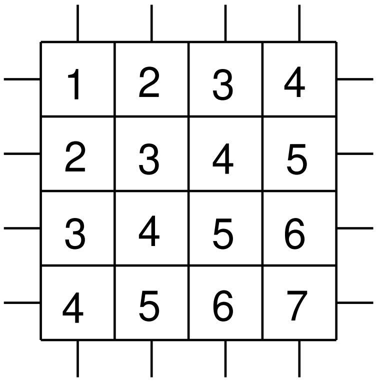


Abbildung 2.17: Paralleisierungsschema für das Gauß-Seidel Verfahren

Abbildung wird angegeben, in welchem Zeitschritt die Relaxation an einem Knoten des Gitters ausgeführt wird. Man erkennt, daß bei maximal $N-1$ gleichzeitig ausgeführten Relaxationen $2N-3$ Ausführungsschritte erforderlich sind. Nach dem gleichen Schema lassen sich auch andere Iterationsverfahren wie z.B. die sogenannten unvollständigen Zerlegungen (ILU, siehe z.B. Wesseling [25]) parallelisieren.

Ein anderer interessanter Aspekt ergibt sich, wenn man die Parallelisierung des Jakobi-Verfahrens nochmals genauer ansieht. Als Beispiel betrachten wir die Situation im Eindimensionalen.

Bei einer maximale Parallelisierung werden sämtliche Werte gleichzeitig berechnet (Gesamtschrittverfahren). Der Abhängigkeitsgraph in Abb. 2.18 zerfällt

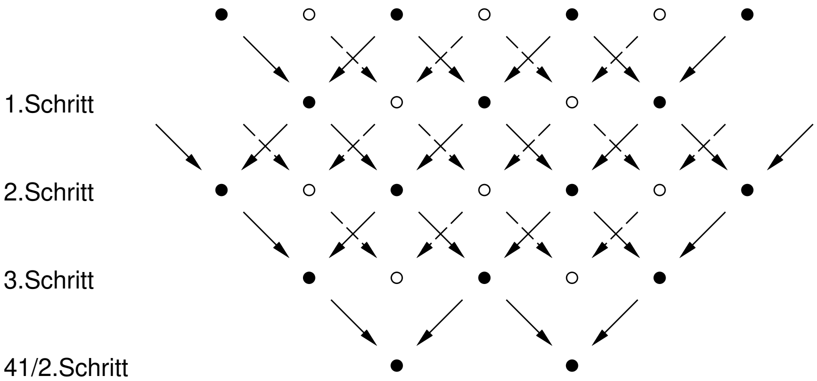


Abbildung 2.18: Paralleisierungsschema für das Gauß-Jacobi Verfahren

in zwei völlig unabhängige Teilberechnungen. Man erkennt, daß einer der beiden Teile überflüssig wird, wenn man noch einen halben Iterationschritt als Abschluß durchführt, um die Werte an den fehlenden Knoten aufzufüllen. Interesanterweise ist der damit beschrieben Algorithmus äquivalent zu einem Gauß-Seidelverfahren in Rot-Schwarz Durchlaufreihenfolge.

Diese Überlegung läßt sich analog auch im Zwei- und Dreidimensionalen machen. Die Idee hängt nur von der Form der Differezensterne ab: Im Fall von 9-Punkt-Sternen für das zweidimensionale Problem muß z.B. anstelle des einfachen Rot-Schwarz-Verfahrens eine Vier-Farb-Partitionierung der Gitterpunkte benützt werden.

2.3.3 Direkte Verfahren

Bandlöser

In Matrixschreibweise ergeben die Gleichungen (2.4) ein Gleichungssystem der Form wie in Abb. 2.19.

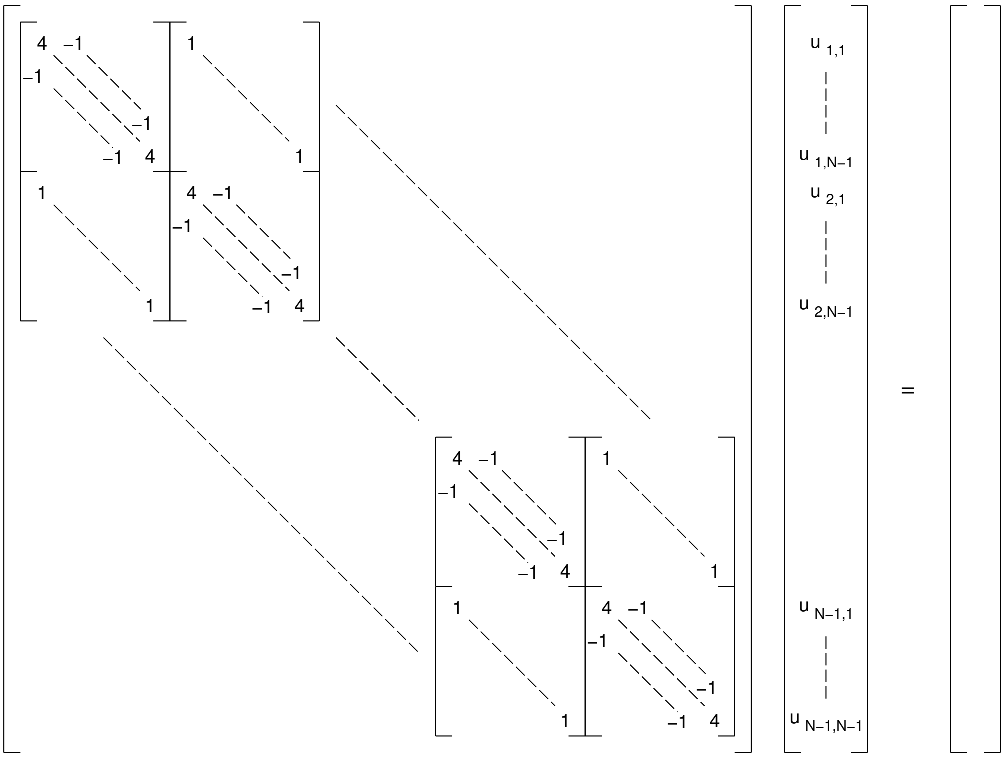


Abbildung 2.19: Gleichungssystem für (2.4)

Es sei angemerkt, daß ganz ähnliche Matrixstrukuren auch bei anderen Diskretisierungstechniken erhalten werden. Genau die gleiche *Steifigkeitsmatrix* erhält man bei Diskretisierung mit dreieckigen linearen finiten Elementen bei einer uniformen Triangulierung wie in Abb. 2.20. Die Randwerte und rechte Seite werden bem Finite-Element-Verfahren aber anders berechnet und stimmen mit den Werten beim Differenzenverfahren nur annähernd überein.

Bei der Verwendung von Blockmatrizen, kann das Gleichungssystem aus Abb. 2.19 auch kompakter mit den Gleichungen

$$A = \mathrm{tridiag}_{N-1}(-I, T, -I)u = f$$

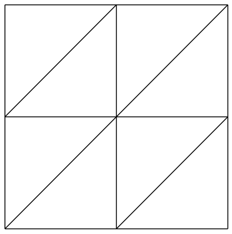


Abbildung 2.20: Quadratgittertriangulierung

und

$$I = \left.\begin{bmatrix} 1 & & 0 \\ & \dots & \\ 0 & & 1 \end{bmatrix}\right\}(N-1)$$

$$T = \operatorname{tridiag}_{N-1}(-1, 4, -1)$$

beschrieben werden.

Dieses System kann nun auch mit direkten Verfahren der numerischen Linearen Algebra gelöst werden. Die Grundalgorithmen zur Lösung von linearen Gleichungssystemen sind z.B. in Stoer [21] beschrieben. Weitere Information findet man in dem Kapitel *Numerical Linear Algebra*[13] aus dem *Computational Science Education Project.*[14]

Die Lösung eines linearen Gleichungssystems mit $O(N^2)$ Unbekannten mit dem Eliminationsverfahren von Gauß erfordert aber $O((N^2)^3) = O(N^6)$ Operationen. Zur Verbesserung ist es also unbedingt erforderlich, die spezielle Struktur des Gleichungsystemes auszunutzen.

Natürlich brauchen Matrixeinträge, die schon 0 sind nicht erst eliminiert werden. Leider aber, kann man so nur die Nullen außerhalb des Bandes ausnützen. Die Nullen innerhalb des Bandes werden schnell durch den Eliminationsprozess zerstört. Dieser *Fill-in* ist bei der oben angegebenen Matrixstruktur unvermeidlich. Zur Abschätzung des Aufwandes können die Bänder deshalb als voll angenommen werden.

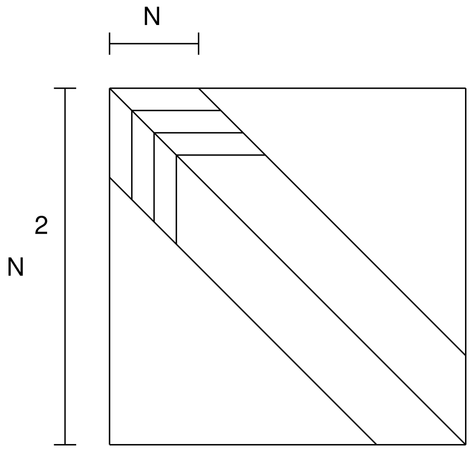
Abb. 2.21 zeigt, daß der auf dieser Idee aufbauende *Bandlöser* vom Aufwand mit $O(N^4)$ also durchaus konkurrenzfähig zur Gauß-Seidel-Relaxation ist. Leider ist der Speicheraufwand höher, denn die Speicherkomplexität ist $O(N^2 \cdot N) = O(N^3)$.

Das Speicherproblem kann (teilweise) entschärft werden, wenn der Aufbau der Matrix (oder ihr Einlesen vom Hintergrundspeicher) mit der Berechnung der Elimination überlagert wird. Eine weitere Möglichkeit zur Beschleunigung wäre die Parallelisierung.

Bei allgemeineren Problemen mit unstrukturierten Netzen ist die Bandstruktur aufgebrochen, und wird zu einer *Profilstruktur.* In Finite-Element-Paketen werden die entsprechend verallgemeinerten Bandlöser häufig verwendet. Einer ihrer Vorteile sind die relative numerische Robustheit und Genauig-

[13] http://csep1.phy.ornl.gov/la/la.html

[14] http://csep1.phy.ornl.gov/csep.html



Zu eliminierende Elemente:	$O(N^2 \cdot N)$	Elemente
Jede Elimination eines Elementes:	$O(N)$	Operationen
Aufwand:	$O(N^4)$	

Abbildung 2.21: Aufwandsabschätzung für den Bandlöser

keit. Trotzdem sind aus heutiger Sicht Bandlöser nur noch bei relativ kleinen Problemen konkurrenzfähig.

Bei Problemem in drei Raumdimensionen wird der durch Fill-In verursachte Mehraufwand schließlich völlig unakzeptabel. 3d-Probleme haben nämlich für das Modellproblem in einem Würfel mit $N-1 \times N-1 \times N-1$ Gitterpunkten eine Matrixdimension von $(N-1)^3$ und eine Bandbreite von $O(N^2)$. Der Aufwand hieraus errechnet sich dann zu $O(N^7)$. Für das Gauß-Seidel-Verfahren beim gleichen Problem wäre der Aufwand $O(N^5)$, bei Überrelaxation sogar nur $O(N^4)$. Bandlöser sind für damit für kleinere 2-D Probleme akzeptabel, jedoch nicht für 3-D Probleme.

Nested Dissection

Zur Verbesserung der direkten Verfahren muß man zurückgehen, und die Matrixstruktur nochmals neu studieren. Obwohl die Lösung des Gleichungssystems natürlich unabhängig von der Reihenfolge der Gleichungen ist, ist offensichtlich das Eliminationsverfahren davon abhängig. Gesucht sind also Eliminationsreihenfolgen, die für die Elimination günstiger sind, weil sie weniger Fill-In prduzieren.

Im Modellproblem teilen wir dazu die Unbekannten so ein, wie in Abb. 2.22 angegeben. Die Menge der Indizes wird also disjunkt in drei Mengen zerlegt.

$$\text{Menge der Indices} = \{1, ..., (N-1)^2\} \;=\; G_1 \cup G_2 \cup S \quad \text{disjunkt.}$$

Daraus ergibt sich eine Matrixstruktur der Form, wie in Abb. 2.23 angegeben. Ein etwas genauerer Blick ergibt, daß die durch A_{11} und A_{22} beschriebenen Teilmatrizen die gleiche Grundgestalt haben wie das Ausgangsproblem. Des-

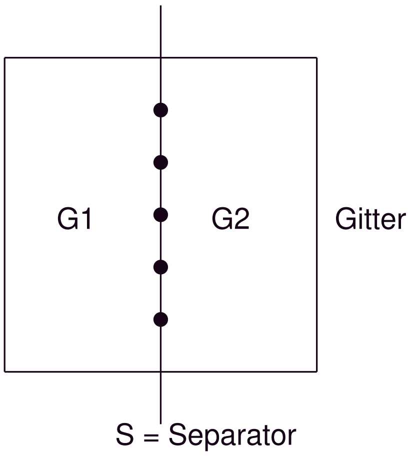


Abbildung 2.22: Zerlegung des Gebietes

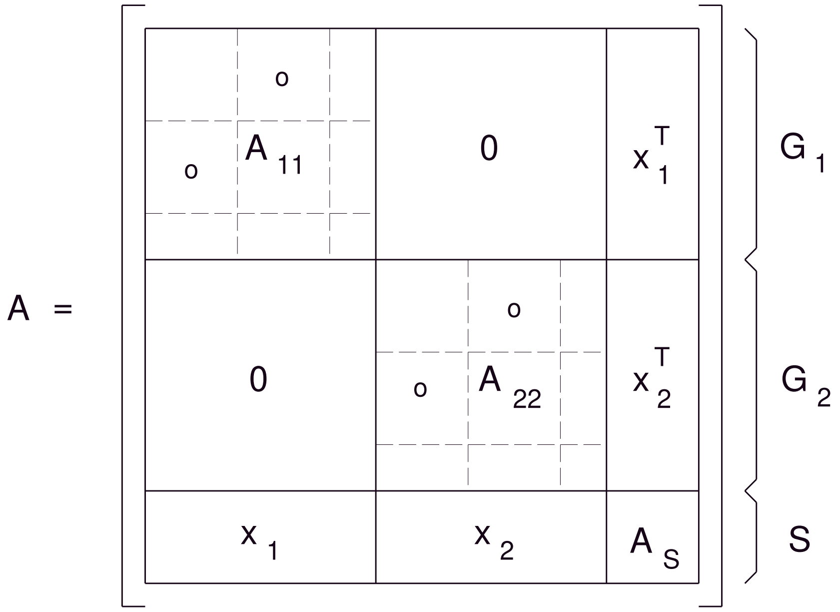


Abbildung 2.23: Nested Dissection Matrixstruktur

halb ist die gleiche Umordungsidee rekursiv für A_{11} und A_{22} anwendbar. Setzt man diese *Divide et Impera* Strategie konsequent fort, bis nur noch einelementige Matrizen übrig sind, so erhält man im vorliegenden Fall tatsächlich eine Struktur, die bei der Elimination den Fill-in minimiert.

Zusätzlich kann man für das Modellproblem ausnützen, daß $A_{11} = A_{22}$. Im allgemeinen Fall, wenn $A_{11} \neq A_{22}$ kann man man immerhin A_{11} und A_{22} parallel bearbeiten. Der Hauptaufwand für das Gesamtverfahren wird jetzt durch die Faktorisierung von A_S bestimmt. A_S ist zunächst eine tridiagonale Matrix der Dimension $N-1$, die jedoch während des Eliminationsprozesses in A_{11} bzw A_{22} voll wird. Der Aufwand für ihre Zerlegung ist also $O(N^3)$.

Für allgemeinere Probleme, insbesondere wenn nur noch die Matrix, aber nicht mehr die Netzstruktur bekannt ist, bedingt das Verfahren, daß der Separator bestimmt wird. Entscheidend ist dabei, daß die Separatormatrix klein ist, und das Problem duch den Separator in zwei annähernd gleich große Teile zerfällt. In vielen technischen Anwendungen sind aus der Aufgabenstellung gute (=kleine) Separatoren bekannt. Die Bestimmung von Separatoren aus der Matrixstruktur (wenn das Anwendungswissen nicht mehr sichtbar ist) ist dagegen mühsam und rechenaufwendig.

Abb. 2.24 soll illustrieren, daß technische Konstruktionen oft sehr kleine Separatoren zulassen, denn in dem Kran sind der Turm und Ausleger sind nur an wenigen Punkten verbunden.

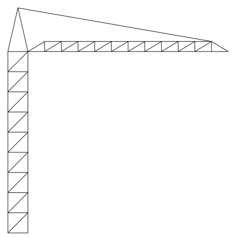


Abbildung 2.24: Typische Finite-Element-Struktur bei einer technischen Anwendung

Leider arbeiten in der Praxis sehr viele Anwendungspakte mit einer horizontalen Modularisierung des Algorithmus, bei der der Lösungsalgorithmus nur noch die Matrix, nicht aber das zusätzliche Wissen über gute Zerlegungen des Problems erhalten.

2.3.4 Mehrgitterverfahren

Dieser Abschnitt ist derzeit nur auf Englisch und als html-Dokument[15] verfügbar.

2.3.5 Zusammenfassung

Selbst *einfache* Aufgaben, die auf Standardprobleme der linearen Algebra führen, erfordern ein genaues Studium der Strukturen, damit große Probleme effizient gelöst werden können. Matritzen sind meist strukturiert und nicht nur beliebige

[15] http://www5/sci-comp/xwb/xwb.html

$N \times N$ Felder von Zahlen. Das Ausnützen dieser Struktur ist für Effizienz der Algorithmen entscheidend.

Der hierarchische Aufbau führt zu Multilevel bzw. zu Divide & Conquer - Algorithmen (Nested Dissection), die in ihrer Klasse jeweils optimal sind. Typisch sind dünn besetzte Matritzen mit rekursiver Blockstruktur. Voll besetzte Matritzen sind vergleichsweise selten.

Wie bei den ökologischen Modellen von Eigen und Winkler in Abschnitt 2.1, ist die Simulationswelt für die Lösung partieller Differentialgleichungen ein Gitter, das hier aber mit reellen Zahlen besetzt ist.

$$\text{Welt} = \mathbb{R}^{N \times N (\times N)} \quad (N \sim 10 \text{ bis } 1000)$$

Die Grundsimulation basiert auch hier auf einer *Übergangsfunktion*

$$f : \mathbb{R}^{3 \times 3} \rightarrow \mathbb{R}^{3 \times 3},$$

der Relaxation, die in den iterativen Algorithmen simultan oder sequentiell und in verschiedenen Reihenfolgen angewandt wird. Das Ziel der Simulation ist es auch hier, die *globale* Auswirkung *lokaler* Wechselwirkungen zu berechnen. Die Simulationen hier ist aber quantitativ, d.h. es werden möglicht genaue Ergebnisse berechnet.

2.4 Kombinatorische Optimierung am Beispiel des Travelling Salesman-Problems

In diesem Abschnitt untersuchen wir das Problem des Handlungsreisenden (engl. Travelling Salesman), der eine Anzahl Städte n einer festzulegenden Reihenfolge besuchen muß, und dabei seine Reisekosten (oder Reisezeit) minimieren will. Die mathematische Beschreibung des Problems ist wie folgt.
Gegeben sind

1. *Vollständig zusammenhängender Graph* $G = (V, E)$ mit n Knoten und mit ungerichteten Kanten und eine
2. *Kostenfunktion:* $c : V \times V \rightarrow \mathbb{R}_0^+$ (also $c : E \rightarrow \mathbb{R}_0^+$)

Die Kosten für einen *Weg* im Graphen isnd rekursiv deviniert durch

$$c((v_1, \ldots, v_m)) := c((v_1, \ldots, v_{m-1})) + c(v_{m-1}, v_m)$$

Gesucht ist

> Ein Weg im Graphen, der alle Knoten genau einmal berührt, dessen *Kosten minimal* sind, und der zum Ausgangspunkt zurückkehrt (eine *Tour*).

Mit $v1, \ldots, v_m$ ist auch jede zyklische Permutation eine Tour mit gleichen Kosten.

Wir stellen nun eine Reihe von algorithmischen Ansätzen vor, um das Travelling-Salesman-Problem zu lösen. Zur Erläuterung verwenden wir das folgende Beispiel[16] in Abb. 2.25.

[16] Entnommen dem Skript Informatik IV (J. Schlichter) an der TU-München

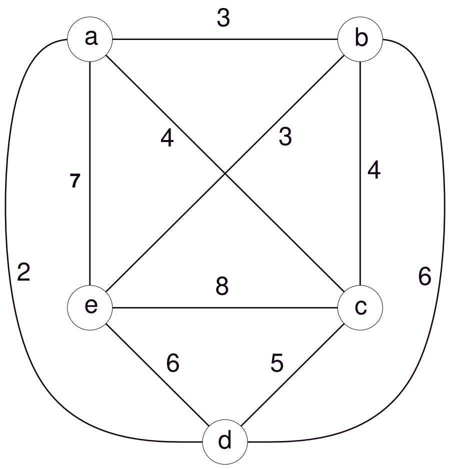


Abbildung 2.25: Beispiel für das Travelling Salesman problem

2.4.1 Greedy-Algorithmus

Der erste Ansatz ist naheliegend:

- Starte beliebig im Graphen
- Wähle jeweils die billigste noch erlaubte Kante

Dieser Algorithmus findet aber im allgemeinen keine optimale Lösung und führt bei unvollständigen Graphen in Sackgassen.

2.4.2 Vollständiges Suchen

Berechne die Kosten *aller möglichen* Touren und suche das Minimum. Der Aufwand für diesen Algorithmus ist $O(n!)$. Er ist damit schon für kleine N extrem teuer.

2.4.3 Suche mit Heuristik

Hier wird ebenfalls von der Gesamtheit aller Touren ausgegangen, aber es wird versucht den Suchraum durch die Verwendung von Zusatzwissen kleiner zu machen. Man berechnet also den Lösungsbaum und verwendet eine *Heuristik* zur Beschränkung des Suchraumes.

Hierzu beschreiben wir eine Tour durch die verwendeten Kanten. Wir verwenden die folgende Notation.

$ab \leftrightarrow$ Tour enthält die Kante ab
$\overline{ab} \leftrightarrow$ Tour enthält die Kante ab nicht

Ein möglicher Lösungsbaum ist in Abb. 2.26 dargestelt. Einige Kombinationen im Lösungsbaum sind unzulässig, z.B. (ab, ac, ad) ist nicht möglich, da Inzidenz$(a) > 2$. $(\overline{ab}, \overline{ac}, \overline{ad})$ ist ebenfalls unzulässig, da Inzidenz$(a) < 2$.

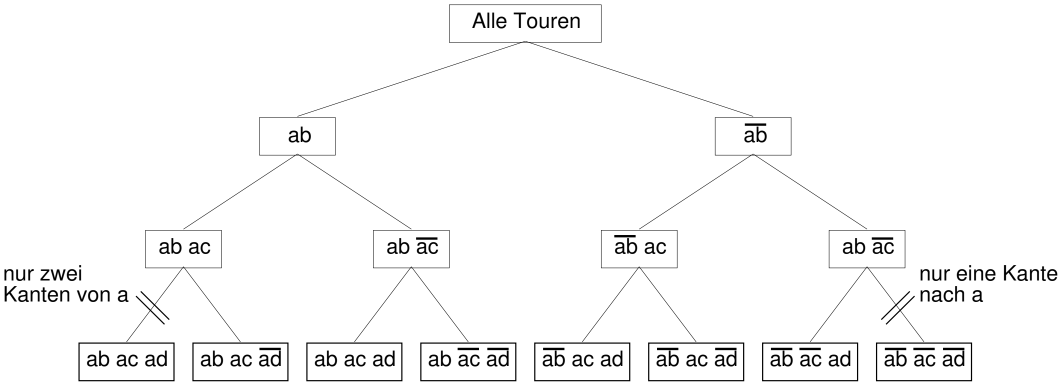


Abbildung 2.26: Beispiel für das Travelling Salesman problem

Der nächste Schritt ist die Bestimmung einer *unteren Kostengrenze* für Knoten im Lösungsbaum. Ist $T = (v_1, \ldots, v_n)$ eine Tour, dann tragen an jedem v_i zwei Kanten zu den Kosten der Tour bei, also

$$c(T) = \frac{1}{2} \sum_{i=1}^{n} c(v_{i-1}, v_i) + c(v_i, v_{i+1}) \quad (\text{mit } v_o \equiv v_n, v_{n+1} \equiv v_1)$$

Eine untere Kostengrenze ergibt sich nun durch die Abschätzung

$$c(T) \geq b(T) = \frac{1}{2} \sum_{i=1}^{n} \min_{k,l \in \{1,\ldots,n\}} \{c(v_k, v_i) + c(v_i, v_l) \mid k \neq l,\ k, l \neq i\}$$

Für das Beispiel 2.25 ergibt sich also die Kostengrenze

$$b(T) = \frac{1}{2}\,(5 + 6 + 8 + 7 + 9) = \frac{1}{2} \cdot 35 = 17.5$$

Diese Kostengrenze kann nun nach jeder Entscheidung, ob eine Kante gewählt oder ausgeschlossen wird, neu angepaßt werden. Diese Berechnung ist *inkrementell* durchführbar, d.h. sie kann lokal im Baum aus der Kostengrenze des Vaterknotens und der lokal gefällten Entscheidung berechnet werden.

Für den Aufbau des Lösungsbaumes verwenden wir folgende Heuristik. An jedem Knoten:

1. Kante xy wird zugefügt wenn

$$(\overline{xy}) \in T \Rightarrow \text{Inzidenz}(x) < 2 \text{ oder Inzidenz}(y) < 2$$

2. Kante xy wird ausgeschlossen, falls

$$xy \in T \Rightarrow \text{Inzidenz}(x) > 2 \text{ oder Inzidenz}(y) > 2$$

3. Wenn die aktuelle Kostengrenze größer ist, als die Kosten einer bereits gefundenen Tour, kann die Suche im Teilbaum beendet werden

4. Falls beide Alternativen untersucht werden müssen, dann wird zuerst die Alternative mit geringerer aktueller Kostengrenze betrachtet

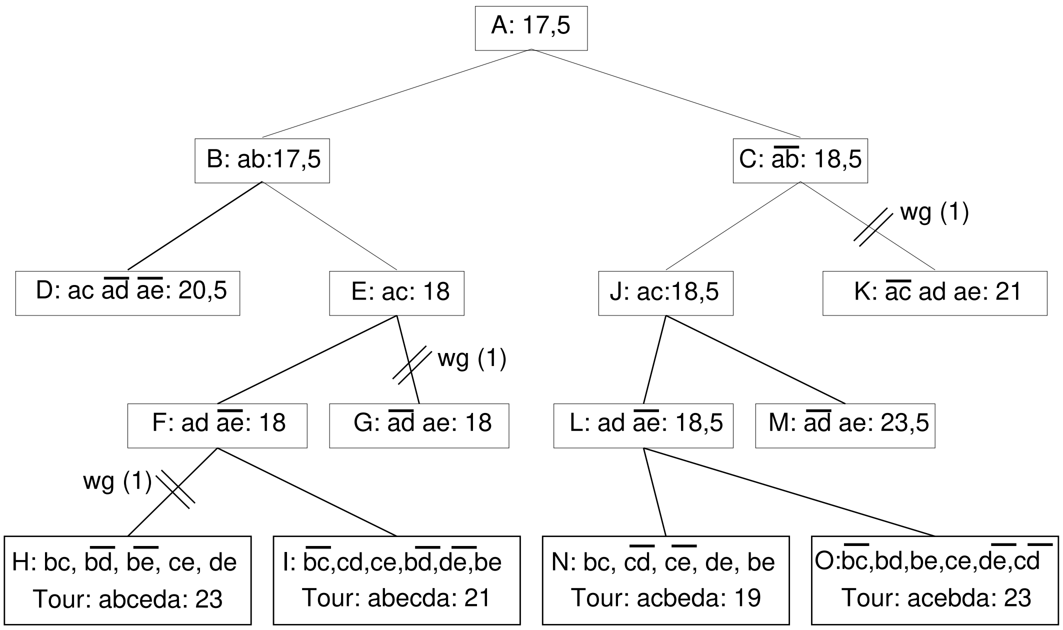


Abbildung 2.27: Beispiel für das Travelling Salesman problem

Wir füren dies am Beispiel des Graphen in Abb. 2.25 durch. Der entstehende Lösungsbaum wird in Abb. 2.27 gezeigt. Erläuterungen:

C: Ausschluß von ab erhöht Kostengrenze von $a : 5 \rightarrow 6, b : 6 \rightarrow 7$

D: ab und ac gehören zu T $\Rightarrow$ ad und ae gehören nicht zu T

H: ab und bc gehören zu T $\Rightarrow$ $\overline{bd}$ und $\overline{be}$ wegen Regel 2.

I: $\overline{ae}$ und $\overline{be}$ $\Rightarrow$ ce und de

N: $\overline{cd}$ da sonst Inzidenz$(c) > 2$ oder Inzidenz$(d) > 2$

Insgesamt wird die optimale Tour **N** wesentlich rascher gefunden, als wenn, der volle Lösungsbaum aufgebaut werden müßte.

Der obige Algorithmus kann relativ leicht verallgemeinert weren, und damit auch folgende Varianten des Problems lösen.

- unvollständige Graphen: Bei dieser Problemvariante gibt es Sackgassen im Baum. Dies kann am einfachsten dadurch simuliert werden, wenn fehlende Kanten mit Kosten gleich ∞ ausgezeichnet werden.

- gerichtete Graphen, bei denen die Kosten von der Reiserichtung abhängen.

2.4.4 Einschub zum Thema NP-vollständig

Das Travelling-Salesman-Problem erzeugt sehr große Suchräume. Vollständiges Suchen im Lösungsbaum ist von der Komplexität $O(2^N)$. Interessanterweise gibt es zum Finden des *kürzesten* Weges zwischen zwei Knoten in einem Graphen den Warshall-Floyd-Algorithmus mit der Komplexität $O(N^3)$. Zum Finden des *längsten* Weges im Graphen ist bisher aber nur das Suchen mit exponentieller Komplexität bekannt.

Man unterscheidet folgende Problemklassen

P: Probleme für die ein (deterministischer) polynomialer Algorithmus existiert.

NP: Probleme, für die es einen nichtdeterministischen Algorithmus mit polynomialem Aufwand gibt (d.h. die Überprüfung ob eine Lösung gefunden wurde ist polynomial)

NP-vollständig: Probleme in NP, auf die jedes andere Problem aus NP mit polynomialem Aufwand reduziert werden kann.

In der Praxis bedeutet dies: Wäre ein NP-vollständiges Problem mit polynomialen Aufwand gelöst, dann wären es alle. Bis dahin sind NP-vollständige Probleme schon für moderates N unlösbar.

Man betrachte als Beispiel ein Problem der Komplexitätsklasse $O(2^N)$, das auf einem bestimmten Prozessor für $N = 20$ gelöst werden kann. Baut man jetzt einen Parallelrechner mit 1000 Prozessoren, dann kann das Problem bei optimaler Parallelisierung bestenfalls bis $N = 30$ gelöst werden.

2.4.5 Näherungsweise Lösung mit Simulated Annealing

Da offensichtlich die echte Lösung von NP-vollständigen Problemen für größeres N fast ausgeschlossen ist, muß man nach Alternativen suchen. In diesem Abschnitt wird ein Algorithmus vorgestellt, der Näherungslösungen liefern kann. Die Darstellung beruht auf Kapitel 10 in Press et. al. [18].

Im Suchraum bildet das zu minimierende Funktional ein sehr zerklüftetes Gebirge. Konventionelle Minimierungsverfahren werten lokale Information aus, z.B. Gradienten die zu einem steilsten Abstieg verwendet werden. In einer zerklüfteten Landschaft führt dies aber nur zu einem lokalen Minimum. Konventionelle Optimierungsverfahren werden deshalb mit großer Wahrscheinlichkeit durch lokale Minima eingefangen.

Simulated Annealing heißt übersetzt *Simuliertes Ausglühen* und deutet bereits die eine Analogie zur Thermodynamik an, auf der der Algorithmus aufbaut. Beim Abkühlen einer Flüssigkeit verlieren die Moleküle ihre Mobilität und richten sich oft über sehr weite Strecken geordnet aus, so daß sie einen Kristall bilden, der energetisch günstiger ist, als eine amorphe Struktur. Wesentlich ist dabei aber die langsame Abkühlung. Dieser Vorgang kann — erstaunlicherweise — auch für unsere diskreten Optimierungsprobleme genutzt werden. Grundlage der physikalischen Beschreibung ist die

$$\text{Boltzmann-Wahrscheinlichkeit} : Prob(E) \sim exp(-E/kT),$$

wobei

$$k = \text{Boltzmannkonstante.}$$

Im thermischen Gleichgewicht bei Temparatur T ist die Wahrscheinlichkeit des Auftretens eines Energiezustandes durch den Ausdruck $Prob(E)$ gegeben. Das System kann also in einen Zustand höherer oder niedrigerer Energie übergehen, aber je niedriger die Temparatur ist, um so weniger wahrscheinlich ist ein Energieanstieg. Dieses Verhalten wird nun simuliert. Der Algorithmus führt zufällige Änderungen im System aus, die einer Energieänderung $E_1 \rightarrow E_2$ entsprechen, Diese Zustandsänderung hat eine Wahrscheinlichkeit $p = exp(-(E_2 - E_1)/kT)$. Falls $(E_2 < E_1) : p > 1 \Rightarrow$ Akzeptiere E_2 in jedem Fall. Die Energie wird also manchmal erhöht und immer, wenn möglich, verringert. Dies ist der *Metropolis-Algorithmus (1953)*

Grundelemente des Simulated Annealing-Verfahrens sind also

1. Beschreibung der Systemkonfiguration

2. Erzeugung zufälliger Änderungen

3. Eine Zielfunktion die einer *Energie* entspricht

4. a) Ein Kontrollparameter T, der einer *Temparatur* entspricht und
   b) ein *Abkühlungsschema*

Für den Travelling Salesman wird dies konkret wie folgt realisiert.

1. Konfiguration = Permutation der Städte.

2. Zufällige Änderungen.

   (a) Ein Teilpfad wird entfernt und mit dem Pfad in umgekehrter Folge vertauscht. siehe Abb. 2.28

   (b) Ein Teilpad wird entfernt und zwischen zufällig gewählten anderen Knoten eingefügt.

3. Zielfunktion: Kosten der Tour

4. Hier muß man experimentieren

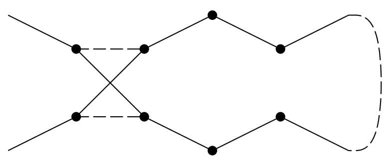


Abbildung 2.28: Vertauschen zweier Teilpfade

Zum Start des Algorithmus wird au einigen zufälligen Modifikationen ein durschnittlicher Wert ΔE für Änderungen in E bestimmt. Dann wird ein Anfangs-T signifikant größer als ΔE gewählt und für ca $100 \cdot N$ Rekonfigurationen konstant gehalten. Danach wird T um den Faktor 0.9 reduziert.

Zusammenfassend erhält man den folgenden Algorithmus.

loop

wähle ein Segment $(v_i, \ldots, v_j)$ aus $(v_1, \ldots, v_N)$
wähle ob Invertierung oder Austausch
Konstruiere daraus T_{neu}
Berechne Kosten: T_{neu} (*)
Akzeptiere neue Konfiguration mit Wahrscheinlichkeit $e^{T_{neu}/T}$
Nach $100 \cdot N$ Schritten oder nach $10 \cdot N$ Modifikationen $T = T \cdot 0.9$
Nach $100 \cdot N$ Schritten ohne akzeptierte Änderungen: break

endloop

Für den Schritt (*) ist aber eine effiziente Programmierung erforderlich, bei der die neu Energie aus der alten möglichst schnell berechnet wird. Im Detail ist dieser Algorithmus in Press et. al. [18] beschrieben. Die Programmquelle ist dort ebenfalls verfügbar.

Zusammenfassung:

- NP-vollständige Probleme, z.B. aus der kombinatorischen Optimierung, entziehen sich oft einer exakten Lösung
- Die exakte Lösung mittels Suchräumen muß durch Heuristiken beschleunigt werden
- Bei großen Problemen kann man eine näherungsweise Lösung mit *Simulated Annealing* verwenden
- Erweiterung: Multilevel-Simulated Annealing, da auch das Simulated Annealing bei wachsendem N langsam wird
- Die Simulationswelt beim Simulated Annealing besteht aus
  - Permutationen der Menge $1, \ldots, n$
  - Übergangsfunktion (zufällig)
  - Bewertungsfunktion

2.5 Chip-Layoutoptimierung mit algebraischen Mehrgitterverfahren

2.5.1 Problemstellung

In diesem Abschnitt wird ein weiteres Optimierungsproblem vorgestellt.

Die Schaltung eines IC besteht aus Transistoren, die zu Zellen zusammengefaßt werden. Eine Zelle ist dabei typisch ein NAND- oder NOR-Gatter. Typische ICs haben derzeit 10^5 oder mehr solche Zellen und werden mit weiter steigender Komplexität der Chips fortlaufend größer. Ein Beispiel ist in Abb. 2.29 gezeigt.

Das Problem ist nun, die optimale Anordnung dieser Zellen auf dem Chip zu finden. Ziel dabei ist es

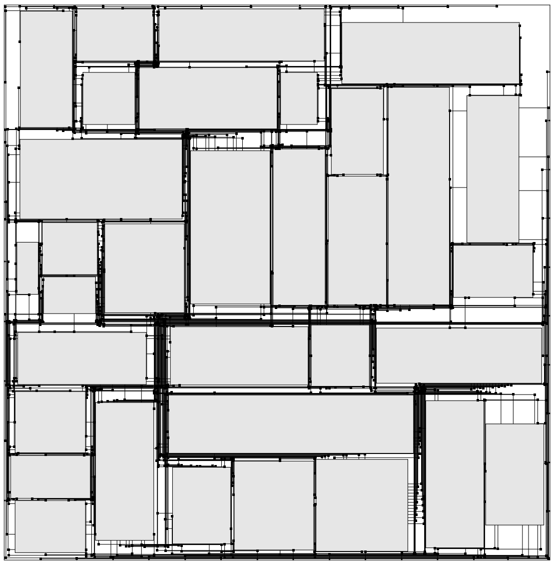


Abbildung 2.29: Plazierung und Routing eines Beispielchips

- die Gesamtfläche zu minimieren (bzw. auf gegebenem Platz möglichst viel an Funktionalität) unterzubringen und
- die Signallaufzeiten zu minimieren, da dies die Voraussetzung für hohe Taktraten ist.

Eine genaue mathematische Beschreibung der Optimalitätskriterien existiert nicht. Die Modellierung des Problems ist Bestandteil der Aufgabenstellung. Dabei ist es primär nicht das Problem, mathematisch saubere Optimalitätskrierien zu definieren, sondern algorithmisch handhabbare Ziele festzulegen und in dem Optimierungsalgorithmus einzubauen.

Fest steht aber, daß sich Zellen nicht überlappen dürfen und deshalb spielt ihre individuelle Form eine Rolle. Verbindungsleitungen dürfen sich je nach Fertigungsprozess des Chips nicht zu stark überkreuzen. Dies führt zu dem sogenannten *Routing problem*. Eigentlich müßten Plazierung und Routing simultan gelöst werden, da sie sich geneseitig beeinflussen, aber in der Praxis werden sie zur Vereinfachung der als separate Probleme gelöst.

Die Vorgehensweise hier lehnt sich an an die das Programmsystems GORDIAN an, siehe [17]. Alternativ könnten Methoden der diskreten Optimierung, so z.B. auch das Simulated Annealing Verfahren (siehe Abschnitt 2.4.5) zum Einsatz kommen.

Das Plazierungsproblem wird in zwei weitere Teilprobleme aufgespaltet, nämlich die *Relativplazierung* und die *Endplazierung*.

Die Aufgabe der Relativplazierung ist es dabei, eine gute Startlösung für Endplazierung bereitzustellen. Dazu wird eine globale Optimierung under vereinfachten Nebenbedingungen verwendet, bei der Überlappung der Zellen noch zugelassen ist.

Im zweiten Schritt wird eine lokale Optimierung zur Einhaltung der Überlappungsfreiheit eingesetzt. In diesem Schritt werden dann globale Umordnungen nicht mehr berücksichtigt.

2.5.2 Relativplazierung

In diesem Abschnitt konzentrieren wir uns ganz auf die Lösung des Relativplazierungsproblems. Die Modellierung der Optimalitätskriterien erfolgt hier mit einem *Kräftemodell*. Die Zellen sind mit *Signalen* verbunden, die Verbindungen kann man sich als elastische Gummifäden vorstellen.

Ein Chip sieht schematisch dann etwa so aus, wie in Abb. 2.30. Diese Schal-

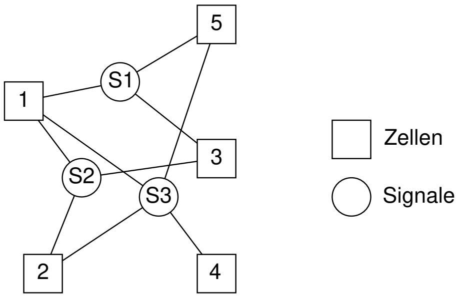


Abbildung 2.30: Zellen und Signale

tung läßt sich auch in Matrixform darstellen. Dies führt zur Matrix C in Abb. 2.31. C repräsentiert hier den Verbindungsgraphen, die Gewichtung der Signale

$$C = \begin{array}{|ccccc|ccc|}\hline 3 & & & & & -1 & -1 & -1 \\ & 2 & & & & & -1 & -1 \\ & & 2 & & & -1 & -1 & \\ & & & 1 & & & & -1 \\ & & & & 2 & -1 & & -1 \\ \hline -1 & & -1 & & -1 & 3 & & \\ -1 & -1 & -1 & & & & 3 & \\ -1 & -1 & & -1 & -1 & & & 4 \\ \hline \end{array} \begin{array}{l} \left.\vphantom{\begin{array}{c}1\\1\\1\\1\\1\end{array}}\right\} \text{M 1–4} \\ \left.\vphantom{\begin{array}{c}1\\1\\1\end{array}}\right\} \text{S 1–3} \end{array}$$

Abbildung 2.31: Verbindungsmatrix der Schaltung und Signale

kann entsprechend ihrer Bedeutung erfolgen. Diese Matrix ist schwach diagonaldominant und symmetrisch.

In der Modellierung treten feste Zellen (z.B. Anschlüsse) auf, die eliminiert werden können. Damit entstehen Zeilen in der Matrix mit starker Diagonaldominanz und C wird positiv definit.

Zu minimieren ist jetzt die Gesamtlänge der Verbindungen. Das Funktional ist also

$$E(x) = x^T C x - 2b^T x \ = \ \min, \quad (*)$$

und analog für die y-Koordinaten der Zellen. Die Minimierung von (*) führt auf das Normalengleichungssystem

$$Cx = b, \quad (b = \text{feste Module}).$$

Erster Schritt: Elimination der M- oder S-Unbekannten (ergibt C'). In realen Anwendungen sind C (oder C') sehr groß (Dimension 10^5 und mehr), dünn besetzt ($\leq 1\%$) und unstrukturiert.

Die direkte Minimierung von (*) führt zur Clusterung der Zellen in der Chipmitte. Da dies noch keine gute Ausgangsposition für die Endplazierung sein kann, muß durch die Hinzunahme von *Nebenbedingungen* eine gleichmäßigere Verteilung der Zellen über den Chip erreicht werden.

Sukzessive werden mehr und mehr Nebenbedingungen eingeführt.

Schritt 1: Schwerpunkt der Zellen (1 NB)
Schritt 2: Schwerpunkt der *linken Zellen* liegt in der Mitte der linken Chiphälfte, analog für rechts (2 NB)
Schritt 3: Partitionierung oben/unten (4 NB)
Schritt 4: Partitionierung rechts/links (8 NB)
...
Schritt m: 2^{m-1} NB

Der Fortgang der Plazierung bei immer weiterer Hinzunahme von Nebenbedingungen wird in Abb. 2.32 und 2.33 gezeigt.

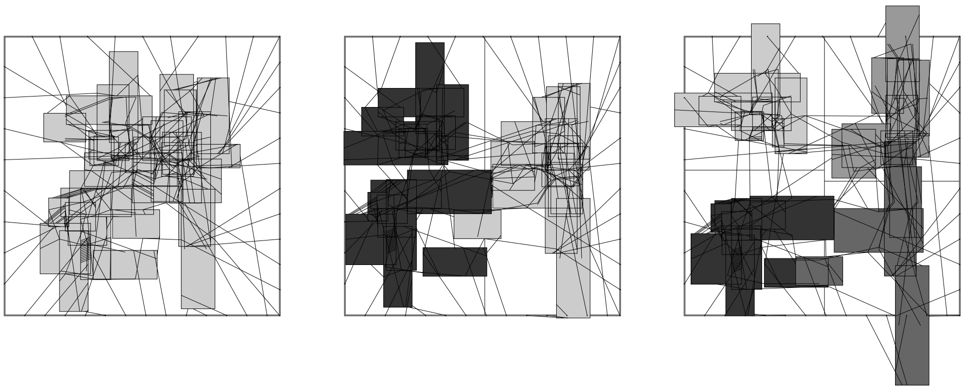


Abbildung 2.32: Optimierung mit 1,2 und 4 Nebenbedingungen

In der Praxis wird in den Schwerpunktsbedingungen die Größe der Zellen mit berücksichtigt, so daß eine wirklich gleichmäßige Verteilung über den Chip erreicht wird. In Matrixform lssen sich die Nebenbedingungen wie folgt schreiben.

$$A^{(m)} \in \mathbb{R}^{2^{m-1},n} \quad \text{, Zeilen orthogonal}$$

$$NM^{(m)} : A^{(m)}x = d$$

Einige Eigenschaften der Verbindungsmatrix C sind

$$C = (\gamma_{ij}), \quad \gamma_{ij} \leq 0, 1 \leq i, j \leq n \quad (i \neq j)$$

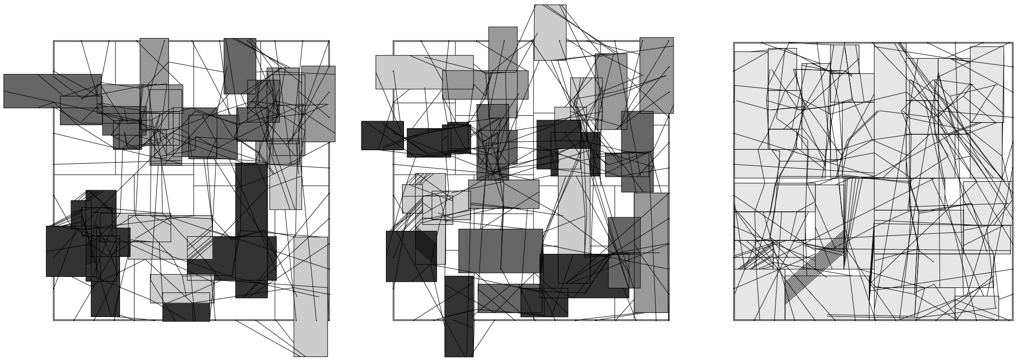


Abbildung 2.33: Optimierung mit 8 und 16 Nebenbedingungen und das Resultat nach der Endplazierung

$$\sum_{j=1}^{n} \gamma_{ij} \geq 0, \quad 1 \leq i \leq n \quad (M)$$

(Über-)Relaxationsverfahren zur Lösung des quadratischen Minimierungsproblems konvergieren, sind jedoch für $n \sim 10^5$ zu langsam. Deshalb muß nach schnelleren Alternativen gesucht werden. Dabei eignen sich zunächst viele der in Abschnitt 2.3 besprochenen Iterationsverfahren, z.B. CG-Verfahren mit ILU-Präkonditionierung. Die Nebenbedingungen werden dabei durch Projektionen bzw. Lagrange- Multiplikatoren realisiert.

2.5.3 Algebraische Mehrgitterverfahren

Dieser und der folgende Abschnitt sind noch unvollstaendig!

Im Rest dieses Abschnittes soll dazu eine Alternative vorgestellt werden, die auf der Idee der Mehrgitterverfahren beruht. Die Ideen und Ergebnisse beruhen auf der Arbeit [19].

Im Unterschied zu den in Abschnitt 2.3.4 vorgestellten Mehrgittermethoden liegt hier keine partielle Differentialgleichung und keine Gitterstruktur einer Diskretisierung zugrunde. Die Matrizen sind deshalb zwar dünn besetzt, aber unstrukturiert. Ein typisches Beispiel wird in Abb. 2.34 gezeigt. Deshalb wird ein Mehrgitteralgorithmus benötigt, der keine geometrische Information verwendet, sondern allein auf der Systemmatrix aufbaut. Grundlage hierfür ist ein Algorithmus, der von Ruge und Stüben in [20] vorgestellt wurde.

Die Grundidee ist, ausgehend von der Matrix $C = C^{(1)}$ eine Folge kleinerer Matrizen und die dazugehörigen Transferoperatoren zu konstruieren.

$$C^k, k = 1, \ldots, K, \quad C^1 = C, \quad \dim C^k = n_k, \quad n = n_1 > n_2 > \ldots n_K$$

Die Transformationsmatrizen sind von der Form

$$I_k^{k+1} : \mathbb{R}^{n_k} \to \mathbb{R}^{n_{k+1}}, \quad I_{k+1}^k = (I_k^{k+1})^T$$

$$C^{k+1} = I_k^{k+1} C^k I_{k+1}^k$$

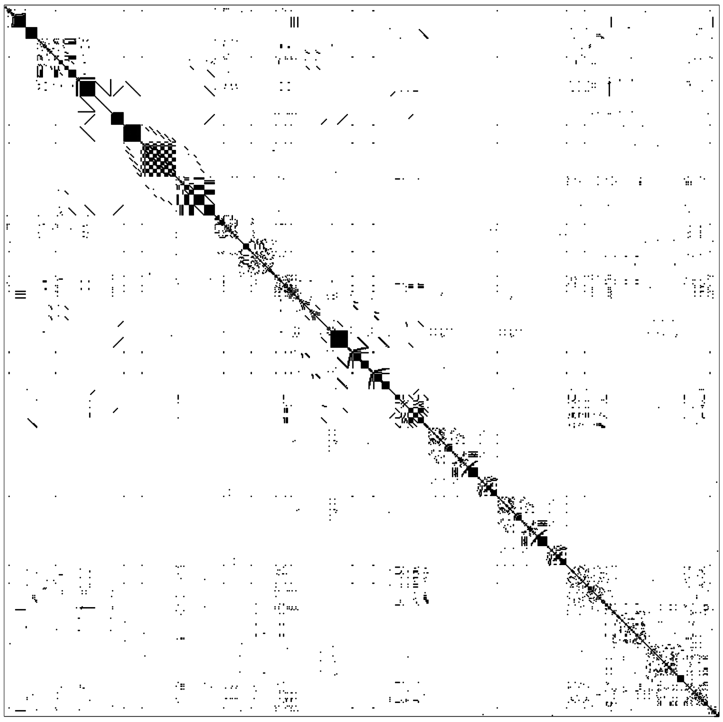
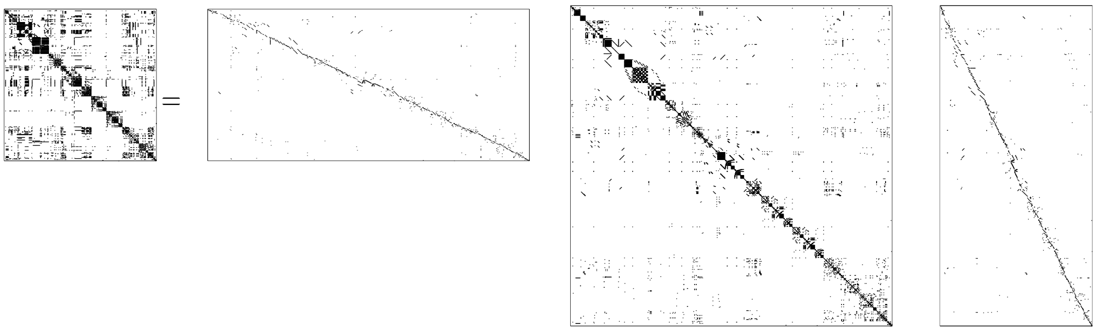


Abbildung 2.34: Besetzungsmuster der Verbindungsmatrix für einen Beispielchip mit 750 Zellen



$$C^{(2)} = I_1^2 \quad C^{(1)} \quad I_2^1$$

Abbildung 2.35: Galerkinprodukt zur Vergröberung von $C^{(1)}$ zu $C^{(2)}$

Für das Beispiel in Abb. 2.34 wird das Resultat in Abb. 2.35 gezeigt. Die *Interpolationsmatrizen* I_{k-1}^{k} werden dazu in einer Setup-Phase berechnet.

Der AMG-Algorithmus folgt nun dem normalen rekursiven Ablauf eines Mehrgitterverfahrens.

AMG($C^{(k)}, x^k, b^k$) {
  if ($k < K$)
    Relaxiere $C^{(k)}x^k = b^k$
    $b^{k+1} = I_k^{k+1}(b^k - C^{(k)}x^k)$ (*)
    AMG($C^{k+1}, x^{k+1} = 0, b^{k+1}$) (*)
    $x^k + = I_{k+1}^k x^{k+1}$ (*)
  else
    löse $C^{(k)}x^k = b^k$ direkt
  endif
}

Die mit (*) markierten Schritte können als näherungsweise Lösung von

$$\min_{v^{k+1}\in V^{k+1}} (x + v^{k+1})^T C(x + v^{k+1}) - 2b^T(x + v^{k+1})$$

$$V_{k+1} = \text{Range}(I_2^1 I_3^2 \ldots I_{k+1}^k), V_1 = \mathbb{R}^n \supset V_2 \supset V_3 \ldots$$

interpretiert werden. Die Relaxation kann interpretiert werden als

$$\min_{\gamma\in\mathbb{R}} (x + \gamma_{e_i})^T C(x + \gamma_{e_i}) - 2b^T(x + \gamma_{e_i})$$

$$e_i = i - \text{ter Einheitsvektor}$$

Rekursiver setup: $\Omega = \{1, \ldots, n_k\}$

Ziel: Partitionierung $\Omega = C \cup F, C \cap F = \emptyset$

C-Unbekannte werden auf Level $k+1$ übernommen

F-Unbekannte werden auf Level $k+1$ nicht übernommen sondern durch Interpolation angenähert

$$|F| \le 0.5|\Omega|$$

Notation:

- u_j^{k+1} „entspricht" $u_{i(j)}^{k+1}$
- Zur Vereinfachung: Numerierung so, daß $i(j) = j$
- $N_i = \{i \in \Omega, j \ne i, \gamma_{ij} \ne \emptyset\}$ (Superskript k unterdrückt)
- $s_i = max\{-\gamma_{ij} | j \ne i\}$
- $(i, j) = \begin{cases} -\gamma_{ij}/s_i, & s_i \ne 0 \\ 0 & \text{sonst} \end{cases}$
- $d(i, P) = \sum_{j\in P} d(i, j)$

- $S_i(\alpha) = \{j \in N_i | d(i,j) > \alpha\}$ „Starke Verbindungen", d.h. Punkte, von denen i stark abhängt

- $S_i^T(\alpha) = \{j : i \in S_j(\alpha)\}$ Punkte, die von i stark abhängen

Achtung: Obwohl C symmetrisch, $S_i^T \neq S_i$ und $j \in S_i(\alpha) \not\Rightarrow i \in S_j(\alpha)$
Partitionierung nach zwei Kriterien:

1. Jeder Punkt, von dem ein F-Punk stark abhängt, mus entweder zur Interpolation verwedet werden, oder selbst von den Interpolationspunkten stark abhängen

$$\forall\ i \in F\ \exists\ S_i^I \subset C \cap S_i\ :\ \forall j \in S - i\ :\ j \in S_i^I \quad \text{oder} \quad d(j, S_i^I) > \alpha$$

2. Die Abhängigkeit der C-Punkte untereinander sollte so schwach wie möglich sein

Ist die Partitionierung erfolgt, dann ist I_{k+1}^k definiert durch

$$v_i^k = \begin{cases} v_i^{k+1} & (i \in C) \\ \sum_{j \in S_i^I} \frac{\gamma_{ij} + c_{ij}}{\gamma_{ii} + c_{ii}} v_j & (i \in F) \end{cases}$$

$$\text{mit} \quad c_{ij} := \sum_{k \neq S_i^I, k \neq i} \frac{\gamma_{ij}\gamma_{kj}}{\gamma_{ki} + \sum_{l \in S_i^I} \gamma_{kl}} \quad (j \in S_i^I \cup \{i\})$$

Probleme: Eigenschaft (M) nicht notwendig erfüllt für $C^k, k > 1$

2.5.4 AMG Erweiterungen

- Nebenbedingungen

$$A^k x^k = d^k \ \rightarrow\ A^k I_{k+1}^k x^{k+1} = d^k$$

Problem: Nebenbedingungen können inkonsistent werden. I_k^{k+1} wählt aus dem $\mathbb{R}^{n_k}$ einen n_{k+1}-dimensionalen Unterraum, der zu den NB inkompatibel sein kann. Abhilfe: Setup wiederholen und NB berücksichtigen: Interpolation nur durch Punkte aus der gleichen Partition.

- Auffüllung der Grobgittermatritzen. Weglassen kleiner Elemente (Störungstheorie)

Präkonditionierung mit additivem AMG:

$$\tilde{I}_k = I_k^{k-1} I_{k-1}^{k-2} \dots I_2^1$$

Betrachte

$$P = \sum_{k=1}^{K} \tilde{I}_k D_k^{-1} \tilde{I}_k^T C^1 \quad \text{mit}$$

$$D_k = diag(C^k) \ \ (\text{oder ähnlich}), k = 1, \dots, K-1 \quad D_K = (C^K)^{-1}$$

P ist nicht mehr dünn besetzt, kann aber effizient berechnet werden. $cond(P) << cond(C)$, deshalb ist $\sum_{k=1}^{K} \tilde{I}_k D_k^{-1} (\tilde{I}_k)^{-1}$ als Präkonditionierer geeignet.

Die Summe kann parallel ausgewertet werden. Das hat aber nur einen Sinn, wenn die Multiplikation mit D_k eine *relativ teure* Operation ist.

2.5.5 Zusammenfassung

- Multilevel-Divide & Conquer-Ansatz für ein Teilproblem der Layoutoptimierung
- AMG als Multigrid für beliebige Matritzen
- Additives AMG als (parallelisierbarer) Präkonditionierer
- Setup mit intensiver Verwendung von Mengenoperationen
- Dünn besiedelte Matritzen

Kapitel 3

Computer Architecture

In this chapter and in chapter 4 below we will discuss aspects of *computer architecture* that are relelvant for the development of numerical simulations. This first chapter focuses on single processor machines. Chapter 4 will then be devoted to parallel computers.

The core topic of computer architecture is the *functional design* of computing machines. The knowledge of the primary computer design principles is of course relevant for the software developers and numerical analysts, because a particular computer architecture may have an essential influence on the performance of a particular algorithm. To the author's experience, the impact of the hardware features for the performance of a numerical simulation is often underestimated and better simulation programs would be written if mathematicians and software developers were more aware of the machine(s) they are working with.

In particular, the trend to more complex computer designs makes a thorough understanding of the basic principles increasingly important for the development of efficient simulation codes. This is reflected in the growing importance of parallelization and parallel algorithm development. Of similar importance, however, are certain aspects of the RISC (see 3.1.2) paradigm, like superpipelined and superscalar central computing units (CPU) (see 3.3.1), and cache based memory designs (see 3.2), which will be one of the main issues discussed in this chapter.

Recommended accompanying literature for this and the following chapter are the books by Giloi [12] and Hwang [16]. On the net we recommend the chapter on computer architecture[1] in the *Computational Science Education Project*[2]. Other sources with interesting hardware related information are the Internet information services of the major manufacturers.

- Cray Computer Corporation[3]
- Cray Research[4]

[1] http://csep1f.phy.ornl.gov/ca/ca.html
[2] http://csep1.phy.ornl.gov/csep.html
[3] http://www.craycos.com/
[4] http://www.cray.com/

- Digital Equipment Corporation[5]
- Hewlett-Packard[6]
- IBM RISC System/6000 Products & Services[7] and IBM High-Performance Computing[8]
- Silicon Graphics[9]
- Sun Microsystems[10]

Another useful source of information is the *CPU Info Center*[11].

In general, the performance of a computer is determined by three aspects

$$\left.\begin{array}{l}\text{Performance of the Components}\\ \text{Functional Design}\\ \text{Systems Software}\end{array}\right\} \Rightarrow \text{Overall Performance}$$

Though many computer design principles have been proposed over the years, the basic machine is still the Von-Neumann computer, as shown in Fig. 3.1. Though the von-Neuman architecture has been refined in many ways, the organization of a computer into CPU and memory is still the basic organizational principle.

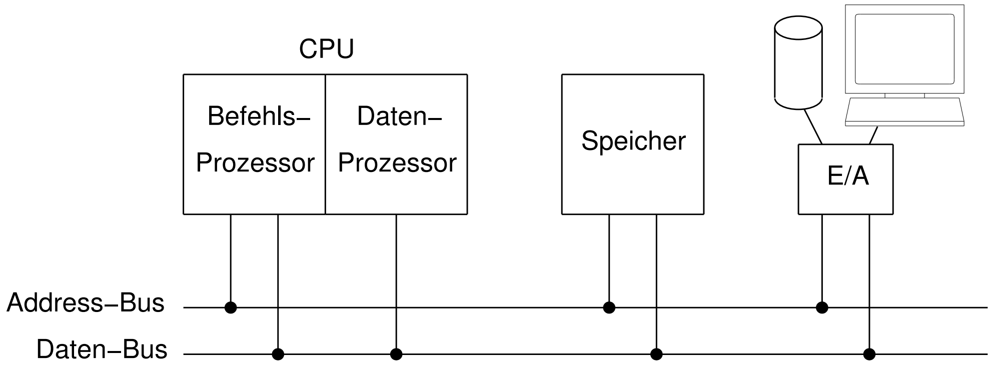


Abbildung 3.1: Von Neumann Computer

3.1 State of the art for hardware components

3.1.1 Processors

In this section we discuss the state of the art and current trends in the development of *central processing units* (CPU).

The conventional design philosophy distinguishes between

[5] http://www.digital.com/
[6] http://www.hp.com/home.html
[7] http://www.austin.ibm.com/
[8] http://ibm.tc.cornell.edu/index.html
[9] http://www.sgi.com/
[10] http://www.sun.com/index.html
[11] http://infopad.eecs.berkeley.edu/ burd/gpp/cpu.html

- VHSIC, very high speed integrated circuits, and
- VLSI, very large scale integrated circuits

VHSIC is typically based on expensive, but fast technology, like ECL (emitter-coupled logic) or GaAs (gallium arsenide) based chips which lead to high clock rates. VLSI chips, in contrast, are typically based with inexpensive CMOS technology. Though the clock rates for such chips may be slower, the high integration rate allows to put significantly more functionality on a chip. With present CMOS technology, several million transistors can be integrated on a single chip. The price-performance rate is also influenced by the number of chips produced, and this gives CMOS based microprocessors a tremendous cost advantage, since they are typically mass products. This is in sharp contrast to specialized high performance (vector) processors which may be produced only in a few hundred units. Additionally, CMOS (and BiCMOS) now reach up to 300 MHz clock rates (as for the new Alpha 21164 chip[12]) and thus have clearly begun to catch up with the typical VHSIC technologies in terms of clock speeds.

However, in oder to utilize the price-performance advantage of VLSI technology, it becomes increasingly important to exploit the (quantitatively) large functionality of these devices by parallelizing the execution. This is accomplished in two completely different approachs.

- Parallelism on the chip may be used by pipelined and superscalar execution, see 3.3.1.
- The parallelism may be made explicit to the user by building parallel computers with many CPUs, see chapter **??**.

From this point of view, the two chapters on computer architecture in this book presnet just two sides of the same problem. Both the single processor and parallel processor design are increasingly becoming dependent on parallelization concepts. The technological progress has made it necessary to study parallelism on both the single CPU as well as the parallel processor level. Only by exploiting this implicit or explicit parallelism, will it be possible to utilize the ci=ontinuing technological development for more powerful numerical simulations. While explicit parallel computing is a main stream of the current scientific development, the implicit parallelism in all modern microprocessors has not received as much attention in the user community. In the remainder of this chapter we will present some aspects which are important also for the user to better understand the performance characteristics of modern computers and to understand current and future questions that this will pose for algorithm developers.

Processor performance is measured in *Million Instructions per Second* (MIPS) which the processor is capable to perform. However, this is a very crude estimate. It depends not only on a certain instruction mix (containing a certain combination of quick and slow instructions), it also depends on the instruction

[12] http://www5.informatik.tu-muenchen.de/persons/ruede/vorles/wiss1/sc/otherdocs/alpha1.ps

set of the processor itself. A processor with a powerful instruction set may have a lower MIPS number than a competitor with only simple instructions which has a higher MIPS number but is eventually slower because it must execute a larger program to perform the same task.

More basic and less vague is the *clock rate* for characterizing a processor. It is typically given in MHz, the number of basic switching steps (in Millions) per second. It is of course equivalent to cite the basic switching time, typically in Nanoseconds = 10^{-9} sec.

The MIPS number can be related to the clock rate by introducing the *cycles per instruction* (CPI). The performance P in MIPS is related to clock rate T and CPI by

$$P = T/\mathrm{CPI}$$

3.1.2 CISC and RISC

In the past decade, two different CPU design philosophies have been much debated. The traditional design is the so-called *complex instruction set computer*, which is characterized by

- a variable and rich set of instructions (> 200), in particular with respect to memory access with
- various indirect addressing modes (> 10)
- often microprogrammed
- needs several cycles per instruction (CPI > 1)
- attempts to support high level languages on the instruction level

The alternative are the so-called *reduced instruction set computers* (RISC), which are designed with the guidelines

- small instruction set (< 100) with
- only 3-5 addressing modes
- performs operations only in registers, thus it is a
- load-and-store – architecture
- is not microprogrammed
- the CPI is low, nowadays often < 1 through superscalar execution.

The original rationale for introducing RISC was machine language statistics which showed that many instructions of a CISC computer were used only quite rarely. Thus a large portion of the CPU (in terms of total functionality) was spent for instructions which were of only little use to the user. In a classical RISC design, such instructions are left out systematically, and the available resources are used to speed up the implementation of the frequent instructions.

Clearly, the art is to find the right compromise, and modern RISC designs now typically have again more than 100 instructions.

Therefore the second characteristic, being a load-and store architecture may be the more important one. separating data transfer between memory and CPU from the operations makes it possible to rearrange instructions in the optimizer or scheduler and therefore improves the potential for overlapping parallel execution.

Of course it is important to keep the instruction set so simple that the need for microprogrammed instructions and the overhead of another level of interpretation is eliminated.

Presently, CMOS based devices, like the fastest chips of the DEC Alpha product line run with up to 300 MHz clock rate.

Besides the microprocessors, there are specialized high performance processors, as used in the classical vector supercomputers. These CPUs are optimized for special operations, like numerical manipulations on long vectors of uniform data. This is supported by special vector instructions. The processors are designed for the highest possible clock rates, but to achieve this, there is usually a considerable overhead in devices, including expensive cooling equipment, etc.

It should be noted that peak performances computed by theoretically achievable CPI and clock rates are very unlikely to be obtained for real life applications.

3.1.3 Memory

Presently, the typical memory chip has 16 MBit capacity, for 1995/96 the 64 MBit chip is expected. Here, 1 MBit $= 2^{20}$ Bit $\approx 10^6$ Bit. Memory is usually organized in units of Bytes (= 8 Bit) or words of 32 or 64 Bit width.

The technological trend results is a reduction of the channel lengths approximately by a factor two every three years which in turn results in a quadrupling of the memory capacities in the same period.

3.1.4 Memory access times

The most commonly used devices are dynamic random access memory (DRAM) and static random access memory (SRAM). They differ primarily in the access times which are typically

DRAM = dynamic random access memory	:	60 – 100 ns
SRAM = static random access memory	:	5 – 50 ns

DRAM is cheaper because it requires only two transistors per bit, while SRAM requires 8 (or 6) transistors per bit. SRAM is therefore significantly more expensive and is typically not used for the regular main memory.

Though the available memory capacity increases quickly with the current technological development, the access times tend to improve only slowly. One reason is the increased complexity of memory access in larger capacity memory chips.

Tabelle 3.1: Performance of floating point performance of some commonly used microprocessors

Processor	Clock rate	CPI
8086/87	5 MHz	70 Cycle/Add
80286/87	12.5 MHz	30 Cycle/Add
80386/87	33 MHz	12 Cycle/Add
80486	50 MHz	8 Cycle/Add
Weitek 4167/486	50 MHz	2 Cycle/Add
⋮	⋮	⋮
PA-RISC 1.1 (HP)	100 MHz	1 Cycle/(Add+Mult)
Dec Alpha 2064	275 MHz	2 Cycle/(Add+Mult)
Power2 (IBM)	71 MHz	1 Cycle/(2 Add+2 Mult)
SGI Power Challenge	150 MHz	1 Cycl/(Add+Mult)

Tabelle 3.2: Supercomputer Performance

Computer	clock rate	Mflops/Processor
Cray Y-MP	6 ns	300
Cray C90	4.2 ns	952
Fujitsu VP 2600/10	3.2 ns	5000
NEC SX 3/14R	2.5 ns	6400
Hitachi S-3800/180	2 ns	8000

3.1.5 General technological development

Halving the channel length on an electronic device permits a quadrupling of the functionality (in transistors). At the same time the speed typically doubles. The speed is determined by the electrical capacity of devices (which is reduced by a factor of 4 proportional to the area used) and is inverse proportional to the electrical currency. Since the width of the channels is also halved, the current is halved. In total this leaves an expected speed up of a factor of two in each channel length halving period.

The increase in number of transistors available in the design can partly be used to improve the internal parallelism of a CPU and thus improve the CPI. In total the technological progress has lead to almost a doubling of the processor performance (at least for floating point operations) each 1 to 2 years.

Table 3.1 gives some relevant data for widely used floating point hardware.

It is interesting to compare these data with performance characteristics of classical vector supercomputer CPUs. The data in table 3.2 is taken from the performance report[13] of Dongarra [7].

It should be noted that the peak rates are usually not reached byœ either supercomputers or RISC microprocessors. It is also remarkable that only the very high end processors have a significant performance advantage against RISC

[13] file://elib.zib-berlin.de/pub/netlib/benchmark/performance.ps

processors.

The key feature in the present CPU and memory performance features is a large gap in the cycle times. While DRAM access may be as slow as 100 ns, the fastest processors have cycle times of as low as 2 ns or 3.6 ns for RISC microprocessors respectively. Thus no presently available memory can support the processing speed of high end processors in a direct and naive design. The Von-Neumann bottleneck (see Backus [2]), has been a serious concern in computer science for a long time. However, now, with current high sped processors, the data transport from the memory to the CPU has become in a very serious constraint for numerical simulations. All current computer architectures involve advanced techniques to overcome the memory bottleneck.

It is also important to distinguish bewteen the *latency* and *bandwidth*. The latency measures the time to *complete* an operation, like a memory read or a multiplication. It is very difficult to improve the latency within a given technology, e.g. it is very costly to build memory modules much faster than the figures given above. The *throughput* (or bandwidth), in contrast, measures the number of operations performed per time, and this can in principle be improved by increasing the the number of functional units and providing data paths with sufficient capacity.

Therefore the typical latency for a floating point operation may be 10 basic machine cycles for a single multiplication, while the maximal throughput may be several multiplications in each cycle. Similar considerations hold for memory throughput and latency. These issues will be discussed in Section 3.2, below.

In consequence, even for numerical algorithms the *storage organization* and *data structures* are gaining in importance. In practice the best algorithm is not necessarily the one with the smallest possible number of arithmetic operations but the one where arithmetic cost is well balanced with memory access requirements. Additionally, all programs must be organized such that the CPU-internal parallelism can be exploited.

This motivates to study memory organization principles in the Section 3.2 of this chapter.

3.2 Memory organization

3.2.1 Registers

Registers are storage cells within the CPU. They are the fastest memory available, which can be accessed without any delay. This is of course only true as long as no ressource conflicts like the multiple usage of the same register occur. On the other hand, the number of registers is very limited. Modern RISC CPUs typically have 32 registers for integer data and 32 registers for floating point data. More registers are not provided, because these registers might use up to much of the space on the CPU chip and because addressing many registers might cause a bottleneck in the maximal possible clock rate Finally, the instruction set having to distinguish bewteen to many registers might become difficult to fit in the common 32 bit instruction words. Despite these arguments there is still a trend to increase the numbr of registers in a CPU. More registers give the compiler more flexibility in using the fastest memory and to avoid access to slow main memory.

The assembly language programmer must make the decision which data is stored in registers explicitly. From many high level languages the registers are not directly accessible. The most prominent exception of this is the C language where data declaration can be qualified with the register attribute. With modern compilers, this is rarely necessary or even useful, because the register allocation algorithms within the compiler usually perform a better job than the average programmer.

The more difficult jobs is to restructure a given algorithm such that the CPU together with the register allocation algorithm in the compiler can obtain optimal performance.

Registers and their optimal use are linked to the following design issue.

3.2.2 Load-and-store architecture

An additional trend in CPU architecture is the separation of memory access from data manipulations in the instruction set. All manipulation instructions ($+, -, *, /$, shift, test, ...) then only work with registers. this makes their execution faster, because they never have to wait for slow memory to supply the data. Furthermore, the separation of data access from its manipulation gives the register allocation routines more flexibility, especially, when a data dependency analysis (as done in all good optimizers) allows the rescheduling of instructions to more favorable places, see teh example in Sect. 3.3.1. Eventually, an intelligent compiler may organize the program execution such that data is being prefetched and the execution is overlapped (that means executed in parallel) with memory access. In view of the above distinction between latency and throughput, this is a key technique to gain good throughput though the latency of memory access is poor.

3.2.3 Memory interleaving

Memory has high *latency* compared to present day CPU speeds. This means that the time for a memory module to respond to a read or write request is large. Another possible bottleneck is the maximal *throughput*, that is the amount of data a memory system can provide per unit time. Memory *interleaving* is a technique to increase the throughput, though the latency may be poor. The core idea is to split the memory system into independent banks, which can answer read or write requests independently, in parallel.

Usually, this is done by *interleaving* the address space. Consecutive cells in the address space are assigned to different memory banks. An example of a four-way interleaved memory, and the mapping of consecutive data cells to it is illustrated in Fig. 3.2.

Alternatively to this *low order* interleaving, a *high order* interleaving is possible, but will not be considered here.

The problem in exploiting interleaved memory is to obtain maximal throughput. This requires a prefetching of the data. A simple loop accessing the data

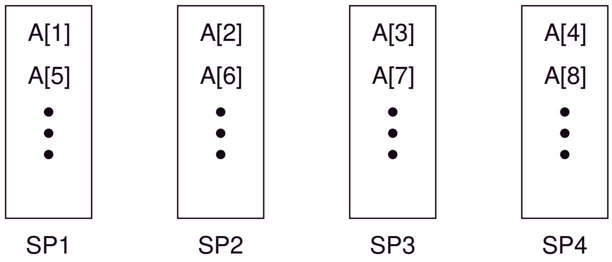


Abbildung 3.2: Four way interleaved memory

by a *stride 1* loop, like

```
        for (i=1; i<n; i++) { ... a[i] ... }
```

should be executed internally in the form

```
        for (i=1; i<n; i++) {
                start read a[i+k] from memory
                process a[i]
                finish read of a[i+1] from memory
}
```

This style of program need not be written explicitly, but is internally effected by pipelined and superscalar designs using prefetch techniques. The most extreme interleaved design is used in current vector supercomputers which may have up to 256 way interleaved memory banks. Here this is necessary to supply several high performance processors with a shared main memory with a sufficient amount of data.

Interleaving is one of the standard techniques of improving the throughput of the memory system. It brings with it a number of problems for the compiler writer and optimizer and it also has direct effects on the applications programmer.

Some of the problems are:

1. Memory access with index increment $! = 1$, as occurs naturally in matrices that are accessed both in column- and row- major ordering. In a naive memory mapping of the matrix

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

   to a linear address space with four-way interleaved memory, either the column or the row access will always hit the same memory bank and therefore have poor performance. The same is of course true for data on multidimensional grids which are relaxed in a the wrong ordering.

Structured meshes can be mapped to memory using appropriate shifts for each row, in the form

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{13} & a_{10} & a_{11} & a_{12} \\ a_{22} & a_{23} & a_{20} & a_{21} \\ a_{31} & a_{32} & a_{33} & a_{30} \end{pmatrix}$$

This now permits optimal access in both column and row major ordering. Of course, if this memory layout is not generated automatically by the compiler, and must be programmed explicitly, it leads to quite awkward indexing problems in the program. Note that it is also possible to find memory layouts which generate good diagonal access.

2. Quite difficult problems can occur when either several arrays are accessed in the same loop or the same array is accessed in several different forms. In the first case, it is sometimes possible to find an allocation pattern of the arrays such that optimal access is possible, however, in general, this involves a complicated global program analysis.

3. Memory accesses with indirect addressing e.g. of the form `a[p[i]]` as they frequently occur in sparse matrix manipulations or in unstructured grid codes.

3.2.4 Word width

The past development of computers has seen a continuous increase in the word width of the processors. The first microprocessors in the 70ies were typical 8 bit processors, then 16 and 32 bit processors followed and now 64 bit processors are becoming the standard. The advantage of a large word length is of course that potentially more data is processed at the same time, and, more data is transported form memory to the CPU. Again the price is the more complex design of CPUs and busses which must support the necessary data transfer rates.

Larger word widths are of course only effective if the data is processed word-wise. A loop which accesses data sequentially Byte by Byte will not profit from a higher word width in memory and processor, unless some (manual or automatic) restructuring is done.

3.2.5 Caches

The idea of a cache is to have an auxiliary fast memory integrated into the CPU chip. Alternatively, *external caches* are built with fast (and expensive) SRAM chips. Though a cache memory is usually much too small to replace the conventional main memory, it can hold frequently used data which can then be accessed much faster In a typical CPU design, two separate caches, say of 8 K Byte capacity each are used for instructions and data. External caches are usually larger, in workstations they typically have a size of 32K Byte up to 4 MByte.

The advantage of a cache is a significantly lower latency, as compared to the main memory, the disadvantage clearly, that it is at one to three orders magnitude smaller than the main memory. The cache is used to store copies of the data in main memory. As long as this memory is only read and left unmodified (as true for the program instructions), this is unproblematic. For data, however, which may also be modified and written back, this leads to the *cache coherence problems*. In the case of multiprocessors, where several processors may store copies of the same data item, this is a very difficult problem.

Any memory access operation of a system using a cache must distinguish bewtween two cases. The acess may be memory request may be supplied from the

- cache, if the data is stored there
- or from the main memory, if the data is not stored in the cache.

The cache organization must therefore resolve the following problems.

- Uniqueness and transparency of addressing: Caches should not affect a user program (except of course by improving the performance)
- Finding a working set, that is having a mechanism to determine what the most useful data is to have in the cache.
- Data consistency with main memory

The cache efficiency is characterized by a *Cache hit ratio* H. If the latency of the cache is T_c, and the latency for main memory is T_m, then the effective (average) latency of a memory architecture with cache is given by

$$T_{\text{eff}} = H \cdot T_c + (1 - H) \cdot T_m$$

and the speed up is

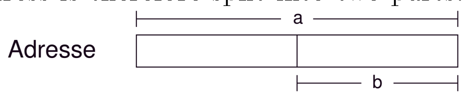
$$\mathrm{S} = \frac{T_m}{T_{\text{eff}}} = \frac{1}{1 - H(1 - T_c/T_m)}$$

For a typical scenario, where the cache is 10 times faster than the main memory, this gives a function with the qualitative dependence on H shown in Fig. 3.3. It is clearly necessary to have the hit ration H significantly larger than 50% in order to obtain good speed ups.

The transfer of information from memory to cache is organized in terms of blocks of memory, also called *cache lines*. In the following we describe a number of different cache organization principles.

Direct mapping cache

Here we assume that the memory has 2^a words, while the cache only has 2^b words, $b < a$. An address is therefore split into two parts:



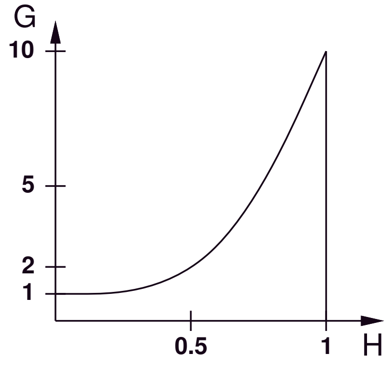


Abbildung 3.3: Speedup as a function of the cache hit ratio

This naturally defines 2^b equivalence classes for the memory cells with each 2^{a-b} elements. Now each equivalence class is assigned to one cache cell, which can therefore hold exactly one element of the equivalence class. This is depicted in Fig. 3.4. The index in the cache cell now defines, which particular element

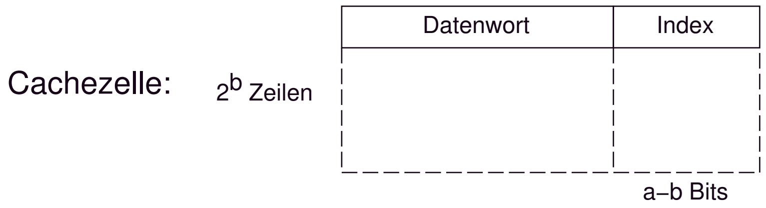


Abbildung 3.4: Direct mapped cache

of the equivalence class is presently stored in the cell.

If memory is accessed, first the corresponding cache cell is checked. If the index of this cell coincides with the address, the element is identified and a *cache hit* has occurred. Otherwise, the opreration has resulted in a *cache miss*, and the element must be transferred from main memory. At the same time the cache word is usually overwritten with the new data and index.

A common generalization of this strategy is to organize the cache not in single words, but to use a cache line size (block size) larger than one single word. This is based on the assumption that neighboring addresses have a high probability to be accessed together.

If e.g. a cache line of length 4 (words) is used, and data is accessed with unit stride, a cache miss will occur only every fourth word. If additionally the throughput of the main memory system is sufficient to transport the necessary amount of data, a significant speed up may result. However, if memory is accessed with a stride which is a multiple of 4 the result will be much unnecessary memory traffic. The cache line size has therefore a similar effect as the interleaving factor of the main memory.

The major advantage of a directly mapped cache is its conceptual simplicity and simple implementation which allows very fast access times. However, a directly mapped cache may produce very poor hit ratios, when more than one element of an equivalence class are accessed cyclically. In this case, we will never hit the right data in the cache. Cache line sizes larger than a word increase the probability of this effect for certain application types.

Fully associative cache

Unlike direct mapping, the *fully associative* cache organization offers the most flexibility in mapping cache blocks. As indicated in Fig. 3.5, each block of the main memory can be placed in any of the available cache lines of 2^d words length. Therefore each of the $2^{(b-d)}$ must have a tag of $a-d$ bits to distinguish

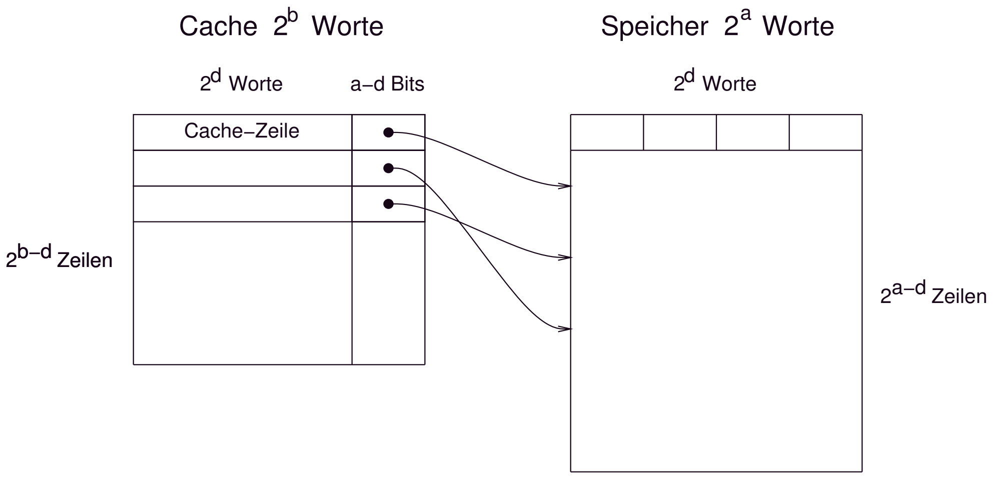
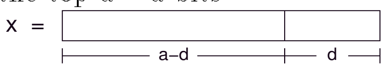


Abbildung 3.5: Fully associative cache

the possible contents.

In a fully associative cache, a $2^{(b-d)}$-way associative search is required in each memory access to determine, whether a address is cached.

Thus of an address, the top $a-d$ bits



must be compared to *all* tags in the cache. This is of course expensive and impossible for large caches in high speed designs. If a cache hit occurs, the data item can be found by using the last d bits of the address as an index into the cache line found. In the case of a cache-miss, the full a bits are used to address the main the memory.

A fully associative cache also allows great flexibility in the implementation of replacement policies which may significantly improve the hit ratios. The most common replacement strategy is the so-called *least recently used* (LRU) strategy. Whenever a cache-miss demands a new line to be loaded into the cache, the line which has not been used for the longest time is deleted to make space for the new one.

Set associative cache

This design can be viewed as a compromise between the two previous ones.

Here, the $2^{(b-d)}$ cache lines are organized into $2^{(b-d-k)}$ sets of 2^k elements. Associative addressing is performed within these sets, so that only 2^k comparisons are necessary per addressing operation. The parameter 2^k is called the *associativity* of the cache.

For $k = 1$, an example is shown in Fig. 3.6

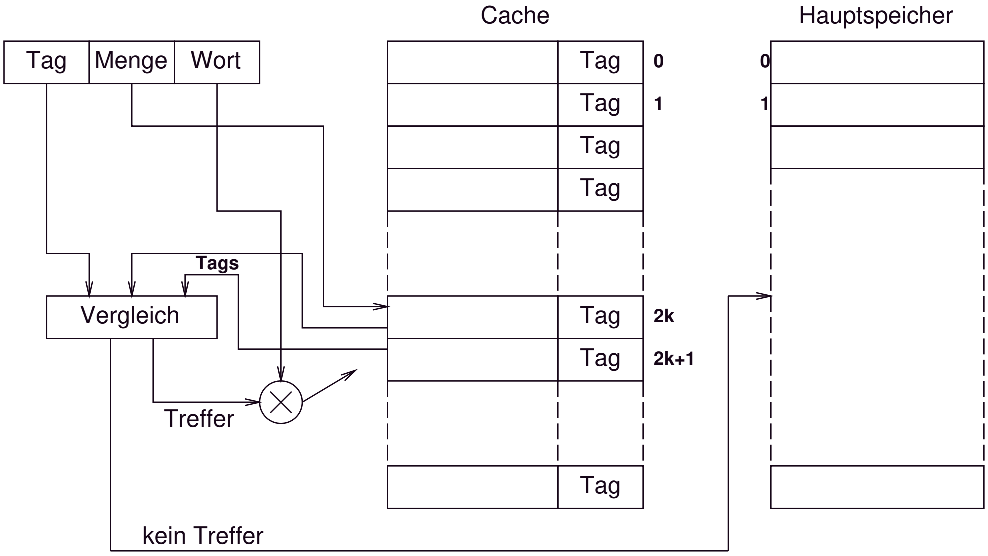


Abbildung 3.6: Set associative cache

In the extreme case of just one single set, we obtain the fully associative cache, in the case of sets with just one element $k = 0$, we recover the direct mapped cache.

Write operations in the cache

Writing data in a cached system may cause consistency problems. The simplest strategy to maintain consistency,is the *write-through* cache, where any write operation is also performed on the main memory. The alternative is a *write-back* strategy, where a cache line is written back when it has been modified (as marked by a dirty bit) and the space is required for other data.

Further aspects of cache designs

Cache design has many further aspects which cannot be discussed in detail here. We refer to the literature, e.g. as in the book of Hwang [16]. The main issues are

- Relation of cache design with the virtual memory system, primarily, whether the cache is implemented on the basis of physical addresses or on the basis of virtual addresses. The first choice requires the virtual-to-physical translation of addresses each time before the cache is used, but it is conceptionally simpler than virtually addressed caches. Virtually addressed caches have an aliasing problem, since two different processes (in particular the operating system and the user process) may use the same virtual addresses. Simply flushing the cache with each process switch is only feasible for small caches.

- Caches in (virtual) shared memory systems. In multiprocessor systems, the access to the shared memory tends to be even slower than in scalar machines, because some kind of switching network must connect processors and memory modules. This makes caches even more important to use, however, it also complicates the consistency problem. Weak consistency models have been developed which offer a compromise between performance and strict consistency of all data.
- The KSR, a typical virtual shared memory machine has had a so-called cache-only memory system, where the collection of all caches of a multiprocessor machine has served as a large global memory.

3.2.6 Storage hierarchy

The current trend in computer architecture is to use several levels of caches. The new Alpha 21164[14] chip already has two levels of on-chip internal cache, and additionally supports a third level of cache built with external SRAM modules. Viewing registers in the CPU, and the background storage of the computer system as additional layers of a memory hierarchy, we get the picture of a multi-layer memory structure of increasingly larger, but also increasingly slower memories, see Fig. 3.7. For the future it is left open to use these structures efficiently and to re-evaluate our algorithms where the *data locality* can be improved in order to improve the performance.

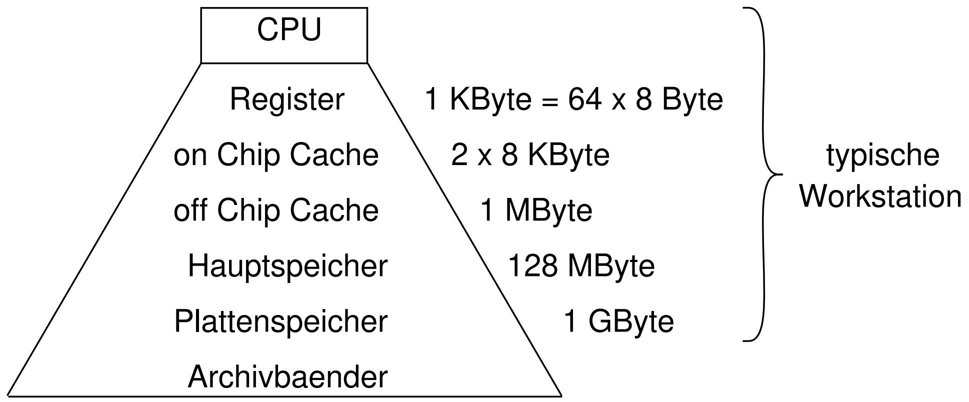


Abbildung 3.7: Memory hierarchy of a typical workstation

Besides the size, there is another distinguishing feature of the different types of memory.

- Memory may be addressed explicitly, like the registers, where either the assembly language programmer or the compiler decides where to put data.

[14] otherdocs/alpha1.ps

- Caches are used implicitly. The machine design itself decides what data is kept in the cache, and the user has only indirect control by arranging the algorithm and data access patterns.

3.3 Processor architecture

3.3.1 Superscalar and Pipeline techniques

Consider a processor with a instruction cycle split into 7 stages, like

F Load Instruction (from cache)
D decode and identify resources (registers)
I reserve registers
E_1 execute
E_2 execute
E_3 execute
W write back results

Let us assume that each stage needs exactly one cycle to execute, then a simple scalar design would need 7 cycles to execute an instruction, and the CPI for such instructions is 7.

However, now assume that each stage of the execution is performed by an independent unit of the processor. Thus, it is possible to execute instructions in a *pipelined* fashion. While the first instruction has completed stage one, and is processed in stage two, the next instruction can begin to execute at stage one. Thus instructions are passed through the CPU like on a assembly line which has 7 different stations.

Consider now an example machine language program containing two arithmetic operations together with instructions for loading the input into registers `R1` – `R6` and storing back the results. A pipelined executing might proceed as follows

```
            0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
            ===================================================
R1 = Mem(Y) F D I E E E W
R2 = Mem(Z)   F D I E E E W
R3 = R1+R2      F D X X X I E E E W
Mem(X) = R3       F X X X D X X X I E E E W
R4 = Mem(B)               F X X X D I E E E W
R5 = Mem(C)                       F D I E E E W
R6 = R4*R5                          F D X X X I E E E W
Mem(A) = R6                           F X X X D X X X I E E E W
```

Note that to execute 8 instructions without a pipeline would require 56 cycles. In the pipelined execution, instructions are blocked (marked by `X`) when their resources cannot be allocated (stage `I`). This typically happens, when the input registers for an instruction are not yet available. A blocked instruction of course also prevents further instructions to enter the blocked stage.

Despite the blocked instructions, the execution time has now been reduced to 25 cycles. This, however, is not a fair measure of the execution time, because a program like this would hardly be used stand-alone. Further instructions could of course be issued much earlier than after 26 cycles, namely right after the issue of the last instruction of the above sequence. In that measure from start of the first to the start of the last instruction, the execution time is only 15 stages.

However, since each instruction following the above sequence would be blocked, this measure is somehwat too optimistic and we may measure the length of the execution time from the termination of the first to the termination of the last instruction. This results in a measure of 19 cycles. If we acceot this as a fair measure of the pipelined execution time, we have obtained a speed-up of almost 3 versus the non-pipelined execution.

Reordering operations

A closer look at the code shows that the operations can be reordered to obtain an equivalent code. Since in the original version, the execution had to be halted, when data (from a slow memory) had not yet arrived, we attempt to issue the memory transfer instructions as early as possible. This increases the likeliness for instructions to find their data ready in the registers. A transformed code may then execute like this

```
                   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
                   ===================================
R1 = Mem(Y)        F D I E E E W
R2 = Mem(Z)          F D I E E E W
R4 = Mem(B)            F D I E E E W
R5 = Mem(C)              F D I E E E W
R3 = R1+R2                 F D X I E E E W
R6 = R4*R5                   F D X X I E E E W
Mem(X) = R3                    F X X D X I E E E W
Mem(A) = R6                      F X X D X X I E E E W
```

This instruction sequence executes in 18 cycles overall, but from the termination of the first to the termination of the last instruction, it is only 11 cycles. If we take this as the measure of the execution time, we see a speed up of more than 5 with respect to the non-pipelined version, and still a speed up of 70% with respect to the original pipelined code. The pipeline is still blocked, however, now primarily, because data is not yet ready to be written back or to be processed further. In a longer code sequence chances are, that the write instructions could be further deferred, and thus the performance might be be even improved even more.

The above example clearly shows the importance of pipelining, and the importance of the load-and-store design which makes the rescheduling of processing and memory transfer operations possible. In all current designs, the optimizer, whose job it is to reorder the instructions, is of vital importance.

3.3.2 Supporting principles

In a high performance CPU design, additional principles are used to speed up the performance. Again, we cannot discuss them in detail, but we refer the reader to special literature (e.g. Hwang [16]) for a more detailed discussion.

- Prefetching of instructions (several sequences in the case of jumps)
- Data forwarding, e.g.
  `Mem(M) = R1, R2 = Mem(M)` is transformed to `Mem(M) = R1, R2 = R1`
  `Mem(M) = R1, Mem(M) = R2` is transformed to `Mem(M) = R2`
- multiple functional units, that is a superscalar design
- dynamic instruction scheduling, which may results in a instruction reordering as above, only that it is performed dynamically after decoding the instructions.

3.3.3 Jump optimization

Jumps are potentially a major problem for any internally parallel processor, in particular also for pipelined processors.

A jump changes the flow of instructions, and therefore instructions whose execution may have begun in the pipeline are useless. After the jump it takes a number of cycles until the pipeline is again filled and operating efficiently.

Many advanced CPU designs therefore attempt a *jump prediction*, which is either done

- statically, by the compiler. This is quite successful in many cases. E.g. loops are coded using jumps, and a good first guess is that the loop will be executed multiple times, and this is the most likely jump to be taken.

```
                                    i=0
                                    ...     <---
for (i=0; i<n; i++) {               X          | Jump
    X                      ->       i++        | is
}                                   test(i-n)  | likely
                                    jump <0 ---
```

- Alternatively delayed jumps may be introduced. The idea here is that jumps are executed with some delay, so that the first few instructions following the jump are always executed. This is only effective, when it is possible to move jumps ahead, that is, if there is some flexibility in reordering the instructions.

3.3.4 Further developments

- Superscalar pipelines. These designs have multiple pipelines to work on several instructions simultaneously. Many current RISC processors already have a *multiply-and-add* instruction, which performs the two operations simultaneously in separate pipelines. Generally, this leads to an instruction execution similar the scheme in Fig. 3.8.

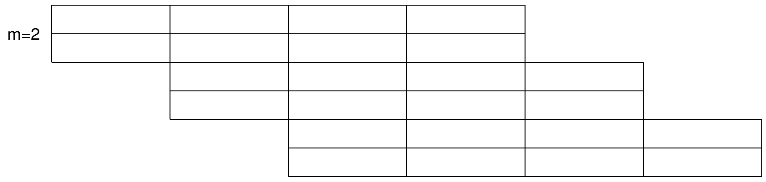


Abbildung 3.8: Superscalar pipeline

- An alternative design are superpipelined architectures, where multiple functional units are used to issue instructions at a very high clock rate, as shown in Fig. 3.9.

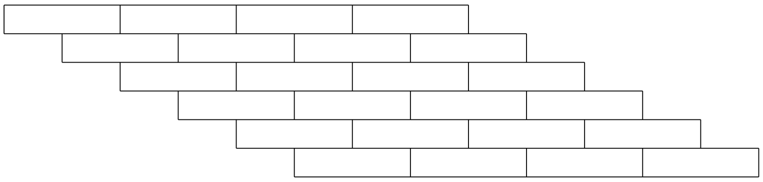


Abbildung 3.9: Superpipeline

- Finally, the two can be combined to the superscalar superpipeline.
- The idea of the very long instruction word (VLIW) idea is to have an instruction set which can decode multiple instructions at the same time, say 2 integer and two floating point operations. The instruction flow then takes a form as in Fig. 3.10.

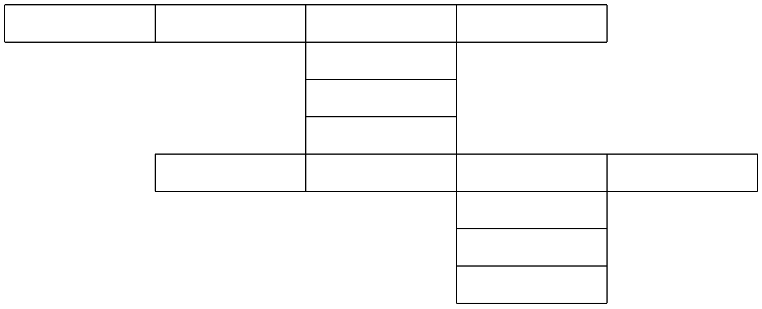


Abbildung 3.10: Very long instruction word execution

3.3.5 Further aspects

There are may other design aspects which influence processor performance. We list a few.

- Instruction word format. While older designs used variable length instruction words, which allow for shorter codes, all new designs use a fixed format. Typically, each instruction is a 32 bit word. This simplify instruction (pre-)fetching and decoding, though it leads to considerably longer programs and thus more memory traffic at execution time.

- A particularly elegant register organization is used in the Sparc architecture. To support fast subroutine calls, the registers are organized in overlapping windows, as indicated in Fig. 3.11. This simplifies procedure

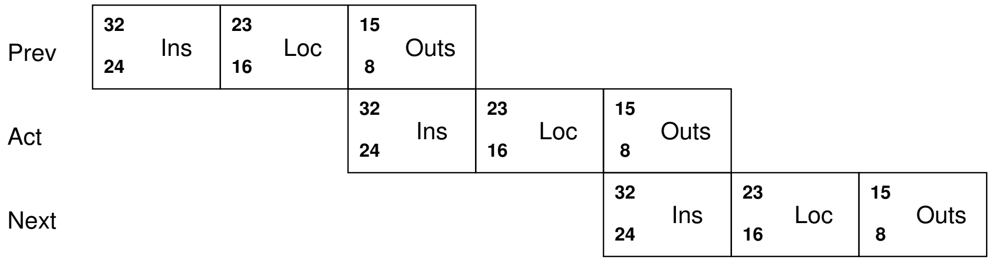


Abbildung 3.11: Overlapping register windows

calls by avoiding the need to save and restore registers on subroutine calls and return. It also has an particularly elegant means of passing arguments into the subroutine, by storing them in the *out* and receiving them in the *in* registers.

3.3.6 Vector computers

Though several manufacturers produce vector computers, the name Cray is used almost synonymously with the vector processing paradigm.

Design principles

The basic components of a vector computer are a

- conventional (but fast) scalar processor
- which dispatches floating point instructions to one or several floating point units.
- The floating point processors are highly pipelined and therefore execute at very high clock rates.
- Memory is usually interleaved, and the busses support very high memory bandwidth (throughput), when the access patterns are uniform enough.
- The CPU may have *vector registers*, that is registers which can hold vectors of floating point data.
- Pipelines can be *chained*, as shown in the multiply-add execution in Fig. 3.12.

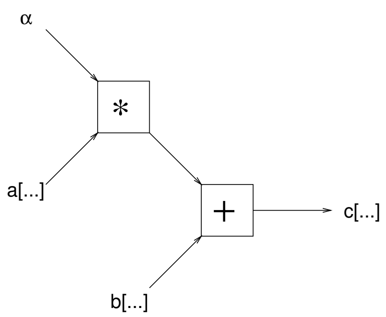


Abbildung 3.12: Multiply-and-add pipeline chaining

- These principles result in high performance for typical linear algebra operations on (dense) matrices and vectors, which permit processing with uniform memory access patterns.
- Since the performance for nonuniform memory access (e.g. when indirect addressing for sparse matrix data structure sis necessary) cannot profit from vectorization, many vector computers have a *gather/scatter* hardware which supports sparse matrix structures.
- Vector machines need (relatively) long vectors to exploit their performance potential. The typical performance in dependence on the vector length is shown in Fig. 3.13. Besides the peak speed, which is never fully obtained, a very characteristic measure is the vector length, where 50 % of the peak performance are obtained.

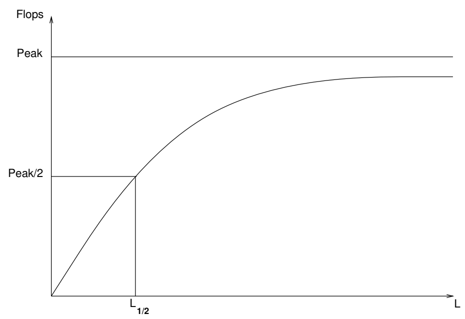


Abbildung 3.13: Vector performance depending on vector length

- The *vectorizer* (a special optimizer) transform loops of typical pattern, like

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

to vector instructions for the hardware. Many languages are not very well suited for vectorization, because determining whether a loop may be vectorized requires to make sure that no side effects can occur. FORTRAN is rigid enough to be reasonably well suited for automatic vectorization, and for FORTRAN codes very good vectorizers exist.

For languages, like C, the vectorizer must exclude the possibility that two vectors *alias*, as may occur, when a subroutine has two different vector parameters. The global program analysis is difficult and often impossible by automatic tools so that the fullmachine potential may not be accessible.

Problems of vector computers

The interleaved memory which is necessary to support the bandwidth required by the CPU is sensitive to strides, as discussed in section 3.2.3. Good vectorizers and compilers may also attempt to organize arrays and matrices in storage such that optimal strides can be used.

Despite the gather/scatter hardware, vector processors cannot support the data structures that are required by adaptive techniques with full efficiency.

High performance vector processors have multiply chainable pipes which are often not practical unless the code is restructured significantly. The classical design with just one multiply and add pipeline does not yield a better CPI than current RISC microprocessors.

Vector computers are a special architecture which is not commonly used, therefore the number produced is small, an vector computers tend to be expensive. There are basically no low cost vector computers because the advantages of vector processors become only cots effective in the supercomputer range.

Architectures not further discussed in this section are processor arrays like the ILIAC IV or processors based on systolic arrays.

3.4 Performance Evaluation of Single Processor Machines

The performance of computers is often evaluated using *benchmarks*. Generally, a benchmark is a program or a collection of programs which are run on the computers under consideration. The relative performace is then derived from the run times of the code, either given in seconds (so that small numbers represent good performance), or the figures are converted to some other metric. For the LINPACK benchmark, e.g. the result is given in MFLOPS, the number of floating point operations per second performed during the run of the program. Here of course a high number means good performance.

For numerical applications, probably still the most popular benchmark is the *Linpack-Benchmark* of Dongarra [7]. The original technical report is updated frequently and contains probably the most complete list of machines used for scientific computing, ranging from the fastest supercomputer prototypes to workstations, and PCs. The comparison of all these machines is also interesting from a historical perspective, and it is amazing to see that machines just a few years old are now historical — that is far from being competetive with newer hardware. The newest version of the LINPACK results is available electronically by ftp from *netlib*[15] (`netlib.att.com`) or one of its mirrors, e.g. the one at *elib*[16] (`elib.zib.berlin.de`) in Berlin.

The linpack results are published in a table of the form in Tab. 3.3, showing the *linpack value*, a column labelled *TPP*, and a column with the peak performance. All values are given in Megaflops.

Tabelle 3.3: Sample Linapck Results

Rechner	Mflop	TPP	Peak
Cray Y/MP	145	332	333
HP 9000/735	41	120	198
SGI Crimson	16	32	50
Compac Descpro 486/33	1.4	-	-

The linpack value itself is obtained by running a strictly prescribed FORTRAN program which solves a 100 by 100 linear system with Gaussian elimination. In this program nothing, not even the comments, must be modified. Comments might be used for compiler directives. The only things which may be modified are compiler flags and for many linpack results the compiler version and compilation flags is documented carefully.

For the TPP column the program code may be modified and even other languages (e.g. assembly language) may be used. The program is applied to a larger problem, namely the solution of a 1000 by 1000 linear system. Many manufacturers have very carefully tuned and optimized the code to their machines in order to obtain an as high as possible figure. The TPP values do not reflect a performance seen in typical applications but a performance the machine may achieve in very favorable circumstances.

The peak performance given in the last column is the optimal theoretical performance on the givens system It is calculated from clock rate and the maximal number of floating point oprations per cycle. Thus the value is like a speed of light for the given machine — a faster performance is technically impossible, and the real performce is typically much lower.

The LINPACK benchmark has a number of advantages, including its wide availability and simple use. It is typically just a matter of minutes to find the LINPACK figure for a given system once the FORTRAN compiler is ready to run.

On the other hand, LINPACK does have a number of problems. The program is very small, and though solving linear equations is certainly one of the core problems in scientific computing, it is far from being a representative program in itself. LINPACK is also restricted to dense matrices and does not measure anything related to more flexible data structures as necessary for adaptive meshes and sparse matrices.

Since the code itself, and for the 100 by 100 system also the data set are very small (for today's standards), LINPACK measures the cache performance

[15] http://www.netlib.org/index.html

[16] ftp://elib.zib-berlin.de/

of systems with hierarchical memory. Larger problems may perform significantly worse, when either the program code or the program data or both must be accessed from main memory with frequent cache misses. Since modern machine architectures (in particular parallelization and vectorization) cannot be exploited efficiently on too small problems, the 100 by 100 LINPACK results are significantly lower than those for the 1000 by 1000 table.

Two more tables in the LINACK report are of general interest. These are the *Parallel Processing* and *Highly Parallel Computing* tables which show the results of parallel processing experiments with various machines.

Other benchmarks include the Dhrystone, Whetstone and Specmark (SPECfp, SPECint) benchmarks. The Specmark values for integer and floating point performance are based on program suites and are mostly used for evaluating and comparing workstations and smaller systems. For supercomputers they tend to show a too pessimistic performance, since they contain code where supercomputers have typically relatively poor performance and which are usually not run on supercomputers.

Benchmarking of sequential computers is nontrivial, and benchmarking of parallelcomputers is even more difficult.

Kapitel 4

Parallel Computer Architecture

In this chapter we will discuss the principles of parallel computers and their impact on numerical algorithms and applications. There are so nmany and different parallel computing architectures and their specific properties have such a great impact on program development and performance that we feel the subject justifies a separate chapter. Most parallel computers are based on conventional scalar processors, as discussed in the previous chapter. So the presentation here will frequently refer back to the previous chapter. Though all modern scalar processors depend strongly on *implicit* parallelism in themselves, the performace obtained in this way is still insufficient for many of the grand challenge 1.3 problems. Therefore computer architectures with *explicit* parallelism are gaining importance. These computers are emplyoing several (from two to several thousand) scalar processors. As we will see in this chapter, the construction of these machines is far from trivial, and further chapters will show that using these architectures as a numerical application developer is far from trivial, too. The departure point for our study is Flynn's classification [9].

4.1 Flynn's Classification

In 1972 Flynn [9] introduced the basic classification of parallel machines using the flow of instructions and data as criterium. This results four classes of parallel execution which are labelled as follows.

SISD: Single Instruction Single Data
SIMD: Single Instruction Multiple Data
MIMD: Multiple Instruction Multiple Data
MISD: Multiple Instruction Single Data

In this scheme, simple scalar machines appear as SISD computers, which have both sequential instruction and data stream. The operatation of such a SISD machine is visualized in Fig. 4.1. Here a single CPU consisting of a command unit (CU) and a processing unit (PC) executes a single stream of

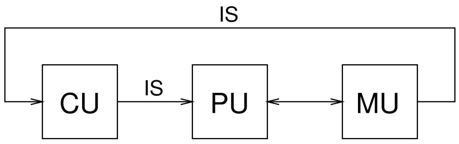


Abbildung 4.1: SISD single processor architecture

instructions on a single stream of data, which is stored in a single memory module. Of course the discussion in chapter 3 shows that there is much more to single processor machines than this simple picture.

The next type of machine is the SIMD architecture which is shown in Fig. 4.2. A SIMD machine has a single stream of instructions which is decoded

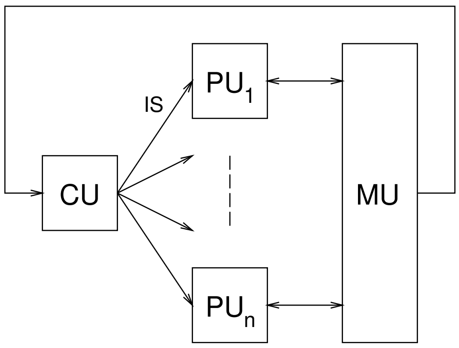


Abbildung 4.2: SIMD architecture

in a single command unit (CU). In contrast to the SISD type there are now several processing units (PU). As shown in Fig. 4.2 a multiple data stream is delivered to n processing units. In this figure there is a single memory module. This corresponds to a so called *global* or *shared* memory architecture. From the discussion in the previous chapter where we have seen that the memory performance is increasingly becoming the bottleneck for the performance of even scalar processors, we may already predict that this type of architecture must be used with a very sophisticated design of the memory access. A typical SIMD machine is the ILLIAC-IV [6] or, more recently, the CM-1 of Thinking Machines Corporation.

A prominent architecture which also fits in the SIMD model (usually with shared memory) are the vector computers as they have been discussed in section 3.3.6. However, the classification of vector computers as SIMD machines is not unique. Vector processing has features which would make it more appropriate to classify it as SISD, see e.g. the chapter on computer architecture[1] in the *Computational Science Education Project*[2]. This ambiguity is caused by how the flow of data is perceived. At any fixed time the vector processor operates on several data items simultaneously (as does any pipelined microprocessor), thus corresponds to the view of the vector processor as a SIMD machine. In a

[1] http://csep1f.phy.ornl. gov/ca/ca.html
[2] http://csep1.phy.ornl.gov/csep.html

more dynamic view, however, one can see that there is just one sequential flow of data from memory to the CPU.

The original Flynn taxonomy does not distinguish between different memory architectures. An alternative SIMD design might not have a single memory module but separate ones for each PU.Such a memory design is called *distributed* or *local*.

The *multiple instruction single data* (MISD) architecture in the Flynn taxonomy is probably the most uncommon one. A MISD computer might operate as shown in Fig. 4.3. Here, one data stream flows through a linear array of pro-

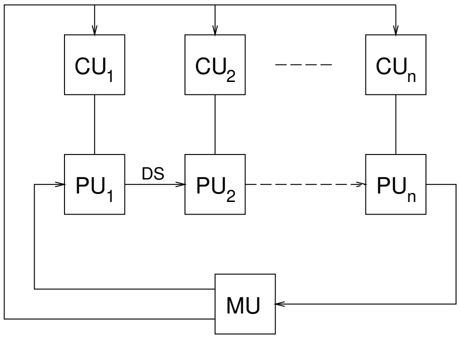


Abbildung 4.3: MISD

cessors, executing different instructions on the stream. This kind of architecture is also known as a *systolic array*. Note that vector processors again fall in the picture, where now the single processing units are interpreted as the the stages of the pipeline.

Finally, the *multiple instruction multipla data* architecture is shown in Fig. 4.4 with *local memory*. The MIMD architecture with global memory is comple-

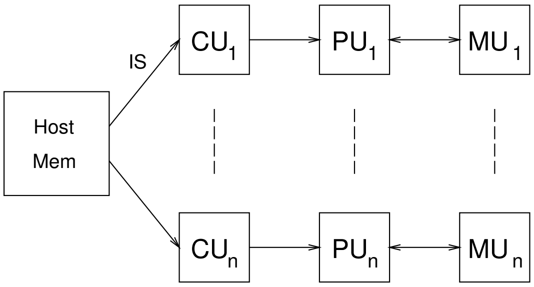


Abbildung 4.4: MIMD with local memory

tely analogous. For both global and local memory, the key feature of MIMD computers is, that n independent instruction streams are executed in n independent processing units. This is presently the most common parallel architecture. Again, a global memory architecture may suffer from insufficient memory throughput (if not designed properly), so that at least for very large processor numbers, the local memory design may be more appropriate. However, as we will see later, it may also be the most difficult one to program.

4.2 PRAM: A theoretical model of parallel computation

Before we will come to discuss real machines we briefly outline the concept of the *parallel random access machine* (PRAM) which is used as a theoretical model machine to analyse parallel processes. The basic concept is that of a multiprocessor machine with global *shared* memory according to Flynn's MIMD rpinciple, see Fig. 4.5. Each processor may execute an independent program.

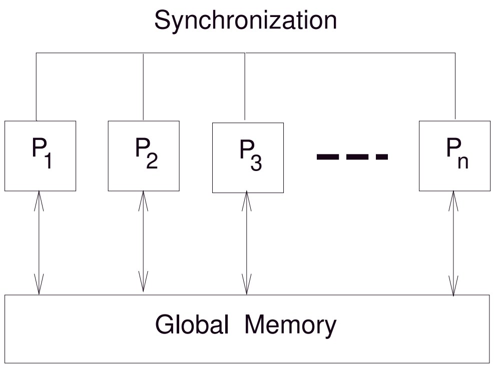


Abbildung 4.5: PRAM machine

The essential features of the concept are

- Synchornized processors with a fixed *read – operate – write* cycle. Thus it is assumed that all read (write) requests arrive at the global memory at the same time.
- uniformly addressable memory
- each processor can access any memory cell in unit time

For the memory access there are four possibilities.

ER: *Exclusive read.* At most one processor can read from one cell at a time.

EW: *Exclusive write.* At most one processor can write to a memory cell at a time.

CR: *Concurrent read.* Several processors can read a cell at the same time.

CW: *Concurrent write.* Several processors can write toread a cell at the same time.

Thus there are four different variants of PRAM machines, depending on how these possibilities are combined. The most restrictive variant of EREW–PRAM, allows only exclusive read and write access to the memory. Another common variant is the CREW-PRAM concept, where processors may read in parallel form the same cell, however only one processor may write to a cell at a time. The combination ERCW is not very common, however the most flexible concept of CRCW is the object of many studies. Here both read and write access may be concurrent, and therefore the concurrent write accesses must be somehow resolved. This leads o four more subclasses of the CRCW–PRAM machines.

- Concurrent write is only permitted, when all processors write the same value.
- The written value is chosen at random (from all write attempts)
- The value of the processor with highest priority (e.g. lowest processor number) is written, all others are discarded.
- A function of all values is computed and written. Of course this is usually a rather simple function, e.g. the *maximum* or *sum* of all values.

The PARM model is useful for theoretical studies. E.g. it is possible to prove that any CRCW algorithm can be simulated on a EREW type machine by using at most $\log(n)$ times as much time, where n is the number of processors. Generally, however, the PRAM assumptions of a uniform memory access in unit time are not realistic, and therefore real-life conditions are significantly more complicated than visible from PRAM based studies.

We conclude this section by describing, how a matrix multiplication algorithm for a CREW–PRAM machine might look like.
Example: Assume that $A, B \in \mathcal{R}^{n\times n}$ is given, and n^3 processors are available. The algorithm in Fig. 4.6 computes $C = AB$ in $\log_2 n$ steps. For the sake of simplicity we assume that the processors are indexed in the form $P_{i,j,k}$, $1 \leq i,j,k \leq n$. This algroithm also uses n^3 storage cells. In the first phase of the

```
Each processor (i,j,k):
     Read  A(i,k)
     Read  B(k,j)
     Compute A(i,k) * B(k,j)
     Store   A(i,k) * B(k,j) in C(i,j,k)
     l:= n
     Repeat
          l:= l/2
          if (k<l){
       Read C(i,j,k)
       Read C(i,j,k+l)
       Compute C(i,j,k) + C(i,j,k+l)
       Store C(i,j,k) + C(i,j,k+l) in C(i,j,k)
          }
     until (l=1)
```

Abbildung 4.6: CREW–PRAM Matrix-multiplication algorithm

algorithm all n^3 products are formed simultaneously in one step. The second phase then needs $\log_2 n$ steps to sum these values recursively. Note that in this phase most processors are idle.

It is therefore natural to modify this algorithm to use only $n^3/\log_2(n)$ processors and memory. For this it is then necessary to program phase 1 such that each processor performs $\log_2(n)$ products and at the same time computes their sum. Thus in this modified algorithm, phase 1 already requires $\log_2(n)$ steps. The second phase then has only to deal with sums of $n/\log_2(n)$ numbers.

4.3 Multiprocessors und Multicomputer

Since the memory access structure is so important for the both the programmer and the system designer it is natural to use the different memory architecures as a classification scheme alternative to Flynn's. We may distinguish between multicomputers and multiprocessors.

Multiprocessor: Shared (global) memory
Multicomputer: distributed (local) memory

4.3.1 Multiprocessors with shared Memory

Shared memory machines are usually called *multiprocessors*. They are characterized by a memory with a single uniform address space which is equally accessible to all processors. We further distingusih between *uniform memory access* (UMA) machines, where the access is totally uniform and the access times from one processor to any memory element are equally fast. Such an UMA architecture is shown in Fig. 4.7.

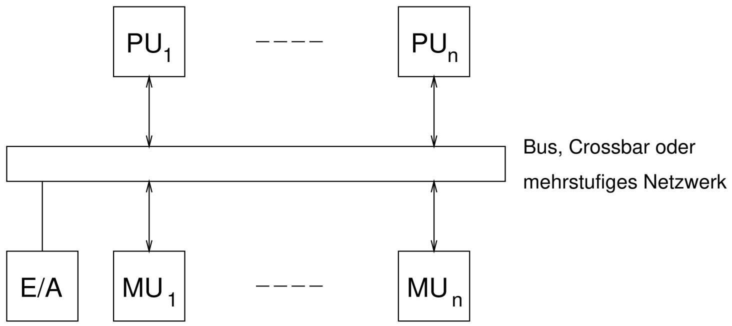


Abbildung 4.7: UMA (uniform memory access)

In programming these machines it can be exploited that there is a single address space, and though the processor structure must be used more or less explicitly by using e.g. parallel loop constructs (`forall`), quite conventional program structures can be used. In Fig. 4.8 a simple example program and its parallel m-processor equivalent are shown. Note that the transformation of the vector addition is quite simple and requires only splitting the execution in m pieces, each of which is envoked in the parallel `forall` loop.

The summation of `a[i]` is more difficult to program efficiently. Here a naive translation using a single variable `s` would be functionally correct only if the C++ *add-into* operation `+=` were defined to be *atomic*. This means that all

```
for (i=1; i<=n; i++)
    a[i]=b[i]+c[i];
s=0;
for (i=1; i<=n; i++)
    s+=a[i];
```

Execution in $2n$ cycles

```
l=n/m;
forall k=1,...,m
    for (i=(k-1)*l+1; i<=k*l; i++)
        a[i]=b[i]+c[i];
    s[k]=0;
    for (j=1; j<=l; j++)
        s[k]+=a[l*(k-1)+1];
endall
"Sum s[1..m] into s"
```

Execution in $2l$ Cycles $+ \log_2 m \cdot t_{sync}$

Abbildung 4.8: Parallelization of two simple vector operations

processors must be blocked on access to `s`, when one processor is in the process of performing the `+=` operation. Imagine what would happen if `s` were simultaneously read by all processors, each processor modifying its local copy of `s`, and then trying to write back to the original storage location. Clearly chances are that a completely wrong result would end up in `s`.

But even with an atomic `+=` operation, using just a single variable with n processors writing to it, the memory module containing `s` would become a severe bottleneck. The correct solution to this programing problem is as in Fig. 4.8 and uses a vector `s[m]` of variables, each of which is used to compute the partial sums corresponding to one of the processors.

In a terminating sequential loop the final result is computed. Note that this can be done trivially by m sequential steps. For large processor number m, this could be reduced (and should be) by using a recursive tree-like summing strategy, where first $m/2$ processors are forming the sums of pairs of values in `s[k]`, then $m/4$ processors compute the sum of pairs of pairs, etc. This requires only $\log_2(m)$ instead of m sequential steps, however, it would of course add a considerable complication to the above program. In many parallel programming systems such functionality is directly available e.g. as library routine.

Technically, a UMA machine needs a component connecting each of n processors to each of m memory modules. The simplest realization of this connecting device is a *bus*. However, since on a bus only one pair of participants can communictate at any time, a bus is acceptable only for small processor numbers. In section 4.4.3 we will discuss alternative connection devices which allow the parallel execution of several communication requests.

In any case, the UMA model in Fig. 4.7 requires each memory request to pass through a network. This poses a number of problems and restrictions. With a given technology, such a network puts a strict limit on the latencies which can be obtained. Therfore UMA architectures are usually combined with techniques for hiding the latencies, e.g. by using caches for each processor, as we will see further below. Of course this raises the question of cache coherence.

The throughput of a network can be improved by using wide enough data paths. Though this is not a fundamamental problem like the latencies, switching networks for say 128 bit wide channels are of course expensive.

To alleviate the latency and throughput problem of a a UMA machine, we note that each processor is likely to access data which is contained in only one or a few of the memory modules. A good example is the program in Fig. **??**, where each processor essentially operates only on its private segments of a few vectors. Therefore it is natural to assign the memory modules to the processors, as in Fig. 4.9. Such a design is now callled a *non-uniform memory access* (NUMA)

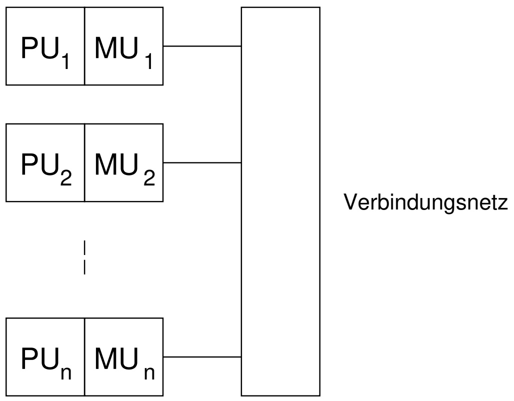


Abbildung 4.9: NUMA (non-uniform memory access)

machine. In a NUMA multiprocessor, there is still a uniform single address space, however, the access of a processor to its private memory is implemented directly and therefore much faster than the access to remote memory which is handled via the connection network. Each processor P_i can therefore address data in each memory module M_j, however, the access time depends strongly on i and j. This model can be augmented by an addtitional global memory, in which case the local memories may only play the role of a fast cache to the global main memory. In any case, the performance of such an architecture will depend strongly on how local the processors can work with their memory modules compared to the frequency of expensive remote memory accesses. The remote access times can of course be influenced by implementation, where a pure software implementation will most likely be very slow, while any hardware support will add significantly to the cost of the machine.

The final variant of this type of architecture is the so-called *cache only memory architecture* (COMA) in Fig. 4.10. Here the local memories are built as large caches for fast access by each processor, while the remote (cache) memories act as the global memory. The D_i are so-called Cache-Directories. The basic difference from the NUMA concept in Fig. 4.9 to the COMA architecture in Fig. 4.10 is not so obvious from the static figures, but the dynamics implied by the cache concept. While in a conventional NUMA architecture data is assigned *statically* to it memory module and has a unique address which remains the same for the lifetime of the variable, a COMA architecture migrates the data to the memory module, whose procesor has issued the latest request for the data. Thus a variable may still have a unique address for access, however, its physical location may be anywhere in the system at any given time. The cache directories are used to organize the access.

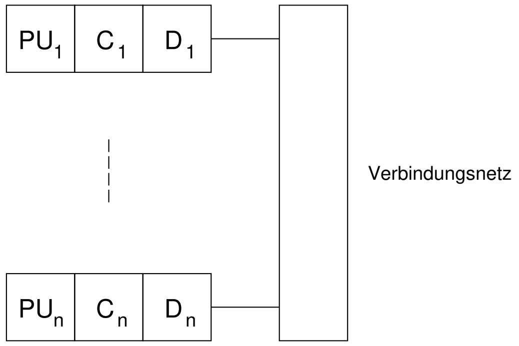


Abbildung 4.10: COMA (cache only memory access)

Since in a COMA architecture data migrates to the processor using it, such machines may have a significant performance advantage compared to conventional NUMA architectures. If the assignement of variables to the processors is static, the processor cannot adapt to different situations in the algorithm. A program may have subroutines where a different assignement of the variables would be much more suitable.

On the other hand, the data exchange in a COMA architecure between the caches is typically organized in sizes of cache lines. If now a single variable or two different variables of the same cache line are used by two different processors, this cache line must migrate back and forth bewteen the two processors for each single access. Such effects may depend on details of memory allocation and may lead to a surprising and almost unpredicatable machine behavior.

Both NUMA and COMA memory organizations are called *virtual shared memory architectures*. They both retain the still comparably simple programming model of a shared memory computer, however, they are essentially implemented with a *physically distributed* memory.

4.3.2 Multicomputers with distributed memory

Above we have seen that performnce considerations have forced us to consider physically distributed memory organizations. If this structure is made explicit and the close coupling of the processors is loosened by discarding the uniform address space, then we arrive at *multicomputers with distributed memory*, see Fig. 4.11. Now each processor is an independent single computer in many cases with even its own operating system and address space. Memory in a remote processor is not (directly) accessible. The basic communication primitive available is the data send or data receive request of a processor to the connection network. In older multiprocessor architectures the network topology was explicit to the machine and communication requests were allowed only between neighboring (directly connected) processors. Many newer designs implement communication primitives, where each processor can send directly to any other processor and the system software does the routing directly (with significant hardware support). In this sense we can distinguish between three different generations of multicomputers

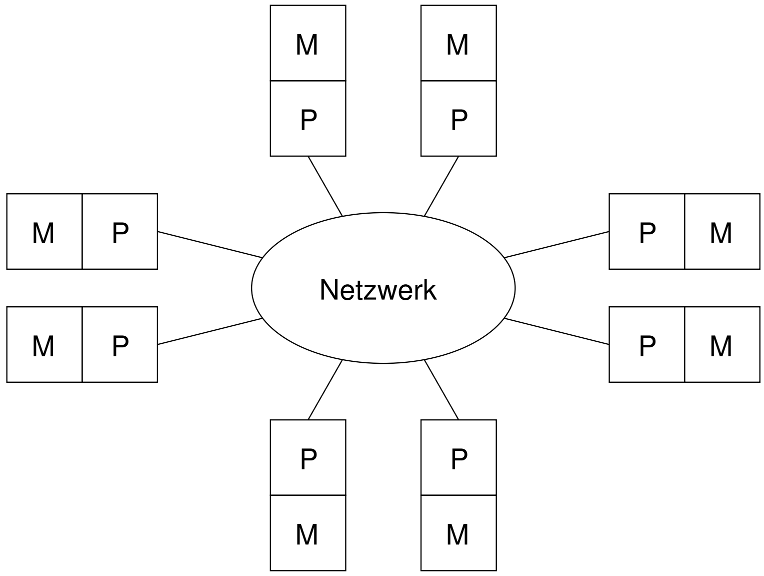


Abbildung 4.11: Multicomputer with shared memory

1. Generation:	communication & routing in software	('87)
2. Generation:	hardware-routing	('92)
3. Generation:	communication hardware on-chip	(?)

4.4 Network topologies

Though hardware routing support has freed the parallel programmer to some extent from considering the particular network architecture and the mapping of an algorithm with its data structures in a network of processors, the study of different network topologies is still of fundamental importance to understanding the difficulties arising in parallel algorithm design.

4.4.1 Static networks

In conventional multicomputers, the network is static. This means the communication channels are built in hardware and are considered as fixed, at least for the run time of a particular algorithm.

With the possibilities of fast hardware-supported routing, some parallel computers offer *virtual communication topologies*, which are simulated by software within the true physical architecture. In this case, several virtual topologies may be used simultaneously.

1. We begin by arranging the processors in a linear array and connecting them to a *chain*, see Fig. 4.12.

2. A chain is *unsymmetric* in the sense that the processors at the end of the chain have just one connection link, while all others have two. This is avoided by closing the chain to a *ring*, see Fig. 4.13. Depending on the

type of connection we may further distinguish between *unidirectional* and *bidirectional* rings.

3. In a ring with large number of processors the distance may grow very large. One way to improve the situation is to increase the *node degree*. The node degree is the number of links at the node. In the ring we only have degree two at all nodes. By using a third link at each node, shortcuts for the communication in the ring can be built. The *chordal ring* of deegree 3 is shown in Fig. 4.14.

4. By further increasing the node degree more shortcuts can be introduced. Alternatively to the chordal ring (of higher degree) we may then also consider the *barrel shifter*, see Fig. 4.15. Here, the basic ring structure with $N = 2^K$ nodes is augmented by connections between all nodes i and j, whenever $|i - j| = 2^k$ is an integer power of 2. The routing algorithms here are particularly elegant and efficient, however, the node degree will now increse with the size of the network.

5. If the barrel shifter is still insufficiently connected, we may proceed to a completely connected graph, shown in Fig. 4.16. Here any pair of processors is connected and the node degree of an N processor network is $N - 1$. For large configurations (N large) a topology requiring $N(N-1)$ connections is not feasible.

6. Since so many algorithms for scientific computing (see chapter 2) use (multidimensional) arrays as data structures, it may seem natural to reflect this structure in the connection toplogy of a parallel machine. The simplest examples are of course the linear array (see 1) and the ring (see 2). For problems in two physiscal dimensions a two-dimensional array of processors may be more suitable. Analogous to the ring, the mesh may be closed. The resulting structure is a *mesh torus*, see Fig. 4.17. For a 4×8 processor array a 3-dimensional picture of the graph is shown in Fig. 4.18.

7. If the horizontal loops are closed in a spiraling form, we obtain the so called ILLIAC mesh, see Fig. 4.19, which happens to coincide with a chordal ring of degree 4.

8. Finally, without pictures we present the idea of three dimensional meshes, and their closure to three-dimensional tori by connecting the processors into rings in the x, y, and z direction. This architecture is presently prominently represented by Cray's T3D[3] architecture.

9. Another seemingly natural simple processor configuration is the *star*, see Fig. 4.20.

10. Alteratively, the processors may be configured in a tree like structure, see Fig. 4.21 Both the star and the tree suffer from the bottleneck caused

[3] http://www.cray.com/PUBLIC/product-info/mpp/CRAY_T3D.html

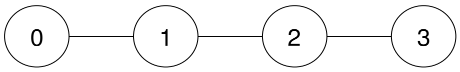


Abbildung 4.12: Processor chain

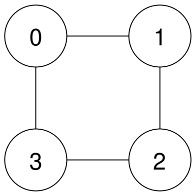


Abbildung 4.13: Ring

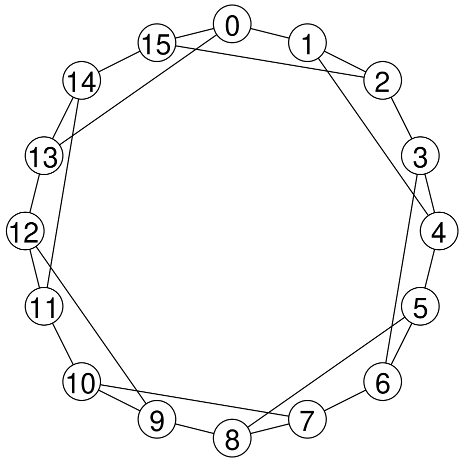


Abbildung 4.14: Chordal ring of degree 3

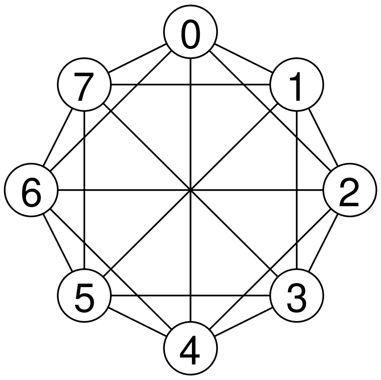


Abbildung 4.15: Barrel shifter

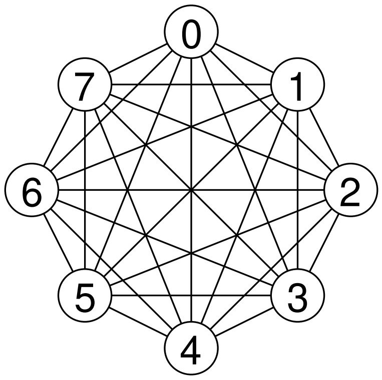


Abbildung 4.16: Completely connected mesh

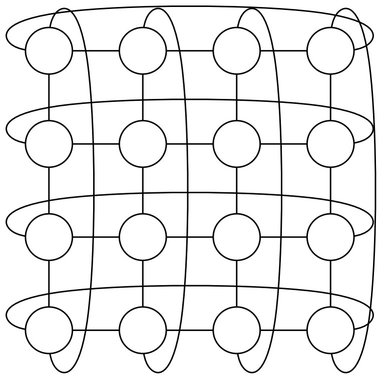


Abbildung 4.17: Mesh and Torus

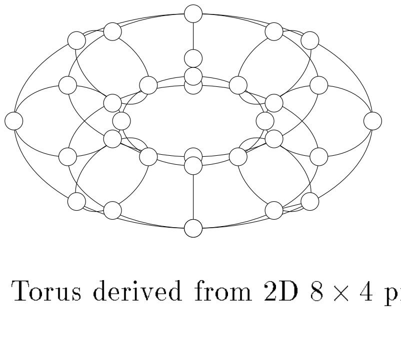


Abbildung 4.18: Torus derived from 2D 8×4 processor array

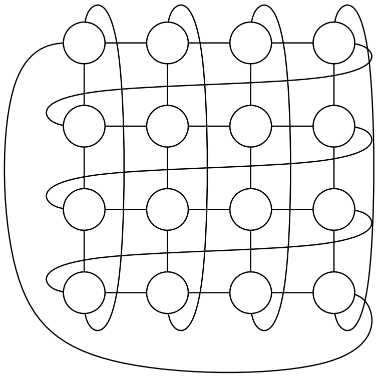


Abbildung 4.19: ILLIAC mesh

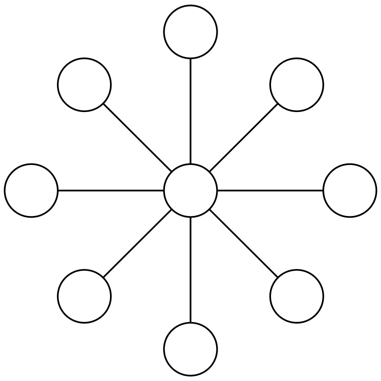


Abbildung 4.20: Star

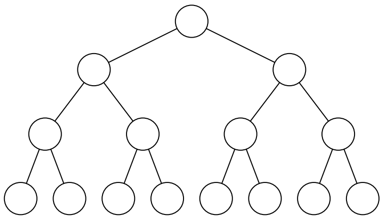


Abbildung 4.21: Tree

by the hierarchical structure for problems which require large amounts of long distance communication. Here all the global communication maust pass through one (or a few) nodes and their channels.

11. For the tree a possible improvement is the so-called *fat tree*, see Fig. 4.22. The idea here is to alleviate the communication bottleneck at higher levels

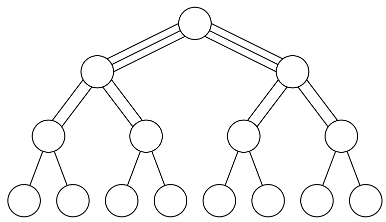


Abbildung 4.22: Fat tree

in the processor hierarchy by incraesing the communication bandwidth. Fig. 4.22 suggests that this is done by having multiple connections between the higher level nodes.

12. The last class of network topologies is a sequence of topologies the simplest of which is the chain of two nodes. The next topology is then the 2×2 mesh and the $2 \times 2 \times 2$ mesh in 3 dimensions which becomes the cube in Fig. 4.23 with $2^3 = 8$ nodes.

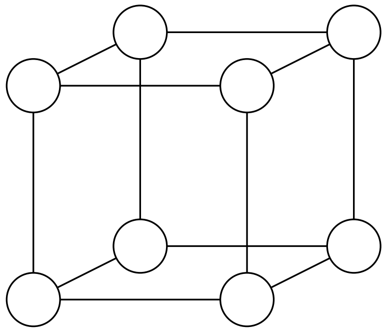


Abbildung 4.23: Cube

13. This sequence is naturally extended to the *hypercube* with $2^4 = 16$ nodes as in Fig. 4.24. The general topology is the d-dimensional hypercube with 2^d nodes.

14. The hypercube offers many interesting properties, in particular a communication distance between any two nodes of at most $\log_2(N)$ links for an N processor configuration. The increase in the node degree with $\log_2(N)$, however, still a severe hardware problem for large N. A compromise which accepts somehwat longer communication distances while retaining the same basic structure is the cube-connected cycle, as are shown (for 3D) in Fig. 4.25). Here the node degree is 3 for any size of the system.

Tabelle 4.1: Network properties

Type	Degree	Diameter	Connections	Bis. width	Symmetry
lin. Kette	2	$N-1$	$N-1$	1	no
Ring	2	$[N/2]$	N	2	yes
Voll. Vern.	$N-1$	1	$N(N-1)/2$	$(N/2)^2$	yes
Bin. Baum	3	$2((\log_2 N)-1)$	$N-1$	1	no
2D-Gitter	4	$2(\sqrt{N}-1)$	$2N-2\sqrt{N}$	$2\sqrt{N}$	no
2D-Torus	4	$2\lfloor\sqrt{N}/2\rfloor$	$2N$	$2\sqrt{N}$	yes
Hypercube	$\log_2 N$	$\log_2 N$	$N\log_2 N$	$N/2$	yes

4.4.2 Properties of network topologies

A network (or generally a graph) is chracterised by a number of features, which are usually given as a function of the size (that is number of nodes) in the graph. Some of these features are relevant to computer architecture and parallel programming.

- *Number of connections.*
- *Node degree*, that is the number of connections at each node. This may be seperated in incoming and outgoing connections. Of special interest is also the maximal node degree in a network.
- *Network diameter*, that is the maximal shortest path between two nodes
- *Bisection width*, that is the minimal number of connections between any two parts of equal size of the graph.
- *Symmetry*, that is whether the graph looks the same from any of its processors.
- *Channel length.* This is a physical parameter also depending on the geometric layout of the graph, however, important for e.g. signal latencies.

Table 4.1 shows some of the characteristics for prominent network topologies discussed above.

4.4.3 Dynamical networks

In section 4.4.1 we have discussed static networks which cannot be cahneged once they are built. Networks, however, may also be *dynamic*. Such networks may be configured to set up connections between any pair of several components.

Typical devices with this functionality are

1. The *bus*, which permits communication between any pair of devices connected to it. However, a bus is an exclusive device, and whenever it services a communication between two devices, any other cummunication is blocked.

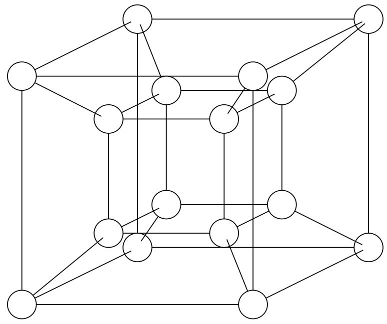


Abbildung 4.24: Hypercube $d = 4$ with 16 processors

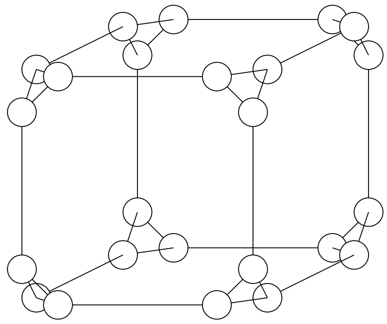


Abbildung 4.25: Cube connected cycle ($d = 3$)

.

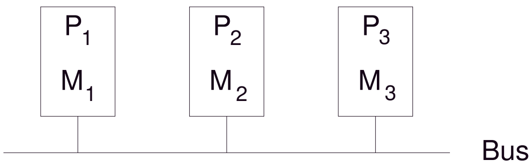


Abbildung 4.26: Bus

Therefore a bus cannot be used with large number of processors. A bus however can *broadcast* messages, that is it can send a message to all devices connected to it.

2. *Switching networks.* Based on the *binary switch* in Fig. 4.27 with four

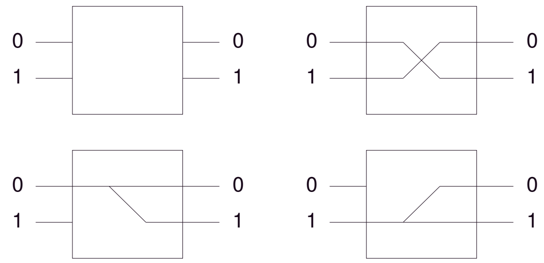


Abbildung 4.27: Binary switch

possible states, it is possible to build complex switching networks. Fig. 4.28 shows an 8×8 switch in a so-called *baseline network*. In such a

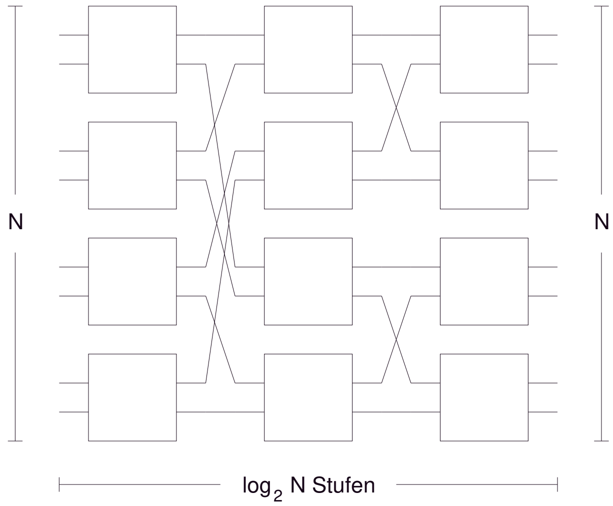


Abbildung 4.28: 8×8 baseline network

network up to eight connections may be used simulatneously, as long there is no access conflict. Not all combinations of connections, however are possible simulatneously. The complexity of the baseline network grows with $N \times \log_2 N$, where the depth of the network with $\log_2 N$ stages influences the latency of messages passing through the switch. Note that again bradcasts of messages from one sender to any of the output devices are possible.

3. Another device capable to perform the functionality of switching from N input to N output devices is the *crossbar*. Here an $N \times N$ grid is built and any connection from an incoming line to an outgoing line can be made. The crossbar has complexity $N \times N$ and saves the $\log_2(N)$ intermediate

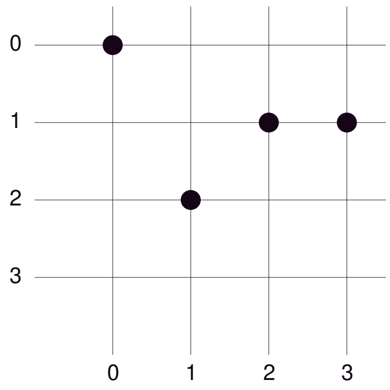


Abbildung 4.29: Crossbar

stages of a basline net. Due to the regular structure it is fairly simple to build, say on a specialized routing chip. However, a crossbar connecting a large number of processors a large number of memory modules where each of the lines in Fig. 4.29 would also represent a bus of say 128 bit width would quickly reach technological limits in its complexity.

4. *Multi-bus systems.* The wide availablity of buses e.g. in the form of ethernet installations make it attractive to devise networks where buses are used. The primary feature of a bus is that though only one connection can be serviced at any time, a potentially large number of devices canbe connected to it. If we think of ethernet as a cheap network technology, the reverse is true for the processing nodes. Typically the number of networks, a workstation can be connected to is very limited. For entry level machine it may be just one, and even for larger ones it is only two. However, the number of workstations that can be connected to an ethernet is almost unlimited. Thus it is a nice idea to reverse the role of networks and workstations. Fig. 4.30 shows the so called em dual hypercube for

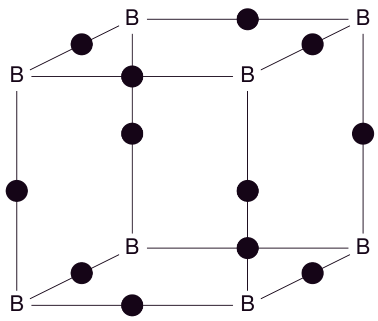


Abbildung 4.30: Dual hypercube

dimension 3, where the edges represent computers and the corners represent the different ethernets. This configuration retains several of the features of a hypercube. However, while the number of networks is now 2^d, and each computer needs exactly two ethernet connections, the number of computers required is $d2^{d-1}$.

4.5 Program Parallelism

In this section some basic concepts of parallel program execution are discussed.

4.5.1 Conditions for parallel execution

Operations may only be executed in parallel, when certain conditions are satisfied. For parallelizing a program certain dependencies which enforce sequential execution must be checked, either automatically or manually by the programmer. The conditions that prohibit parallelization are the following.

- *Data flow dependence.* $S1 \longrightarrow S2$: $S1$ produces input for $S2$.
- *Antidependence.* $S1 \not\longrightarrow S2$: $S2$ overwrites input of $S2$.
- *Output dependence.* $S1 \circ\!\!\longrightarrow S2$: $S1$ und $S2$ write same variable.
- *I-O-Dependence*: $S1$ und $S2$ write same file.
- *Unknown dependence*: This includes all cases which cannot be easily analysed and which may contain a dependence prohibiting parallelization. These include
  - indexed indices,
  - indices which do not contain the loop variable,
  - variables which apear several times with different indices,
  - nonlinear indices.
- *Control dependence.* This includes all cases, when the control flow depends on the data like in the following example.

```
for (i=1; i<n; i++)
    if (a[i-1]==0) a[i]=0;
```

4.5.2 Granularity

In practice, it is also necessary to classify parallism according to its *granularity.* We distinguish between parallelism on the level of

- machine instructions
- loops
- subroutines
- programs.

4.5.3 Parallel execution paradigms

Parallelism can be described by a number of different paradigms. These parallelism paradigms are also reflected in different parallel programming languages. In present scientific computing practice, only the first, the so-called *control driven* style is used widely. However, we believe that the alternatives are at interesting for the algorithm developer because they may prsent useful insight about he algorithms, even when such less common techniques cannot be used for a direct implementation.

We distingusish between

- *Control driven parallelism.* This programming model is based on conventional procedural/imperative programming paradigm, which is augmented by explicit parallel constructs in the form of message passing or synchronization primitives. The basis of this model is a machine architecture with several conventional CPUs, each having its own program counter. The state of processing is characterized by the collective value of all the program counters.

- *Data flow driven parallelism.* In this programming model data is pointing to executable code which may be executed as soon as all inputs are available. This corresponds to an *eager*, *bottom up* evaluation of parallel code fragments. Fig. 4.31 shows a program fragment computing a few simple vector operations. Below its data flow graph is shown. Data flow paralle-

```
input d,e,f
c[0]=0
for (i=1; i<=4; i++) {
    a[i] = d[i] + e[i];
    b[i] = a[i] * f[i];
    c[i] = b[i] + c[i-1];
}
```

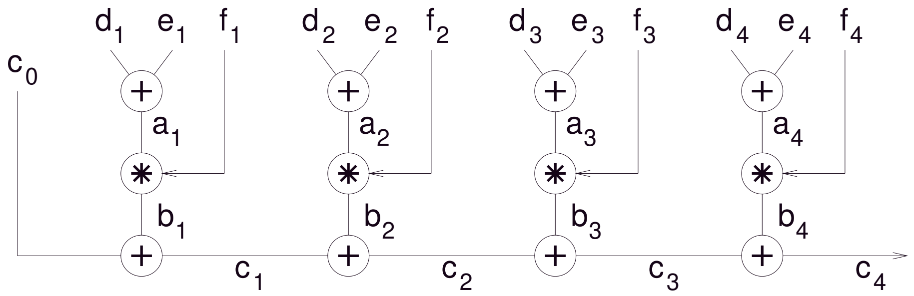


Abbildung 4.31: Data flow parallelism

lism requires an efficient technique to check whether data is available for computation. Data that is required as input for several operations must be copied. One disadvantage of data flow parallelism is associated with its *eager* evaluation strategy. Operations my be executed whose result is later found to be irrelevant. The typical example is the parallel execution of an *if–then–else* construct. Here both branches may be executed in parallel, even though the evanulation of the condition must make one of the branches irrelevant.

- *Reduction driven parallelism.* Different from the data flow parallelism, here the operations point to their operands, which must be evaluated. The parallel execution of a simple expression leads to a tree, as shown in Fig. 4.32 This programming paradigm corresponds to a *functional programming style.*

$$a := (b + 1) * c - d/e$$

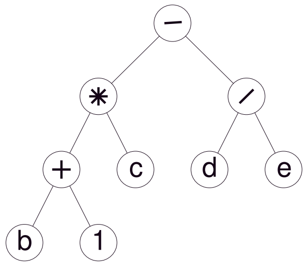


Abbildung 4.32: Reduction driven parallelism

4.6 Evaluation of parallel perfomance

4.6.1 Program parallelism profile

We consider a parallel program which executes $D(t)$ parallel instructions at time step t. $D(t)$ is called the *parallelism profile* of a program. A typical parallelism profile for a *divide-and-conquer* algorithm is shown in Fig. 4.33. Thus, a program

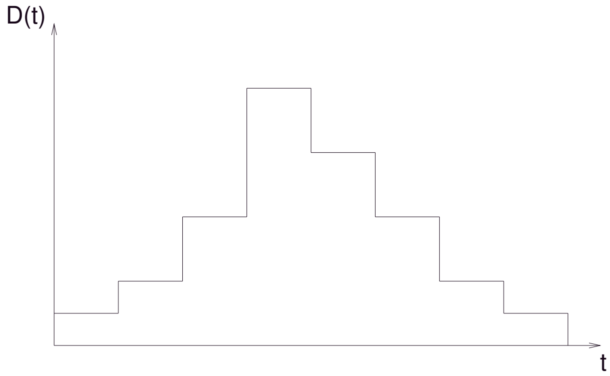


Abbildung 4.33: Parallelism profile for typical divide-and-conquer algorithm

run from the start time t_s to t_e needs a total of W operations.

$$W = \Delta \int_{t_s}^{t_e} D(t)dt = \Delta \sum_{i=1}^{m} i \cdot t_i$$

Here Δ is the computing capacity of a single processor. We can now define the *average parallism* as

$$A = \frac{1}{t_e - t_s} \int_{t_s}^{t_e} D(t)dt.$$

If we denote by t_i the time spent using i processors, then the execution time $T(n)$ with n available processors can be expressed for the two extreme cases of n=1 and $n = \infty$ processors by

$$T(1) = \frac{W}{\Delta} = \sum_{i=1}^{n} \frac{W_i}{\Delta}, \quad W_i = i \cdot t_i,$$

and

$$T(\infty) = \sum_{i=1}^{n} \frac{W_i}{i\Delta} = \sum_{i=1}^{n} t_i,$$

respectively. The *asymtotic speedup* S_∞ is defined correspondingly as

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{n} W_i}{\sum_{i=1}^{n} W_i/i}.$$

4.6.2 Average execution times

One of the basic problems of parallel execution is to determine a total overall performance if either several programs with different parallel execution characteristics are executed, or when in a single parallel program phases with different parallel execution characteristics occur.

If we denote by R_i the performance (say in MFlops) in the ith of m program parts, we may then compute the *arithmetic mean* defined as

$$R_a = \frac{1}{m} \sum_{i=1}^{m} R_i$$

Since R_i is proportional to the inverse of the execution time, the average of these inverses does *not* represent the total execution time.

The *geometric mean*, defined as

$$R_g = \sqrt[m]{\prod_{i=1}^{m} R_i}$$

is sometimes used to compare the performance with respect to a reference machine, however, it again does not represent a total execution time.

Only the *harmonic mean*, defined as

$$R_h = \frac{m}{\sum_{i=1}^{m} R_i^{-1}}$$

represents the total number of operations divided by the total execution time and is thus an appropriate measure for the average performance.

In practice the harmonic mean is often generalized to a *weighted harmonic mean* in the form

$$G_h^* = \frac{1}{\sum f_i/R_i},$$

where the f_i for $i = 1, \ldots, m$ are weights.

4.6.3 Harmonic speedup

The harmonic average defined above is the basis to define an *harmomic mean speedup* for a parallel program. Assume that n processors are given. Thus a program may execute with $1, \ldots, n$ processors at any time. Assume that

$$T_1 = \frac{1}{R_1} = 1$$

is the sequential execution time on a uniprocessor. $R_i = i$ is the idealized performance with i processors resulting in an execution time of

$$T_i = \frac{1}{R_i} = \frac{1}{i}.$$

Suppose now that the program is executed in n execution modes involving $1, \ldots, n$ processors. With a weight distribution of f_i, $i = 1, \ldots, n$, the *harmonic mean speedup* is defined as

$$S = \frac{T_1}{T^*} = \frac{1}{\sum_{i=1} n \frac{f_i}{R_i}}$$

This formula can of course be used only, if the weights are either known or certain assumptions about them are made. We study the effect of these formulas for a few examples.

Examples

1. First, we assume that on an n-processor machine any degree of parallelism (taht is any number of processors) is used to eexcute $1/n$th of the total problem. Thus $f_1 = f_2 = \ldots f_n = \frac{1}{n}$ and thus

$$S_n = \frac{1}{\sum_{i=1}^{n} \frac{1}{in}} = \frac{n}{\psi(n+1) + \gamma}$$

Here, the sum is represented in closed form with

$$\psi(x) := \ln(\Gamma(x)), \quad \gamma = 0.5772156\ldots$$

where $\Gamma(x)$ is the Gamma function and γ is Euler's constant. A few typical values for the speedup are:

$$S_2 = 1.33\ldots \quad S_{32} = 7.88\ldots \quad S_{1024} = 136.$$

These figures are not very impressive and the efficiency deteriorates quite significantly for larger processor numbers.

2. Next we consider a more optimistic case. We assume that each number i of processors $i = 1, \ldots, n$ is used in parallel for a portion of the overall work which is proportional to i. Thus the time when the program operates in highly parallel mode is longar than in example 1. Thus

$$f_1 = \frac{1}{\sigma}, f_2 = \frac{2}{\sigma}, \ldots f_n = \frac{n}{\sigma}, \text{ where } \sigma = \sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

   and

$$S_n = \frac{1}{\sum_{i=1}^{n} \frac{i}{i\sigma}} = \frac{\sigma}{n} = \frac{n+1}{2}.$$

   This leads to a parallel efficiency tending asymptotically towards 50%.

3. Finally, we examine the pessimistic case that the portion of work executing with high parallelism is inverse proportional to the number of processors. This means

$$f_1 = \frac{n}{\sigma}, f_2 = \frac{n-1}{\sigma}, \ldots, f_n = \frac{1}{\sigma}.$$

   Thus

$$S_n = \frac{1}{\sum_{i=1}^{n} \frac{n+1-i}{i\sigma}} = \frac{1}{2} \frac{n(n+1)}{-n + n\psi(n+1) + \psi(n+1) + \gamma n + \gamma}$$

   and

$$S_2 = 1.2 \ldots \quad S_{32} = 5.18 \ldots \quad S_{1024} = 78.64 \ldots$$

   Here the speedups and efficiency are even worse than for example 1.

4. Finally, we consider the trivial but famous case

$$f_1 = \alpha, f_2 = \ldots = f_{n-1} = 0, f_n = 1 - \alpha.$$

   Here the program is executed in only two modes. A purely sequential mode is needed for a fraction of α of the total computation. In a second phase the problem can be optimally parallelized with n processors. This phase covers the remaining part of $1 - \alpha$ of the computation. This leads to a speedup

$$S_n = \frac{1}{\alpha/1 + (1-\alpha)/n} = \frac{n}{n\alpha + 1 - \alpha} = \frac{n}{\alpha(n-1)+1}$$

   Here, even if the number of processors tends to ∞, the speedup is limited by

$$\lim_{n \to \infty} S_n = \frac{1}{\alpha}$$

   This effect is known as *Amdahl's law.*

4.6.4 Efficiency, Utilization, and Quality

We consider a parallel program execution with the following characteristics:

$O(n)$ Number of operations on an n processor system
$T(n)$ execution time in unit time steps on an n processor system
$T(n) \leq O(n)$

The the *speedup* is now defined as

$$S(n) = \frac{T(1)}{T(n)} \quad (1 \leq S(n) \leq n)$$

Typically, the speedup satisfies

$$S(n) \leq n.$$

(Counterexamples will be given below). Next, the *efficiency* is defined as

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

The efficiency usually satisfies $\frac{1}{n} \leq E(n) \leq 1$. For a more realistic description of parallel performance it is necessary model the situation that a parallel algorithm may need more operations than its sequential counterpart. Quite often organizing th eexecution on n processors has a significant overhead. Therefore it makes sense to introduce the *redundancy* as

$$R(n) = \frac{O(n)}{O(1)}$$

We further define the *utilization* as

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)}$$

to see that

$$\frac{1}{n} \leq E(n) \leq U(n) \leq 1$$

and

$$1 \leq R(n) \leq \frac{1}{E(n)} \leq n.$$

The *quality* of a parallelization is finally defined as

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)}, \quad Q(n) < S(n)$$

In exceptional cases the speedup S_n may be larger than n. This may happen because of practical side conditions.

- A sequential program may execute in a too small memory cuasing swapping/paging, or less obvious, may suffer more from cache misses than the parallel version where typically each parallel portion of the code executes in amuch smaller data set.

- Another cause for superlinear speedup is shown in the folowing example. Assume that we must to evaluate

$$A_1 \vee A_2$$

where both A_1 and A_2 are true with a probablity of 50%. However, to evaluate A_i the average execution $T(A_i)$ time is significantly different, depending on whether the result is true or false. Assume

$$T(A_i) = \begin{cases} 1 \text{ sec. (average)} & \text{for } A_i = 1 \\ 100 \text{ sec. (average)} & \text{for } A_i = 0 \end{cases}$$

Now there are four cases with equal probablity.

A_1	A_2	sequentiell	parallel
T	T	1 sec + 0	1 sec
T	F	1 sec + 0	1 sec
F	T	100 sec + 1 sec	1 sec
F	F	100 sec + 100 sec	100 sec
Sum		$\sim \frac{303}{4}$ sec ≈ 76 sec	$\sim \frac{103}{4} \approx 26$ sec

Thus a parallel execution with two processors leads to an average speedup of

$$S(2) = \frac{T(1)}{T(2)} = \frac{303}{103} \approx 3$$

Note that the superlinear speedup is caused by the inflexibility of the sequential execution. The sequential execution of A_1 should be stopped after 1 second of computation without result (at latest) to check A_2 first, whether a quick decision is possible here. Thus a pseudo-parallel execution of $A_1 \vee A_2$ would have the a similar effect and would reduce the average compute time of the sequential algorithm.

4.6.5 Scalability of parallel algorithms

In this section we will further analyse the *scalability* of parallel algorithms. Generally, the factors determining parallel performance are the

- granularity
- parallelism profile
- communication pattern
- synchronization pattern
- uniformity
- storage requirement and storage organization

In section 4.6.3 we have seen that for a fixed algorithm, e.g. Amdahl's law predicts a rather pessimistic paralell behavior. For a fixed algorithm any sequential part (or part with limited parallelism) limits the achievable speedup independent of the processor number. However, since parallel computers are usually applied to *larger* problems than sequential machines, the concept of a fixed algorithm in itself may be doubtful. The topic of this section is to study concepts anlogous to speedup and efficiency, when the problem is allowed to grow with the increased compute (and storage) capacity associated with lareger paralell machines.

We therefore introduce an abstarct parameters s, which represents the problem size. For full matrix operations, e.g. s may be the dimension of the matrix. For a mesh it may be the number of unknowns, etc. Each sequential algorithm has a characateristic operation count $w(s)$, the work. For a matrix multiply e.g., $w(s) = O(s^3)$. On a parallel processor, however, some overhead must be taken into account. For a parallelization with n processors, we denote this overhead by $h(n, s)$. This function describes the effects of communication delays as well as extra cost necessary to split the algorithm in independent processes.

With these convenions, we can compute the efficiency of an n processor parallelization of a problem of size s by

$$E = \frac{w(s)}{w(s) + h(s,n)}. \tag{4.1}$$

We now wish to keep the efficiency constant. Of course, our previous considerations show, that for fixed problem size, the efficiency will decrease when the number of processors is increased. Thus, a constant efficiency will require to increase the problem size with the number of processors. In particular, solving (4.1) for w, leads to

$$w(s) = \frac{E}{1-E} h(s,n).$$

Thus for $E = \text{const}$, $w(s) \equiv f_E(n)$ is the so called *iso-efficiency function.*

$$f_E(n) = C \cdot h(s,n)$$

For a given algorithm, $f_E(n)$ describes how fast the workload must increase (with the processor number) to maintain a constant efficiency.

Example 1. Multiplication of two square matrices. Clearly, the work depends on s (defined as the dimension of the matrices) by $w(s) = s^3$. The Fox-Otto-Mey-Algorithmus [10] for a n processor torus with $1 \leq n \leq s^2$ is characterized by

$$h(s,n) = O(n \log n + s^2\sqrt{n}).$$

By using $w(s) = O(s^3)$ we derive the iso-efficiency function

$$O(s^3) = O(n \log n + s^2\sqrt{n})$$

Thus

$$O(s^3) = O(n \log n) \quad \text{and} O(s) = O(\sqrt{n})$$

The iso-efficency function here is

$$w(s) = O(s^3) = O(n^{3/2}) = f_e(n),$$

so that to maintain a constant efficiency, we expect the dimension s to grow like *sqrtn*.

Example 2. Fast Fourier transormation for s points.

The Fast Fourier Transformation FFT has a work cont of

$$w(s) = s \log s$$

We now consider two different algorithms on different architectures.

1. On the Hypercube, a parallel implementation has

   $$h_1(s, n) = O(n \log n + s \log n)$$

   and thus

   $$O(s \log s) = O(n \log n + s \log n) \Rightarrow s = n$$

   giving the iso-efficiency function

   $$f_e(n) = O(n \log n).$$

2. On a 2D grid of processors, the algorithm has an overhead

   $$h_2(s, n) = O(n \log n + s\sqrt{n}$$

   and thus

   $$O(s \log s) = h_2(s, n) \Rightarrow \sqrt{n} = \log(s) \Rightarrow s = 2^{\sqrt{n}}$$

   giving the iso-efficiency function

   $$f_e(n) = O(2^{\sqrt{n}} \cdot \sqrt{n}.$$

Comparing these iso-efficiency functions shows that the FFT scales well on the hypercube, but not on the grid.

4.6.6 Speed-up Laws

In this section we again discuss properties of speedup. Konventionally, a fixed problem is considered, and a variable number of processors is used for the solution.

Let W_i denote the number of operations which can be performed with i processors in parallel. Then, when n processors are available the execution of W_i requires a time of

$$t_i(n) = \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil,$$

and, as already seen in Section **??** the execution of a program employing m different execution modes corresponding to W_i, $1 \leq i \leq m$, needs an overall time of

$$T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil .$$

Therefore the speedup is

$$S(n) = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}} \left\lceil \frac{i}{n} \right\rceil$$

and

$$S(n) < S_\infty < A$$

More realisticlly we additionally introduce the overhead $Z(n)$, and find

$$S(n) = \frac{T(1)}{T(n) + Z(n)}$$

At this stage we recall again Amdahl's law as introduced in Section 4.6.3: $W_2, \ldots, W_{n-1} = 0$

$$S(n) = \frac{W_1 + W_n}{W_1 + \dfrac{W_n}{n}} \quad \overset{n \to \infty}{\longrightarrow} \quad \frac{W_1 + W_n}{W_1} = 1 + \frac{W_n}{W_1}$$

This says that a given problem with a fixed sequential part can not be solved below a certain minimal time no matter how many pocessors are used.

4.6.7 Scaled Speedup Laws

Now we keep the execution times fixed an determine which problem size can be solved depending on the number n of processors. By m' we denote the maximal parallelism of the problem and by W_i' the scaled work for parallelism with i processors. Thus we have for the scaled execution time $T'(n)$ with n processors

$$T(1) = T'(n)$$

, where $T(1)$ is the sequential time, as usual. Thus

$$\sum_{m=1}^{m} W_i = \sum_{i=1}^{m} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Z(n)$$

and

$$S'(n) = \frac{T'(1)}{T'(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} W_i}$$

We are now in a position to derive the analogon of Amdahl's law for the case that the execution time (and not the problem size) is kept fixed. We again assume

$$W_2 = \ldots = W_{n-1} = 0$$

and thus obtain

$$S'_{(n)} = \frac{W'_1 + W'_n}{W_1 + W_n} = \frac{W_1 + nW_n}{W_1 + W_n} \overset{n \to \infty}{\longrightarrow} n\left(1 - \frac{W_1}{W_1 + W_n}\right)$$

where

$$W'_n = W_n \text{ and } W_1 + W_n = W'_1 + \frac{W'_n}{n}$$

corresponding to the fixed time constraint.

$$\lim_{n\to\infty} \frac{S'(n)}{n} = \lim_{n\to\infty} \frac{\frac{W_1}{n} + W_n}{W_1 + W_n} = \frac{W_n}{W_1 + W_n} = (1 - \frac{W_1}{W_1 + W_n})$$

We see that once the problem grows with the machin, the seqeuntial part is no longer a bottleneck. This modification of Amdahl's law is also called *Gustafson's law*, see [13].

4.6.8 Speedup with fixed memory size

The idea here is to consider speedup subject to the condition that the memory requirements of the problem are kept fixed. This is motivated by studying scalable computer architectures where n processors usually also have n times as much storage than a single processor system.

We illustrate the concept briefly for the matrix multiplication algorithm. We assume the algorithm has a fixed seqeuntial part and then a part which parallelized fully.

For one processor, the matrix is assumed to have dimension m. The seqeuntial execution time is

$$T(1) = W_1 + W_n = W_1 + m^3.$$

With n processors we can handle matrices of size

$$dim' = \sqrt{n}m,$$

and therefore the hypothetical time for the sequential process is

$$T'(1) = W_1 + n^{3/2}m^3$$

However, with n processors, we obtain

$$T'(n) = W_1 + n^{1/2}m^3$$

Thus the speedup is

$$S^*(n) = \frac{T'(1)}{T'(n)} = \frac{W_1 + n^{3/2}W_n}{W_1 + n^{1/2}W_n} \overset{n\to\infty}{\longrightarrow} n.$$

For the matrix multiplication, the compute time grows faster than the memory requirements. Thus, the larger the machine, the larger a problem can be handled, and the compute time grows even though the machine gets faster and faster. In terms of speedup this yields an even more optimistic picture than the fixed time speedup.

4.6.9 Scalability of computing environments

Example. We consider a simple example with a Laplace equation in 3D. The equation is discretized by a 3D 7-point stencil, as shown schematically in Fig. 4.34.

7-Punkt-Stern

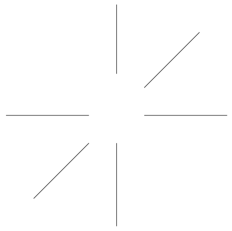


Abbildung 4.34: 3D 7-point stencil

Assume that we have a processor performing at 100 MFlop. Further assume that any two nodes can excahnge data at a rate of μ seconds per Byte.

Thus the relaxation of a single node of the grid requires $7 \cdot 10^{-8}$ sec. For a parallelization we assume a standard domain decomposition, where each processor handles (and stores) s^3 nodes.

Thus the compute time per processor, required for relaxing all nodes is given by $7s^3 10^{-8}$ seconds, and the storage requirement is (for double precision words) $8s^3$ Byte. On the other hand, processors must exchange the *surface* of the grid cube after each step of computation. In our case this is $6 * 8s^2$ Byte.

In order to balance the communication and compute requirements, we assume that both can be fully overlapped, so that ideally both times are equal. This leads to

$$8 * 6 * s^2 * \mu = 7s^3 * 10^{-8}$$

and

$$s\frac{7}{6} = \mu.$$

For connections capable of transferring 1 Byte in approx. 10^{-6} seconds (like Ethernet) we set $\mu \approx 10^{-5}$ and obtain

$$s = 10^3.$$

In this case, a balnced communication/computation realtion will thus require to store a $(10^3)^3$ grid in each node and therefore requires each processor to have about 8 GigaByte of memory.

If we alternatively assume that the communication speed is increased by a factor of 10 to $\mu = 10^{-6}$, then the analogous computation yields $s = 10^2$ and the storage requirement for each processor is a moderate $8 MegaByte$ to obtain balanced performance. Thus, the well-balancing of a computation depends very sensitively on the relationship of compute- and communication speed.

4.7 Program optimization for cache-based architectures

Note: In the final version, this section will be moved elsewhere.

The work-horse in many scientific computing applications is nowadays the workstation and thus the processor most commonly used has a hierarchical memory architecture with several layers of caches. Parallel computers are also increasingly beoming dependent on standard processors and thus they also employ cache designs. Thus it becomes important to study the performance of scientific computing applications on hierarchical memory architectures, and — if possible — to find means for optimizing the programs and algorithms to better exploit these machines. An interesting discussion of these issues can also be found in the report RISC Microprocessors and Scientific Computing[4] by D. Bailey [3] or in Performance analysis and optimization of numerically intensive programs.[5]

In this section we will study two examples of optimization for scientific computing applications. The reference timings are given for a Hewlett-Packard HP9735 workstation with a 99 MHz processor.

4.7.1 Optimized matrix multiplication

Our first example is the product of two square $n \times n$ matrices $A, B \in \mathbb{R}^{n\times n}$. The product matrix $C = AB$ can be computed by the standard program

```
#define N 512
#define ND (N)
double a[ND][ND], b[ND][ND], c[ND][ND];
// ... initialize
          for(i= 0; i<N; i++)
          for(j= 0; j<N; j++){
                c[i][j]= 0.0;
                for(k= 0; k<N; k++) c[i][j]+= a[i][k]*b[k][j];
          }
```

Abbildung 4.35: Standard matrix multiplication

We wish to point out that there are few scientific computing applications which require the computation of a matrix product directly. In most of the cases it is not necessary to compute the matrix represenattion of a composed mapping directly by computing the product matrix. It is usually more effective to keep the matrices stored separately and to compute the application of $Cx = (AB)x$ by applying the matices separately in the form $A(Bx)$.

[4] http://www.nas.nasa.gov/NAS/TechReports/RNRreports/dbailey/RNR-93-004/RNR-93-004.html

[5] http://www5.informatik.tu-muenchen.de/persons/ruede/info/refs.html

The importance of matrix multiplication arises indirectly. When performing LU decomposition for solving linear systems, the main body of the work can be organized in multiplications of (block) submatrices. Thus matrix multiply is a core task in linear system solving.

The simple algorithm in Fig. 4.35 will now be improved in several steps. Note that the innermost loop computes a scalar product of a row- and a column vector.

For a vector processor it would be worthwile to consider a rearrangement of the loops such that the innermost loop updates a (row-) vector of C by exchanging the k and j loops. For a cache-based system this would not improve the performance.

Our first modification of the algorithm introduces *padding fields* by modifying the declaration of the data arrays in the form shown in Fig. 4.36. In a

```
#define N 512
#define ND (N+1)

double a[ND][ND], b[ND][ND], c[ND][ND];
```

Abbildung 4.36: Data declaration with padding fields

next modification, we introduce an axiliary variable in the innermost loop for assembing the scalar product. This helps the compiler to optimize the code by allocating this variable in a register. The resulting code is shown in Fig. **??**.

```
for(i= 0; i<N; i++)
for(j= 0; j<N; j++){
        double s= 0;
        for(k= 0; k<N; k++)
                s+= a[i][k]*b[k][j];
        c[i][j]= s;
}
```

Abbildung 4.37: Scalar product to be assembled in register

In the next step, we notice that the innermost loop traverses the b-vector in column-wise ordering. Since in C the matrix elemnst are stored by row, we expect the column-wise access to be a potential performance bottleneck. Thus we modify the access to b by introducing an auxiliary (row-) vector v, which is used to hold intermediate copies of columns of the matrix b. To make this effective we must of course also try to reuse these copies of data from b as much as possible. This is accomplished by additionally reversing the order of the i and j loops. Together these changes result in the program given in Fig. 4.38.

The program in Fig. 4.38 still has an innermost loop which executes just one multiply-add pair in each pass. In the next step we attempt to reduce the

```
for(j= 0; j<N; j++){
double v[N];
for(i= 0; i<N; i++)      v[i]= b[i][j];
for(i= 0; i<N; i++){
        double s= 0;
        for(k= 0; k<N; k++)
                s+= a[i][k]*v[k];
        c[i][j]= s;
}
}
```

Abbildung 4.38: Auxiliary vector for improved column access

.

loop overhead by *unrolling*. Here, we unroll the i and j loop by a factor of 2 each, so that the innermost loop is changed to have four multiply-add pairs in each pass. This transformation can also be interpreted as a *blocked* algorithm, where the product matrix is computed in units of 2×2 blocks.

The final modification introdues an additional blocks structure. The algorithm of Fig. 4.39 is applied to blocks of size 64×64. This blocksize is such that the corresponding matrices A, B, C fit in the cache. The total product is then assembled in terms of $64 = 8 \cdot 8$ blocks each of size 64×64. The total algorithm now requires 8^3 multiplications of 64×64 matrices. This code is shown in Fig. **??**.

In the following table we summarize the run times and flop rate for each program We note that the optimizations have almost improved the performance

Program variant	Execution time	Megaflop rate
Fig. 4.35	74.3 sec	3.6
Fig. 4.36	39.5 sec	6.7
Fig. **??**	34.2 sec	7.8
Fig. 4.38	16.7 sec	16.1
Fig. 4.39	8.5 sec	31.6
Fig. 4.40	3.8 sec	69.9

by a factor of 20. Additionally we remark that the padding fields introduced in 4.36 are not necessary when the other optimizations are carried out. The code shown in Fig. 4.40 can be further improved by some fine tuning. Note, however, that so far all optimizations have been done without using any machine-specific information. The same program would improve the performance on any other cache-based machine.

Further improvements could be obtained by further work on details of the algorithm, possibly using machine specific characteristics. With C programs alone it is possible to obtain 100 MFlop performance on the HP735, which is about 50% of the 198 MFlop peak performance. In order to obtain this, the

```
for(j= 0; j<N; j+=2){
double v1[N];
double v2[N];
for(i= 0; i<N; i++){
        v1[i]= b[i][j];
        v2[i]= b[i][j+1];
}
for(i= 0; i<N; i+=2){
        double s00=0.0, s01=0.0, s10=0.0, s11=0.0;
        for(k= 0; k<N; k++){
                s00+= a[i][k]*v1[k];
                s01+= a[i][k]*v2[k];
                s10+= a[i+1][k]*v1[k];
                s11+= a[i+1][k]*v2[k];
        }
        c[i][j]= s00;
        c[i][j+1]= s01;
        c[i+1][j]= s10;
        c[i+1][j+1]= s11;
}
}
```

Abbildung 4.39: Unrolling the outer loops

```
for(jo= 0; jo<N; jo+=64)
for(io= 0; io<N; io+=64)
for(ko= 0; ko<N; ko+=64){

for(j= jo; j<jo+64; j+=2){
double v1[65];
double v2[65];
for(i= io; i<io+64; i++){
        v1[i]= b[i][j];
        v2[i]= b[i][j+1];
}
for(i= io; i<io+64; i+=2){
         double s00=0.0, s01=0.0, s10=0.0, s11=0.0;
         for(k= 0; k<64; k++){
                s00+= a[i][k+ko]*v1[k];
                s01+= a[i][k+ko]*v2[k];
                s10+= a[i+1][k+ko]*v1[k];
                s11+= a[i+1][k+ko]*v2[k];
        }
        c[i][j]+=     s00;
        c[i][j+1]+=   s01;
        c[i+1][j]+=   s10;
        c[i+1][j+1]+= s11;
}
}
}
```

Abbildung 4.40: Doubly blocked matrix multiply program above code must be further optimized by e.g. unrolling (the i and j loop, etc.). By additionally using assembly routines for the innermost loop, a performance of up to 120 MFlop on the same machine is possible, see the Linpack 1000×1000 results in [7].

4.7.2 Detour: Winograd-Strassen multiplication

In the previous section 4.7.1 we have seen that *blocking* is an important technique to improve the cache-performance of the matrix multiplication algorithm. In the following we will briefly outline a different idea whose origin is the blocking of matrices.

Note that all versionsof the sandard matrix multiplication algorithm need the same amount of operations. For general $n \times n$ matrices the cost is $O(n^3)$ operations.

In his paper 1969 paper *Gaussian elimiation is not optimal*, Strassen [23] showed that the $O(n^3)$ complexity of matrix multiplication algorithms was not optimal. Here refer here to the variant of the algorithm described by Winograd [26].

We begin by studying 2×2 matrices

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} C = AB = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

C can be computed by the following algorithm.

Algorithm R

p_1	=	$(a_{11}+a_{22})(b_{12}+b_{22})$	2A	+	1M
p_2	=	$(a_{21}+a_{22})b_{11}$	1A	+	1M
p_3	=	$a_{11}(b_{12}-b_{22})$	1A	+	1M
p_4	=	$a_{22}(b_{21}-b_{22})$	1A	+	1M
p_5	=	$(a_{11}+a_{12})b_{22}$	1A	+	1M
p_6	=	$(a_{21}-a_{11})(b_{11}+b_{12})$	2A	+	1M
p_7	=	$(a_{12}-a_{22})(b_{21}+b_{22})$	2A	+	1M
c_{11}	=	$p_1+p_4-p_5+p_7$	3A		
c_{12}	=	p_3+p_5	1A		
c_{21}	=	p_2+p_4	1A		
c_{22}	=	$p_1+p_3-p_2+p_6$	3A		
			18 A	+	7M
		Standard Algorithm	7A	+	8M

The surprising result is that this modfied algorithm ends upt with one multiplication less than the standard algorithm. Of course the price to be payed is a substantially larger number of additions. The new algorithm would therefore be slower, whenever a multiplication and an addition have roughly the same cost. This would be true on most modern CPUs, and when the matrices are really just elementary 2×2 matrices.

The picture changes, when the matrices are block-matrices of considerable size. If we assume that the computation of the $p_1, \ldots, p_7$ is performed with the standard $O(n^3)$ matrix multiplication algorithm, one of those $O(n^3)$ operatios has been saved, and only additional $O(n^2)$ matrix addition operations have been introduced. Additionally, when the matrices are sufficiently large, these multiplications can be performed recursively by the same algorithm. In a parallel environment it is important, that all the $p_1, \ldots, p_7$ can be computed in parallel.

To analyse the effect of this recursion, we consider matrices with

$$\dim(A) = \dim(B) = \dim(C) = m2^k$$

and define a corresponding recursive class of algorithms $\mathcal{M}_{m,k}$, defined by

$\mathcal{M}_{m,0}$: Standard algorithm for multiplying two $m \times m$ matrices using m^3 elementary multiplications and $m^2(m-1)$ additions.

$\mathcal{M}_{m,k}, k > 0$: Application of algorithm **R** to

$$A, B, C \in \mathbb{R}^{m2^k \times m2^k}$$

using a block partion with

$$a_{i,j}, b_{i,j}, c_{i,j} \in \mathbb{R}^{m2^{k-1} \times m2^{k-1}}, \quad \text{for } i, j = 1, 2$$

and applying algorithm $\mathcal{M}_{m,k-1}$ to the 7 multiplications occuring in the execution of algorithm **R**.

We can now prove the following complexity result.

Algorithm $\mathcal{M}_{m,k}$ needs m^3 multiplications and $(5+m)m^2 7^k - 6(m2^k)^2$ additions.

This is proved by induction.

$k = 0$: Multiplications:

$$m^3 7^0 = m^3$$

Additions:

$$(5+m)m^2 7^0 - 6(m2^0)^2 = (m^3 - m^2)$$

$k > 0$: Multiplications:

$$7\text{Mult}(\mathcal{M}_{m,k-1}) = 77^{k-1} = 7^k$$

Additions:

$$18(m2^{k-1})^2 + \text{Add}(\mathcal{M}_{m,k-1}) = 18m^2 2^{2k-2} + (5+m)m^2 7^{k-1} - 6(m2^{k-1})^2 = (5+m)m^2 7^k - 6(m2^k)^2$$

Therefore, the product of two $n \times n$ matrices can be computed with less than $28n^{\log_2 7}$ operations. For general (odd) n, algorithm $\mathcal{M}_{m,k}$ must be modified to include a suitable treatment of the "rest". One possiblity is, to extend the matrices by a 0 row and column.

Since $\log_2 7 \approx 2.807 < 3$, we have found a better complexity algorithm than standard matrix multiplication. Since the initial paper by Strassen many more variants ofofast matrix multiplication algorithms have been devloped. The best algorithm presently known has a complexity of $O(n^{2.376\cdots})$ and quite possiblyeven faster algorithms exist. At presnt a best lower bound for matrix multiplication algorithms is uunknown.

The asymptotic complexity advantage of the fast multiplication algorithms, however, will only dominate for sufficiently large matrices. For small n, the standard algorithm will be better. In practice it does not suffice to simply compute the number of arithmetic operations for each n to decide wich algorithm should be preferred. As shown above in section 4.7.1, cache effects and other machine dependent features like vector lenths may play an as important role as the lower numberof operations. Additionally, it should be noted that the Strassen algorithm is less stable than conventional multiplication, though it has been shown that it can still be used within usual matrix algorithms, see Higham [15]. Blocking may in any case have different stability properties than the conventional algorithm. In particular, the use of exact scalar product routines for computing improved matrix products is more difficult.

More information and numerous adidtional references on the design and optimization of dense linear algebra computations can be found in [11]. On the use of Strassen's multiplication in linear systems solvers, see Using Strassen's Algorithm to Accelerate the Solution of Linear Systems[6] [4].

[6]http://www.nas.nasa.gov/NAS/RNRreports/dbailey/RNR-90-001/RNR-90-001.html

Literaturverzeichnis

[1] O. Axelsson, *Iterative solution methods*, Cambridge University Press, 1994.

[2] J. Backus, *Can programming be liberated from the von neumann style? a functional style and its algebra of programs*, Communications of the ACM, 21 (1978).

[3] D. Bailey, *RISC microprocessors and scientific computing*, RNR Technical Report 93-004, NASA Ames Research Center, March 1993.

[4] D. Bailey, K. Lee, and H. Simon, *Using Strassen's algorithm to accelerate the solution of linear systems*, RNR Technical Report 90-001, NASA Ames Research Center, January 1990.

[5] F. Bause and W. Tölle, *C++ für Programmierer*, Vieweg, Braunschweig, 1990.

[6] W. Bouknight and et. al., *The ILLIAC-IV system*, in Proc. IEEE, 1972, pp. 369 – 388.

[7] J. Dongarra, *Performance of various computers using standard linear equation software*, Tech. Report CS 89-85, Computer Science Department, Univ. of Tennessee, Knoxville, 1989. Version January 7, 1995.

[8] M. Eigen and R. Winkler, *Das Spiel: Naturgesetze steuern den Zufall*, Piper, München, Zürich, 1975.

[9] M. Flynn, *Some computer organizations and their effectiveness*, IEEE Trans. Computers, 21 (1972), pp. 948 – 960.

[10] G. Fox, S. Otto, and A. Hey, *Matrix algorithms on a hypercube I: Matrix multiplication.*, Parallel Computing, 4 (1987), pp. 17–31.

[11] K. Gallivan, R. Plemmons, and H. Sameh, *Parallel algorithms for dense linear algebra computations*, SIAM Review, 32 (1990), pp. 54–135.

[12] W. Giloi, *Rechnerarchitektur*, Springer, 2 ed., 1993.

[13] J. Gustafson, *Reevaluating Amdahl's law*, Commun. ACM, 31 (1988), pp. 532 – 533.

[14] W. Hackbusch, *Iterative solution of large sparse systems of equations*, vol. 95 of Applied mathematical sciences, Springer Verlag, 1994.

[15] N. J. Higham, *Exploiting fast matrix multiplication within the level 3 BLAS*, 16 (1990), pp. 352–368.

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

[17] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreich, *Gordian: VLSI placement by quadratic programming and slicing optimization*, IEEE Trans. Computer-Aided Design, CAD-10 (1991), pp. 356–365.

[18] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical recipes in C : the art of scientific computing*, Cambridge University Press, second ed., 1992.

[19] H. Regler and U. Rüde, *Layout optimization with algebraic multigrid methods (AMG)*, in Proceedings of the Sixth Copper Mountain Conference on Multigrid Methods, Copper Mountain, April 4-9, 1993, Conference Publication, NASA, 1993.

[20] J. Ruge and K. Stüben, *Efficient solution of finite difference and finite element equations by algebraic multigrid (AMG)*, Arbeitspapiere der GMD, 89 (1984).

[21] J. Stoer, *Einführung in die Numerische Mathematik I*, Springer Verlag, Berlin, Heidelberg, New York, 1979.

[22] J. Stoer and R. Bulirsch, *Einführung in die Numerische Mathematik II*, Springer Verlag, Berlin, Heidelberg, New York, 1978.

[23] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[24] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, 1991. Second Edition.

[25] P. Wesseling, *An introduction to multigrid methods*, John Wiley & Sons, Chichester, 1992.

[26] S. Winograd, *On multiplication of 2x2 matices*, Linear Algebra and its Applications, 4 (1971), pp. 381–388.