Ausgewählte Kapitel aus dem Übersetzerbau

Prof. Gerhard Goos
Fakultät für Informatik
Universität Karlsruhe

Sommersemester 2006

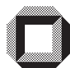
©Goos 2006

http://www.info.uni-karlsruhe.de/

Nebenläufige Sprachen

Inhalt - Nebenläufige Sprachen

- Begriffe und Konzepte
  - Parallelität
  - Granularität
  - Programmiersprachen Konzept: Thread
- Parallele Hardware-Architekturen
  - MIMD: Multiprozessoren
    - Kopplung von Hardware mittels Software
      - Speicher
      - Nachrichten
  - SIMD: Feldrechner, Vektorrechner
- Implementierung von Parallelität
  - Manuell
  - Automatische Vektorisierung
  - Automatische Parallelisierung
  - Werkzeuge und Sprachen



Nebenläufigkeit: Aufgaben für den Übersetzer

Alle Aufgaben werden im Kontext von FORTRAN diskutiert.
Grund: Alle Programme im wissenschaftlichen Rechnen in FORTRAN.
Andere Anwendungen: Transaktionsverarbeitung für große Datenbanken (in C, C++).

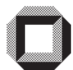
- Verarbeite parallele Erweiterungen von FORTRAN, z. B. HPF.
- Nutze potentielle Parallelität in sequentiellen Programmen.

Berücksichtige Hardwareabhängigkeiten:

- Datenflußrechner (hier nicht behandelt)
- Prozessor-interne Parallelität: superskalar, VLIW
- SIMD-Rechnern, speziell Vektorrechner,
- MIMD-Rechner mit/ohne gemeinsamem Speicher
  Modell Synchronisation im Speicher oder Botschaftenaustausch
- gekoppelte Rechner (NOW, Parastation, . . . , “the Grid”)
- Theoriegestützte Modelle: PRAM, BSP- und LogP-Modell

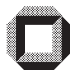
Einsichten:

- Parallelisierung und Optimierung von Hardwarearchitektur abhängig.
- Architekturunabhängiges paralleles Programmieren: ungelöst
- Übersetzer oft auf Unterstützung (Übersetzeroptionen) des Programmierers angewiesen.



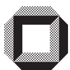
Ebenen der Parallelität

- Ein (paralleles) Programm läßt sich als halbgeordnete Menge von Befehlen darstellen, wobei die Ordnung durch die Abhängigkeiten der Befehle untereinander gegeben ist (siehe SSA mit Speicherkanten).
  Befehle, die nicht voneinander abhängig sind, können parallel ausgeführt werden. Weitere Parallelisierung mit Synchronisation.
- Unterscheide:
  - Ebenen der Parallelität: in einem Programm
    * Wegen Datenabhängigkeiten müssen die parallelen Einheiten zumindest am Anfang (und Ende?) miteinander kommunizieren.
    * Abhängigkeiten dazwischen können den Ablauf einer Einheit anhalten, bis eine andere bestimmte Operation beendet hat.
    * Die Kommunikation kann durch Ereignisse oder Bedingungen erfolgen.
  - Techniken der Parallelarbeit: in der Hardware
    * Speichergestützte Kommunikation über einen gemeinsamen Speicher (Semaphore), oder Botschaftenaustausch.



Endebehandlung

- alle warten aufeinander
  Beispiele: Konsensus-Verfahren beim verteilten Schreiben in eine Datenbank, alle Standard-Schleifenparallelisierungen
- warten auf den Ersten "*winner takes all*"
  Beispiele: parallele Suche, RAS Entschlüsselung
- kein Warten
  Beispiel: Druckjob

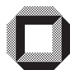


Fünf Ebenen der Parallelität

1. **Programmebene** (oder Jobebene)
2. **Prozeßebene** (oder Taskebene): “*tasks*” (schwergewichtige Prozesse, “*heavy-weighted processes*”, “*coarse-grain tasks*”)
3. **Blockebene**: Anweisungsblöcke oder leichtgewichtige Prozesse (“*threads*”, “*light-weighted processes*”). Innere oder äußere parallele Schleifen in Fortran-Dialekten, Microtasking und “*large-grain*”-Datenfluß als Programmiertechnik.
4. **Anweisungsebene** (oder Befehlsebene): Elementare Anweisungen (in der Sprache nicht weiter zerlegbare Datenoperationen)
5. **Mikrooperationsebene**: Eine elementare Anweisung wird durch den Übersetzer oder die Hardware in Mikrooperationen aufgebrochen, die parallel ausgeführt werden. Beispiel: VLIW, Vektorrechner, superskalarer Rechner

Ebenen der Parallelität und Körnigkeit

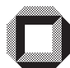
- Die **Körnigkeit** oder **Granularität** (grain size) ergibt sich aus dem Verhältnis von Rechenaufwand zu Kommunikations- oder Synchronisationsaufwand.
  Sie bemißt sich nach der Anzahl der Befehle in einer sequentiellen Befehlsfolge.
- Programm-, Prozeß- und Blockebene werden häufig auch als **grobkörnige** "*large grained*" Parallelität,
- die Anweisungsebene als **feinkörnige** "*finely grained*" Parallelität bezeichnet.
- Seltener wird auch von **mittelkörniger** "*medium grained*" Parallelität gesprochen, dann ist meist die Blockebene gemeint.



Technische Realisierungen der Parallelitätsebenen

(1 Programmebene, 2 Prozeßebene, 3 Blockebene,
4 Anweisungsebene, 5 Suboperationsebene)

Parallelarbeitstechniken	**1**	**2**	**3**	**4**	**5**
Techniken der Parallelarbeit durch Rechnerkopplung					
Hyper- und Metacomputer	×	×			
Workstation-Cluster	×	×			
Techniken der Parallelarbeit durch Prozessorkopplung					
Nachrichtenkopplung	×	×			
Speicherkopplung (SMP)	×	×	×		
Speicherkopplung (DSM)	×	×	×		
Grobkörniges Datenflußprinzip		×	×		
Techniken der Parallelarbeit in der Prozessorarchitektur					
Befehlsfließband				×	
Superskalar				×	
VLIW				×	
Überlappung von E/A u. CPU				×	
Feinkörniges Datenfluß prinzip				×	
SIMD-Techniken					
Vektorrechnerprinzip					×
Feldrechnerprinzip					×



Amdahls Gesetz – Grenzen der Parallelisierung

Die **Beschleunigung** σ wird als Quotient

$$\frac{\text{Ausführungszeit ohne Optimierung}}{\text{Ausführungszeit mit Optimierung}}$$

definiert.

Amdahl beschrieb 1967 die Beschleunigung σ in Abhängigkeit von

α Der Bruchteil der Arbeit, der sequentiell abgearbeitet werden muß.

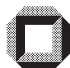
p Die Anzahl der Einheiten, die parallel Arbeit verrichten können.

$$\sigma(\alpha, p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Offenbar wird die Beschleunigung durch $1/\alpha$ beschränkt. Wenn also ein Programm einen nicht marginalen sequentiellen Anteil hat, lohnen sich viele Paralleleinheiten überhaupt nicht!

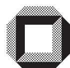
Prozeß/Fäden

- Eine sequentiell ausgeführte Folge von Operationen heißt ein (sequentieller) **Prozeß**.
- In der Terminologie der Betriebssystemliteratur ist dies ein **Faden**, "*thread*". Er besitzt nicht zwangsläufig einen eigenen Adreßraum wie ein Betriebssystemprozeß.
- Bei parallelem Rechnen werden mehrere solche Prozesse nebenläufig ausgeführt.
- Nebenläufige Programmiersprachen können Prozesse implizit oder explizit definieren z.B. `forall ...do ...` oder `new Thread.start()`. Die Kommunikation ist dann ebenfalls implizit oder explizit.
- Die Prozesse können auf einen oder mehrere Betriebssystem/Hardware-Prozesse abgebildet (serialisiert) werden. So kann eine nebenläufige Programmiersprache auf Parallel-Hardware eingesetzt werden, muß es aber nicht.



Inhalt - Nebenläufige Sprachen

- Begriffe und Konzepte
  - Parallelität
  - Granularität
  - Programmiersprachen Konzept: Thread
- Parallele Hardware-Architekturen
  - MIMD: Multiprozessoren
    - Kopplung von Hardware mittels Software
      - Speicher
      - Nachrichten
  - SIMD: Feldrechner, Vektorrechner
- Implementierung von Parallelität
  - Manuell
  - Automatische Vektorisierung
  - Automatische Parallelisierung
  - Werkzeuge und Sprachen



Parallele Rechenmodelle: PRAM

PRAM: Parallele Registermaschine

- $p \geq 1$ Prozessoren: $p_0, p_1, ...$
- gemeinsamer Speicher unbeschränkter Größe von Speicherzellen R_i, können ganze Zahlen unbeschränkter Größe aufnehmen
- übliche sequentielle Operationen
- Parallelanweisung: for $v \in M$ do in parallel Rumpf end
  $\Longrightarrow \mid M \mid$ Prozesse werden gestartet

Parallele Rechenmodelle: Variationen der PRAM

- **synchrone PRAM**: Prozesse der Parallelisierung werden synchron im Takt ausgeführt
- **EREW-PRAM**: Exclusive Read Exclusive Write
- **CREW-PRAM**: Concurrent Read Exclusive Write
- **CRCW-PRAM**: Concurrent Read Concurrent Write
- **APRAM**: asynchrone PRAM: Barrieresynchronisation: Snyhronisation aller Prozessoren

Netzwerkmodelle

- lokaler versus globaler Speicher
  Zugriffe in lokalem Speicher viel schneller
- gemeinsamer Speicher bei steigender Prozessorzahl immer teurer
  $\Longrightarrow$ dupliziere evtl. Daten in lokalem Speicher
- betrachte statt Modell mit speichergestützter Kommunikation Modell mit synchronem Botschaftensystem
  Grundoperationen:
  - send(v:T, P:Prozeß) – sende Datum an Prozeß P
  - receive(P:Prozeß):T – warte auf Datum vom Typ T von Prozeß P
- jeder Prozeß hat eigenen lokalen Speicher
- kein globaler Speicher
- asynchrone Ausführung von Prozessen
- Kommunikation durch Sende- und Empfangsoperationen

Netzwerkmodelle: BSP-Modell

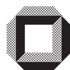
BSP: bulk-synchroneous parallel model (Valiant 1989, 1990)

- statt Parallelanweisung Superschritt: 2 Phasen: Rechenschritte und Kommunikationsoperationen + Barrieresynchronisation
- versandte Daten liegen erst zu Beginn des nächsten Superschritts beim Empfänger vor, Sender wartet nicht auf Empfänger
- BSP-Programm: Menge von Superschritten
- Ausführung BSP-Programm: beeinflusst durch drei Parameter:
  - p: Anzahl Prozessoren
  - L: Dauer Barrieresynchronisation und Kommunikation zwischen aufeinanderfolgenden Superschritten
  - g: Bandbreite eines Prozessors (nur alle g Zeiteinheiten kann Sende-/Empfangsoperation durchgeführt werden
- Nachteil BSP-Modell: Barrieresynchronisationen, die in Realität teuer sind

Netzwerkmodelle: LogP-Modell

(Culler, Karp, Patterson etal., 1996)

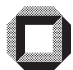
- keine Superschritte und keine Barrieresynchronisation
- stattdessen: Synchronisation durch explizite Kommunikationsoperationen
- Parameter **LogP**:
  - **L**: Kommunikationsverzögerung (latency)
  - **o**: Verwaltungsaufwand (overhead): Rechenzeit eines Prozessors beim Senden bzw. Empfangen einer Nachricht
  - **g**: Totzeit (gap) zwischen aufeinanderfolgenden Sende- bzw. Empfangsoperationen eines Prozessors
  - **P**: Anzahl Prozessoren der LogP-Maschine



Parallelrechner - Flynnsche Klassifikation

Flynn charakterisiert Rechner als Operatoren auf zwei verschiedenartigen Informationsströmen: dem **Befehlsstrom** "*instruction stream*" und dem **Datenstrom** "*data stream*".

- **SISD** "*Single Instruction stream over a Single Data stream*"
  - die von-Neumann-Architekturen (Einprozessorrechner)
  - seriell, deterministisch
- **MISD** "*Multiple Instruction streams over a Single Data stream*"
  - multiple Frequenzfilter, Quantenrechner
  - nicht klassische Rechner
- **SIMD** "*Single Instruction stream over Multiple Data streams*"
  - die Feldrechner und die Vektorrechner
  - synchron, deterministisch
- **MIMD** "*Multiple Instruction streams over Multiple Data streams*"
  - die Multiprozessorsysteme
  - synchron/asynchron, deterministisch/nicht deterministisch

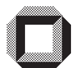


Wichtige parallele Architekturen

- **VLIW-Rechner** "*Very Large Instruction Word*": mehrere parallele Rechenwerke (praktisch: 3-6), Beispiel: IA-64, EPIC "*Explicitly Parallel Instruction Computer*"
- **Feldrechner** (SIMD): Rechner mit einem Feld von regelmäßig verbundenen Verarbeitungselementen, die unter Aufsicht einer zentralen Steuereinheit immer gleichzeitig dieselbe Maschinenoperation auf verschiedenen Daten ausführen.
  Heute praktisch keine Bedeutung.
  Zukünftig: Verheiraten von Speichern und (Grafik-)Prozessoren.
- **Vektorrechner** (SIMD): Rechner mit parallel arbeitenden (Gleitpunkt-)Rechenwerken, Arbeit im Fließband verknüpft. Parallelitätsgrad 16 bis 64.
- **Multiprozessoren** (MIMD): Rechner mit mehreren konventionellen Prozessoren (SISD), die entweder über den Speicher oder über Nachrichten gekoppelt sind.

Vektorrechner

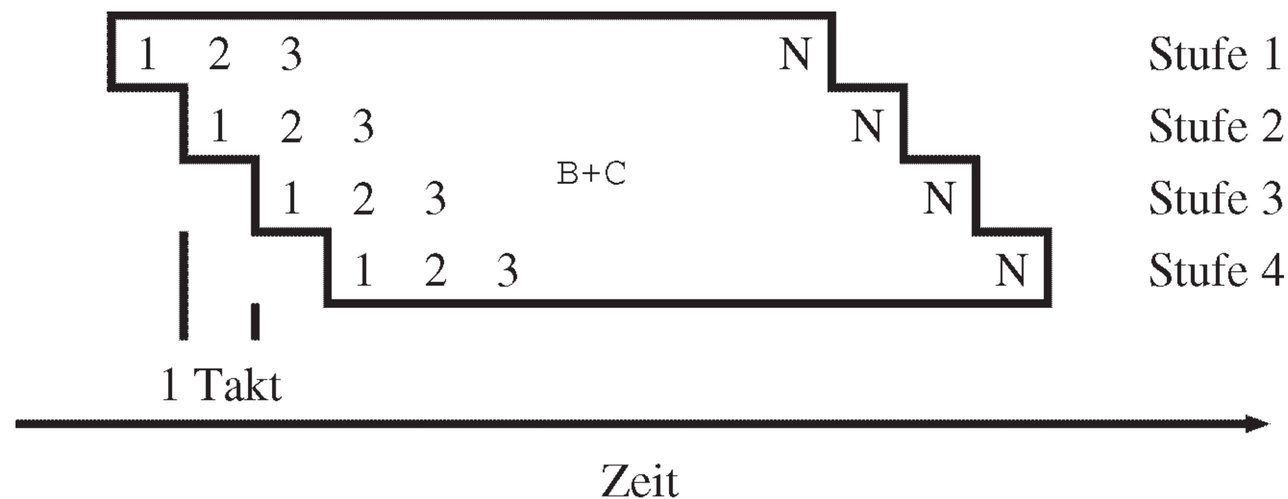
- **Vektorrechner**: Rechner mit parallel arbeitenden (Gleitpunkt-) Rechenwerken, Arbeit im Fließband verknüpft.
- **Vektor** = Reihung von Gleitpunktzahlen.
- Jeder Vektorrechner besitzt einen Satz von **Vektorfließbändern** (**Vektoreinheiten**).
- Im Gegensatz zur Vektorverarbeitung heißt die Verknüpfung einzelner Operanden **Skalarverarbeitung**.
- Ein Vektorrechner enthält neben Vektoreinheiten auch eine oder mehrere **Skalareinheiten**. Dort werden die **skalaren Befehle** ausgeführt, d.h. Befehle, die nicht auf ganze Vektoren angewendet werden sollen.
- Die Vektoreinheit und die Skalareinheit(en) können parallel zueinander arbeiten, d.h. Vektorbefehle und Skalarbefehle können parallel ausgeführt werden.



Beispiel Addition

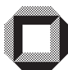
```
A(J) = B(J) + C(J),     J = 1,2,...,N
```

Hier werden die Vektoren `B` und `C`, d.h. die Reihungselemente `B[1]`, `...`, `B[N]` und `C[1]`, `...`, `C[N]`, mit einem Befehl komponentenweise addiert und im Ergebnisvektor `A` abgespeichert.
Die Vektoren werden dabei sequentiell und überlappt abgearbeitet, d.h. zuerst wird die Berechnung `B[1]+C[1]` gestartet, dann `B[2]+C[2]`, usw.



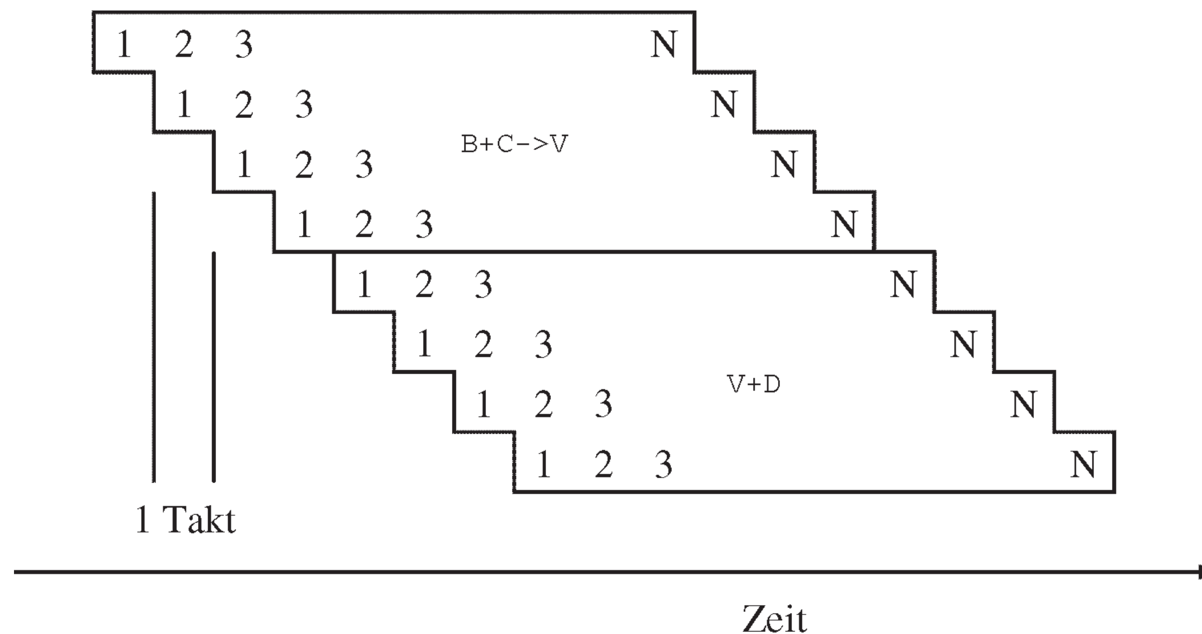
Besonderheit eines Vektorrechners

- Die Fließband-Verarbeitung wird mit einem Vektorbefehl für zwei Reihungen von Gleitpunktzahlen durchgeführt.
- Die bei den Gleitpunkteinheiten skalarer Prozessoren nötigen Adreßrechnungen entfallen.
  Genauer: werden implizit von der Lade-/Speichere-Einheit des Prozessors miterledigt.
- Bei ununterbrochener Arbeit im Fließband erhält man nach einer **Einschwingzeit** bzw. **Füllzeit** mit jedem Fließband-Takt ein Ergebnis.
- Die Fließband-Taktdauer ist durch die Dauer der längsten Teilverarbeitungszeit zuzüglich der Stufentransferzeit gegeben.



Verkettung

- **Verkettung** "*chaining*": Das Fließband-Prinzip kann auch auf eine Folge von Vektoroperationen erweitert werden.
- Zu diesem Zweck werden die (spezialisierten) Fließbänder miteinander verkettet, d.h. die Ergebnisse von einem Fließband werden sofort dem nächsten Fließband zur Verfügung gestellt.
- Beispiel:
  Fließband-Verkettung von `B(J)+C(J)+D(J),     J=1,2,...,N`

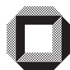


Arten von Multiprozessoren

- Bei **speichergekoppelten Multiprozessoren** besitzen alle Prozessoren einen gemeinsamen Adreßraum.
  Kommunikation und Synchronisation geschehen über gemeinsame Variablen.
  - Symmetrischer Multiprozessor **SMP**: ein globalen Speicher
  - Distributed-shared-memory-System **DSM**: gemeinsamer Adreßraum trotz physikalisch verteilter Speichermodule
- Bei den **nachrichtengekoppelten Multiprozessoren** besitzen alle Prozessoren nur physikalisch verteilte Speicher und prozessorlokale Adreßräume.
  Die Kommunikation geschieht durch Austauschen von Nachrichten.
  Mit wachsender **Kommunikations-Verzögerung** kommt man zu:
  - Verteiltem Rechnen in einem **Workstation-Cluster**
  - **Metacomputer**:
    Zusammenschluß weit entfernter Rechner oder Cluster
    * Kopplungsgrad nimmt ab, Programme müssen immer grobkörniger sein.
    * Skalierbarkeit der Hardware nimmt zu.

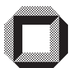
Vergleich Speicher-/Nachrichten-Kopplung

- Speicher-Kopplung ist
  - leichter programmierbar,
  - auch für kommunikations- und synchronisationsaufwendige Programme geeignet
  - SMPs skalieren bis ca. 20 Knoten, DSMs bis ca. 256
- Voraussetzung für den Einsatz von Nachrichten-Kopplung
  - Auf „Prozeßebene“ parallelisierbares Problem, wenig Kommunikationsaufwand
- Für Höchstleistungen: Nachrichtenkopplung (z.B. IBM SP2)



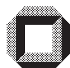
Grenzbereiche

- **Eingebettete Systeme** als spezialisierte Parallelrechner.
- **Superskalarprozessoren**, die feinkörnige Parallelität durch Befehls-Fließband und durch die Superskalartechnik nutzen.
- Ein Mikroprozessor arbeitet als Hauptprozess oder teilweise gleichzeitig zu einer Vielzahl von spezialisierten Einheiten wie der Bussteuerung, DMA- und Graphikeinheit.
- **Ein-Chip-Multiprozessoren**
- **Mehrfädige Prozessoren** "*multithreaded processors*" führen mehrere Ablauffäden überlappt oder simultan innerhalb eines Prozessors aus. IBM, Intel und α sehen in dieser Architektur das größte Potential für die nächsten 15 Jahre.
- **Datenflußrechner**: Steuerung durch Gültigkeit von Daten. Im Grenzfall als mehrfädige Prozessoren zu betrachten.
- **VLIW-Prozessoren** "*Very Long Instruction Word*":
  Viele Recheneinheiten werden durch einen Befehl parallel genutzt (IA-64, SSE, MMX).



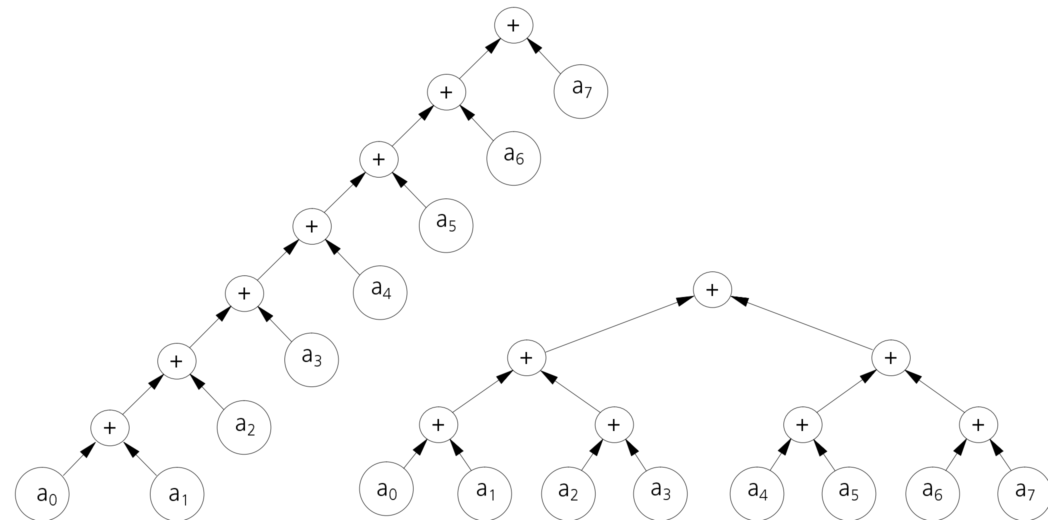
Inhalt - Nebenläufige Sprachen

- Begriffe und Konzepte
  - Parallelität
  - Granularität
  - Programmiersprachen Konzept: Thread
- Parallele Hardware-Architekturen
  - MIMD: Multiprozessoren
    - Kopplung von Hardware mittels Software
      - Speicher
      - Nachrichten
  - SIMD: Feldrechner, Vektorrechner
- Implementierung von Parallelität
  - Manuell
  - Automatische Vektorisierung
  - Automatische Parallelisierung
  - Werkzeuge und Sprachen



Vektorisierung / Parallelisierung

Serielle vs. parallele Sicht: Falls möglich im Entwurf darauf achten, dass Algorithmen parallel konstruiert werden.



- Prinzipielles Vorgehen (manuell):
  - Algorithmisch: z.B. durch Teile-und-Herrsche-Entwurf
  - Explizites Einfügung von Bibliotheksroutinen
  - Vorverarbeitung (+ graphische Visualisierung)
  - Restrukturierung
  - Einsatz von Werkzeugen, z.B. zur Programmanalyse
  - Abhängigkeiten erhalten
- Bei Vektorisierung / Parallelisierung durch den Übersetzer Annotationen in der Quelle und zusätzlich Analysen.

Vorverarbeitung

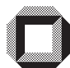
Diese Transformationen dienen der Normalisierung. Die eigentliche Vektorisierung / Parallelisierung findet im Anschluß daran statt.

- **Operatorvereinfachung** insbesondere Reihungszugriff- bzw. Index-Normalisierung
  In diesem Kontext ist die Standardisierung der affinen Index-Funktion wichtig.
- **Schleifen-Normalisierung**
  Z.B. eine Fortran DO-Schleife ist in Normalform, wenn sie bei 1 beginnt und die Schrittweite 1 ist.
- **Umbenennung von Skalaren**
  verschiedene Definitionen (Zuweisungen) einer Variablen durch verschiedene Variable ersetzen. (wird durch SSA-Form schon geleistet)

Grundlegende Transformationsschritte

- Zunächst Transformationsschritte, die der Vektorisierung dienen, jedoch noch nicht maschinenabhängig sind.
- Das Ziel der Transformationen ist das Herausfaktorisieren einzelner Schleifen, die alle uniform auf Reihungen zugreifen. Diese Schleifen können dann als einzelner Vektorbefehl von einer Vektoreinheit verarbeitet werden.
- Die Schleifen können ggf. zu mehrstufigen Vektorbefehlen kombiniert werden. Dafür müssen sie manchmal erst in Streifen geschnitten werden.
- Beispielprogramm:

```
DO I=1,100
   A(99-I)  = B(102-I) + C(101-I)
   E(101-I) = F(98-I) / A(99-I)
END DO I
```



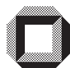
Maschinenunabhängige Transformationen

- **Schleifenteilung**, "*loop distribution*": man erhält zwei Schleifen:

```
DO I=1,100
   A(99-I)  = B(102-I) + C(101-I)
END DO I

DO I=1,100
   E(101-I) = F(98-I) / A(99-I)
END DO I
```

- Falls die Schleifenteilung nicht nur zwei einzelne Anweisungen, sondern zwei Anweisungsblöcke im Schleifenrumpf betrifft, spricht man auch von "*loop fission*".
- Ähnliche Technik: **Anweisungsumordnung**, "*statement reordering*", innerhalb eines Schleifenkörpers, bei der die Position zweier aufeinanderfolgender Anweisungen geändert wird. Möglich, wenn keine Abhängigkeit zwischen den beiden Anweisungen besteht.



Techniken zur Erleichterung der Vektorisierung I

- **Schleifenvertauschen**, "*loop interchange*": Vertauschen der inneren und äußeren Schleife.
  Anwendung: Verketten der Fließbänder, zeitliche Abfolge der Abhängigkeiten ändern.
- **Skalar Expansion**, "*scalar expansion*": Skalare Variable in einem Schleifenrumpf durch eine Reihung ersetzen, so dass in jeder Schleifeniteration eine eigene Kopie der Variablen vorkommt.
- **Kopieren von Variablen**, "*variable copying*".
  Anwendung: Abhängigkeiten reduzieren (SSA-Idee), Variable expandieren.
- **Auseinanderziehen der Indexmenge**, "*index set splitting*"
  Anwendung: getrennt Vektorisieren wegen Abhängigkeiten.
- Ersetzen eines Anweisungsknotens in einem Abhängigkeitsgraphen durch einen Teilgraphen, um einen verfeinerten Abhängigkeitsgraphen zu erhalten, "*node splitting*".

Techniken zur Erleichterung der Vektorisierung II

- **Schleifenumkehr** wegen Zählerhardware oder Organisation der Fließbänder.
- **Ausrollen** einer einzelnen Schleifeniteration “*loop peeling*”.
- Ausrollen *aller* Schleifeniterationen “*loop unrolling*”.
- **Aufrollen** von Codeblöcken zu einer Schleife “*loop rerolling*”.

  Die „Rolltechniken“ werden benötigt, um genügend Code für eine Verkettung der Fließbänder zu haben oder um Randfälle los zu werden.

- **Erkennung Standardfall**, “*idiom recognition*”: häufig verwendete Konstruktionen erkennen und durch hoch optimierte Codeteile ersetzen.

→ Danach Vektorisierung durchführen, d.h. Vektorbefehle erzeugen.

Vektor-Befehlserzeugung

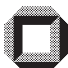
```
DO I=1,100
        A(99-I)  = B(102-I) + C(101-I)
END DO I

DO I=1,100
        E(101-I) = F(98-I) / A(99-I)
END DO I
```

- Die **Vektorisierung** der beiden Schleifen des Beispielprogramms führt zu den folgenden beiden Vektorbefehlen:

```
A(-1:98) = B(2:101) + C(1:100)
E(1:100) = F(-2:97) / A(-1:98)
```

- `A(-1:98)` Vektor mit den Komponenten `A(-1),A(0),...,A(98)`
- Die Operatoren + und / werden komponentenweise angewandt und die Resultatvektoren den Vektoren auf der linken Seite der Anweisungen zugewiesen.



Korrekte Vektorisierung

- Die folgende Vektorisierung ist korrekt:

  ```
  DO I=1,100
     X(I) = X(I) + Y(I)
  END DO
  ```

  wird vektorisiert zu:

  ```
  X(1:100) = X(1:100) + Y(1:100)
  ```

- Beachte: `X(1:100) = X(1:100) + ...`
  auf die rechte Seite muss zugegriffen werden, bevor die linke Seite gespeichert wird

Korrekte Vektorisierung

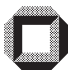
- Die folgende Vektorisierung ist nicht korrekt:

```
DO I=1,100
   X(I+1) = X(I) + Y(I)
END DO
```

  kann nicht vektorisiert werden zu:

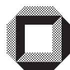
```
X(2:101) = X(1:100) + Y(1:100)
```

- Um derartige Fehler zu vermeiden, muss bei der Vektorisierung der Abhängigkeitsgraph mitbetrachtet werden.
- Nicht ganz präzise Faustregel: es darf nicht über Zyklen im Abhängigkeitsgraphen vektorisiert werden.
- Ein darauf beruhender Algorithmus für die Vektor-Befehls-erzeugung ist der **Allen-Kennedy-Algorithmus**.



Maschinenabhängige Transformationen I

- **Schleifenpartitionieren** “*loop sectioning*” oder “*strip mining*”: eine Schleife wird in eine äußere partitionierende Schleife “*sectioning loop*” und einen Block mit Vektoroperationen “*strip loop*” unterteilt.
- **Trade-off-Punkt** “*vector breakeven length*”: diejenige Vektorlänge, ab der eine Vektoroperation effizienter als die Verwendung von skalaren Operationen durchgeführt wird.
- Fließbänder können zu einem langen Band verkettet werden oder die Bänder können parallel benutzt werden. Dabei ist der jeweilige Trade-off-Punkt zu beachten.
  Um die Fließbänder optimal auszunutzen, muß die Schleife oft partitioniert werden.
- Verbesserung der **Datenlokalität**



Maschinenabhängige Transformationen II

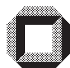
- Bei der **Ablaufflluß-Vektorisierung** werden bedingte Ausdrücke und Verzweigungen mit **maskierten Befehlen** vektorisiert.
  Beispiel:

Original:

```
DO I=1,N
1: A(I)=A(I)+B(I)
2: IF A(I)=0 THEN
3:    GOTO 7 FI
4: IF A(I)>C(I) THEN
5:    A(I)=A(I)-2
   ELSE
6:    A(I)=A(I)+1
   FI
7: D(I)=A(I)+1
END DO
```

Mit Maskierungsbedingungen:

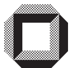
```
DO I=1,N
1: A(I)=A(I)+B(I)      TRUE
2':%C2=( A(I)=0 )      TRUE
2: IF %C2 THEN         TRUE
3:    GOTO 7 FI        %C2
4':%C4=( A(I)>C(I) )   NOT(%C2)
4: IF %C4 THEN         NOT(%C2)
5:    A(I)=A(I)-2      NOT(%C2) AND %C4
   ELSE
6:    A(I)=A(I)+1      NOT(%C2) AND NOT(%C4)
   FI
7: D(I)=A(I)+1         TRUE
END DO
```



Beispiel – Vektorisierung

- Die Variablen `I` und `K` sollen sonst nicht im Programm vorkommen.

```
K = 5
DO I = 100, 0, -2
   A(K) = B(I+3) + C(K+I)
   D(I) = E(I*2) - A(K)
   K = K + 3
END DO
```



DO-LOOP-Normalisierung

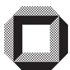
```
K = 5
DO I = 100, 0, -2
   A(K) = B(I+3) + C(K+I)
   D(I) = E(I*2) - A(K)
   K = K + 3
END DO
```

```
   K = 5
   DO I = 1, 51
* I = 100+(2*I-2)*(-1)
      A(K) = B(100+(2*I-2)*(-1)+3) + C(K+100+(2*I-2)*(-1))
      D(100+(2*I-2)*(-1)) = E((100+(2*I-2)*(-1))*2) - A(K)
      K = K + 3
   END DO
   I = -2
```

Toten Code eliminieren

```
K = 5
DO I = 1, 51
   A(K) = B(100+(2*I-2)*(-1)+3) + C(K+100+(2*I-2)*(-1))
   D(100+(2*I-2)*(-1)) = E((100+(2*I-2)*(-1))*2) - A(K)
   K = K + 3
END DO
I = -2
```

```
K = 5
DO I = 1, 51
   A(K) = B(100+(2*I-2)*(-1)+3) + C(K+100+(2*I-2)*(-1))
   D(100+(2*I-2)*(-1)) = E((100+(2*I-2)*(-1))*2) - A(K)
   K = K + 3
END DO
```



Operatorvereinfachung / Index Normalisierung

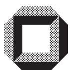
```
   K = 5
   DO I = 1, 51
      A(K) = B(100+(2*I-2)*(-1)+3) + C(K+100+(2*I-2)*(-1))
      D(100+(2*I-2)*(-1)) = E((100+(2*I-2)*(-1))*2) - A(K)
      K = K + 3
   END DO
```

```
   DO I = 1, 51
*  K = 2+3*I
      A(2+3*I) = B(100+(2*I-2)*(-1)+3) + C(2+3*I+100+(2*I-2)*(-1))
      D(100+(2*I-2)*(-1)) = E((100+(2*I-2)*(-1))*2) - A(2+3*I)
   END DO
```

Konstantenfaltung

```
DO I = 1, 51
   A(2+3*I) = B(100+(2*I-2)*(-1)+3) + C(2+3*I+100+(2*I-2)*(-1))
   D(100+(2*I-2)*(-1)) = E((100+(2*I-2)*(-1))*2) - A(2+3*I)
END DO
```

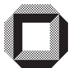
```
DO I = 1, 51
   A(2+3*I) = B(105-2*I) + C(104+I)
   D(102-2*I) = E(204-4*I) - A(2+3*I)
END DO
```



Schleifenteilung

```
DO I = 1, 51
   A(2+3*I) = B(105-2*I) + C(104+I)
   D(102-2*I) = E(204-4*I) - A(2+3*I)
END DO
```

```
DO I = 1, 51
   A(2+3*I) = B(105-2*I) + C(104+I)
END DO
DO I = 1, 51
   D(102-2*I) = E(204-4*I) - A(2+3*I)
END DO
```



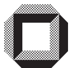
Vektor-Befehlserzeugung

```
DO I = 1, 51
   A(2+3*I) = B(105-2*I) + C(104+I)
END DO
DO I = 1, 51
   D(102-2*I) = E(204-4*I) - A(2+3*I)
END DO
```

```
A(5:155,3)  = B(103:3,-2) + C(105:155)
D(100:0,-2) = E(200:0,-4) - A(5:155,3)
```

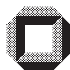
Parallelisierung

- Parallele Schleifen – implizite Synchronisation
- Parallele Schleifen – explizite Synchronisation
- SMP-Parallelisierung
- DSM-Parallelisierung



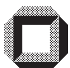
Symmetrischer Multiprozessor (SMP)

- Bei einem SMP wird zweckmäßigerweise die Parallelisierung durch **funktionale Partitionierung** erreicht.
- Die Techniken, die zur Vektorisierung eingesetzt werden, sind auch in diesem Fall anwendbar.
- Dabei arbeitet eine kleine Menge von Einheiten gemeinsam an den Daten. Motto: Was vorher ein Fließand war, ist jetzt ein Faden.
- Auf eine explizite Synchronisation wird sogar verzichtet, wenn Synchronität per Systemtakt sichergestellt werden kann.
- Im Fall von echten (Daten-)Abhängigkeiten wird bei SMP Synchronisation eingesetzt. Die eigentlichen Daten werden nicht kopiert, sondern **uniform zugegriffen**, egal, von welchen Prozessor sie berechnet wurden.



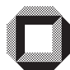
Distributed-shared-memory-System (DSM)

- Da bei DSM die Daten nicht uniform zugreifbar sind, sondern im Arbeitspeicher eines Prozessors immer nur ein lokale Kopie liegt, muß bei Abhängigkeiten das betroffene Datum transferiert werden.
- Beobachtung: Numerische Programme greifen auf die Daten **regelmäßig** und mit hoher **räumlicher Lokalität** zu.
- Idee:
  - Wir betrachten einzelne kritische Porgrammstellen: Schleifen
  - Auf allen Einheiten läuft das gleiche Programm ab
  - Es werden nur Teile der Daten auf den Einheiten abgelegt
  - Wenn Daten außerhalb diese Teils benötigt werden, wird kommuniziert
  - Bei geeigneter Wahl der Zerteilung ist der Kommunikationsaufwand gering.
- Achtung: Bei Standard HPF muß die Daten-Verteilung vom Benutzer spezifiziert werden. Dies erfordert eingehende Analyse des Programms von Hand oder mit Visulisierungswerkzeugen.



Parallelisieren für DSM

- Hauptidee ist **Kacheln**.
- Probleme: Welche Felder sind wo zu kacheln?
  Wie ist bei gegebener Kommunikationinfrastruktur zu verteilen?
- Lösungen:
  - Benutzung von statischen und dynamischen Informationen (Profile), um kritische Bereiche zu finden.
  - Benutze Datenflußinformationen, um Abhängigkeiten zu finden. Wähle Kachelung so, dass die Abhängigkeit (Abhängigkeitsvektor) so selten wie möglich aus der Kachel führt
  - Transformiere die Schleife und Datenanordnung so, dass Zugriffsmuster lokaler werden (Schleifen vertauschen, umkehren, neigen, etc.).
  - Vergebe Prioritäten für Daten (häufig verwendete Datenbereiche bilden lokale Gruppe) und repliziere ggf. entferntere Daten.

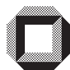


Forge xHPF

- Forge xHPF ist einer der wenigen veröffentlichten parallelisierenden Übersetzer.
- xHPF ist ein Vor-Übersetzer für Fortran 77, 90 und HPF mit den Merkmalen:
  - Automatische Parallelisierung von DO-Schleifen
  - Partitionierung der Daten (Reihungen)
  - Bewertung relevanter Stellen mit statischer Analyse oder Profilinformationen.

Literatur

- Hans Zima, “Supercompilers for Parallel and Vector Computers”, Frontier Series, ACM Press, 1991
- M. J. Wolfe, “Optimizing Supercompilers for Supercomputers”, Research Monographs in Parallel and distributed Computing, MIT Press, Cambridge
- Für Hardware: J. Hennessy and D. Patterson, “Computer Architecture A Quantitative Approach”, Morgan-Kaufmann, SanMateo, 1995



Zusammenfassung

- Techniken der Vektorisierung und Parallelisierung sind vor allem für wissenschaftliches Rechnen interessant.
- Durch die IA-64 sind viele Techniken nun zum Pflichtteil eines „normalen“ Übersetzers geworden.
- In der Algorithmen-Theorie werden BSP und LogP als von konkreter Hardware abstrahierte Modelle verwendet (hier nicht besprochen).
- Probleme gibt es bei automatischer Verteilung von Daten; manuelle Techniken verkomplizieren die Entwicklung um Faktoren.
- Amdahls Gesetz gibt eine Schranke an, die der beste Prarllelisierende Übersetzer bzw. Parallelrechner nicht unterbieten kann. Mit einem parallel entworfenen Algorithmus kann das jedoch leicht gelingen.