Optimierte Programmierung

für die naturwissenschaftliche Anwendung

12. Januar 1995

Abstract: *Die optimierte Programmierung von skalaren Programmen ermöglicht die effizientere Nutzung von Ressourcen, sowohl in Hinsicht auf den Speicher als auch in Hinsicht auf die Rechenzeit. Auch im Parallel–Computing ist die Optimierung der skalaren Programmteile von Vorteil, da zum einen die Parallelisierung überflüssig wird, zum anderen für langsame skalare Teile die Prozessorzahl dementsprechend hoch sein muß, um die Mängel des skalaren Codes wieder wett zu machen.*

Fehler und Anregungen an `c4049 at rphs1.physik.uni-regensburg.de`

Inhaltsverzeichnis

Kapitel 1

Programmierung

Schlechter Programmierstil hat nicht nur „ästhetische“ Nachteile, sondern erschwert generell die Arbeit, egal ob in Simulationen der Computational Physics, bei Auswertungen in der Experimentalphysik oder bei numerischen Berechnungen für die theoretische Physik. Viele Bücher der Informatik sind einseitig auf informatische Probleme ausgerichtet (Sortieren, Datenstrukturen), während in der naturwissenschaftliche Programmierung vor allem die effiziente Organisation von Gleitkommarechnungen erforderlich ist. Beschleunigung von solchen Rechnungen führt zu kürzeren „Turnaround–Zeiten“, d.h. die Programme sind schneller durchgerechnet und man hat mehr Zeit, etwas nützliches zu tun.

1.1 Programmiersprachen auf verschiedenen Architekturen

Die Wahl der richtigen Programmiersprache kann die Arbeit an einem Projekt erheblich erleichtern. Prinzipiell ist auch zu beachten, daß manche Programme auf andere Plattformen portiert werden sollen, aber nicht jede Programmiersprache auf jedem Rechner gleichgut läuft. Wenn sich die notwendigkeit der Portierung abzeichnet, sollte man sich frühzeitig nach der vorhandenen Software erkundigen.

1.1.1 General-Purpose

Pascal: Vor allem für didaktische Zwecke

+ „sicheres“ und „schönes“ Programmieren stehen im Vordergrund.
– Kaum schnelle Compiler auf richtigen Computern, auf vielen Installationen in Forschungsinsituten nicht vorhanden, Turbo–Pascal ist nicht Standard und deswegen nicht portabel.
– Viele elementare Funktionen müssen selbst programmiert werden.

C: Betriebssystemprogrammierung, vor allem unter Unix

+ Sehr verbreitet auf Low–Cost–Rechnern (Gnu-C Public Domain) und oft recht schnell.
+ Auf Unix–Systemem die Standardprogrammiersprache für Betriebssystemprogrammierung.
– Nicht immer sehr sicher, da Pointer nur sehr schwer vom Compiler überprüft werden können (auch in F90 etc).

- Keine komplexen Zahlen ⇒ Keine komplexen Intrinsics.
- Vektorisiert auf vielen Hochleistungsrechnern. nicht (NEC, Fujitsu, Hitachi)

Fortran77: Formula Translation, klassische Programmiersprache der naturwissenschaftlichen Anwendung

+ Sehr zuverlässig (Check von Arraygrenzen auf allen wichtigen Compilern vorhanden).
+ „Einfache“ Programmiersprache.
+ Umfangreiche numerische Software.
+ Komplexer Datentyp mit dazugehörigen Intrinsics.
- I/O oft umständlich.
- Nicht die modernsten Datenstrukturen.
- Im PC-Bereich (auch SUN) über lange Jahre nur sehr schlechte Compiler.

Fortran90

+ Alles, was C++ hat, mit der Programmiersicherheit von Fortran77 .
- recht umfangreich.
- noch relativ wenig Installationen.

Generell Vorsicht bei der Verwendung sehr spezieller Datenstrukturen. Oft ist bei ihrer Verwendung die Programm–Optimierung sehr erschwert.
Soweit nichts dagegenspricht, die Programmiersprache verwenden, in der man sich wohlfühlt, weil das wahrscheinlich auch die Sprache ist, in der einem am wenigsten Fehler unterlaufen!
Unübersichtliche Programme habe nichts mit Programmiersprachen zu tun, sondern sind ein Ergebnis disziplinloser Rumprogrammiererei !

1.1.2 Special Purpose Languages

+ Vor allem Interpreter: Höhere Geschwindigkeit bei der Programmentwicklung, langwieriges Compilieren entfällt.
+ Umfangreiche mathematische Bibliotheken.
- Oft sehr langsame Abarbeitung der Programme, für rechenintensive Aufgaben ungeeignet.
- Gibt's häufig nicht auf Supercomputern.

APL

- Schon in den 60ern entwickelte IBM APL (für „A Programming Language“), in der beliebige Typen und Operationen mit diesen Typen vereinbart werden konnten, um z.B. die Matrizenverarbeitung zu vereinfachen. Die Flopraten für Matrizenverarbeitung sind hoch, da der Interpreter auf Array–Verarbeitung abgestimmt ist. Die weitgehend Linpack–kompatible ESSL–Library von IBM kann eingebunden werden.

- Durch die ungewohnte Befehlsstruktur, die notwendigen APL–Sonderzeichen auf der Tastatur (Hieroglyphen wie $\rightarrow, \perp, \triangle, \square, \bowtie$ statt „normaler“ Befehle) und die Beschränkung auf IBM–Rechner konnte sich diese Programmiersprache auf dem Softwaremarkt aber nicht durchsetzen.
- Beipiel für eine Array–Zuweisung in APL: Feldelement kann ein Skalar oder auch ein Vektor sein.
  `A ← 12 (3 4 5) 67`
  `A[1]` hat den Wert `12`,
  `A[2]` hat die Werte `3 4 5`
  `A[3]` hat den Wert `67`.

MAPLE Computer–Algebra–Programm,

- Auf einem C–Kernel basiert ⇒ auf jedem System mit C–Compiler installierbar, Syntax C–ählich, Befehlszeilen mit „;“ abschliessen, Zuweisungen erfolgen mit „:=“, Groß– und Kleinschreibung wird beachtet.
- Attraktiven Vorrat von Numerik–Funktionen, Aufruf ohne lästige Überträge von Feldparametern und Anlinken von Libraries:
  `gausselim(A,'rank','piv')` ist eine Gauß–Elimination auf Matrix `A` mit Rangbestimmung und Pivotisierung
- Kann mit beliebig vorgegebener Stellenzahl rechnen

MATHEMATICA wie MAPLE

MATLAB dient speziell zur Verarbeitung von Matrizen (MATrix LABoratory). So enthält MATLAB in der Workstation–Version die Exponentialfunktionen von Matrizen nach drei verschiedenen Algorithmen. Die Syntax ist FORTRAN–ähnlich, Zuweisungen erfolgen mit =, es ist kein Abschluß von Befehlszeilen notwendig, Variablennamen können für beliebige Typen, Skalare, Vektoren, Matrizen, usw. stehen. Vorsicht, einziger Datentyp ist Array of Real!
Mathematische Operationen werden dann als Operation auf der Matrix etc. verstanden, im Gegensatz zu FORTRAN90 und MAPLE, wo die Operationen und Funktionen auf die Matrixelemente angewendet werden. Eine Matrizenmultiplikation der Matrizen A und B erfolgt einfach durch $A * B$, ohne lästige Indizes. Die Rechengenauigkeit ist als `REAL*8` voreingestellt.
Durch seine spezielle Ausrichtung übertrifft dieser Interpreter in der reinen Numerik der linearen Algebra auf PC's auch gute Compiler in den Flopraten (optimierte Programmierung in Assembler, z.B. durch Übergehen von überflüssigen Zwischenspeicher–Schritten etc).
Gauss wie MATLAB, Gaussian92 ist ein Chemie–Anwendungsprogramm

1.1.3 Parameterüberträge in Unterprogramme

Nicht alle Programmiersprachen übergeben Daten auf die selbe Art und Weise an ihre Unterprogramme. Die Kenntnis von der Art der Datenübergabe ist aber sehr wichtig beim Umschreiben von Programmen sowie bei der Optimierung, da der Compiler nicht das tut, was man naiverweise voraussetzt. Spezialitäten wie globale Variable oder Common–Blöcke sollen hier nicht behandelt werden, sondern nur Parameterübergaben in den Argumentlisten von Unterprogrammen.

Sprache	Datentyp		Übergabemodus
Pascal	Var–Parameter	veränderbar	Übergabe über Zeiger der Anfangsadresse
	Wert–Parameter	nicht veränderbar	Kurze Datentypen als Wert, lange über den Zeiger der Anfangsadresse
C	Einfacher Datentyp	nicht veränderbar	Übergabe einer Kopie, die im Unterprogramm zwar verändert werden kann, aber nicht ans Hauptprogramm zurückgeliefert wird.
	Pointer auf einfachen Datentyp	veränderbar	Veränderung des einfachen Datentyps durch Veränderung der Speicherstelle, auf die der Pointer zeigt
F77	Alle Daten	veränderbar	Übergabe des Zeigers in der Unterprogrammroutine, alle Speicherstellen können verändert werden, leider auch ganzzahlkonstanten (Verändern der 1 reduziert die Lebenserwartung von Programmen erheblich)
F90	Daten	wie F77	
	Daten mit `intent in`	nicht veränderbar	Veränderung führt zu Fehlermeldung
	Daten mit `intent out`	veränderbar	Einlesen aus dem Hauptprogramm führt zu Fehlermeldung
MATLAB	Daten in den Argumentlisten		`[a]=sub(b)`, `a` ist nur Ausgabevariable, `b` ist nur Eingabevariable

1.2 Programmentwicklung

1.2.1 Programmierstrategien

Erst denken, dann hacken !

Programmierumgebung: Vernünftige Editoren und Compiler suchen: z.B. Mausorientierte Editoren, F77 nicht auf der SUN entwickeln, C nicht auf IBM370 Maschinen, History beim Compilieren nutzen, wenn immer wieder dieselben Befehle eingegeben werden.

Datenmengen und Datenstrukturen:

1. Speicherplatzanforderungen
2. Rechenzeitbedarf
3. Voraussichtliche Codegröße: Zerlegung in Unterprogramme, Unterprogramme getrennt entwickeln, spart Zeit und Resourcen
4. Vorhandener Code: Übernehmen, Umschreiben, neu Programmieren ? Code aus Programmbibliotheken oder der Literatur verwenden, wenn ja, gibt's die Codes online, soll man sie eintippen oder scannen ?
5. Was sind Variable, was sind Parameter?
6. Was sind Indizes, was sind Pointer?
7. Bei welchen Datenänderungen neucompilieren, welche Daten durch Inputfiles ändern?

8. Wieviele Inputfiles (MSDOS: Autoexec.bat, Command.Com, Config.sys, io.sys ist nicht der beste Stil, da man bei Systemmodifikationen mehrere Startupskripten Files modifizieren muß)?

Ein– und Ausgabe: Die Ein– und Ausgabe numerisch rechenintensiver Programme sollte sich auf das Nötigste beschränken, um die Programmausführung nicht zu verzögern. Graphische Online–Visualisierungen kosten ebenfalls relativ viel Rechenzeit. Wenn man nicht wirklich am Geschehen interessiert ist, keine Online–Graphikausgabe an die Produktionsläufe hängen.

Wieviele Programme: Datenvisualisierung online oder 1 Datengenerierungsprogramm, 1 Visualisierungsprogramm: Turbo–Pascal– gegen Supercomputing–Philosophie. Es gibt aber genug Public–Domain Programme für Graphik (z.B. Gnuplot, pgplot), so daß man nicht gezwungen ist, das Rad selber zu erfinden und Spline–Interpolationen (gibt's leider bei gnu noch nicht) etc. selber programmieren muß, um seine Daten zu visualisieren.

Portabilität: Soll das Programm nur auf 1 Hardwareplattform laufen ⇒ Special Features ausnutzen, sonst meiden wie die Pest und sich strikt an den Standard halten (macht bei I/O immer Probleme).

Include–und Header–Files sorgen bei häufigem Plattformwechsel für Probleme, wenn auf verschiedenen Betriebssystemen die Konventionen für Filenamen verschieden sind.

Unterprogramme: Die Verwendung von Unterprogrammen erleichtert die Lesbarkeit des Quelltextes, das Austesten und die Fehlersuche in neuen Programmen. Der Rechenzeitaufwand für Programmläufe von undurchsichtigen fehlerhaften Programmen übersteigt in der Regel den Rechenzeitaufwand der `SUBROUTINE`–Aufrufe. Unterprogrammaufrufe sind bei Verwendung von optimierenden Compilern in der Regel nicht rechenzeitkritisch (Inlining). Unterprogramme können bequem auf dem PC geschrieben und getestet werden und dann in ein Hauptprogramm eingebaut werden. Unterprogramme sollten nach Möglichkeit nicht länger als 1 Bildschirmseite sein, da dann das Debuggen extrem schwierig wird.

Amdahls Law: Anteil der einzelnen Programmteile an der Gesamtrechenzeit beachten: Initialisierungen „sicher", Kernel „schnell" ⇒ für die Kernel die schnellsten Algorithmen wählen, für die Initialisierungen die übersichtlichsten. Optimiert werden

- rechenzeitintensive Unterprogramme bei Online-Verarbeitung,
- rechenzeitintensive Produktionsprogramme (alle Monte–Carlo und Molekulardynamik–Simulationen),
- häufig verwendete Programme.

EDV gegen Physik: Unterschied in den Datenstrukturen naturwissenschaftliche Programmierung # Informatik: Viele Daten, wenig Strukturen gegen wenig Daten, viele Strukturen.

Destruktives Programmieren: In der Regel kombiniert man bereits lauffähige Programme so miteinander, daß man für neue Programme nur einzelne Unterprogrammroutinen neu schreiben muß. Dadurch macht man dieselben Syntaxfehler immer nur einmal, Programmierfehler sind leichter zu finden. Diese Strategie wird durch Einsatz von Unterprogrammen erleichtert, denen es egal ist, von welchem Hauptprogramm aus sie aufgerufen werden.

Keine Algorithmen programmieren, bei denen man nicht weiß, was man tut. Dies gilt insbesondere für Abbruchkriterien!

1.2.2 Preprocessing und optimierende Compiler

Der Computer führt ein Programm eventuell nicht so aus, wie es im Quelltext vorliegt. Umstellen von Programmteilen, Ersetzen von Funktionen etc. ermöglichen eine schnellere Programmausführung. Je nach Fabrikat und Befehl optimiert ein Preprocessor (Textverarbeitungsprogramm) oder der Compiler selber.
Angesprochen werden die Optimierungsstufen des Compilers in der Regel mit der Option `-O`, gefolgt von einer Zahl für die Optimierungsstufe (IBM RISC 6000 `O1–O3`, Intel IPSC und Paragon `O1–O4`, SUN `O1–O4`). Es empfielt sich in jedem Fall das Studium der Man–Pages des Compilers.
Manchmal führen Programmierfehler, die ohne Optimierungsstufen fast keine Auswirkung haben, bei Optimierung zu Katastrophen, da die Datenstrukturen vom Compiler völlig umgestellt werden. Zugriff auf vorher leere Felder wird dann zum Zugriff etwa auf Betriebssystemadressen.
In seltenen Fällen sind auch die Optimierer fehlerhaft. Dann ist mit dem Auftreten von sogenannten Heisenbugs zu rechnen: Jede Veränderung des Codes oder Änderung der Compiler–Optionen (in Fortran: `-C`), Verwendung des Debuggers etc. kann dazu führen, daß der Fehler, der beobachtet werden soll, nicht mehr auftritt. Läßt man den Code wieder in der ursprünglichen Form laufen, hat man wieder die alten Fehler.
Programmoptimierung durch Compiler und Preprozessoren:
Je übersichtlicher ein Programm geschrieben ist, deso wahrscheinlicher ist es, daß der Compiler Optimierungen vornehmen kann.
Programmoptimierung ist leider oft ziemlich speicherintensiv, da der Compiler eine umfangreiche Buchhaltung über die vorhandenen Daten und Abläufe führen muß. Je kürzer die Bestandteile eines Codes sind (Unterprogramme, Funktionen, Code in Schleifen), desto mehr Aussicht auf Erfolg hat der Optimierer. Auf der 10–Prozessor IRIS des Rechenzentrums erfordert die Compilation mancher Programme in der höchsten Optimierungsstufe bis 300 MB, auf den PC's unter WatcomFortran77 ist eine Standardmeldung, daß aus Speicherplatzgründen nicht weiter optimiert werden konnte. Hier hilft oft das Zerlegen des Codes in weitere Unterprogramme.

Inlining: Bei Unterprogrammaufrufen wird oft der Inhalt der Register auf den Stack gelegt. Bei kurzen Unterprogrammen ist daher der Overhang oft höher als die eigentliche Ausführungszeit. Dies kann durch Inlining verhindert werde, da hier das Unterprogramm einfach ins Hauptprogramm übertragen wird. Inlining wird je nach Art des Unterprogramms (mit oder ohne globale Variablen) vom Preprocessor oder vom Compiler vorgenommen.

Code Löschen: Zur Compilezeit bekannte Block–`if`'s können aus dem Code entfernt werden, wenn auf Parameter/Konstanten abgefragt wird.
Quellcode, mit `if`:

```
parameter (a=5)
if (a.gt.4) then
  b=a*4
else
  b=3
endif
```

Optimiert:

```
parameter (a=5)
b=a*4
```


Ersetzen von Ausdrücken Sind z.B. algebraische Ausdrücke bereits zur Compilezeit bekannt (Verwendung von Konstanten/Parameter, Variablen können Ärger machen), werden sie bereits vom Compiler berechnet und eingesetzt.

```
parameter (a=5)
b=a*4
```

Optimiert:

```
parameter (a=5)
b=20
```

Ersetzen langsamer durch schnelle Funktionen: Divisionen brauchen sehr viel Rechenzeit. Bei Division durch Potenzen von 2 kann auch eine Bitshift–Operation verwendet werden, die viel schneller ist. Divisionen durch Konstanten werden schon zur Compile–Zeit in Multiplikationen umgewandelt.

Chaining: Unnötiges Abspeichern von Zwischenresultaten, etwa bei Summation in eine Variable, wird unterdrückt, wenn das Ergebnis wieder für den folgenden Rechenschritt benötigt wird (etwa bei Summation in eine Variable).

```
a=c*(a+b)
```

gliedert der Computer auf in

```
a=a+b
a=a*c
```

Abgearbeitet wird folgendermaßen (angenommen, es existieren nur numerische Operationen mit 2 Argumenten):
1a) Hole `a` aus dem Speicher.
1b) Hole `b` aus dem Speicher.
1c) Addiere `a` und `b`.
1d) Speichere `a` wieder ab.
2a) Hole `a` aus dem Speicher.
2b) Hole `c` aus dem Speicher.
2c) Multipliziere `a` und `c`.
2d) Speichere `a` wieder ab.
Bei Chaining entfallen die Schritte 1d) und 2a), wenn jede der angegebenen Operationen einen Taktzyklus braucht (realistisch für RISC's), wird der Programmteil damit um 25% beschleunigt.

Verwendung bereits berechneter Ausdrücke: Viele Compiler legen Zwischenergebnisse in Registern ab, und so können bei mehrfacher Verwendung desselben Zwischenergebnisses Rechenoperationen gespart werden:
Quellcode, 6 Additionen:

```
u=c+a+b
v=d+a+b
w=e+a+b
```

Optimiert, 4 Additionen:

```
help=a+b
u=c+help
v=d+help
w=e+help
```

Ob ein Compiler `b+a` als `a+b` erkennt, ist Glückssache, je „symmetrischer“ ein Code ist, desto besser ist die Aussicht auf Erfolg.

```
u=c+a+b
v=d+a+b
w=e+b+a
```

wird eventuell durch 5 Additionen berechnet. Dies gilt sinngemäß für alle Rechenoperationen, besonders bei Divisionen (Siehe auch Loop–Invarianten).

Ersetzen duch maschinenkodierte Routinen: Viele Rechner verfügen über Assembler–codierte Routinen zur Abarbeitung linearer Algebra. Erkennt der Compiler z.B. eine Matrix–Matrix–Multiplikation, kann er sie durch seine eigene Routine ersetzen (gibt eine Nachricht beim Compilieren).

NoIEEE: Zur Abarbeitung von Floating–Point Operationen existiert ein Standard von IEEE. Wenn niedrigere Präzision ausreicht (Programläufe mit und ohne IEEE–Genauigkeit vergleichen), kann der Rechner auf schnellere, weniger genaue Routinen zugreifen, wenn entsprechende Compileroptionen spezifiziert wurden.

Loop–Invarianten: Werden innerhalb eines Loops Variable unnötig immer wieder berechnet, wird die Berechnung vor den entsprechenden Loop gezogen.

Quellcode:

```
      do 10 i=1,m
        a=1./m
        b(i)=b(i)*a
10    continue
```

Optimiert:

```
      a=1./m
      do 10 i=1,m
        b(i)=b(i)*a
10    continue
```

Die Berechnung von `a=1./m` kann genausogut außerhalb des Loops stattfinden. Allerdings ist bei Divisionen Vorsicht geboten: Der IEEE–Standard fordert die Genauigkeit von Rechengergebnissen bis auf das letzte Bit. Aufgrung von Rundungsfehlern kann sich im

```
a=1./m
b=2.*a
c=2./m
```

nebenstehenden Programmierbeispiel `b` im letzten Bit durch Rundungsfehler von `c` unterscheiden, der Compiler zieht dann `1./m` nicht als Loopinvariante aus einer Schleife. In Fortran90 ist explizit gefordert, daß solche Anwendungen des Assotiativgesetzes vom Compiler druchgeführt werden dürfen, in Fortran77 sollte man bei Divisionen die Loopinvarianten von Hand programmieren.

Vertauschen von Loops: Der Compiler kann die Reihenfolge zweier ineinandergeschachtelter Loops vertauschen, wenn dadurch die Speicherzugriffe zeitgünstiger ausfallen.

Loop–Ersetzung: Ist ein Loop sehr kurz, kann der Compiler die Operationen ohne Loop schreiben.

Quellcode:

```
      DO 10 i=1,3
        c(i)=0.0
10    CONTINUE
```

Optimiert:

```
      c(1)=0.0
      c(2)=0.0
      c(3)=0.0
```

Loop–Unrolling: Lange Schleifen werden durch kürzere ersetzt, wenn durch den Loop zuviel Overhang erzeugt wird:

Quellcode:

```
      do 50 i = mp1,5*n
        dx(i) = da*dx(i)
   50 continue
```

Optimiert:

```
      do 50 i = mp1,5*n,5
        dx(i) = da*dx(i)
        dx(i+1) = da*dx(i+1)
        dx(i+2) = da*dx(i+2)
        dx(i+3) = da*dx(i+3)
        dx(i+4) = da*dx(i+4)
   50 continue
```

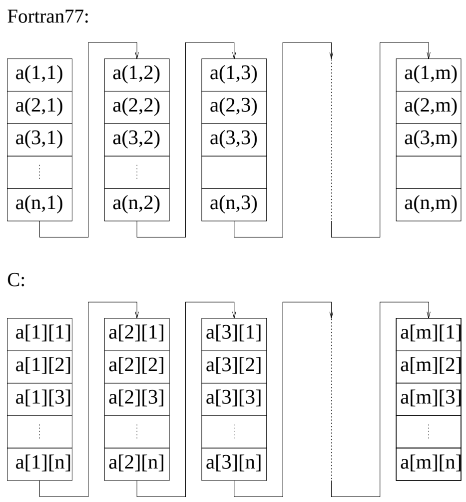
Verringerung der Felddimension: Es ist möglich, n–dimensionale Felder in 1–dimensionale umzuformen. Dadurch muß statt der n Indizes nur ein einziger berechnet werden.

1.3 Speicherorganisation und Programmierung

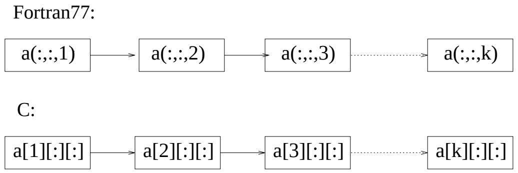
1.3.1 Datenauslegung von Arrays

Mehrdimensionale Arrays werden vom Compiler auf einen Vektor von aufenanderfolgenden Speicherzellen im Memory abgebildet. Die Art der Abbildung, nämlich zeilenweise oder Spaltenweise, ist von einer Programmiersprache verschieden.
Ungeschickte Addressierung von Speicherzellen führt jedoch dazu, daß
immer Feldelemente außerhalb des Cache angesprochen werden und sich die Laufzeiten der Programme verlängern.

Lineare Auslegung des 2–dimensionalen Arrays `a(l,m)` im Speicher :



Lineare Auslegung des 3–dimensionalen Arrays `a(l,m,n)` im Speicher :



Slogan: Der erste Index wackelt in Forgram am meisten, in C der letzte.

Beispiel: Skalierung einer Matrix, $\underline{A} = b \cdot \underline{A}$

günstig (Zugriff auf aufeinanderfolgende Speicherzellen):

```
      DO 10 i=1,l
        DO 20 j=1,l
          A(j,i)=b*A(j,i)
20      CONTINUE
10    CONTINUE
```

ungünstig (Überspringen von je `l` Speicherzellen beim Datenzugriff):

```
      DO 10 i=1,l
        DO 20 j=1,l
          (A(i,j)=b*A(i,j)
20      CONTINUE
10    CONTINUE
```

1.3.2 Hierarchische Speicherorganisation

Die Speicherhierarchie für Workstations und PC's sieht in etwa folgendermaßen aus (von links nach rechts in aufsteigender Reihenfolge der Zugriffszeiten/Speichergrößen). Die angegebenen Werte sind Größenordnungen:

CPU	Cache	Hauptspeicher	Virtueller Speicher (Platten)	
	0 KB - 1 MB	640KB - GB	MB - GB	Speichergröße
	$\mathcal{O}(1)$	$\mathcal{O}(10)$	$\mathcal{O}(50)$	Zugriffszeit
			in Taktzyklen	

Ziel der optimierten Programmierung ist das Operieren auf den lokalen=schnell zugänglichen Daten im Cache. Falsche Programmierung durch irreguläre Speicherzugriffe erhöht die Laufzeit manchmal um Größenordnungen, da aus dem jeweils langsameren Memory in den Cache nachgeladen werden muß (Cache–Miss, im Gegensatz zum Cache–Hit). Ein Problem ergibt sich bei indirekten Adressierungen, wenn statt auf einen Vektor mit den Elementen `real a(l)` durch indirekte Adressierung mit einem Vektor `integer i(l)` häufig auf `a(i(j))` zugegriffen werden muß: Je nach `i(l)` kann der Speicherzugriff sehr irregulär ausfallen.

1.3.3 Branching

Vorsicht bei der Verwendung von `if`'s (Branching). Möglicherweise wird in `if–else` auf sehr verschiedene Bereiche im Speicher zugegriffen, so daß ein Cache–Miss auftritt. Branching verzögert eventuell die Programmausführung mehr als nötig. Oft ist die Progammierung ohne if möglich:

falsch:

```
      DO 10 i=1,l
        DO 20 j=1,l
          IF(j.eq.l)THEN
            SUB1
          ELSE
            SUB2
          END IF
20      CONTINUE

10    CONTINUE
```

richtig:

```
      DO 10 i=1,l
        DO 20 j=1,(l-1)
          SUB2
20      CONTINUE
        j=l
        SUB1
10    CONTINUE
```

Bei Vektorrechnern hängt die Performance bei der Verwendung von `if`s von Hersteller, Modell und Compilerversion ab. Unbedingt die Manuals konsultieren.

1.3.4 Summation in Zwischenspeicher

Beispiel: $\vec{c} = \underline{A} * \vec{b}$ (Matrix-Vektor-Produkt)

Schlecht für RISC,PC:

```
      DO 10 i=1,l
        c(i)=0.0
        DO 20 j+1,m
          c(i)=c(i)+a(j,i)*b(j)
20      CONTINUE
10    CONTINUE
```

besser:

```
      DO 10 i=1,l
        d=0.0
        DO 20 j=1,m
          d=d+a(j,i)*b(j)
20      CONTINUE
        c(i)=d
10    CONTINUE
```

Durch die Verwendung eines indexlosen Zwischenspeichers spart man sich zwei von 5 Indexberechnungen, manchmal werden bei hoher Optimierungsstufe diese Ersetzungen auch automatisch ausgeführt.
Für manche Vektorrechner ist aus nicht–trivialen Gründen eventuell das linke Modell günstiger.
Vor Benutzung von Vektorrechnern Leute fragen, die sich auskennen.

1.4 Algorithmen und Big–O Terminologie

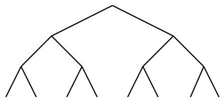
$\mathcal{O}$ ist das Landausches Größenordnungssymbol, das bedeutet, daß nur die höchste Potenz angegeben wird, von der eine Funktion abhängig ist, nicht die niedrigeren Potenzen oder Vorfaktoren. Manchmal werden auch die Vorfaktoren angegeben, $\mathcal{O}$ besagt dann, daß Terme niedrigerer Ordnung vernachlässigt werden. l ist im folgenden die Systemgröße.
Rechenregeln:
$\mathcal{O}(l^m + l^n) = \mathcal{O}(l^{\max(m,n)})$
$\mathcal{O}(l^m) * \mathcal{O}(l^n) = \mathcal{O}(l^{m+n})$
$\mathcal{O}(l^m + \text{const}) = \mathcal{O}(l^m)$
Beispiele:
Skalierung eines Vektors Y der Länge l: $\mathcal{O}(l)$
Skalierung einer Matrix A mit l Zeilen und Spalten: $\mathcal{O}(l^2)$
Direkte Multiplikation zweier Matrizen mit je l Zeilen und Spalten: $\mathcal{O}(l^3)$

Vorsicht: Die Aussagen der Big-O-Terminologie gelten nur für die Größenordnung. Bestimmend ist allerdings oft auch der Vorfaktor. Dadurch kann manchmal die Verwendung von „theoretisch“ ungünstigeren Algorithmen zu schnelleren Programmen führen, wenn l klein ist.

1.4.1 Divide–and–Conquer Algorithmen

Die einfachste Methode zur Beschleunigung von Programmen ist die Wahl von Algorithmen, die in der Big-O-Terminologie schneller sind. Am deutlichsten sieht man mögliche Unterschiede zwischen linearen und Divide–and–Conquer Algorithmen.
Beispiel: Austeilen von l Aufgabenblättern an l Vorlesungsteilnehmer.
1.) Jeder nimmt von dem Stoß, den er bekommt, ein Blatt und gibt den Stoß weiter : $\mathcal{O}(l)$
2.) Divide–and–conquer, jeder nimmt von dem Stoß, den er bekommt, ein Blatt und gibt jeweils die Hälfte des Stoßes an 2 verschiedene Nachbarn weiter (Jeder Schritt dauert zwar länger, aber Vorfaktoren werden nicht berücksichtigt): $\mathcal{O}(\log_2 l)$

Schritt	Ausgeteilt im n-ten Schritt	Ausgeteilt bis zum n-ten Schritt
1	1	1
2	2	3
3	4	7
4	8	15
n	2^{n-1}	$2^n - 1$



Weitere Beispiele:

Linsort: $\mathcal{O}(l^2) \rightarrow$ Quicksort: $\mathcal{O}(l \log l)$
Lineares Sortieren findet so statt, daß die obersten beiden Elemente der zu sortierenden Liste nach Bedarf umgeordnet werden, das 3. Element dann in der richtigen Reihenfolge eingefügt wird usw. Dies ist ein Prozeß, der $\mathcal{O}(l^2)$ erfordert. Teilt man die Liste in 2 Teillisten auf, erfordert das Sortieren der Teillisten nur $\mathcal{O}(2 \cdot (l/2)^2) = \mathcal{O}(l)^2/2$. Bei der Verwendung von sehr kurzen Teillisten und anschließendem geschickten Zusammenfügen der Teillisten erhält man einen Algorithmus, der nur noch $\mathcal{O}(l \ln l)$ erfordert – Quicksort. Heapsort ist ebenfalls ein Algorithmus, der sich bemüht, nur mit Teilen der zu sortierenden Liste zu arbeiten, Details in [Pre92].

Matrix–Matrix–Multiplikation: $\mathcal{O}(l^3) \rightarrow$ Winograds Variante von Strassens Algorithmus in [Loc92]: $\mathcal{O}(l^{2.4\cdots})$ Umordnen des Matrixprodukts mit $\mathcal{O}(l^3)$ in Blockmatrix–Produkte mit $1/2$, $1/4$, ...der Zeilenlänge, so daß die Blockmatrixprodukte nur $\mathcal{O}(l^3/8)$, $\mathcal{O}(l^3/64)$, ...der Rechenzeit benötigen. Anschließen kombiniert man die Teilprodukte mit Additionenn und Subtraktionen der Ordnung $\mathcal{O}(l^2)$.

Konventionelle Fouriertransformation: $\mathcal{O}(l^2) \rightarrow$ Schnelle Fouriertranformation durch Aufteilung in Untergitter für Systeme, die 2^n Gitterplätze haben: $\mathcal{O}(l \log_2 l)$
Die konventionelle diskrete Fouriertransformation erhält man aus der kontinuierlichen als ein Matrix–Vektor–Produkt($\mathcal{O}(l^2)$):

$$\sum_n f(x_n) = \sum_n \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(y) \mathrm{e}^{-\mathrm{i}x_n y} \mathrm{d}y \approx \sum_n \sum_m F(y_m) \mathrm{e}^{-\mathrm{i}x_n y_m} y_m = \sum_n \sum_m \tilde{F}_{nm} y_m$$

Man kann allerdings zeigen [Pre92], daß diskrete Fouriertransformationen für N Gitterpunkte als die Summer von 2 Fouriertransformationen für $N/2$ Gitterpunkte geschrieben werden kann, dieser Prozeß kann iteriert werden:

$$\mathcal{O}(l^2) \rightarrow 2\mathcal{O}((l/2)^2) \rightarrow \ldots \rightarrow \mathcal{O}(l \log_2 l)$$

Auch ohne Divide–and–conquer lassen sich Algorithmen verbessern:
Determinantenberechnung durch Enwicklung nach Zeilen/Spalten (obendrein numerisch instabil): $\mathcal{O}(l!) \rightarrow$ Trigonalisierung der Matrix und Berechnung der Determinante der Diagonale (mit dem Gauß–Eliminationsverfahren): $\mathcal{O}(l^3)$

1.4.2 Computersimulation und Big–O

Messungen:

Im Allgemeinen ist die Größenordnung an Rechenzeit, die in einer Simulation für Messungen verbraucht wird, sehr viel kleiner als für die Simulation selbst. Wenn nach jedem m-ten Update eines physikalischen Systems gemessen wird, ist der Anteil an der Messung im Programm von $\mathcal{O}(1/m)$. Es lohnt sich deshalb im allgemeinen nicht, Unterprogrammroutinen zur Messung auf Geschwindigkeit zu optimieren. Mehr Wert sollte auf Übersichtlichkeit gelegt werden, da eine falsche Meßroutine in einer richtigen Simulation viel Verwirrung stiftet, weil man den falschen Teil des Programms debuggt.

Dimension von Feldern:

Fortran–Veteranen bevorzugen in der Regel das Programmieren von 1–dimensionalen Feldern auch z.B bei 2–dimensionalen Systemen. Weil man Angst hat, daß das Programm zuviel Zeit an den Systemgrenzen verliert, nimmt man lieber die umständliche Programmierung und Fehlersuche in kauf. Abgesehen davon, daß ein optimierender Compiler im Umsetzen von 2–D Feldern in 1–D Feldern keine Probleme hat, zeigt eine einfache Abschätzung:
Rechenzeitverbrauch innerhalb des Systems $\mathcal{O}(l^2)$
Rechenzeitverbrauch auf den Rändern $\mathcal{O}(4l)$
Verhältnis $\mathcal{O}(4/l)$, mit steigender Systemgrösse und damit steigender Rechenzeit werden die Ränder immer unbedeutender. Außerdem ist es sehr wahrscheinlich, daß durch die lineare Abarbeitung der Felder ein Overhang entsteht, da die Geometrie dann durch zusätzliche Abfragen berücksichtigt werden muß.

1.4.3 Speicher gegen Rechenzeit

Oft ist es möglich ein Problem durch intensive Speichernutzung oder durch intensive Rechenleistung zu Parametrisieren, sehr gut ist das anhand eines Travelling Salesman auf einem Gitter zu sehen:
Im einfachsten Fall soll die kürzeste Verbindung von verschiedenen Orten gefunden werden:

Konnektivität: Bei wenigen Verbindungen: Abspeichern von Abstandstabellen (Speicherintensiv), sonst neuberechnen bei jedem Verändern der Route (rechenzeitintensiv): Entscheidend ist die Größe des Executables und der zur Verfügung stehende Speicher.

Parameter: Falls Verbindungsstrecken paarweise vertauscht werden sollten: Geradenschnittalgorithmen, Orientierungsalgorithmen oder Abstandsalgorithmen

Mies: Gut:

Hat das Problem ein krumme Metrik, so wird man auch für größere Systeme auf Abstandtabellen zurückgreifen müssen: Tabellen, die größer sind als der vorhandere Hauptspeicher verlängern die Ausführungszeiten von Programmen allerdings um Größenordnungen, wenn ständig geswappt werden muß.

1.5 Profiling und Zeitmessung

1.5.1 Was es alles für Zeiten gibt ...

User–Zeit ist die Zeit, die Programmteile verbraucht haben, die keine System–Calls enthielten.

Systemzeit ist die Zeit, die System–Calls verbraucht haben.

CPU–Zeit = User–Zeit + Systemzeit.

Echtzeit ist die Zeit, die während der Programmabarbeitung wirklich vergangen ist.

Echtzeit $\gg$ Cpu–Zeit
Unix–Maschinen haben Probleme bei der Speicherverwaltung vieler Jobs. Pflastern irgendwelche Idioten eine Maschine mit Programmen voll, wird nur noch vom Memory in den Hauptspeicher geswappt und umgekehrt. Da Quotierungen in Unix nicht ohne weiteres zu erstellen sind, ist dies der Grund für die weitgehende Unbrauchbarkeit von Unix–Clustern für numerisch rechenintensive Aufgaben, die Leistung der Prozessoren ist in der Regel ausreichend. Abhilfe schafft zum Teil ein NQS (Network Queuing System), an das alle Hintergrundjobs geschickt werden müssen. Das NQS läß nur wenige Jobs gleichzeitig laufen, so daß keine Performance–Einbrüche entstehen.
Echtzeit $<$ Cpu–Zeit
Auf Multiprozessor–Maschinen wie der ZEUS in Regensburg (10 MIPS–Prozessoren), CRAY in Jülich (4 bzw. 8 Prozessoren) und München (8 Prozessoren) sind intelligente Compiler in der Lage, bei unabhängigen Unterprogrammen oder langen Schleifen die Arbeit auf mehrere Prozessoren aufzuteilen. Dabei ist vorsicht geboten: Die Aufteilung der Arbeit wird zur Laufzeit vom System versucht, wenn keine effiziente Aufteilung möglich ist, geht sehr viel Systemzeit dabei verloren. Der Autor hatte einmal ein Programm mit 20.000 Sekunden User–Zeit und 25.000 Systemzeit, kurz darauf meldete sich die Systemadministration. Näheres zum Thema Shared–Memory Parallelisierung im Kapitel über Hardware.

1.5.2 Profiling unter Unix

Profiling bedeutet das Aufstellen eines Rechenzeit–„Profils" des Programms, daher der Name. Das Profiling hat nichts mit dem Login–File `.profile` der Unix–Bourne–Shell zu tun. Durch Compileroptionen und Hilfsprogramme wird dabei eine Tabelle über den Rechenzeitverbrauch der einzelnen Unterprogramme erstellt. Unter Unix: Kompilieren mit Option `-p` (für Profile):

`> f77 -p name.f` [RET]
`> a.out` [RET] (Programmlauf, Anlegen von `mon.out`)
`> prof > name.prof` [RET] (Schreiben der Profile-Tabelle nach `name.prof`)
oder
`> f77 -p -o name name.f` [RET]
`> name` [RET] (Programmlauf, Anlegen von `mon.out`)
`> prof name > name.prof` [RET] (Schreiben der Profile-Tablelle nach `name.prof`)
Beispiel für ein Profile:

```
\%time  cumsecs  #call  ms/call  name
 28.5  8062.33-1118859084    -0.01  .mul
 17.9 13137.24                    mcount
 17.7 18139.93   1000  5002.69  _nodeclus_
 13.2 21881.991696265920     0.00  _irandom_
  9.7 24636.621328432862     0.00  _findowner0_
  7.8 26832.36   1000  2195.74  _changenodelab2_
  2.0 27406.22   1000   573.86  _flipupdate_
  1.7 27901.17   1000   494.95  _update_
  1.1 28224.18   1000   323.01  _update4_
  0.2 28287.18     20  3150.00  _measenergy_
  0.1 28308.43     20  1062.50  _measmag_
  0.0 28319.15     20   536.00  _meassize_
```

Die Profiletabelle enthält, defaultmäßig geordnet nach dem Anteil an der Gesamtrechenzeit,

1. den Anteil an der Gesamtrechenzeit in %,
2. die kumulierte Ausführungszeit in Sekunden,
3. die Anzahl der Aufrufe für jedes Unterprogramm,
4. CPU–Sekunden pro Aufruf,
5. Namen der Subroutines, Funktionen und System–calls.

Vorsicht!

- Mit `-p` kompilierte Programme haben bei häufigen Unterprogrammaufrufen um 10–15% längere Ausführungszeiten (bei seltenen Unterprogrammaufrufen weniger als 1 % länger). Die Summe aller geprofileten Zeiten liegt dann 10–15% unter dem tatsächlichen Bedarf an CPU–Zeit für den Profiling–Programmlauf. Der Zeitverbrauch der Profiling–Routinen erscheint nicht im Profile, die Ergebnisse stimmen aber nur auf etwa 5 %.
- Das Executable (im obigen Beispiel `a.out`, `name`) darf vor dem Profilen mit `prof` weder gelöscht noch modifiziert werden, weil die Information über die Lage der Unterprogramme benötigt wird.

- Die `mon.out`-files können mehrere hundert KB umfassen. Wenn man knapp mit Plattenspeicher dran ist, `mon.out` so bald wie möglich löschen.

Für C–Programme ist die Option `-p` ebenfalls vorhanden, auf einigen Installationen kann die Option `-gp` benutzt werden, dementsprechend `gprof` statt `prof`. Falls irgendwas nicht klappt, Handbücher und Systemverwalter konsultieren.

1.5.3 Amdahls Gesetz

Durch Optimierung eines Unterprogramms, das $n\%$ eines Programmlaufs beansprucht, kann das Programm maximal um den Faktor $(100\%)/(100\% - n\%)$ beschleunigt werden.
Wenn man ein Unterprogramm optimiert, das 5% der Rechenzeit frißt, wird das Programm um höchstens 5% schneller. Deshalb müssen zuerst die Unterprogramme optimiert werden, die am meisten Rechenzeit beanspruchen, die Optimierung der übrigen Routinen ergibt eventuell keinen Zeitgewinn mehr.

1.5.4 Sonstiges Profiling

Auf PC's gibt es keine Standardumgebung für Profiling–Tools. Vektorrechner haben im Allgemeinen spezielle Profiling–Tools, um nicht nur den Zeitanteil der einzelnen Subroutines zu ermitteln, sondern auch für die einzelnen Schleifen den Vektorisierungsgrad zu ermitteln. Auf den Vektorrechnern der KFA Jülich wurde ein eigenes Profilingtool namens Looptool entwickelt, das Profiling und Vektorisierungsanalysen erstellt.
Auf der Cray existieren darüberhinaus X–basierte Analysetools (Atexpert, Atscope), die in hochauflösender Graphik Säulendiagramme des Rechenzeitbedarfs erstellen.
Auf Parallelrechnern existieren bis jetzt nur sehr rudimentäre Möglichkeiten, solche Analysen durchzuführen, die Tools arbeiten in der Regel auch nicht immer fehlerfrei.

1.5.5 Subroutine–Namen zur Zeitmessung

Man kann Zeitmessungen auch direkt über die Systemroutinen vornehmen. Diese System–Calls sind allerdings herstellerabhängig, das Ergebnis ist vom Typ `REAL*8`, vor der Benutzung empfielt sich jedoch immer die Konsultation der Handbücher (in diesem Fall: Language Reference Manual), da die Einheiten manchmal in Sekunden, hundertstel Sekunden oder Mikrosekunden angegeben werden.

Cray	`call second()`
IBM VS Fortran	`call cputime()`
IBM AIX	`call mclock()`
Intel	`call dclock()`
SUN	`t=secnds(u)` `t` und `u` sind `Real*4`
IRIS	`()` nicht bekannt

Zu beachten ist, daß bei Mehrbenutzersystemen die CPU–Zeit und nicht die Systemzeit/-Echtzeit gemessen werden muß.
In seltenen Fällen kann es notwendig sein, auch den Verbrauch der System–Zeit während des

Programmlaufs zu messen, wenn z.B. sehr viel geswappt wird. Für solche Messungen sollte die Maschine leer sein (dedizierte Messung, bzw. „Dedicated Machine").

Wenn man nur an der Gesamtlaufzeit eines Programms interessiert ist, kann man die Laufzeit auch mit dem `time`–Kommando bestimmen, für das Executable `a.out`:

```
time a.out
```

Will man die Ausgabe des Time-Kommandos in ein File umlenken, muß man dabei beachten, daß `time` nicht auf Standardoutput (Umlenkung durch > `filename`), sondern nach Standard–Err schreibt. In der C–Shell erfolgt die Umlenkung mit `>& filename`, in der K–Shell mit 2 > `filename`. Die Umlenkung von Compile–Messages auf Unix–Systemen ist ebenfalls nur durch Umlenkung von Standard–Err möglich. Beispiel:

```
rphs7:/home3/morgenstern/c4049> time ps
  PID TT STAT  TIME COMMAND
 4187 p2 S     0:00 -tcsh (tcsh)
 4205 p2 R     0:00 ps
0.120u 0.270s 0:00.58 67.2% 0+160k 10+0io 7pf+0w
```

Etwas komfortabler ist die Ausgabe mit dem neueren Time–Kommando in `/usr/bin`, Beispiel:

```
rphs7:/home3/morgenstern/c4049> /usr/bin/time ps
  PID TT STAT  TIME COMMAND
 4187 p2 S     0:00 -tcsh (tcsh)
 4206 p2 S     0:00 /usr/bin/time ps
 4207 p2 R     0:00 ps
        0.4 real         0.0 user         0.3 sys
```

1.5.6 Speicherbedarf

Größe Executables erfordern Laufzeiten, die deutlich höher als die erforderliche CPU–Zeit, wenn das Programm mehr Speicher erfordert als im Hauptspeicher vorhanden ist. Dann werden Daten ständig von und auf Platte geswappt, was in Echtzeit sehr lange dauert.

Der Speicherbedarf eines Executables kann in jedem Betriebssystem durch das Auflisten des Directories festgestellt werden. Dabei handelt es sich aber nur um die Speicheranforderungen auf der Platte, wobei im Code noch nicht der Speicherbedarf für anzulegende Felder berücksichtigt ist.
Unter Unix kann man für das Executable `name` den gesamten Speicherbedarf zur Laufzeit mit dem Befehl `size` erfahren, hier ist der gesamte anzufordernde Speicher beruücksichtigt, die Größe unter `size` kann deutlich von der unter `ls -l` verschieden sein:

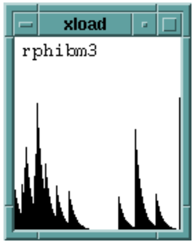
```
c4049@rphibm1:for/finidiff > ls -l a.out
-rwxr-xr-x   1 c4049    imo        21453 Apr  7 12:17 a.out
c4049@rphibm1:for/finidiff > size a.out
a.out: 20 + 13132 + 180 + 124 + 35088 + 388 + 801 = 49733
c4049@rphibm1:for/finidiff >
```

`size` gibt mehrspaltige Infomationen aus, allerdings varriert die Liste von Unix zu Unix, man sollte sich die Aufgliederung der Aufgelisteten Zahlen mit `man size` erklären lassen.

`size` kann allerdings nur solange richtig funktionieren, solange nicht zur Laufzeit Speicher alliziiert wird (Vorsicht C–Programmierer)

Grundsätzlich gibt es 2 Möglichkeiten, den Object–Code von Programmbibliotheken an das Programm anzulinken: Entweder dynamisch, dann wird beim Compilieren nur der Quelltext kompiliert und auf Platte abgespeichert, zur Laufzeit wird der restliche Object–Code angelinkt und das Programm läuft los. Dies ist von Vorteil, wenn viele große Programmbibliotheken angelinkt werden müssen, da die Anforderungen an den Plattenzugriff dann erheblich sind. Statisch gelinkte Programme haben den Vorteil, daß die Executables schon lauffähig sind und durch binären Filetransfer auch auf anderen Maschinen desselben Typs lauffähig sind.

1.5.7 Ausnutzung von Rechnern



Ein grundlegendes Problem ist die Ausnutzung von Rechnern bei Workstation–Clustern, da hier nicht zentral wie auf Supercomputern und Großrechnern der ordnungsgemäße Betrieb überwacht werden kann.
Zwei Faktoren spielen hierbei eine Rolle: Die Geschwindigkeit/ Auslastung der CPU und, oft wichtiger, die Größe des zur Verfügung stehenden Speichers.
Das auf X11–Oberflächen so beliebte `xload`, das die Auslastung der CPU zeigt, ist sehr irreführend.

1. Auf Sun führt die 100% –ige Auslastung der Maschine zu einem Strich, auf IBM–RISC Rechnern wird bei 100% –iger Auslastung gerade noch nichts angezeigt, nur wenn sich mehrere Anwendungen eine CPU teilen müssen, ist überhaupt etwas zu sehen.

2. `xload` zeigt nur an, wie die CPU ausgelastet ist. Werden gleichzeitig mehrere Jobs abgearbeitet und müssen dazu ständig Daten von der Platte geladen werden, weil der Speicher nicht reicht, so kann die Load zurückgehen, obwohl die Maschine (nämlich der Speicher) hoffnungslos überlastet ist, nur befinden sich nie genügend Daten im Speicher.

3. Mehrere Striche bedeuten nur, daß sich mehrere Jobs dieselbe CPU teilen müssen, die Auslastung ist keineswegs effizienter, im Gegenteil, der Verwaltungsaufwand für mehr als 3 Jobs führt unter Unix oft zu deutlichen Performanceeinbrüchen, außerdem Steigt der Schaden bei Systemabstürzen.

Es gibt, abhängig von der benutzten Shell, unter Unix die Familie der `ps`–Kommandos.

1. Mit geeigneten Optionen (mit `man ps` erfahrbar) kann nicht nur die momentane CPU–Auslastung, sondern auch der aktuelle Speicherbedarf für die einzelnen Job's erfragt werden.

2. Nähert sich der gesammte Speicherbedarf den 100 %, so sollte man sich hüten, auf so einer Maschine noch Hintergrundjobs abzusetzen, da jeder neu eingeloggte User die Maschine zum swappen bringt.

3. Wichtig ist auch, daß sich der Speicherbedarf einer Anwendung währedn der Laufzeit ändern kann, `ps` zeigt aber nur die momentane Speicheranforderung an, die beim Loslaufen der Programme noch gering sein kann, später aber wächst. Man sollte schon die maximale Speicheranforderung der eigenen Programme kennen.

Aus Suns kann der maximale momentane CPU–Verbrauch auch mit der Funktion `top` ausgegeben werden. Angegeben werden unter anderem Prozess–Id, User–Name, Nice-Level, CPU–Verbrauch und Größe des benutzten Speichers für den Prozess.
Die Ausgabe bleibt am Bildschirm stehen und wird ständig upgedated. Mit Eingabe von `q` kann der laufende Bildschirm verlassen werden. Der Bildschirm sieht dann etwa so aus:

```
last pid: 11448;  load averages:  1.06,  1.00,  1.00                  15:13:18
39 processes:  37 sleeping, 2 running
```

```
Cpu states:     % user,     % nice,     % system,     % idle,     % spin
Memory: 26456K available, 24416K in use, 2040K free, 1848K locked

  PID USERNAME PRI NICE   SIZE   RES STATE   TIME   WCPU    CPU COMMAND
11418 c4133    102   19 10844K 10992K run     3:11 91.92% 91.80% relax
11448 c4049     29    0   500K  616K run     0:00  7.69%  0.39% top
11421 c4049     15    0   428K  632K sleep   0:00  0.00%  0.00% tcsh
  119 root      15    0    12K    8K sleep  39:52  0.00%  0.00% update
11380 c4055     15    0   380K    0K slee2  0.00%  0.00% update
11380 c4055     15    0   380K    0K sleep   0:00  0.00%  0.00% tcsh
    1 root       5    0   108K   24K sleep   0:05  0.00%  0.00% init
21475 c4133      5   19   116K    0K sleep   0:00  0.00%  0.00% sh
11188 c4383      3    0   456K    0K sleep   0:00  0.00%  0.00% tcsh
11385 c4227      3    06K    0K sleep   0:00  0.00%  0.00% tcsh
11385 c4227      3    0   452K    0K sleep   0:00  0.00%  0.00% tcsh
11041 root       3    0    64K    0K sleep   0:00  0.00%  0.00% getty
11383 c4055      1    0   264K 1108K sleep   0:00  0.00%  0.00% xload
10733 c4133      1    0   264K  892K sleep   0:00  0.00%  0.00% xload
c4133      1    0   264K  892K sleep   0:00  0.00%  0.00% xload
  110 root       1    0   468K  336K sleep  15:40  0.00%  0.00% automount
11420 root       1    0    32K  268K sleep   0:00  0.00%  0.00% in.rlogind
 9823 root       1    0    64K  240K sleep   0:01  0.00%  0.00% rpc.rstatd
```

Eine ähnliche Funktion erfüllt auf IBM–RISC's der Befehl `monitor -top`, der Anzeigen der folgenden Form liefert:

```
Cpu states:      94.3% user,  5.7% system,  0.0% wait,  0.0% idle
Real memory:     63.5M used,   0.5M free,  64.0M total
Virtual memory:  62.8M used,  97.2M free, 160.0M total

   PID USER     PRI NICE   SIZE     RES STAT      TIME   CPU% COMMAND
 22368 c4321    124  14   4643K   4712K run       1:16  49.6% lambdaslave
 19849 c4283    124  20   9553K   6700K run     384:26  43.7% gttp8neu
 12187 root      63   0    930K    392K sleep   694:26   4.0% glbd
 24425 c4049     60   0    300K    332K run       0:00   0.9% monitor
 25972 c4050     60   0    320K    300K sleep     3:05   0.7% monitor
 16616 c4050     60   0   3707K   1912K sleep     3:59   0.4% X
 13666 root      60   0    293K    328K sleep     0:00   0.2% rlogind
   771 root      39  21     16K     16K sleep    15:01   0.2% netw (kproc)
  5007 root      60   0    192K    200K sleep     0:27   0.1% cron
 24690 c4050     60   0    473K    212K sleep     0:19   0.1% xterm
  8832 root      60   0    194K     16K sleep    13:17   0.1% nfsd
```

Nur das Virtual Memory ist wirklich benutzbar, ohne daß die Performance der Maschine zusammenbricht. Auch wenn man als einzelner User eine Maschine für sich alleine hat, sollte man dennoch nicht mehr als 1 Job im Hintergrund laufen lassen: Angenommen, die mittlere Zeit für einen Systemcrash beträgt 3 Tage. Wenn 1 Job der Länge 12 Stunden submittiert

wird, beträgt die Verlustwahrscheinlichkeit 1/6, verloren sind dann im Mittel 6 Stunden CPU–Zeit. Bei 3 gleichzeitig laufenden Jobs der Länge 12 Stunden ist die Wahrscheinlichkeit 1/2, da sich die Abarbeitung über 1.5 Tage hinzieht. Darüberhinaus gehen dann im Mittel nicht 6, sonden 18 CPU–Stunden verloren, also Vorsicht: Mit steigender Anzahl von Hintergrundjobs wächst nicht nur das Verlustrisiko, sondern auch der Verlust an CPU–Zeit. Deswegen ist auch das Submittieren von Jobketten mit kurzen Jobs dem Submittieren von Riesenjobs vorzuziehen, da bei Abstürzen weniger Rechenzeit in den Sand gesetzt wird.

Kapitel 2

Hardware

2.1 Größenklassen

Man kann Computer nach fallender Leistung einteilen in

- Supercomputer
- Mainframes / Großrechner
- Workstations / Minicomputer
- Mikrocomputer / PC's

Die Klassifikation ist fließend und durch den technologischen Fortschritt wechseln beständig Maschinen von den oberen Kategorien in die unteren: Viele Features heutiger PC's, wie Cache, Pipelining und Coprozessoren waren einmal typisch für Rechner der höchsten Leistungsklassen und sind heute Standard. Mit IBM/Apples Power-PC verschwimmt bei der Hardware die Grenze zwischen Workstations und Mikrocomputern, und auf der Softwareseite ist es unter dem PC-UNIX Linux durchaus möglich, den PC zu einer Workstation aufzurüsten, siehe z.B. im Graphikbereich Motif für Linux, mc 4/94.

Die Kategorien Mikrocomputer / Workstation / Großrechner / Supercomputer waren einmal fast synomym mit den „klassischen“ Herstellernamen die Firmen Apple / Vax / IBM / Cray. Bei Mikrocomputern redet man landläufig von PC's, obwohl das mal ein bestimmtes Modell war. Genauso spricht man häufig von einer „Cray“, wenn man irgendeinen Vektor-Supercomputer meint, auch wenn es sich um eine Fujitsu, NEC, Cyber oder Convex handelt. Der Markt ist wesentlich unübersichtlicher geworden.

Das untere Spektrum der Maschinen ist mit Silizium–Chips in CMOS–Logik ausgeführt, die schnellen Maschinen benutzen BiMOS und die schnellsten ECL-Logik, die mit jeweils höherer Frequnz getaktet werden können. Bei CMOS– und BiMOS–Chips werden elektrische Felder aufgebaut, was einige Zeit kostet, sie schalten mit Spannung. Bei ECL–Chips wird ein Stromsignal moduliert, das geht schneller, kostet aber mehr Strom. Auch wenn die Leistungsaufnahme von hochgetakteten CMOS-Chips wie der DEC 2166 mit 20 Watt beträchtlich zu sein scheint, die 8 Prozessoren der Cray Y/MP des Höchstleistungsrechenzentrums Jülich verbrauchen 1.6 Megawatt!

2.1.1 Mikrocomputer

Was ein Mikro ist, weiß jeder Leser. Die Spezifikationen sollen hier auch vor allem unter dem Gesichtspunkt aufgezählt werden, um sie mit den anderen Rechnerkategorien zu kontrastieren. Heutzutage hat eine solche Maschine einige hundert KB bis mehrere MB Memory, ein bis mehrere Floppy-Laufwerke und einige Dutzend bis hundert MB Plattenplatz, vielleicht auch etwas mehr.
Der Preis bewegt sich zwischen tausend und zehntausend Mark, je nachdem, was die Maschine an Extras enthält. Mikros haben einen meist graphikfähigen Bildschirm, vor dem der Benutzer auch sitzen muß, wenn er die Maschine bedienen will. Läuft auf einem Mikro einmal eine Anwendung (ein „Task“), dann läuft sonst darauf fast nichts mehr, woran Prozessor und Betriebbsystem etwa zu gleichen Teilen schuld sind.
Microsoft Windows ist das typische Betriebssystem: Graphisch einigermaßen aufwendig, aber die Multi-Tasking Fähigkeiten der Maschine sind stark eingeschränkt.
Der typische Prozessor sind die der Intel 386/486 Familie: Um die Modelle kompatibel zu machen, schleift man technologischen Balast mit (Speicherorganisation, Betriebssystem oder veraltete Hardware, siehe auch Editorial der mc 3/94 zum Thema Busse und Power PC), über den die einstmals stolzen Erfinder heute auch nicht mehr so glücklich sind.
Die Mikros sind in der Regel standalone-Maschinen, d.h. sie sind nicht mit anderen Rechnern vernetzt, und die Software wird entweder von der lokalen Platte oder dem Diskettenlaufwerk geholt. PC-Netzwerke mit der entsprechnden Software (z.B.Novell-Net) erlauben es, Daten für viele Maschninen auf einer einzigen Platte (der des „Servers“) unterzubringen. Kommunikation ist auf solchen Systemen nur sehr eingeschränkt möglich, etwa das Versenden von Nachrichten im Stil „Mittagessen?“.
Die Systemsicherheit ist einfach geregelt: Wenn man den Rechner verläßt, sperrt man das Zimmer zu, und wenn man einen Virus auf der Platte hat, hat man Backup–Kopien auf Diskette oder eben Pech.

2.1.2 Workstations

hießen früher einmal Mini–Computer. In der guten alten Zeit des kalten Krieges konnte man dann des öfteren lesen, daß irgendwelche Zollämter Vax-Computer, damals noch die Workstations schlechthin, beschlagnahmt hatten, die in den Osten geschmuggelt werden sollten, weil die bösen Russen damit ihre Atomraketen berechnen wollten.
Workstations sind in Verwaltungs– und Forschungseinrichtungen in der Regel vernetzt (im Gegensatz zu den Stand-alone-PC's). Oft werden sie auch gar nicht direkt von der Konsole bedient, sondern von einem bis mehreren Terminals aus, die ganz woanders stehen als der Rechner selber. Auf solchen Workstation-Clustern ist es auch möglich, Terminal-Sitzungen statt auf der eigenen auf einer völlig anderen Maschine aufzubauen, wenn z.B. der eigene Rechner überlastet ist.
Komfortable Bildschirme und Softwareentwicklungsumgebungen machen die Workstations beim Programmieren konkurrenzlos.
Typische Anwendungen sind aufgrund der graphischen Leistungsfähigkeit der Bildschirme und Prozessoren Computer Aided Design (CAD, Statische Konstruktionen auf dem Computer). Multimedia–Graphik ist ebenfalls eine Stärke dieser Maschinen, deswegen hüpften auch Steven Spielbergs Parksaurierzuerst zuerst über den Bildschirm von Silicon-Graphics Hochleistungsworkstations.

Hohe Rechenleistung ermöglicht es auch, viele naturwissenschaftliche Anwendungen statt auf Großrechnern und Supercomputern auf Workstations durchzuführen.
Der Preisverfall, der langsam auch auf dem Workstation-Markt einsetzt, sorgt hier dafür, daß oft schon neue Workstations statt Terminals angeschafft werden.
Wenn zuviele Leute auf einer Workstation arbeiten, bekommt das Betriebssytem in der Regel Probleme, die Antwortzeiten vom Rechner werden recht lang.
Der Hauptspeicher liegt in der Größenordnung von 10 bis 50 Megabyte, aber die modernsten Rechner dieser Art lassen auch Hauptspeicher in der Größenordnung von Gigabyte zu.
Im Gegensatz zum PC unterstützen die Betriebssysteme auf Workstations auch die Nutzung von virtuellem Speicher, d.h. was nicht in den Hauptspeicher paßt, wird auf Platte ausgelagert (Swappen, Pagen).
Klassische Workstations-Betriebssysteme sind Vax/VMS und Unix. In der Regel ist auch die Verwendung von graphischen Benutzeroberflächen, Windows halt, möglich. Dabei handelt es sich aber nicht um das Microsoft-Produkt, sondern um eine Entwicklung des Massachusetts Institue of Technologie (sog. MIT-Windwows) oder eine seiner vielen kommerziellen Ableitungen.
Unix unter MIT-X11 Windows ist übrigens auch eine Domäne von Public-Domain-Programmen: Es gibt säckeweise Editoren und Graphiksoftware, auch Compiler, die man nur vom nächsten PD-Server über das Internet kopieren braucht (Workstations sind in der Regel angeschlossen an irgendein internationales Kommunikationsnetz).
Prozessoren entwerfen die Workstation-Hersteller mittlerweile selber, gefertigt werden sie in eigenen Anlagen (z.B. IBM) oder in Lizenz (für Sun stellt unter anderem Texas Instruments die Prozessoren her). Der Workstation-Markt ist in den letzten Jahren explodiert, und die RISC-Technologie begünstigt Neuentwicklungen (dazu mehr im Abschnitt CISC, RISC und Compiler).
Die Datensicherheit auf Workstations ist in der Regel deutlich besser als auf PC's: Zugang hat sowieso nur, wer einen „Account" (einen Eintrag in die Benuzterliste und ein eigenes Directory, unter Unix im NFS, dem Network FileSystem) besitzt und ein Passwort zu Maschine hat. Unter Unix ermöglicht es ein NIS-Passwort, sich auf jeder Maschine eines Clusters einzuloggen. NIS, das Network Information System, ist auf Unix-Clustern die zentrale Datenbank für die Verwaltung des Clusters. Wenn der Systemverwalter keinen Mist macht, kommt man nur mit Hacker-Fertigkeiten an die Dateien anderer Benutzer. Workstation-Cluster können zentral gebackupt werden.

2.1.3 Mainframes

oder Großrechner sind heutzutage die Dinosaurier unter den Computern: Groß, schwer und es gab schon mal mehr davon.
Geld ist mit Mainframes heutzutage nicht mehr zu verdienen: Mitsubishi gab vor einiger Zeit die Großrechnerfertigung auf, die Abteilungen der anderen Hersteller schreiben rote Zahlen. Früher schafften die Unternehmen der freien Wirtschaft eine neue Mainframe an, wenn die alte steuerlich abgeschrieben war, aber heute kauft niemand mehr einen neuen Computer, bloß weil vom alten die Platte voll ist.
Viele Anwendungen wurden bereits auf kleinere Systeme wie PC- oder Workstation-Netze migriert, die im Unterhalt nicht so kostspielig sind (auch Mainframes können hunderte von Kilowatt fressen). Nicht zuletzt dieses Downsizing ist ein Grund, warum IBM finanziell ins Trudeln geriet, als das einstige Standbein Nr. 1 wegbrach.

Großrechner waren von Anfang an als Mehrbenutzersysteme konzipiert, und sowohl die Hardware als auch die Betriebssysteme sind darauf speziell ausgelegt. Auf diesem Gebiet sind die Mainframes selbst schnellen Workstations heute noch überlegen: Auf dem zentralen Großrechner des Forschungszentrums Jülich (einer 4-Prozessor IBM ES 9000) sind schon mal 500 Leute eingeloggt, ohne daß das System in die Knie geht, während man auch bei Hochleistungsworkstations of schon bei 4 bis 5 eingeloggten Usern deutliche Performance-Einbrüche erlebt.
Großrechner sind auf die Verarbeitung großer Datenmengen abgestimmt: Ihre Input/Output-Fähigkeiten werden nur noch von Supercomputern übertroffen, ein Grund, warum sie in Verwaltungszentralen noch immer ihren Platz haben, wobei nicht nur die Geschwindigkeit, auch die hohe Datensicherheit eine Rolle spielt.
Großrechner werden in der Regel von speziellen Terminals aus bedient (nur Text oder auch mal graphikfähig), in den letzten Jahren häuft sich der Einsatz von Mikrocomputern und Workstations mit Kopplungsprogrammen als Terminal, weil 1. die Terminals aufgrund der niedrigen Stückzahlen so billig auch nicht sind und 2. die Kiste dann auch ohne den zentralen Rechner was tun kann.
Datensicherheit wird bei Großrechnern großgeschrieben: Systeme werden jede Nacht gebackupt, und das Herumhacken in den Dateisystemen sieht in Hollywoodfilmen auch etwas leichter aus, als es in Wirklichkeit ist: Das Betriebssytem VM/CMS von IBM (nicht zu verwechseln mit VMS von Digital Equipement) kennt nicht einmal Directories im eigentlichen Sinn, und wer dort seine Daten unter dem falschen „Filemodus" ablegt, kann seine eigenen Daten nicht mehr selber mit ftp unter TCP/IP von einer anderen Maschine übertragen.

2.1.4 Supercomputer

sind in jeder Beziehung die Exoten unter den Computern. Während die vom Raumvolumen etwa ebenbürtigen Mainframes und Workstations businessmäßig–weiß–solide rumstehen, dürfen sich bei den Supercomputern in der Regel die Designer austoben.
Cray gruppierte seine X-MP Prozessoren mit der hohen Kühlsäule (Wasserkühlung !) in der Mitte, so daß eine Rundum-Sitzbank mit Rückenlehne daraus wurde — Omas Kachelofen lässt grüssen. Die CMOS-Cray Y-MP EL sieht aus wie ein Schlafzimmerschrank von Ikea.
Auch Multiprozessorsysteme haben ihre optischen Reize: Intels Paragon (bis zu 1000 Prozessoren Intel i860) hat für jeden Prozessor und für jede Interprozessorverbindung eine Leuchtdiode an der Gehäusefront, die beim Betrieb der entsprechendne Einheit losfunzelt: Bei Volllast leuchtet der Hobel wie ein Christbaum.
Durch die esoterischen Prozessorkonfigurationen bedingt, ist die Bedienerfreundlichkeit der Supercomputer in der Regel Glückssache. Daß diese Maschinen an der Grenze des technisch Machbaren angesiedelt sind (ab und zu auch auf der falschen Seite), vergrößert das Hochgefühl beim Programmieren von so einem Sensibelchen nicht gerade.
Wenn man nicht priviligierter Anwender ist, also z.B. Systemverwalter, Wetterfrosch, Flugzeugdesigner oder Raketenbastler bei der NASA, so ist man in der Regel nicht auf der Maschine selber eingeloggt, sondern auf einem Vorrechner. Dann compiliert man Programme remote, d.h. der Vorrechner schickt das Programm zum Compilieren an den Supercomputer, (und der schickt ein Listing eventuell mit Fehlermeldungen zurück- Syntax error in Line 1, das ist so mühselig, wie es sich anhört), oder der Vorrechner übernimmt das compilieren selber (sogenanntes Cross-Compiling).

Bei Supercomputern sind 2 Betriebsarten üblich: Rechenzentrumsbetrieb und Laborbetrieb. Beim Rechenzentrumsbetrieb erhält ein Benutzer für einen festen Zeitraum eine bestimmte Menge CPU-Zeit, die er nutzen darf, Filesysteme werden regelmäßig gebackupt, es gibt Dokumentation und manchmal auch Einführungskurse für die Benutzer. Grundbedingung für eine solche Art des Betriebs ist die Zuverlässigkeit des Systems, die vor allem von den mittlerweile ausgereiften Vektorrechnern geboten wird: Das ist der Grund, warum es zu solchen Maschinen heute noch immer keine Alternative gibt, obwohl (auf dem Papier) schnellere und (im Unterhalt) billigere Parallelrechnersysteme existieren.

Das Gegenteil vom Rechenzentrumsbetrieb ist der Laborbetrieb, häufig bei neuen, noch nicht so ausgereiften Systemen: Dokumentation ist zwar vorhanden, aber nicht unbedingt zur aktuellen Softwareversion (manchmal zur nächsten, manchmal zur vorletzten). Neue Benutzer dürfen die Maschine so viel nutzen wie sie wollen, aber das ist in der Regel Nervensache: Auch wenn die Systemadministration optimal arbeitet, kann es zu unvorhergesehnen Ausfällen kommen, die die Maschine aber nicht immer mitteilt. Programm-Output wie: „Gewicht des Neutronensterns: 20 Gramm“ bei einem Programm, das auf einer anderen Maschine tadellos lief, kann ein Anzeichen dafür sein, daß das Betriebssystem noch ein bisserl ein Eigenleben führt oder der Compiler noch Macken hat.

Erst wenn die Versionsnummer von Compiler und Betriebssytem eine 3 vor dem Komma hat, kann man einigermaßen sicher arbeiten.

Supercomputer bestehen entweder aus sehr wenigen, sehr leistungsfähigen Prozessoren (Hitachis HITAC S3800 mit 2ns Zykluszeit, entspricht 500 MHz und ist momentan wohl Weltrekord) oder aus vielen „Standardprozessoren“, etwa die Cray TD3 auf vernetzten DEC-Alph-Prozessoren. Das Hauptanwendungsgebiet dieser Rechner sind naturwissenschaftlich/technische Probleme, und im englischen Computerjargon werden diese Maschinen dann auch als „Number Cruncher“, Zahlenfresser, bezeichnet.

Die Connection Machine 2 war ursprünglich auch für die Anwendung in der KI-Forschung (künstliche Intelligenz) vorgesehen (Lisp ist vorhanden), und so gab sich die Company schließlich den Namen Thinking Machines — Die Denkmaschinen. Der Physiknobelpreisträger Richard P. Feynman, der damals an der Entwicklung des Router's für die Vielprozessormaschine mitarbeitete, kommentierte das mit:„Prima. Jetzt brauche ich niemandem mehr zu erklären, daß ich mit einem Haufen Irrer zusammenarbeite, ich sag nur noch den Firmennamen“ [1].

Supercomputer werden eingesetzt, wo mit normalen Computern oder mit Papier und Bleistift nichts mehr zu holen ist: Außerhalb des Gültigkeitsbereichs analytischer Rechenverfahren, bei der Wettervorhersage und rechenzeitintensiven Problemen der Strömungsmechanik, Teilchen- und Festkörperphysik. Die sonstigen Ingenieurswissenschaften sind beim Einsatz von Supercomputern etwas zurückhaltender: Was hilft eine durch eine 30-Stunden Computersimulation auf einer Cray Y-MP erzielte Verbesserung, die eine Ersparnis von 50.000 DM bringt, wenn die Stunde CPU-Zeit wie im HLRZ Jülich mit 2.000 DM abgerechnet wird, das ist ein Minus von 10.000 DM.

[1] zitiert nach W.Daniel Hillis in Physics Today, Februar 1989, Vol42, 2, Special issue: Richard Feynman. Im englischen Original: „Thats good. Now I don't have to explain to people that I work with a bunch of loonies. I can just tell them the name of the company.“

2.2 CISC, RISC und Compiler

Je weniger Informationen vom Speicher in den Prozessor geladen werden müssen, desto schneller kann ein Programm ausgeführt werden. Die natürliche Folge war, daß Befehlssätze mit sehr mächtigen Befehlen entwickelt wurden. Ein einziger dieser Befehle enthielt viele einzelne Arbeitsschritte und beschäftigte die CPU für ein Weilchen, wenn man schon die Menge der Daten so nehmen mußte, wie sie eben kamen. In der CPU selber wurden diese „komplexen Instruktionen“ in elementarere Anweisungen zerlegt, denen die harwaremäßig verdrahteten Funktionen des Prozessors entsprachen. Dieses Konzept nennt man CISC – Complex Instruction Set Coding. Das Konzept behält erstmal seine Gültigkeit, wenn ein Cache, also ein schneller Speicher vorhanden ist: Hier hält das Laden der Instruktionen aus dem langsamen Speicher in den Befehlscache die Programmausführung auf.

Dieses Konzept birgt auch Probleme:
1. Ein Befehl muß erst dekodiert werden, und das kostet Zeit, denn solange kann die CPU nichts tun.
2. Die komplexen Anweisungen werden auf komplexe Hardware abgebildet, damit es sich lohnt, und die muß dann erstmal als Transistorschaltung auf dem Chip realisiert werden, mehr Instruktionen kosten Entwicklungsarbeit und bedeuten höhere Produktionskosten.

Als Alternative bot sich an, einfach die Masse der selten benötigten Befehle wegzulassen, und in der Hardware nur diejenigen zu implementieren, die auch häufig benutzt werden. Alle anderen Befehle werden dann vom Compiler aus diesen elementaren Befehlen aufgebaut, der stark reduzierte (RISC — Reduced Instruction Set) Befehlssatz erfordert keine Dekodierung, und das spart (Entwicklungs-)Zeit bei den Prozessoren und Platz. Die Reduzierung von interner Prozessorverwaltung bei gleichzeitiger effizienterer Nutzung von vorhandenen funktionalen Einheiten ermöglicht es, den Chip mit weniger Transistorschaltungen zu bauen, der Entwicklungsaufwand wird drastisch reduziert.

Den Compiler liefert jetzt nicht mehr eine Softwarefirma, sondern wer den Chip gebaut hat, weiß auch am besten, was der Hobel so drauf hat, RISC-Compiler werden speziell auf die Prozessorarchitektur zugeschnitten. Vorbei sind die Zeiten, als die schnellsten Programme in Assembler geschrieben waren: Der Grund war einfach, daß die Compilerbauer keine Lust hatten, für jeden neuen Compiler die speziellen Features eines schon wieder anderen Prozessors zu implementieren. Wer sich die Mühe machte, die Eigenheiten des Prozessors per Assembler auszunutzen, konnte so Programme doch noch beschleunigen. Wer auf einer superskalaren RISC in Assembler programmiert, riskiert in der Regel, daß das Programm langsamer wird. Wahrscheinlich zwingt man dem Chip eine Befehlsreihenfolge auf (bezüglich Registerausnutzung, gleichzeitige Nutzung funktionaler Einheiten etc.), bei der sich verschiedene Einheiten gegenseitig blockieren, wo der der Compiler vielleicht schon weiß, daß man's besser machen kann. Codeumstellungen reichen auf RISC's in der Regel aus, um bei Verwendung eines effizienten Compilers das Maximum der Prozessorleistung zu erhalten.

Für schnelle Programmabarbeitung sind gute Compiler so wichtig wie gute Hardware, und die Programmiersprache ist da oft Nebensache.
Compilerbauer haben auch den Ruf von Fortran77 als schneller Programmiersprache auf dem Gewissen: Da Fortran für naturwissenschaftlich–technische Anwendungen eingesetzt wird, und Mikros für derartige Aufgaben in den ersten Jahren zu langsam waren, war die Nachfrage nach Fortran-Compilern für solche Maschinen entsprechend gering, verkauft und entwickelt

wurden vor allem Pascal– und C–Compiler.
Beim Vergleich schnitt Fortran dann entsprechend schlecht ab, was aber in erster Linie ein marktwirtschaftlicher Effekt war. Genau umgekehrt verhält sich die Situtation im High-End Bereich: C ist für Supercomputeranwendungen noch wenig üblich, die Compiler sind jünger und noch nicht so ausgereift wie die Fortran-Compiler derselben Hardware–Plattform und der generierte Code ist dann auch langsamer. Pointer–Programmierung reißt bei Gleitkommarechnungen nichts raus, sondern verwirrt den Compiler eher bei effizientem Memory-Management: Manche Compiler möchten nämlich selber gern herumpointern, aber man muß ihnen eine Chance geben. Auf den japanischen Vektorrechnern, wird die Vektorisierung von C-Programmen völlig boykottiert, und das sind immerhin die schnellsten Monoprozessoren der Welt. Der Grund ist, daß die Vektorisierung voraussehbare Speicherzugriffe erfordert– bei indirekte Addressierung (Pointer) ist dies in der Regel nicht mehr möglich.

In–core und Out-of-Core
CISC–Prozessoren zerstören häufig den Inhalt der Speicherzellen, auf die sie zugreifen, RISC's lassen ihn unversehrt. Dadurch gibt es auf den beiden Rechnertypen verschiedene Programmierstrategien:

CISC/Out of Core:	RISC/In–Core:
`a1=5`	`a=5`
`a2=a1+c`	`a=a+c`
`a1=a2+d`	`a=a+d`

Analog geschieht die optimierte Programmierung für Felder, wobei bei ungeschickter Programmierung optimierende Compiler den Code zum Teil umsetzen könnnen. Vorsicht bei alten Programmen aus der Großrechnerära: Sie sind in der Regel nach Out–Of–Core Techniken geschrieben und für moderne RISCs nicht ideal. Out–Of-Core erfordert mehr Eingabe/Ausgabeoperationen in und aus dem Cache, wofür Mainframe–CISCs eher geeignet sind als Workstation–RISCs.

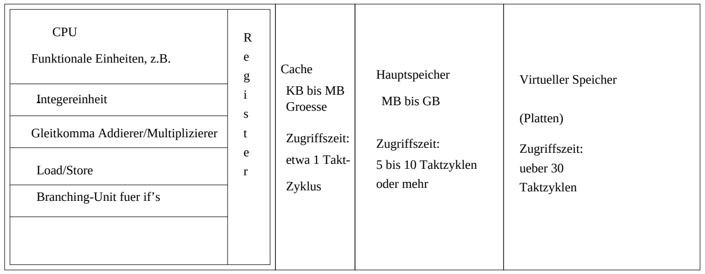
2.3 Skalare und Vektorrechner

Im folgenden sind die „typischen" Skalararchitekturen mit den Vektorarchitekturen auf dem Niveau der funktionalen Bestandteile kontrastiert. Aufgrund der verschiedenen Konzeption wurde bewußt auf eine Angleichung der Diagramme verzichtet: Bei skalaren Maschinen ist bei der Programmabarbeitung die Aufsetzzeit des Datentransfers in Taktzyklen nicht von Belang, wenn nur die entsprechenden Puffer groß genug sind. Vektormaschinen sind auf die Abarbeitung großer, kontinuierlicher Datenströme ausgelegt, darum wurde die Datenübertragungsrate pro Echtzeit angegeben, lange Aufsetzzeiten bei kleinen Datenmengen können diese aber deutlich verlangsamen.
Die angegebenen Werte sind Größenordnungen.

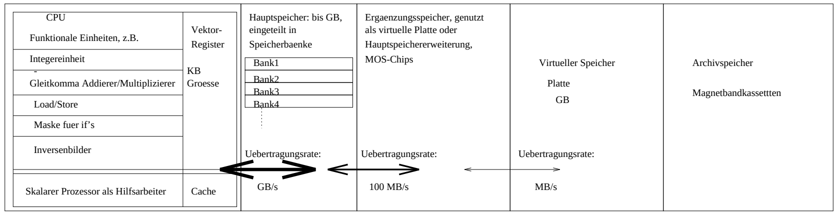
2.3.1 Skalare, Superskalare und Vektorrechner

1. Skalare Prozessoren
Hierarchischer Speicher: Schneller Zugriff der Register (Speicherzellen, auf denen die funktionalen Einheiten des Prozessors operieren) auf die Daten im Cache (1 Zyklus), langsamerer Zugriff auf die Daten im Hauptspeicher. Ungünstige Speicherzugriffe erfordern häufiges Nachladen und verlangsamen die Programmausführung. Dementsprechend gibt es je nach Cache–Größe Prozessoren, die bei ungeschickter Programmierung trotzdem wenig Performance–Einbußen

erwarten lassen: Intels i860 besitzt etwa 16 KB Daten– und Befehlscache, die IBM RS6000 Serie etwa 64 KB und SUN10 1 MB.



2. Vektorprozessoren
Vektorprozessoren haben keinen Cache: aus dem gesamten Speicher kann in der Regel innerhalb eines Taktzyklus ein Wert in einen Platz im Vektorregister geladen werden, sobald die Vektorpipe initialisiert ist. Vorausetzung ist, daß die angesprochenen Speicherzellen in aufeinenanderfolgenden Speicherbänken liegen und daß nach jedem Bankzugriff eine bestimmte Zeit kein Zugriff erfolgt. Wenn doch, muß der Prozessor warten, ein sogenannter „Bankkonflikt" tritt auf. Vektorprozessoren zeichnen sich gegenüber skalaren Prozessorn in der Regel durch breitere Datenpfade aus.



2.3.2 Shared memory

Beim verteiltem Speicher (**distributed memory**) ist jedem Prozessor ein Teil des Speichers zugeordnet, beim Konzept des gemeinsamen Speichers (**shared memory**) greifen alle Prozessoren auf denselben Speicher zu. Während das Konzept des distributed Memory bei jeder Vernetzungsmethode verwirklicht werden kann, ist shared Memory nur bei Bus–basierten Systemen und bei Cross–Bar anwendbar. Allerdings verfügt auch hier jeder Prozessor zumindest über einen eigenen Cache, Inter–Prozessor-Kommunikation erfolgt über den Hauptspeicher selber oder spezielle Kommunikationsregister.
Bei der klassischen shared–Memory Parallelisierung versucht das Betriebssystem, mehrere Programmteile auf verschiedenen Prozessoren gleichzeitig abzurabeiten. Einschränkung ist, daß die Programm–Ergebnisse gleich denen bei der Abarbeitung auf nur 1 Prozessor sein müssen. So können Unterprogramme, die auf verschiedene Daten zugreifen, auch ohne weiteres auf verschiedenen Prozessoren abgearbeitet werden.
Probleme entstehen, wenn Unterprogrammaufrufe nach dem Schema

```
call sub1(einfeld,ausfelda)
call sub1(einfeld,ausfeldb)
call sub1(einfeld,ausfeldc)
call sub1(einfeld,ausfeldd)
call sub1(einfeld,ausfelde)
```

ein Feld wie `einfeld` in ansonsten unabhängigen Unterprogrammaufrufen verwenden. Ist sich der Compiler nicht sicher, ob `einfeld` verändert wird, arbeitet er die Subprogramm–Calls vorsichtshalber nur nacheinander ab. Durch Compileroptionen ist es möglich, die unabhängige Abarbeitung der Calls zu erzwingen, man sollte aber auch wissen, was man tut.

Je nachdem, nach welchen Kriterien die parallele Abarbeitung erfolgt, spricht man auf Cray–Rechnern von Mikrotasking, Makrotasking oder Autotasking.

Die shared Memory Parallelisierung muß durch Compileroptionen vereinbart werde, linkt man allerdings Programmbliotheken an, die mit solchen Optionen kompiliert wurden, kann es sein, daß man diese Parallelität in diesen Programmteilen gar nicht wünscht. Dann muß beim Starten des Programms explizit die Verwendung von nur einem Prozessor vereinbart werden, etwa auf einer Multiprozessor–Cray durch setzen der Umgebungsvariable Anzahl der Prozessoren:

```
NCPUS=1
export NCPUS
```

Für Bus–basierte Systeme gilt das gleiche wie für Netzwerke: Bei metallischer Vernetzung und 250 MHz Datenübertrag und Prozessoren mit 40 MHz können von einem Bus theoretisch 6 Prozessoren versorgt werden, ohne daß die Performance darunter leidet.

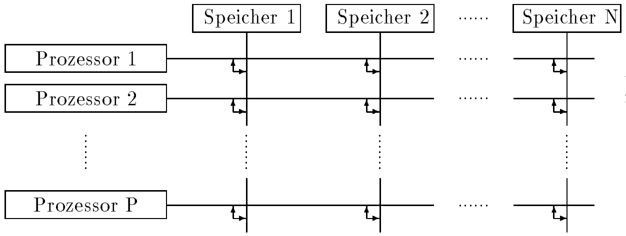


Abb.2.1 Cross–Bar für shared Memory, nach [Sto90]

Bus–basiert sind die Rechner von Cray, NEC, SGI (Zeus hier im RZ) und z.B. Alliant, von allen Parallelrechnern sehen sie den „normalen“ Coputern noch am ähnlichsten, nur daß der Bus, der zwischen Prozessor und Speichern hängt, nicht einen Prozessor versorgt, sondern mehrere.

Bei Parallel–Verarbeitung auf vielen Prozessoren entstehen durch den Bus Engpässe: Bei der parallelen Nutzung aller 16 Prozessoren einer CRAY Y–MP C90 sinkt für ein 1000×1000–Linpack Problem die Effizienz auf 75 Prozent [Don94].

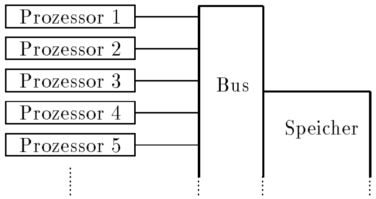


Abb.2.2 Schema des busbasierten Parallelrechners

2.4 Rechnen auf Hochleistungsarchitekturen

Aus Materialgründen wird die Taktrate und die Integrationsdichte der Mikroprozessoren nicht mehr beliebig weiter gesteigert werden können.
Der Schlüssel für weitere Leistungssteigerung von Computern liegt in der Ausnutzung paralleler Techniken:

- Unter **Kontrollparallelismus / interner Parallelismus** versteht man die fließbandartige Abarbeitung (Pipelining) von von Operationen auf einem Prozessor, und **funktionaler Parallismus** ist die unabhängige Abarbeitung von Ein/Ausgabe–Einheit, Fließkommaprozessor etc. . Die Auswirkungen derartiger Möglichkeiten auf die Performance sind in diesem Kapitel behandelt. Einflußnahme auf Kontrollparallelismus ist in der Regel nur bedingt möglich: Es kann vom Programmierer nur ausgenützt werden, was vom Hardwarearchitekten und vom Compilerprogrammierer vorgegeben ist.

- **Datenparallelismus** (im weiteren Sinn) ist dagegen die Methode, mit der durch Verändern der Algorithmen eine parallele Abarbeitung von Programmteilen durch verschiedene Prozessoren erreicht werden kann (nächstes Kapitel).

In diesem Abschnitt werden die Begriffe der skalaren, der superskalaren, der Vektor– und der Parallelarchitektur anhand von Diagrammen veranschaulicht. Für den Programmierer ist vor allem die Veranschaulichung der Unterschiede in der Abarbeitung von Algorithmen wichtig. Auf zu feine Strukturierung der Abläufe wurde verzichtet, weil darunter die Übersichtlichkeit gelitten hätte.

- S bedeutet Zwischenspeicherung (Store) eines Elements und L sein Wieder-Laden (Load).
- Es wird davon ausgegangen, daß keine Speicherkonflikte auftreten. Diagramme für Verarbeitung mit Speicherkonflikten siehe [Sto90], aber dann wird das Leben etwas kompliziert.
- Eine Zeiteinheit wird durch einen Kasten $\square$ dargestellt, Leer–Abstände sind nur zur Verbesserung der Lesbarkeit der Diagramme eingefügt. Ob hier wirklich vier RISC–Instruktionen ausgeführt werden oder ob vier Zeiteinheiten für einen CISC–Befehl verbraucht werden, ist nicht unterschieden.
- Es wurde davon ausgegangen, daß der „Compiler“ optimalen Code erzeugt, d.h. daß die funktionalen Einheiten des Prozessors optimal genutzt werden. In der Praxis erzeugen verschiedene Compiler und verschiedene Optimierungsstufen desselben Compilers deswegen verschieden schnellen Code.

- Aufgetragen wurde in x–Richtung der Zeitverlauf und in y–Richtung die Anzahl gleichzeitiger Operationen.
- Alle Operationen sollen gleichschnell ablaufen. In der Praxis dauern Gleitkommaoperationen „+,-,*“ manchmal länger als Speicherbefehle, Divisionen und die Berechnung transzendenter Funktionen sind beim Fehlen entsprechender Hardware deutlich langsamer.
- Es wird immer der Speedup gegenüber der skalaren Version mit Zwischenspeicherung angegeben. Wie man sich leicht überzeugt, ist der Speedup von der Rechenoperation abhängig, für einfache Skalierung von Vektoren (Unterprogramm, das $a \cdot \vec{x}$ ausführt) ist der Speedup geringer.

Die Programmzeile `a + b = c` wird dementsprechend als

a	b	+	c

dargestellt, als Abkürzung für:

1. Hole `a` aus dem Speicher,
2. hole `b` aus dem Speicher,
3. addiere beides,
4. speichere das Ergebnis unter c ab.

Es wird angenommen, daß zur Berechnung von Feldindizes keine Rechenzeit notwendig ist (stimmt in der Realität nicht, irgendeine Integer–Einheit muß den Index liefern).

Als Beispiel werden 2 aufeinanderfolgende sogenannte SAXPY–Routinen aus dem BLAS–Standard betrachtet, wie sie etwa als Kernel in einer Matrizen–Multiplikation erscheint. Dabei wird zu einem Vektor $\mathbf{y}$ der Vektor $\mathbf{x}$ mal dem Skalar a addiert und dem Vektor $\mathbf{z}$ zugewiesen,

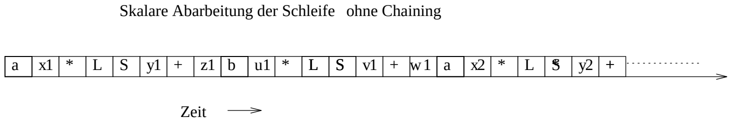
$$\mathbf{z} = a \cdot \mathbf{x} + \mathbf{y} \text{ und } \mathbf{u} = b \cdot \mathbf{v} + \mathbf{w}.$$

Das Programm dazu lautet in MATLAB-Syntax

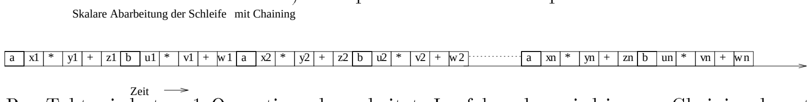
```
% a_Malx_Plus_y
for i=1,n
  z(i)=a*x(i)+y(i)
  u(i)=b*v(i)+w(i)
end
```

2.4.1 Skalarer Prozessor

Alle Operationen müssen hintereinander ausgeführt werden. Es wird angenommen, daß Compiler und Prozessor zwei Arithmetik–Operationen in einer Programmzeile zu zwei Befehlsschritten zerlegen müssen. Entsprechend wird für die Zwischenspeicherung und für das wieder–Laden zusätzlich eine Zeiteinheit benötigt.



Die Zwischenspeicherung ist unnötig. Stattdessen findet bei modernen Rechnern **Chaining** statt: Alle Operationen werden hintereinander ausgeführt, der Output einer Operation kann (ohne Zwischenspeicherung, d.h. ablegen in den Cache oder ein Zwischenregister und anschließendes wieder hervorholen) als Input für die nächste Operation verwendet werden.



Pro Takt wird etwa 1 Operation abgearbeitet. Im folgenden wird immer Chaining benutzt, der Speedup wird gegenüber diesem gechainten Programmablauf berechnet.
Beispiele für skalare Konzepte sind die Intel 80XX–Serie und die Motorola 680XX–Serie, wobei die höheren Modelle auch Konzepte des Pipelining und der Superskalarität benutzen.

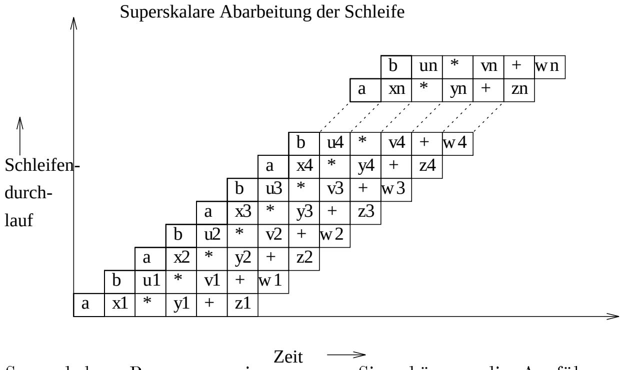
2.4.2 Pipelining

Die Ausführung der Programmzeilen wird nacheinander initialisiert, aber es können verschiedene Operationen gleichzeitig ausgeführt werden, Chaining wird benutzt.

Man unterscheidet bei den Pipelining–Typen superskalare und Vektor–Architekturen, die im folgenden erläutert werden.

2.4.3 Superskalare Prozessoren

Lade- und Rechenoperationen werden auf einem Prozessor von verschiedenen Einheiten ausgeführt. Dementsprechend wird bei modernen Prozessoren versucht, in jedem Takt eine beliebige Gleitkommavariable zu laden und gleichzeitig Arithmetikbefehle abzuarbeiten. Bei sogenannten LIW-Prozessoren (Long Instruction Word) wird die Befehlsfolge schon zur Compilezeit festgelegt, wie beim Intel i860-Prozessor.



Superskalare Prozessoren im engeren Sinn können die Ausführung der Befehle noch zur Laufzeit entscheiden (je nach Verfügbarkeit der Einheiten), der Compiler macht den Programmablauf nur mundgerechter.
Pro Takt werden jetzt 5 Operationen abgearbeitet (die Konstanten bleiben im Register, müssen nicht aus dem Cache geladen werden und zählen nicht als eigene Operation). Voraussetzung ist aber, daß der Rechner 3 Gleitkommazahlen pro Takt laden kann, bei Busbreiten von in

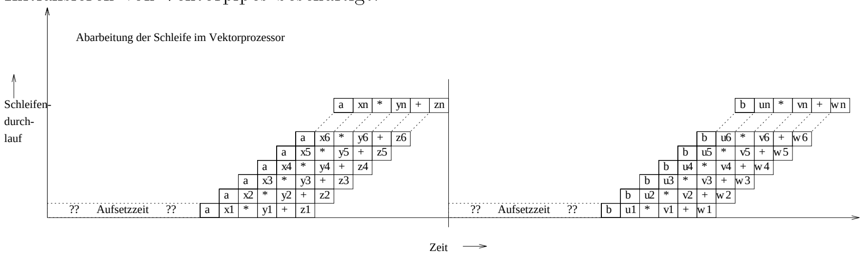
der Regel 64 BIT für eine RISC ist das in der Regel nicht möglich, die Treppe wird dann flacher und verschiebt sich nach links.
Superskalare Prozessoren sind die IBM–RS 6000–Serie, Super-Sparc von SUN, der Alpha-Chip von Digital Equipement und die Mips-Prozessoren von Silicon Graphics (fast monatlich kommen neue Modelle heraus).
In allen Fällen handelt es sich um RISC–Prozessoren, da durch den kleineren Befehlssatz die gleichzeitige Abarbeitung verschiedener Befehle (Store, Load und Arithmetik) übersichtlicher als auf CISC–Prozessoren implementiert werden kann.

2.4.4 Vektorisierung

Vektorisierung ist ein Pipelining mit jeweils einem ganzen Vektor. Der Prozessor ist in der Lage, pro Zeiteinheit jeweils ein Element der Vektoren $\mathbf{x}, \mathbf{y}, \mathbf{z}$ aus dem Speicher nachzuliefern. Am Anfang der Abarbeitung wird diese „Vektorqueue“ initialisiert, was größenordnungsmäßig 50 Taktzyklen erfordert.
Die Länge der Vektoren ist für solche Rechner hardwaremäßig vorgegeben, sie reicht von 64 bei der CRAY–Y/MP für einen „short vector loop“ über 256 bei der IBM ES 9000 bis über 2000 bei einigen japanischen Modellen (NEC). Zu beachten ist, daß der Faktor der Geschwindigkeitssteigerung nicht gleich der Länge der Vektorpipe ist.

Ausgabeoperationen oder Forward/Backward Dependencies[2] im innersten Loop machen Vektorisierung unmöglich. Auch if–Abfragen im Loop können die Vektorisierung beeinträchtigen. Ein „intelligenter“ Vektor–Compiler maskiert hier die if–Bedingungen mit einem Vektor von logischen Variablen und ist so in der Lage, die if–Bedingung vektorisiert abzuarbeiten.

Wird eine neue SAXPY–Schleife mit den Vektoren $\mathbf{u}, \mathbf{v}, \mathbf{w}$ und dem Skalar b abgearbeitet, so muß von neuem eine Vektorqueue initialisiert werden. Ist in der vorherigen Schleife nicht die gesamte Schleifenlänge ausgenutzt worden, entsteht der berüchtigte Vektor–Overhang, der die Performance drückt. Statt mit der Arithmetik ist der Rechner vorwiegend mit dem Initialisieren von Vektorpipes beschäftigt.



Es werden auch hier, wie im superskalaren Fall, 5 Operationen pro Takt abgearbeitet. Der Vektor–Overhang ist ein Ärgernis, wenn man Probleme hat, in denen zwar Arrays bearbeitet werden müssen, bei denen aber die Looplänge „verschwindend“ gering ist. In diesem Fall arbeiten die Superskalaren Prozessoren deutlich näher an der theoretischen Peak–Performance als ein Vektorrechner.

[2] In diesen Fällen benutzt ein Durchlauf „`i`“ des Loops Resultate eines vorhergehenden Duchlaufs „`i-1`“. Durch die gleichzeitige Abarbeitung kann aber auf dieses Ergebnis noch nicht zugegriffen werden.

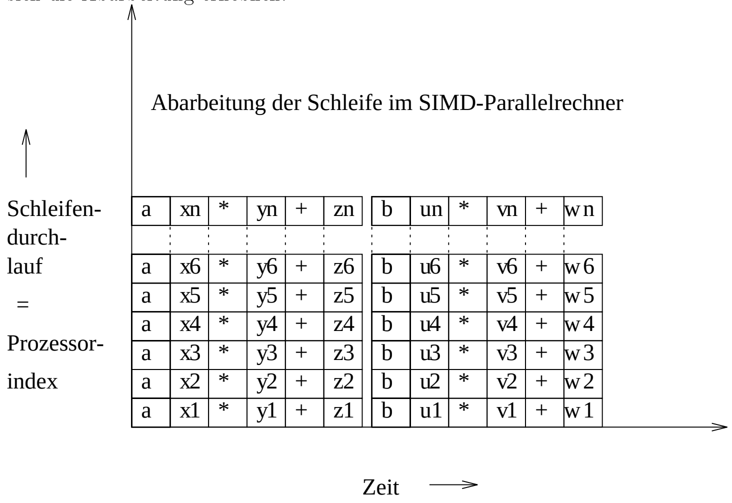
Auch der Vektorrechner muß 3 Zahlen pro Zyklys laden, allerdings ist hierfür die Hardware, die sogenannten Vektorpipes in der Regel vorhanden (bis zu 4 pro Prozessor).
Die NEC SX-3 kann mit bis zu 4 Prozessoren und zu je 4 Vektorpipes aufgerüstet werden, NEC SX-3/44 bedeutet 4 Prozessoren mit je 4 Vektorpipes, NEC SX-3/21 bedeutet 2 Prozessoren mit je 1 Vektorpipe usw.
Der Unterschied zwischen vektorieller und skalarer Abarbeitung einer Schleife ist also kurz gesagt der: Ein Programm

```
for i=1,n
  Zeile 1
  Zeile 2
end
```

wird auf einem Superskalaren Rechner abgearbeitet als `Zeile 1 Zeile 2` `Zeile 1 Zeile 2` `Zeile 1 Zeile 2` `Zeile 1 Zeile 2` `Zeile 1 Zeile 2` `Zeile 1 Zeile 2` . . . (n mal), auf dem Vektorrechner als `Zeile 1` `Zeile 1` `Zeile 1` `Zeile 1` . . . (n mal), dann `Zeile 2` `Zeile 2` `Zeile 2` `Zeile 2` (n mal).

2.4.5 SIMD–Datenparallelismus

Bei einem datenparallelen Rechner werden gleiche Operation gleichzeitig ausgeführt. Die folgende Graphik geht davon aus, daß alle Datenelemente sich bereits im Speicher ihrer zugehörigen Prozessoren befinden. Dies ist der Idealfall, ist Kommunikation nötig, verlangsamt sich die Abarbeitung erheblich.



Der Speedup ist hier gleich der Länge des Loops, bzw. der Anzahl der Prozessoren, wenn die Looplänge die Prozessorzahl übersteigt. Durch andere Variablenvereinbarung könnten beide SAXPY–Loops gleichzeitig abgearbeitet werden, wodurch sich der Speedup noch einmal verdoppeln würde.

Typische Beispiele sind die Hypercube–Architekturen der CM2 (bis zu 64K=64 mal 1024 Prozessoren) sowie die Maschinen von NCube und Parsytec.
Das Problem bei datenparallelen Rechnern ist, daß im Programm über Sprachelemente, die nicht den Standardsprachen entsprechen, die Datenauslegung beinflußt werden muß: Jeder Prozessor erhält ein Feldelement zugeteilt, wenn jeder Prozessor versorgt ist, wird wieder bei Prozessor 1 angefangen:$x(1), y(1), z(1)$ auf dem 1. Prozessor, $x(2), y(2), z(2)$ auf dem 2. und so weiter. Wählt man die falsche Datenauslegung, die häufiges Umkopieren oder ungüngstige Datenzugriffe erfordert, so erhält man erhebliche Einbußen bei der Performance. Das linke Programm erfordert Kommunikation zwischen dem i-ten mit dem $n-i$-ten Prozessor, das rechte nicht.

```
% Dumme_Summe                  % Summe
for i=1:n                      for i=1:n
  a(i)=b(i)+c(n-i)               a(i)=b(i)+c(i)
end                            end
```

2.4.6 Aussagen der Diagramme:

Die Diagramme geben einen groben Überblick, wie sich bei optimaler Ausnutzung der Architektur Unterprogramme beschleunigen lassen.
Anschaulich sieht man, daß der Speed–up dann verbessert wird, wenn viele gleiche Gleitkommaoperationen hintereinander folgen, die dann „gepiped“ werden können. Für Programmteile, in denen das nicht der Fall ist, verhält sich die Performance „skalar“.
Leider ist die Effizienz der einzelnen Architekturen für verschiedene Programmkerne sehr stark unterschiedlich. In den obigen Programmzeilen kommen auf jede Gleitkommaoperation eine Lade–operation (die Konstante a wird im Register der CPU gehalten und muß nicht neu geladen werden), und auf 2 Ladeoperationen kommt eine Speicheroperation, d.h. das Verhältnis von Rechenoperationen zu Speicherzugriffen ist 2/3. Ein Programm, das stattdessen ein Skalarprodukt berechnet, etwa

```
% Dot_Produkt
z=0
u=0
for i=1:n
  z=z+x(i)*y(i)
end
```

hat ein Verhältnis von Rechenoperationen zu Speicherzugriffen =1 für große n, denn z und u werden in Registern gehalten und erst am Ende der Summation abgespeichert. Vektorrechner haben keine Probleme bei häufigen Ladeoperationen, lieben es aber nicht, Vektoren auf Skalare zusammenzustauchen. Sie erreichen bei `a_mal_x_plus_y`-Programmen die höchsten Flopraten. RISC und PC-Prozessoren hassen häufige Speicherzugriffe, sie sind aufgrund der niedrigeren Datenübertragungsraten vom Cache zum Prozessor am schnellsten bei Dot-Produkt-Programmen. Das Problem ist, daß man nicht jeden Algorithmus so umstricken kann, daß er mal den einen, mal den anderen Programmkern enthält.

Kapitel 3

Praktische Numerik

3.1 Typische Probleme

Dieser Abschnitt ist zum Teil den Anfangskapiteln von *Computer Methods for Mathematical Computations* von Forsythe, Malcolm und Moler entnommen. Es handelt sich dabei um ein Buch, das leider nicht mehr im Druck ist und scheinbar auch nicht mehr aufgelegt wird (nur noch in Rußland).

3.1.1 Gleitkommazahlen

Mathematiker lieben die reellen Zahlen. Das Problem ist: in der EDV sind sie nicht realisierbar. Üblicherweise gibt es dort:

Ganze Zahlen: Wie in der „theoretischem“ Mathematik, nur daß es auf dem Computer ein maximales und ein minimales Element gibt. Die Integer sind in Fortran77 normalerweise `*4` (das heißt 4 Byte), der verwendbare Bereich geht von ungefähr -2^{31} bis -2^{31}, wo die Grenzen genau liegen, entnimmt man am besten dem Language Reference Manual, `Integer*1` und `Integer*2` sind kein Standard. In C gibt es die Integer als signed, unsigned, short und long, wobei long signed etwa den Fortran–Integern entspricht.

Festkommazahlen: Hier wird mit einem String von Integern gerechnet, die reelle Arithmetik wird dann auf einen String von Integer übertragen. Diese Art der Zahldarstellung ist beliebt bei Interpretern, wenn man mit hoher Stellenzahl rechnen will (Maple, Mathematika), sie ist aber nicht sehr schnell. In Matlab werden Festkommazahlen als 2 Integer abgespeichert, 1 Integer steht für die Matisse, eine für den Exponenten. Der Unterschied zwischen Festkommazahlen und Gleitkommazahlen besteht darin, daß der Prozessor direkt mit Gleitkommazahlen rechnet, während Festkommazahlen mit einer Kombination von Integerarithmetik für Matisse und Exponent behandelt werden müssen. Die Basis für Festkommazahlen ist 10, für Gleitkommazahlen ist die Basis eine Potenz von 2 (2 für moderne RISC's, 16 für die IBM 370 Serie).

Gleitkommazahlen: Die einzig wahren Zahlen, da nur mit diesen Zahlen schnelle „reelle“ Arithmetik realisierbar ist. Bis zu $8*10^9$ Gleitkommaoperationen (Flops=Floating Point Operations per second) können heute auf den schnellsten Monoprozessoren realisert werden. Charakterisiert ist ein Gleitkomma–Zahlensystem durch die Basis β, die Präzision

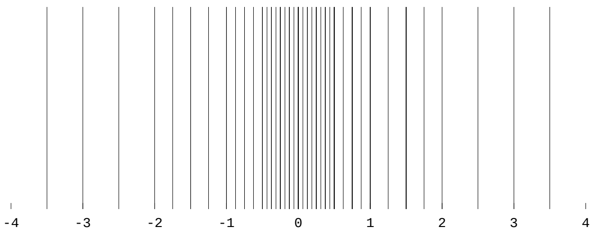
t und die untere und obere Grenze für den Exponenten e, $L \le e \le U$. Jede Gleitkommazahl ist darstellbar als

$$x = \pm \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \dots \frac{d_t}{\beta^t} \right) \cdot \beta^e$$

mit

$$0 \le d_i \le \beta - 1, \qquad (i = 1, \dots, t)$$

- Die obige Darstellung liefert keine äquidistanten Zahlen, wie man aus der Darstellung des „Zahlenstrahls“ für $\beta = 2$, $-1 \le e \le 2$, $t = 3$ entnehmen kann:



  Zahlen von sehr verschiedener Größenordnung sind in der Festkommadarstellung mit verschiedener Genauigkeit dargestellt, Rechenoperationen zwischen verschieden großen Zahlen sind immer mit dem jeweils gößeren Fehler behaftet (Fehlerfortpflanzung!)

- Die kleinste Zahl ϵ, mit der $\oplus$, die numerische Gleitkommaaddition mit dem Ergebnis

$$1 \oplus \epsilon > 1$$

  noch ausführbar ist, heißt **Maschinenepsilon**.

- **Underflow** tritt auf, wenn das Ergebnis einer Rechenoperation betragsmäßig kleiner ist als die kleinste darstellbare Gleitkommazahl. Einige Compiler „maskieren“ Underflow durch runden nach 0 selbständig, bei anderen muß die Maskierung erzwungen werden (IBM VS: `call xuflow` am Programmanfang).

- **Rundungsfehler** treten auf, wenn eine reelle Zahl im Gleitkommasystem nicht dargestellt werden kann und zur nächsten Gleitkommazahl gerundet werden muß, bei einem System zur Basis 2 etwa entsteht für 0.1 ein periodischer Bruch:

$$0.1 = \frac{1}{10} = \frac{0}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \dots$$

- Es gibt weder ein Assoziativgesetz noch ein Distributivgesetz für Addition und Multiplikation!

- Zieht man zwei fast gleich große Gleitkommazahlen voneinander ab, so sind die letzten Stellen zufällig. Dies kann numerische Verfahren empfindlich beeinträchtigen und heißt **Catastrophic Cancellation**. Auf der Sun lieferte das Programm `caca` das Ergebnis

```
program caca
real a,b
a=1.
b=1.000001
c=.999999
d=b-a
e=a-c
write(*,*) 'd=',d
write(*,*) 'e=',e
stop
end
```

```
d=    9.53674E-07
e=    1.01328E-06
```

  ab der 2. Stelle ist das Ergebnis also falsch, als Differenz kommt nicht, wie erwartet, exakt `1.E-0.6`.

  Für doppelt genaue Zahlen gilt singemäß dasselbe, bei Weiterverarbeitung der Ergebnisse ist also nur die 1. Dezimale zuverlässig.

3.1.2 Polynome

In den „Numerical Recipes“ ist folgendes Beispiel für eine ungeschickte Programmierung von Polynomen zu finden:

```
p=c(1)+c(2)*x+c(3)*x**2+c(4)*x**3+c(5)*x**4
```

Originalkommentar: „Come the (computer) revolution, all persons found guilty of such criminal behaviour will be summarily executed, and their programs won't be!“ Bei Polynom–Berechnungen (auch Polynomen von z.B. Matrizen) sollte man soweit wie möglich vom Assoziativgesetz Gebrauch machen:

```
p=c(1)+x*(c(2)+x*(c(3)+x*(c(4)+x*c(5))))
```

Insbesondere umgeht man hier auch die Berechnung der Potenz `**`, die unter umständen nicht als normale Multiplikation, sondern als Kombination von Exponentialfunktion und Logarithmus ausgeführt wird (für $a > 0$) $a^x = \exp(x \log a)$) und dementsprechend viel Rechenzeit kostet.
Für große n wird man die Klammerebenen mit einer Schleife organisieren, etwa

```
p=c(n)
do 10 j=n-1,1,-1
  p=p*X+C(j)
10 continue
```

3.1.3 Transzendente Funktionen

Vorsicht ist geboten, wenn man Funktionen, die als unendliche Taylorreihen definiert sind, mit dem Computer berechnen will. Zum einen hat man keine Garantie, daß die Polynomdarstellung in vernünftiger Zeit (Critical Slowing down) oder überhaupt (Maschinenepsilon) konvergiert, z.B. Kettenbrüche haben in der Regel bessere Konvergenzeigenschaften. Wenn man schon Taylorreihen benutzt, sollte man mit den betragsmäßig kleinsten Komponenten beginnen, da auf eine große Summe kleine Gleitkommazahlen nicht mehr aufaddiert werden können.

Ebenfalls ist Vorsicht geboten beim Evaluieren von Taylorreihen mit alternierendem Vorzeichen, hier kann Catasrophic Cancellation auftreten, kann aber auf einfache Art und Weise umgangen werden:

$$e^{-x} = \frac{1}{e^x}$$

Üblicherweise wird versucht, die Taylorreihe in einen ganzzahligen und einen gebrochenzahligen Anteil aufzuspalten:

$$e^x = e^{m+f} = e^m e^f$$

Die Konvergenz der Taylorreihe für Zahlen < 1 ist deutlich besser als für Zahlen > 1, und die Exponentialfunktion von ganzen Zahlen kann auch in einer Tabelle abgespeichert werden und braucht nicht jedesmal neu berechnet zu werden.

3.1.4 Instabile Algorithmen

Gesucht sei das Integral

$$E_n = \int_0^1 x^n e^{x-1} \mathrm{d}x, \qquad n = 1, 2, \ldots$$

Aus partielle Integration folgt

$$\int_0^1 x^n e^{x-1} \mathrm{d}x = x^n e^{x-1}\Big|_0^1 - \int_0^1 n x^{n-1} e^{x-1} \mathrm{d}x = 1 - n \int_0^1 x^{n-1} e^{x-1} \mathrm{d}x$$

oder

$$E_n = 1 - nE_{n-1}, \qquad n = 2, \ldots$$

mit $E_1 = 1/e$. Mit `REAL*4` erhält man auf einer SUN

$E_1 = 0.367879$	$E_8 = 1.00563E-01$	$E_{15} = 11965.0$
$E_2 = 0.264241$	$E_9 = 9.49326E-02$	$E_{16} = -191438.$
$E_3 = 0.207277$	$E_{10} = 5.06744E-02$	$E_{17} = 3.25445E+06$
$E_4 = 0.170893$	$E_{11} = 0.442581$	$E_{18} = -5.85801E+07$
$E_5 = 0.145534$	$E_{12} = -4.31097$	$E_{19} = 1.11302E+09$
$E_6 = 0.126796$	$E_{13} = 57.0427$	$E_{20} = -2.22605E+10$
$E_7 = 0.112430$	$E_{14} = -797.597$	$E_{21} = 4.67470E+11$

obwohl der Integrand im Intervall von 0 bis 1 immer Positiv ist. Der Grund liegt darin, daß der Rundungsfehler bei der Berechnung von $1/e$ bei jedem Iterationsschritt vergrößert wird, erst mit -2, dann mit -3, so daß der Rundungsfehler von anfangs etwa $4.14 \cdot 10{-}7$ in der 12. Iteration um den Faktor $(-2) \cdot (-3) \cdot \ldots (-12)$ vergrößer wurde.
Vermieden werden kann dieser Fehler durch Umformen der Iteration zum Schema

$$E_{n-1} = \frac{1 - E_n}{n}, \qquad n = \ldots 3, 2$$

Hierbei wird der Fehler bei jedem Rechenschritt verkleinert, allerdings muß E_n für ein großes n bekannt sein, dann allerdings ist der Algorithmus stabil. Für den Startwert E_n für große n gilt

$$E_n = \int_0^1 x^n e^{x-1} \mathrm{d}x \le \int_0^1 x^n \mathrm{d}x = \frac{x^{n+1}}{n+1}\Big|_0^1 = \frac{1}{n+1}$$

Wählt man als Startwert $E_{20} = 0$ im Vertrauen darauf, daß dieser Rundungsfehler im Laufe der Iteration immer weiter verkleinert wird, so erhält man

$E_{20} = 0.$	$E_{13} = 0.6694769859E-01$	$E_6 = 0.1268023551$
$E_{19} = 0.4761904851E-01$	$E_{12} = 0.7177325338E-01$	$E_5 = 0.1455329359$
$E_{18} = 0.5012531206E-01$	$E_{11} = 0.7735222578E-01$	$E_4 = 0.1708934158$
$E_{17} = 0.5277081579E-01$	$E_{10} = 0.8387707174E-01$	$E_3 = 0.2072766423$
$E_{16} = 0.5571936443E-01$	$E_9 = 0.9161229432E-01$	$E_2 = 0.2642411292$
$E_{15} = 0.5901753902E-01$	$E_8 = 0.1009319648$	$E_1 = 0.3678794503$
$E_{14} = 0.6273216754E-01$	$E_7 = 0.1123835072$	

wobei $1/e = 0.3678794503 = E_1$ bis auf die letzte Stelle übereinstimmen, der geratene Startwert war also gar nicht so schlecht.

3.2 Typische Diskretisierungsstrategien

Im allgemeinen kann man sagen: Symmetrische Formeln $2n$-ter Ordnung sind mindestens so genau wie asymmetrische Formeln $2n+1$–ter Ordnung.

3.2.1 Differentiation

Formeln für die 1. Ableitung:
Die Funktion auf diskreten Gitterplätzen ist f_i, f_{i+1}, die analytische Funktion an der entsprechenden Stelle sei $f(x)$ mit den Ableitungen $f'(x), f''(x) \ldots$.

Name:	Differenzenquotient:	Führender Fehlerterm
Vorwärts–Differenz	$(f_{i+1} - f_i)/\Delta x$	$\Delta x f''(x)/2$
Rückwärts–Differenz	$(f_i - f_{i-1})/\Delta x$	$-\Delta x f''(x)/2$
3–Punkt symmetrisch	$(f_{i+1} - f_{i-1})/(2\Delta x)$	$\Delta x^2 f'''(x)/6$
3–Punkt asymmetrisch	$(-1.5 f_i + 2 f_{i+1} - .5 f_{i+2})/\Delta x$	$-\Delta x^2 f'''(x)/3$
5–Punkt symmetrisch	$(f_{i-2} - 8 f_{i-1} + 8 f_{i+1} - f_{i+2})/(12\Delta x)$	$-\Delta x^4 f'''''(x)/3$

nach [Fle90], p.57, Tab. 3.3, dort gibt's auch noch Formeln für die 2.Ableitung.
Zumindest für die 2–Punkt Formeln kann man sich als Faustregel für die Programmierung merken, daß der höhere Index immer links steht (vergißt man oft im Eifer des Gefechts).
Obwohl die Differenzenformeln mit höherer Stützstellenzahl genauer sind, ist ihre Verwendung nicht immer empfehlenswert:

- höheren Ableitungen sind umständlicher zu programmieren,
- die Anzahl der Rechenschritte ist größer, was ein Nachteil ist, wenn die Genauigkeit des Algorithmus nicht in erster Linie von der Ortsdiskretisierung abhängt, sondern z.B. von der Implementierung der Randbedingungen oder genäherten Modellannahmen.
- Randbedingungen sind umständlicher und unübersichtlicher zu programmieren.

Die Effizienz verschiedener Diskretisierungsstrategien ist in [Fle90] 4.5 zu finden.

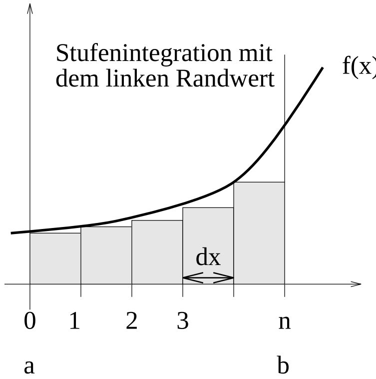
Wichtig: eine numerische Differentiation rauht Funktionen auf, bedingt durch die endliche Genauigkeit. Aus der Funktionentheorie weiß man, daß Funktionen schon dann nicht gleich

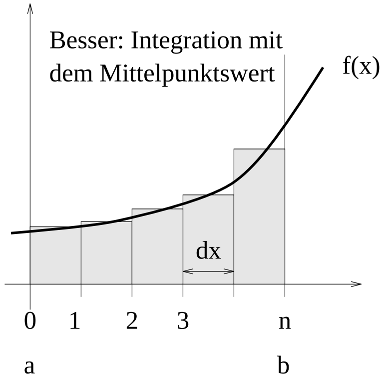
sind, wenn sie sich in einer ϵ–Umgebung um einen Punkt unterscheiden. Bei numerischer Differentiation hat man nicht einmal eine ϵ–Umgebung, sondern einen ziemlichen Brummer von Intervall, die numerische Differentiation von äquidistanten Punkten ist sehr unzuverlässig.
Manchmal hat man nur äquidistante Daten (z.B. Meßwerte aus Experimenten und Computersimulationen). Dann ist es oft besser, die Daten an eine Funktion zu fitten (Gauß–Kurve, Lorenz–Kurve) und den Fit zu differenzieren.
Interessieren die Ableitungen einer Funktion, deren analytische Form man vorliegen hat, sollte man die Funktion mit einem Computer–Algebra–Programm ableiten, wenn man selber dazu zu faul ist, analytische Ableitung ist immer wohldefiniert.

3.2.2 Integration

Im Gegensatz zur analytischen Ableitung ist die analytische Integration nicht immer möglich. Dafür sind die numerischen Integrationsalgorithmen aber auch beliebig stabil (zumindest in 1 Dimension, in höheren Dimensionen gibt's sowieso immer Probleme).

Stufenfunktionsmethode:
Der Kurvenverlauf wird durch eine Stufenfunktion approximiert. Die einzelnen Flächen werden aufsummiert und geben so eine Abschätzung des Integrals. Je kleiner das Intervall dx gewählt wird, d.h. je größer die Anzahl der Stützstellen, desto genauer wird das Ergebnis. Vorsicht, die Genauigkeit hängt davon ab, ob als Funktionswert der Funktionswert in der Intervallmitte oder am Rand benutzt wird!





Es ergibt sich folgende Formel für die Stufenintegration mit dem Mittelpunktswert:

$$x_k = a + (k + .5)\Delta x \qquad \text{mit} \qquad \Delta x = \frac{b-a}{n} \tag{3.1}$$

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{k=0}^{n-1} f(x_k + \Delta x/2)\Delta x \tag{3.2}$$

Die Realisierung im Programm würde folgendermaßen aussehen:

```
      DX  = (B-A)/N
      INT = 0
```



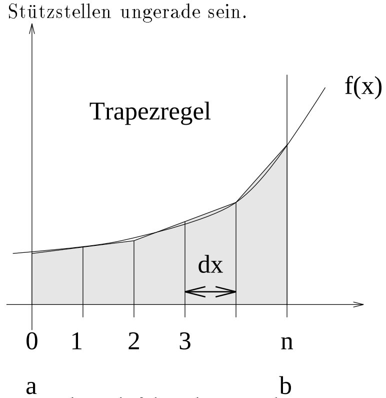
```
      DO 100 K=0,N-1
        XK = (K+.5)*DX + A
        INT = INT + F(XK)
 100  CONTINUE
      INTEGRAL = INT * DX
```

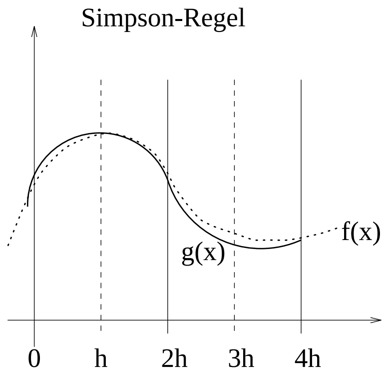
Trapez-Formel: Die Konvergenz der Integration ist sicherlich besser, wenn die Funktion statt durch eine Stufenfunktion durch einen Polygonzug approximiert wird. Man verbindet alle Funktionswerte an den Stützstellen durch Geraden.
Es ergibt sich folgende Formel:

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{k=1}^{n-1} f(x_k)\Delta x + \frac{1}{2}(f(x_0) + f(x_n))\Delta x$$

Es fällt auf, daß der Unterschied zur Stufenfunktion gering ist. Die Ausdrücke unterscheiden sich nur in einem Term.

Simpson-Algorithmus: Die Konvergenz läßt sich weiter verbessern, wenn man den Funktionsverlauf durch Parabeln approximiert. Eine Parabel ist durch drei Punkte festgelegt, man braucht also immer zwei Intervalle für eine Parabel. Aus diesem Grund muß die Anzahl der Stützstellen ungerade sein.





Es ergibt sich folgende Formel:

$$g(x) = ax^2 + bx + c \tag{3.3}$$

$$\int_0^{2h} g(x)\mathrm{d}x = \frac{h}{3}(f(0) + 4f(h) + f(2h)) \Rightarrow \tag{3.4}$$

$$\int_a^b f(x)\mathrm{d}x \approx \frac{\Delta x}{3}(f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4)\ldots f(b)) \tag{3.5}$$

```
      DX = (B-A)/(2*N)
      INT = 0
      DO 100 K=1,2*N-1
         XK = K*DX + A
```

```
         IF (MOD(K,2).EQ.1) THEN
            INT = INT + F(XK)*4
         ELSE
            INT = INT + F(XK)*2
         ENDIF
100   CONTINUE
      INT = INT + F(A) + F(B)
      INTEGRAL = INT * DX / 3
```

Newton–Coates Formeln:
Für die Integration mit äquidistanten Stützstellen lassen sich für die geschlossenen Formeln die Fehler angeben. Die Formel erhält man durch die Integration der Taylorreihe der Funktion, wobei bei den Formeln höherer Ordnung Taylorentwicklungen immer höherer Ordnung integriert werden:

Name	Intervallformel	Fehlerterm
Trapez–Regel	$\int_{x1}^{x2} f(x)dx = h[\frac{1}{2}f_1 + \frac{1}{2}f_2]$	$+\mathcal{O}(h^3 f'')$
Simpson	$\int_{x1}^{x3} f(x)dx = h[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3]$	$+\mathcal{O}(h^5 f^{(4)})$
Simpson 3/8	$\int_{x1}^{x4} f(x)dx = h[\frac{3}{8}f_1 + \frac{9}{8}f_2 + \frac{9}{8}f_3 + \frac{3}{8}f_4]$	$+\mathcal{O}(h^5 f^{(4)})$
Bode	$\int_{x1}^{x5} f(x)dx = h[\frac{14}{45}f_1 + \frac{64}{45}f_2 + \frac{24}{45}f_3 + \frac{64}{45}f_4 + \frac{14}{45}f_5]$	$+\mathcal{O}(h^7 f^{(6)})$

Die Formeln höherer Ordnung sind zwar aufwendiger zu berechnen, die Gesamtzahl der Stützstellen und Intervalle ändert sich für konstantes h nicht. Die Genauigkeit wächst mit immer höheren Potenzen von h.

Vorsicht bei der Programmierung der Ränder: Wenn man dort durch Schlamperei Fehler der Ordnung $\mathcal{O}(h^h)$ macht, helfen auch Integrationsalgorithmen höherer Ordnung nichts, der Fehler bleibt $\mathcal{O}(h^h)$.

Mit nicht–äquidistanten Stützstellen ergeben sich oft wesentlich schnellere Algorithmen. Der Gauß'sche Integrationsalgorithmus beruht darauf, daß eine hinreichend differenzierbare Funktion durch Glockenkurven interpoliert wird und die Glockenkurven dann integriert werden (Integral ist nur abhängig von der Höhe und der Breite der Kurve).

3.2.3 Die Trotter–Suzuki–Zerlegung

Häufig hat man es in der Physik mit Operatoren der Form $\exp(\theta(A+B))$ zu tun (A und B hermitesch, θ reell oder komplex), etwa im Boltzmannfaktor oder in den Greensfunktionen. Dabei kann man in der Regel A und B getrennt diagonalisieren, nicht aber $A+B$, wenn A und B nicht miteinander kommutieren.

Die Trotter–Suzuki–Transformation (im folgenden TS–Trafo abgekürzt), auch allgemeine Trotter–Formel oder Lie–Trotter–Zerlegung genannt, dient zum Aufspalten von solchen Exponentialoperatoren.

Im Gegensatz zum Baker–Hausdorff–Theorem[1] für $[[A,B],A] = [[A,B],B] = 0$,

$$\mathrm{e}^A \mathrm{e}^B = \mathrm{e}^{A+B+1/2[A,B]} \tag{3.7}$$

Dagegen gilt die Trotter–Suzuki–Zerlegung

$$\mathrm{e}^{A+B} = \lim_{n\to\infty} \left(\mathrm{e}^{A/n}\mathrm{e}^{B/n}\right)^n \tag{3.8}$$

für beliebige beschränkte Operatoren A, B.
Hierfür gibt es auch einige verbale Umschreibungen, etwa unter dem Namen Äquivalenztheorem: „Ein m–dimensionales quantenmechanisches System wird auf ein $m+1$–dimensionales klassisches System abgebildet.“ Klassisch ist das System insofern, als man es jetzt nur noch mit kommutierenden Operatoren $\frac{A}{n}$, $\frac{B}{n}$ für große n zu tun hat. Die $(m+1)$–te Koordinate wird dann auch Trotter–Zeit oder imaginäre Zeit (vor allem bei Algorithmen bei endlicher Temperatur) genannt. Produkte mit gleichen m–Koordinaten nennt man Slices.
Eine andere Umschreibung aus ist: „The additional dimension plays the role of path integrals in a discrete space ...“.

Für die numerische Anwendung muß der Grenzwert ($n \to \infty$) abbrechen. Der Exponent mit dem Vorfaktor θ kann durch eine Produktzerlegung in τ–slices immer beliebig klein gehalten werden:

$$\mathrm{e}^{(\theta(A+B))} \approx \left(\mathrm{e}^{\tau(A+B)}\right)^n, \quad \tau = \theta/n. \tag{3.9}$$

Der Fehler für endliche n wird ermittelt, indem man die Näherung von einem Parameter λ im Exponenten abhängig macht, den Ausdruck durch sein Integral über die Ableitung nach λ ersetzt und dafür die Norm abschätzt. Man kann zeigen[2], daß gilt:

$$\mathrm{e}^{\tau(A+B)} = \mathrm{e}^{\tau A}\mathrm{e}^{\tau B} + \mathcal{O}(\tau^2), \tag{3.10}$$

mit dem Landauschen Größenordnungssymbol $\mathcal{O}$. Da diese Näherung richtig bis zur Ordnung τ ist, nennt man sie auch 1. Ordnung. Eine Zerlegung 2. Ordnung (korrekt bis zur Ordnung τ^2) ist

$$\mathrm{e}^{\tau(A+B)} = \mathrm{e}^{\tau A/2}\mathrm{e}^{\tau B}\mathrm{e}^{\tau A/2} + \mathcal{O}(\tau^3). \tag{3.11}$$

Bei der praktischen Anwendung einer der obigen Zerlegungen ist zu beachten, daß mit der Genauigkeit gleichzeitig der Rechenaufwand steigt, es dafür aber möglich ist, eine größere Trotterschrittweite zu wählen.

Die Zerlegung 4. Ordnung[3] nach dem obigen Schema liefert

$$\mathrm{e}^{\tau(A+B)} = \mathrm{e}^{\tau A/2}\mathrm{e}^{\tau B/2}\mathrm{e}^{-\tau\mathcal{C}(A,B)}\mathrm{e}^{\tau B/2}\mathrm{e}^{\tau B/2}\mathrm{e}^{\tau B/2} + \mathcal{O}(\tau^5), \tag{3.12}$$

[1] Nomenklatur wie nach Gutbrod in [IFF89], in Maham, Many Particle Physics, Kapitel 3 wird unter „Baker–Haussdorff–Theorem“ die Operatoridentität

$$\mathrm{e}^A B \mathrm{e}^{-A} = B + [A,B] + \frac{1}{2!}[A,[A,B]] + \frac{1}{3!}[A[A,[A,B]]] + \ldots \tag{3.6}$$

verstanden.
[2] M.Suzuki, S.Miyashita, A.Kuroda, Monte Carlo Simulation of Quantum Spin Systems. I, Progress of Theoretical Physics, Vol. 58,5 (1977)
[3] H.DeRaedt, *Product Formula Algorithms for Solving the Time dependent Schrödinger Equation*, Computer Physics Reports 7, North Holland 1987

mit $\mathcal{C}(A,B) = [A+2B,[A,B]]/24$.
Ein Verfahren 4. Ordnung ohne die lästige Berechnung des Doppelkommutators ist die sogenannte „fraktale“ Zerlegung, die auf der Auslöschung von Fehlertermen höherer Ordnung beruht:

$$e^{\tau(A+B)} = e^{s\tau A/2} e^{s\tau B} e^{(1-s)\tau A/2} e^{(1-2s)\tau B} e^{(1-s)\tau A/2} e^{s\tau B} e^{s\tau A/2} + \mathcal{O}(\tau^5) \tag{3.13}$$

$$\text{mit} \quad s = \frac{1}{2-\sqrt[3]{2}} = 1.35230719195965\ldots \tag{3.14}$$

Hierbei ist zu beachten, daß bei einer Anwendung für große τ der Fehler in der TS–Zerlegung kleiner wird, daß aber eventuell das Aufmultiplizieren der Slices instabil wird und daduch numerische Fehler bei der Invertierung auftreten.

3.3 Programmpakete der Numerischen Mathematik

Sie sind am einfachsten über netlib–server zu bekommen, z.B. am Konrad–Zuse Institut in Berlin:durch

```
mail  elib@sc.zib-berlin.dbp.de
send index
```

oder am Oakridge National Laboratory (USA, also sparsam benutzen):

```
mail netlib@ornl.gov
send index
```

Im folgenden werden die wichtigsten mathematischen Bibliotheken kurz charakterisiert. Zu beachten ist, daß viele käufliche Routinen nur hardwareoptimiert laufen, wenn sie auf herstellereigene, optimierte Routinen (z.B. IBM: ESSL, Cray: Scilib, Intel Paragon: BLAS von Kuck and Associates) zurückgreifen können.

BLAS sind die Basic Linear Algebra Subroutines. Sie stellen eine Standard–Schnittstelle für die Abarbeitung linearer Algebra dar und sind Public–Domain. BLAS ist unterteilt in BLAS1 (Vektor–Operationen), BLAS2 (Matrix–Vektor–Operationen) und BLAS3 (Matrix–Matrix–Operationen).

LINPACK, das LINear algebra PACKackge, benutzt Blas–Routinen für höhere Operationen der linearen Algebra (Orthogonalisierungen, Zerlegungen, Invertierungen) und ist ebenfalls Public–Domain.

EISPACK ist ein Public–Domain–Produkt wie Linpack mit dem Schwerpunkt auf Eigenwertproblemen.

LAPACK ist ein neuer Release, der die Funktionalität von Linpack und Eispack unter einer Library zusammenfassen soll, insbesondere auch mit Blick auf Parallelisierung. Einige Features der alten Bibliotheken sind aber dabei verloren gegangen, andere ergeben bitgenau dieselben Ergebnisse.

NAG ist die Programmbibliothek der Numerical Algorithm Group und wohl zur Zeit die umfangreichste kommerzielle Programmsammlung. Die Anwendung umfaßt Graphik, Lineare Algebra (Routinen aus Lapack, aber mit anderen Aufrufen), Differentialgleichungen, Statistik, Zufallszahlen und vieles mehr. Die Dokumentation umfaßt mehr als ein halbes Dutzend DINA4 Ordner. Vor der Benutzung unbedingt die Konventionen (Band1 der NAG–Dokumentation) lesen und die vorhandenen Beispielprogramme durcharbeiten, damit man vor bösen Überaschungen sicher ist: Fettgedrucktes REAL in der Dokumentation bedeutet, daß entweder REAL*8 oder REAL*4 verwendet wird, je nach Installation.

IMSL ist die Programmbibliothek der International Mathematical Society, ebenfalls kommerziell und im Umfang eine Art abgespeckte NAG. Wichtig bei der Cray–Programmierung, da IMSL auch für `real*16` verfügbar ist.

Alle Libraries verfügen über eigene Konventionen bei der Namensgebung ihrer Unterprogramme. Vielfach können aber BLAS–, Linpack–, Eispack– und Lapack–Routinen, die in NAG und IMSL enthalten sind, mit ihren generischen Namen aufgerufen werden. Vor dem Verwenden der Bibliotheken sollte man die Routinen zuerst in kleinen Beispielprogrammen testen.

3.4 Numerische Lineare Algebra und Auswahl von Algorithmen

Viele Zusammenstellungen sind enthalten im *Handbook for Matrix Computations*, Thomas F. Coleman, Charles Van Loan, SIAM (Society for Industrial and Applied Mathematics) 1988.

3.4.1 Die wichtigsten BLAS–Routinen

BLAS (Basic Linear Algebra Subroutines) hat sich als Standard für die elementaren Operationen in der linearen Algebra durchgesetzt. Es ist auch von Vorteil, nicht mehr nur in Programmzeilen und Schleifen, sondern in ganzen Subroutines zu denken, genauso, wie die Verwendung der Vektor– und Matrixschreibweise statt der Indexschreibweise beim analytischen Rechnen oft übersichtlicher ist.
BLAS wurde ursprünglich in und für Fortran entwickelt, die Einbindung in andere Programmiersprachen ist herstellerabhängig, Handbücher konsultieren!
Die Quellcode–Version von Blas ist optimiert für Mainframes, d.h. auf Vektorrechnern und RISCS können sich Modifikationen lohnen, wenn man mit dem Quellcode arbeiten muß, oder wenn keine optimierte Library vorhanden ist.

Verwendete Datentypen: Die Subroutines für verschiedene Gleitkommadatentypen beginnen mit verschiedenen Anfangsbuchstaben,
`s` für Single Precision,
`d` für Double Precision Daten,
`c` für Complex Single Precision,
`z` für Complex Double Precision.

`x,y`	Vektor der entsprechenden Rechengenauigkeit
`n`	Anzahl der Elemente eines Vektors
`incx,incy`	Schrittweite: `inc` Elemente eines Arrays werden übersprungen

Unterprogrammroutinen in Auswahl:

`call scopy(n,x,incx,y,incy)`	Kopieren $\vec{x} \rightarrow \vec{y}$
`call sswap(n,x,incx,y,incy)`	Swappen $\vec{x} \leftrightarrow \vec{y}$
`call sscal(n,a,x,incx)`	Vektorskalierung $\vec{x} \rightarrow a\vec{x}$
`call saxpy(n,a,x,incx,y,incy`	SAXPY $\vec{y} \rightarrow a\vec{x} + \vec{y}$
`w=sdot(n,x,incx,y,incy)`	Skalarprodukt $w = \vec{x} \cdot \vec{y}$
`sw=snorm2(n,x,incx)`	2–Norm
`sw=sasum(n,x,incx)`	1–Norm, größtes Vektorelement

Ausführliche Beschreibungen im „Handbook for Matrix Computations“, Thomas Coleman, Charles Van Loan, SIAM Philadelphia 1988 sowie in vielen Library Reference Manuals, wenn in die numerischen Biliotheken die optimierten BLAS aufgenommen wurden (Cray: Scilib, IBM: ESSL, Intel: BLAS von Kuck and Associates, auch auf SGI Challenge vorhanden).

Bei Matrix–Matrix und Matrix–Vektor–Operationen muß außer der Grösse der Matrizen auch die führenden Dimension angegeben werden, das ist die Länge der Spalten des Array's. Man kann ja eine 10×10 Matrix auch auf einem Array der Größe 20×5 abspeichern, die führende Dimension ist dann 20,.

3.4.2 Linpack, Eispack und MATLAB

Im folgenden werden die wichtigsten Operationen der numerischen linearen Algebra unter ihrem Namen in MATLAB aufgelistet sowie die Subroutine–Namen der Linpack– oder Eispack–Routinen, nach denen sie programmiert sind. Nähre Informationen zu

- Linpack: Linpack User's Guide, J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W.Stewart, SIAM 1979

- Eispack: Matrix Eigensystem Routines, EISPACK GUIDE, B.T. Smith, J.M.Boyle, B.S.Garbow, Y.Ikebe, V.C.Klema, C.B.Moler, Springer 1976 und Matrix Eigensystem Routines, EISPACK Guide Extension, B.S.Garbow, J.M.Boyle, J.J.Dongarra, C.B.Moler Springer 1977

- MATLAB: Es gibt eine Reihe von Handbüchern, die mit MATLAB selber geliefert werden, sowie eine Kurzzusammenfassung als Postscript–File.

Die angegebenen Linpack–Namen sind für komplexe Arithmetik, die Namenskonventionen für die Anfangsbuchstaben **s**,**d**,**c**,**z** sind denen in BLAS analog.

Aufruf in MATLAB	Typ der Zerlegung	Name in Linpack/ Eispack
`[L,U,P]=LU(X)`	$P \cdot X = L \cdot U$, Faktorisierung durch Gauß–Elimination, obere rechte Dreiecksmatrix in U, Permutation einer unteren rechten Dreiecksmatrix in L	ZGEDI, DGEFA
`[Q,R,E]=QR(X)`	$E \cdot X = Q \cdot R$, Orthogonalisierung (stabiler, aber langsamer als das modifizierte Gram–Schmidt verfahren), Q orthogonal, R obere rechte Dreiecksmatrix	ZQRDC, ZQRSL
`[V,D] = EIG(A)`	$A \cdot V = V \cdot D$ Eigenwertzerlegung, Eigenwerte als Vektor in D	BALANC, BALBAK, ORTRAN, ORTHES, HQR2
`[V,D] = EIG(A,B)`	$A \cdot V = B \cdot V \cdot D$ Verallgemeinterte Eigenwertzerlegung	QZHES, QZIT, QZVEC
`[U,S,V]=SVD(X)`	$U \cdot S \cdot V = X$ Singulärwertzerlegung, S Vektor mit den Singulärwerten, U und S Orthogonal	ZSVDC

Die Singulärwertzerlegung ist eine der wichtigsten, da nützlichsten Zerlegungen der numerischen Linearen Algebra. Sie kann durch Iterative Anwendung von QR–Zerlegungen erreicht werden. Man kann zeigen, daß die Singulärwertzerlegung einer reellen Matrix immer reell ist. Das Verhältnis vom größten zum kleinsten Singulärwert einer Matrix nennt man ihre Konditionszahl. Sie gibt an, wie stark die numerischen Größenskalen einer Matrix bei Matrizenoperationen durchmischt werden, so daß Fehler bei der Matrizenberechnung entstehen können (insbesondere bei Inversionen). Die Berechenbarkeit der Singulärwerte selbst ist selbst von der Konditionszahl abhängig, d.h. sehr ungüngstig konditionierte Matrizen sind auch als solche schlecht zu klassifizieren.

3.4.3 MFLOPS und Linpack als Benchmark

Böse Zungen behaupten:„Benchmarks lie, and Liars do Benchmarks“ (Benchmarks lügen, und Lügner benchmarken). So schlimm ist es auch wieder nicht, aber man muß schon wissen, was man von einem Benchmark erwarten kann. Ein Benchmark beschreibt die Geschwindigkeit eines Rechners bei der Abarbeitung bestimmter Programmpakete, Geschwindigkeit eines Rechners „an sich“ gibt es nicht, und theoretische Spitzenleistungen sind zur Performance-Bewertung eines Rechners sicher weniger aussagekräftig als „richtige“ Benchmarks.

Ein beliebter Test für naturwissenschaftlich–technische Anwendungen ist der sogenannte Linpack–Benchmark, benannt nach dem Public–Domain Programmpaket für numerische Lineare Algebra, auf dem er beruht. Der Kern besteht, stark vereinfacht ausgedrückt, aus der numerischen Gauß–Elimination. Die vollständige Tabelle, der fairerweise eine Einleitung und eine Warnung vor Überinterpretationen vorangestellt ist, umfaßt an die 20 Seiten.

- Die 2. Spalte gibt die Geschwindigkeit für den kompilierten Originalquellcode und Matrizen der Größe 100 mal 100 wieder, die 3. Spalte die handoptimierten Routinen für Matrizen der Größe 1000 mal 1000.

- Für die direkt compilierten Programme ist nicht nur der Rechnertyp, sondern auch der Compiler zitiert (Manche Rechner sind einfach schon durch bessere Compiler schneller geworden).

- Die Linpack-Routinen erfordern vor allem Speicherzugriffe, Additionen und Multiplikationen und können keine Anhaltspunkte z.B. über das Verhalten bei Divisionen liefern.
- Die Rechengenauigkeit beträgt `REAL*8`, also etwa 16 Dezimalstellen.

Computer	"LINPACK Benchmark" n=100 OS/Compiler	Mflops/s	"TPP" Best Effort n=1000, Mflops/s	"Theoretical Peak" Mflop/s
Cray C90 (1 proc. 4.2 ns)	CF77 5.0 -Zp -Wd -e68	387	902	952
NEC SX-3/14R (1 proc. 2.5 ns)			5199	6400
IBM Power2 model 590 (66MHz)	-O-Pv-Wp-ea478-gl-qauch=pwrx	130	236	264
DEC 2000-600 Alpha AXP 200 MHz	1.3 -O5 -fast	44	153	200
IBM Power Sys 6000-570 (50MHz)	v2.3.0 xlf-O-Pv-Wp,-ea478	22	73	100
IBM POWERPC 250	-O-Pv-Wp-ea478-gl-qarch=pwrc	13		
SUN Sparc10/40 (40 MHz) (1proz)	f77 1.4 -O3 -cg89 -dalign	8.9	23	40
Gateway P5-60 (60MHz Pentium)	F77L-EM32 5.01 /4 Z1	5.4		
SUN Sparcstation 1	f77 1.3 -O3 -cg89 -dalign	1.3		
Apple Mac Quadra	Absoft -w -v -O -f -s -N40	1.4		
IBM -486 33MHz	Microsoft 5.1	.94		

Der Inhalt der Tabelle ist aus vielen Gründen interessant, aber viel interessanter ist, was nicht in der Tabelle steht.

- Die Reihenfolge der Rechner ergibt sich aus den Flopraten in der 2. Spalte. So kommt es, daß die schnellsten Monoprozessoren der Welt, die NEC SX-3, erst nach den Cray-Prozessoren stehen: Die Systeme sind so klein, daß die NEC die Matrix im Speicher erst mal nicht findet. Die technische Praxis erfordert eher Matrixgrößen wie in der 2. Spalte, da moderne Rechner auch in der Regel über optimierte Subroutines zum Abarbeiten der Linpack-Prozeduren verfügen.
- Man sieht auch, daß die SUN Sparc 10 ihren Namen ungefähr zu recht hat, ist sie doch in etwa 10 mal so schnell wie die Sparc 1, abhängig von der Größe des zu lösenden Problems.
- Man sieht auch, daß klassische Mikrocomputer, egal ob Apple oder solche mit den Intel-Prozessoren 486 und Pentium, auch mit hoher Taktrate nicht mit den RISC-Prozessoren mithalten können.
- Der schnellste RISC-Chip ist nicht der am schnellsten getaktete 200MHz Prozessor von DEC. Der mit „nur“ 66 MHz getaktete IBM Power2-Chip ist deutlich schneller, was die Überlegenheit durchdachter Architektur und ausgereifter Compilertechnik verrät.

Insgesamt basiert der Linpack-Test auf sehr vielen SAXPY-Operationen in der innersten Schleife. Dies benachteiligt im Allgemeinen die RISC–Prozessoren, da hier relativ viele Daten geladen werden müssen. Für Vektorsupercomputer sind die gerechneten Systeme etwas zu klein, wer sein Problem auf einem PC lösen kann, verwendet keine Cray.

Für eigene Programme kann man folgende Abschätzungen treffen:

Auf RISC–Maschinen:
Enthalten Programme prozentual weniger Speicherzugriffe als Linpack–Programme (SAXPY als innerste Schleife), kann man höhere Flopraten erwarten als in der 2. oder 3. Spalte.
Enthält ein Programm prozentual mehr Indexberechnungen und Speicherzugriffe als eine Gauß–Elimination, so ist mit niedrigeren Flopraten als in der 2. oder 3. Spalte zu rechnen.

Auf Vektor–Supercomputern:
Enthalten Programme größere Vektorlängen als 100 bzw. 1000, so kann man höhere Flopraten erwarten als in der 2. oder 3. Spalte, sonst mit niedrigeren Flopraten.

Enthält der Code einen großen Anteil von transzendenten Funktionen und Divisionen, kann man keine Aussagen mehr treffen, das Verhalten wird dann von vorhandenen Coprozessoren und der Qualität der mathematischen Bibliotheken dominiert.

3.4.4 Verhältnis von Datentransfer zu Rechenaufwand

Insbesondere bei Unterprogrammen der numerischen Linearen Algebra fallen Unterschiede in der Anzahl für die benötigten Ladeoperationen im Verhältnis zu den Rechenoperationen auf. Da Workstations in der Regel so schnell rechnen wie laden können, ist eine Minimierung der Ladeoperationen anzustreben. Im folgenden werden die Blas–Namen für die entsprechenden Rechenoperationen verwendet. Die Systemgröße ist jeweils l, Konstanten vor den Größenordnungen werden diesmal berücksichtigt.

SAXPY steht für A mal X Plus Y und addiert das A–fache eines Vektors X zum Vektor Y. Die Anzahl der Rechenoperationen ist $\mathcal{O}(2l)$, die Anzahl der Loads $\mathcal{O}(2l)$ und die Anzahl der Stores $\mathcal{O}(l)$. Das Verhältnis von Memory–Zugriffen zu Rechenoperationen ist also $\mathcal{O}(1.5)$.

SDOT steht für ein Skalarprodukt. Die Anzahl der Rechenoperationen ist $\mathcal{O}(2l)$, die Anzahl der Loads $\mathcal{O}(2l)$ und die Anzahl der Stores $\mathcal{O}(1)$. Das Verhältnis von Memory–Zugriffen zu Rechenoperationen ist also $\mathcal{O}(1)$.

SGEMM steht für ein Matrizenprodukt. Die Anzahl der Rechenoperationen ist $\mathcal{O}(2l^3)$, die Anzahl der Loads $\mathcal{O}(2l^2)$ und die Anzahl der Stores $\mathcal{O}(l^2)$. Das Verhältnis von Memory–Zugriffen zu Rechenoperationen ist also $\mathcal{O}(1/l)$.

Damit ist zu erwarten, daß die Anzahl der Gleitkommaoperationen für die zitierten Subroutines in der gegebenen Reihenfolge monoton ansteigt. Dies ist auch in der Realität der Fall. Gemessen wird diese Geschwindigkeit in MFLOPS (Million FLOating Point operations per Second). Bei den Gleitkommaoperationen werden in der Regel nur Multiplikationen und Additionen berücksichtigt. Wenn die maximale erreichbare Floprate von Workstations angegeben wird, bezieht sie sich in der Regel auf Matrix–Matrix–Multiplikationen.

Warnung:
Eine LU–Zerlegung ist eine Faktorisierung einer Matrix A in ein Produkt zweier Matrizen, von denen eine eine obere rechte Dreiecks–Matrix ist: $A = L * U$ (Dreiecksmatrizen können sehr leicht invertiert werden, das ist der Sinn des ganzen). Dies kann sowohl SAXPY–basiert (Linpack) oder SDOT–basiert (sog. Crout–Algorithmus, Numerical Recipes) programmiert werden. Während aber das normale Skalarprodukt von Vektoren durchaus schneller ist als eine vergleichbare SAXPY–Operation, muß bei Crout das Skalarprodukt von Zeilen und Spalten

berechnet werden: Der spaltenweise Zugriff dauert selbst auf skalaren Prozessoren um soviel länger, daß Linpack–artige LU's etwa doppelt so schnell sind als Crout–artige.

Vorsicht ist bei kleinen Systemgrößen geboten: Hier kann eventuell die Anzahl von Divisionen dominieren (z.B. Gauß–Eliminationsverfahren), die in die Abschätzung der Flops nie eingehen.

Literaturverzeichnis

[Aho92] Alfred V.Aho, Jeffrey D.Ullman, *Foundations of Computer Science*, Freeman and Company, 1992, sehr gute Einführung in das Fachchinesisch der Computer Science, nicht so theoretisch wie die Knuth–Bände, Big-O erklärt.

[Bod91] Arndt Bode (Hrsg.), *RISC-Architekturen*, BI-Wissenschaftsverlag Mannheim (1991)

[Dow93] Kevin Dowd, *High Performance Computing*, O'Reilly & Associates (1991)

[Knu73] Donald E. Knuth, *Fundamental Algorithms*, Addision Wesley 1973

[Met] Michael Metcalf, *Effective Fortran77*, Tips zu Programmoptimierung. Teilweise leider etwas überholt, da auf die Speicherorganisation der Großrechner, weniger der RISC's abgestimmt.

[Sed92] Robert Sedgewick, *Algorithmen*, Addision Wesley 1992, Standardalgorithmen der Computer Science wie Suchen, Sortieren etc., Bände für C, C++ und Pascal, Addison–Wesly 1992

[Sto90] H.Stone, *High Performance computer architecture*, Addison–Wesley Publishing Company (1990)

[War91] Stephen A. Ward, Robert H.Halstead, *Computation Structures*, The MIT Press, 1991

Einführung in die Computersimulation:

[All87] M.P.Allen, D.J.Tildesly, *Computer Simulation of Liquids*, Oxford Science 1987 Schwerpunkt auf MD–Algorithmen/ Integration der Bewegungsgleichungen.

[Hee90] D.W.Heermann, *Computer Simulation Methods*, Springer 1990, Schwerpunkt auf der Statistischen Physik.

[Hoc88] R.W.Hockney, J.W.Eastwood, *Computer Simulation Using Particles*, IOP Publishing 1988 Schwerpunkt auf Molekulardynamik in der Plasma–Physik (Teilchen und Felder).

[IFF89] 20. IFF Ferienkurs, *Computersimulation in der Physik*, KFA Jülich 1989 Zentralbibliothek

Numerik:

[Abr] Milton Abramowitz, Irene Stegun, *Handbook of Mathematical Functions*, NBS 1964, Nachdruck Dover, Tabellenwerk für Funktionen, mit Formeln zur Berechnung derselben, Standardwerk

[Act90] F.S.Acton, *Numerical Methods that (usually) work*, The Mathematical Asscociation of America, 2. ed. Washington 1990, Gute problemorientierte Einführung, sehr verbal, keine Programmierbeispiele

[Don94] Jack J. Dongarra, aktueller Linpack–Benchmark in *Performance of Various Computers Using Standard Linear Equations Software*, erhältlich als Postscript-File vom Netlib–Server des Oakridge National Laboratory:
`mail netlib@ornl.gov`
send performance from benchmark

[Fle90] Clive A.J. Fletcher, *Computational Techniques for Fluid Dynamics*, Vol.1, 2.ed. Springer 1990, Einführung in die numerische Behandlung von partiellen Differentialgleichungen, nicht Navier–Stokes, das kommt im 2. Band und der ist momentan vergriffen.

[For77] George E. Forsythe, Michael A. Malcom, Cleve B. Moler, *Computer Methods for Mathematical Computations*, Prentice–Hall 1977 Beste Numerik–Einführung aller Zeiten, leider nicht mehr im Druck (Fernleihe!), die angegebenen Algorithmen sind aber zum Teil überholt

[Gol89] Golub, VanLoan, *Matrix Computations*, The Johns Hopkins University Press, London and Baltimore (1989), Bibel für die numerische lineare Algebra.

[Hor91] Roger A. Horn, Charles R. Johnson, *Matrix Analysis* Cambrigde University Press 1991, All you need in Linear Algebra

[Loc92] Franz Locher, *Numerische Mathematik für Informatiker*, Springer 1992, Formale Einführung, im Gegensatz zu vielen Numerik–Büchern aber nützlich und verständlich

[Pre92] W.H.Press, B.P.Flannery, S.A.Teukolsky, W.T.Vetterling, *Numerical Recipes, The Art of Scientific Computing*, Cambridge University Press, 2.ed. 1992, Kochrezepte und Beispielprogramme für die Numerik–Programmierung, Bände für Fortran, C, Pascal und BASIC. Aus dem Beipielen kann man viel für den eigenen Programmierstil lernen, sollte jeder haben, der viel programmieren muß. Trotzdem Vorsicht: Viele Algorithmen sind nicht optimal