3. Parallele Programmierung

Parallele Programme orientieren sich stark am verwendeten parallelen Rechnersysteme (Hardware + Systemsoftware)

$\Rightarrow$ parallele Versionen eines sequentiellen Algorithmus
können stark differieren

Um

- Programme portabel zu halten,
- moderne Prinzipien der Programmentwicklung zu ermöglichen,

werden Parallelrechner in Klassen eingeteilt

Übliche (oftmals zu grobe) Sichtweise

- Rechner mit geteiltem Adressraum
- Rechner mit verteiltem Adressraum

Sichtweisen führen zu unterschiedlichen Programmen

Programme sollten portabel sein zwischen
Rechensystemen einer Klasse

Klassifizierung von Rechensystemen durch Definition von
abstrakten Modellen

Entwurf von Programmen auf Modellebene

Auf Modellebene

- können Programmstrukturen entworfen werden
- Programme funktional analysiert werden
- Programmlaufzeiten abgeschätzt werden

Aber Modell muss auf realem Rechner realisiert werden

I. d. R. Realisierung der im Modell definierten Schnittstellen und des Ablaufs durch Systemsoftware (Laufzeitbliotheken)

- Thread Bibliotheken realisieren geteilten Adressraum
- Message Passing Bibliotheken realisieren verteilten Adressraum

In diesem Kapitel behandelt

3.1 Parallele Programmiermodelle

3.2 Message Passing Programmierung mit MPI
(separate Folien auf der Web-Seite zugreifbar)

3.3 Programmierung mit gemeinsamen Variablen (Pthreads)

3.4 Laufzeitanalyse paralleler Programme

3.1 Parallele Programmiermodelle

Modelle für Rechensysteme werden auf unterschiedlichen Ebenen entworfen

- Maschinenmodelle
  - niedrigste Abstraktionsstufe, Hardware + BS
- Architekturmodell
  - abstraktere Sichtweise wie in Kap. 2
- Berechnungsmodell
  - Erweiterung des Architekturmodells durch parallele Operationen + zugehörige Kosten
  - übliches Modell für sequentielle Rechner Random Access Machine (RAM)
  - davon abgeleitet Parallel Random Access Machine (PRAM)
- Programmiermodell
  - weitere Abstraktionsstufe durch Beschreibung aus Sicht einer Programmierumgebung

Kriterien zur Spezifikation paralleler Programmiermodelle

- Parallelitätsebenen (Instruktionsebene, Anweisungsebene, Prozedurebene, parallele Schleifen)

- implizite oder explizite Spezifikation von Parallelität
- Form der Spezifikation von Parallelität
  (unabhängige Tasks während der Laufzeit generiert,
  Prozesse zum Programmstart generiert)
- Abarbeitung paralleler Einheiten
  (synchron - asynchron, SIMD - SPMD)
- Kommunikation zwischen parallelen Einheiten
  (Nachrichtenaustausch - gemeinsame Variablen)
- Möglichkeiten der Synchronisation

In jeder parallelen Programmierumgebung sind diese Kriterien eindeutig festgelegt

Paralleles Programm spezifiziert, welche Berechnungen (potentiell) parallel ausführbar sind

Berechnungen sind je nach Programmiermodell:

- einzelne Instruktionen
- arithmetische/logische Anweisungen
- Prozeduren
- oder parallele Schleifen

In allen Ansätzen werden unabhängige Module oder Tasks spezifiziert, die parallel abgearbeitet werden können

Definition der Tasks

- explizit vom Programmierer oder
- implizit durch das Laufzeitsystem

Abstrakte Sicht der Parallelisierung

Unterschiedliche Programmiermodelle weisen
unterschiedliche Charakteristika auf,
d.h. stellen Programmierer unterschiedliche Mechanismen zur Verfügung
Jeder Realisierung eines parallelen Programms liegt

- Parallelisierung unter Berücksichtigung der Daten- und Kontrollabhängigkeiten zu Grunde

Vorgehen bei der Parallelisierung eines sequentiellen Algorithmus

Ziel: paralleler Algorithmus mit

- identischen Resultaten
- aber kürzere Ausführungszeit bei Verwendung mehrerer Prozessoren

u. U. Entwicklung spezieller Algorithmen, die sich gut parallelisieren lassen (später mehr dazu)

Schritte der Parallelisierung

- Zerlegung der Berechnung in parallel ausführbare Tasks
  (= kleinste Einheiten der Parallelität die genutzt werden soll)
- Bestimmung der Abhängigkeiten zwischen Tasks
- Zuweisung der Tasks an Prozesse
  (= abstrakter Begriff für Kontrollfluss, der von realem Prozessor ausführbar ist, Zuordnung wird als Scheduling bezeichnet)
- Abbildung der Prozesse auf Prozessoren
  (im einfachsten Fall ein Prozess pro Prozessor, aber auch mehrere Prozesse pro Prozessor sind möglich)

Allgemeines Schema kann zu sehr unterschiedlichen Realisierungen führen

Beispiele:

- dynamische Threadgenerierung - statische Prozessgenerierung
- statisches Scheduling - dynamisches Scheduling
- Kommunikation über Nachrichten - Kommunikation über gemeinesame Variablen

Ebenen der Parallelität

Parallelität kann auf unterschiedlichen Ebenen auftreten

(von unterschiedlicher Granularität sein)

- **Feinkörnige Granularität**:

  Parallelität auf Instruktions- oder Anweisungsebene

- **Mittlere Granularität**:

  Parallelität auf Schleifenebene

- **Grobkörnige Granularität**:

  Parallelität auf Prozedurebene

Granularität bestimmt Einsatzmöglichkeit unterschiedlicher Rechnertypen und Scheduling-Algorithmen

Parallelität auf Instruktionsebene

Gleichzeitige Abarbeitung der Instruktionen I_1 und I_2

erfordert Unabhängigkeit der Instruktionen

Mögliche Datenabhängigkeiten zwischen I_1 und I_2

Fluss-Abhängigkeit: I_1 berechnet Ergebnis in einem Register, welches von I_2 als Operand verwendet wird

Anti-Abhängigkeit: I_1 verwendet ein Register als Operand, welches von I_2 verwendet wird, um ein Ergebnis abzulegen

Ausgabe-Abhängigkeit: I_1 und I_2 verwenden das gleiche Register zur Ergebnisausgabe

Beispiele

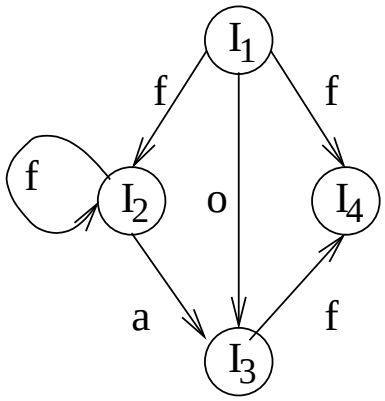
Fluss-Abh.	Anti-Abh.	Ausgabe-Abh.
$I_1 : R_1 \leftarrow R_2 + R_3$	$I_1 : R_1 \leftarrow R_2 + R_3$	$I_1 : R_1 \leftarrow R_2 + R_3$
$I_2 : R_5 \leftarrow R_1 + R_4$	$I_2 : R_2 \leftarrow R_4 + R_5$	$I_2 : R_1 \leftarrow R_4 + R_5$

Darstellung von Abhängigkeiten durch Abhängigkeitsgraph

- mit einem Knoten pro Instruktion und
- gerichteter Kante zwischen Knoten, falls Abhängigkeit existiert

Beispiel:

$$
\begin{aligned}
I_1 &: \quad R_1 \leftarrow A \\
I_2 &: \quad R_2 \leftarrow R_2 + R_1 \\
I_3 &: \quad R_1 \leftarrow R_3 \\
I_4 &: \quad B \leftarrow R_1
\end{aligned}
$$



Abhängigkeitsgraph definiert Halbordnung
zwischen Instruktionen

Abhängigkeitsgraph kann automatisch generiert werden

- teilweise statisch vom Compiler
- sonst dynamisch zur Laufzeit

I_i muss vor I_j ausgeführt werden $\Leftrightarrow$

es existiert eine gerichtete Verbindung von I_i nach I_j

im Abhängigkeitsgraph

I_i kann parallel zu I_j ausgeführt werden $\Leftrightarrow$

es existiert keine gerichtete Verbding von I_i nach I_j

oder von I_j nach I_i

Potentielle Parallelität kann (automatisch) erkannt und genutzt werden (durch Hardware oder Compiler)

Datenparallelität

Oft werden identische Operationen auf unterschiedliche Elemente einer Datenstruktur angewendet

Einfaches Beispiel: Vektor-Matrix-Berechnungen

Falls Operationen unabhängig sind $\Rightarrow$

Daten können auf verschiedene Prozessoren verteilt und parallel manipuliert werden

$\Rightarrow$ datenparallele Programmiersprachen (SIMD Modell)

Beispiel FORTRAN 90:

Vektoranweisung: $a(1:n) = b(0:n-1) + c(1:n)$

ist äquivalent zu $for\ (i = 1:n)$

$$a(i) = b(i-1) + c(i)$$

$$endfor$$

Semantik der Vektoranweisung:

- erst auf rechte Seite zugreifen
- dann Berechnungen durchführen
- schließlich Zuweisung der Ergebnisse

Datenparallelität im MIMD Modell durch

SPMD-Konzept (single program multiple data)

Identisches Programm wird auf unterschiedlichen

Prozessoren asynchron ausgeführt

Jede Inkarnation des Programms bearbeitet einen Teil der

Daten

$\Rightarrow$ Datenverteilung hat großen Einfluss auf Laufzeitverhalten

Die meisten MIMD Programme sind nach dem

SPMD Paradigma realisiert

Datenverteilung für Felder

Felder sind Basisdatenstruktur für Vektoren und Matrizen

$\Rightarrow$ Basis vieler Anwendungen im technisch/wissenschaftlichen

Bereich

Effiziente Realisierung der Algorithmen hängt von Verteilung der Daten im Speicher ab

Hier betrachtetes Modell: Prozessoren $P = \{P_1, \ldots, P_p\}$ mit lokalem Speicher (in ähnlicher Form auch für andere Konfigurationen verwendbar)

Verteilung eindimensionaler Felder (Vektoren):

Verteilung eines Feldes $v = (v_1, \ldots v_n)$

blockweise Verteilung:

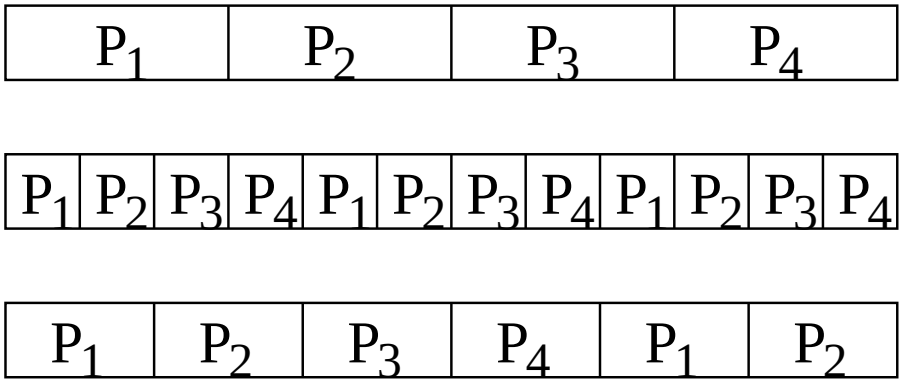
Prozessoren erhalten benachbarte Feldelemente, so dass sich die Zahl der Elemente pro Prozessor sich um maximal 1 unterscheidet

zyklische Verteilung:

Feldelemente werden reihum an die Prozessoren verteilt

blockzyklische Verteilung:

Kombination der vorherigen, Blöcke vorgegebener Größe werden zyklisch verteilt

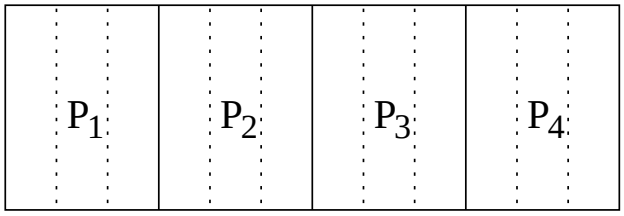


Verteilung zweidimensionaler Felder (Matrizen):

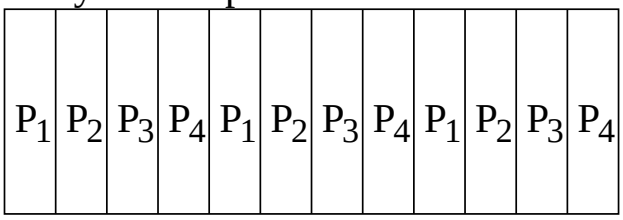
- Verteilung der Spalten
- Verteilung der Zeilen
- Schachbrettverteilung
  - ▷ jeweils blockweise, zyklisch oder blockzyklisch

Beispiele:

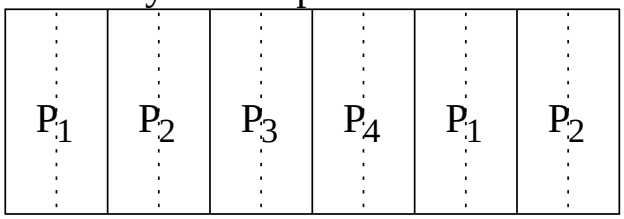
blockweise Spalten



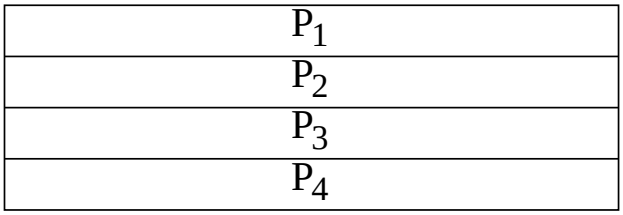
zyklisch Spalten



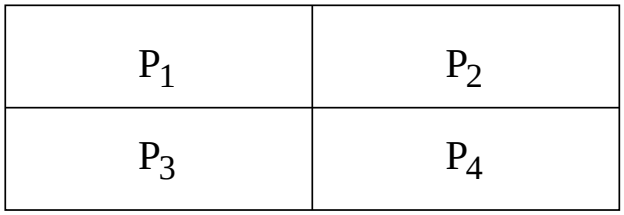
blockzyklisch Spalten



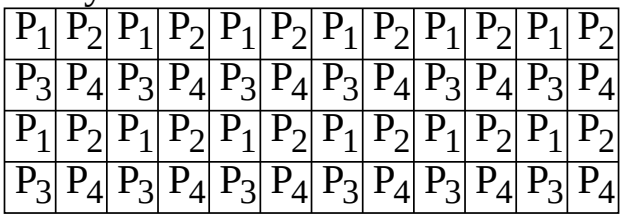
Zeilen



blockweise Schachbrett



zyklisch Schachbrett



Verfahren können auf beliebig dimensionale Strukturen erweitert werden

Zuteilung mehrdimensionaler Spalten, Zeilen oder Blöcke zu einem Prozessor

Optimale Anordnung hängt vom Algorithmus ab
(später mehr)

Parallele Schleifen

Schleifen sind Basiskonstrukte aller Programmiersprachen

übliche Ausführung sequentiell

(i-ter Durchlauf beginnt nachdem $i-1$-ter abgeschlossen)

Falls Schleifenrümpfe unabhängig sind,

können sie parallel ausgeführt werden

Unabhängigkeit

- wird vom Compiler oder Laufzeitsystem erkannt oder
- wird explizit vom Programmierer vorgegeben

Explizite parallele Schleifenkonstrukte (FORTRAN 90):

forall-Schleife $forall\ (i = 1 : n)$

$$a(i) = a(i-1) + a(i+1)$$

$endforall$

entspricht der Vektoranweisung

$a(1:n) = a(0:n-1) + a(2:n+1)$

$\Rightarrow$ parallele Abarbeitung aller Schleifenrümpfe

sequentielle Abarbeitung innerhalb eines Schleifenrumpfes

Alternative dopar - Schleifen

$\Rightarrow$ parallele Abarbeitung aller Schleifenrümpfe

Änderung der Variablenwerte erst nach Abarbeitung des gesamten Schleifenrumpfes

Unterschiede am Beispiel:

$$
\begin{aligned}
&\mathit{for}\ /\ \mathit{forall}\ /\ \mathit{dopar}\ (i = 1:4)\\
&\quad a(i) = a(i) + 1\\
&\quad b(i) = a(i-1) + a(i+1)\\
&\mathit{endfor}
\end{aligned}
$$

liefern für Startwerte $1, \ldots, 6$ in $a(0), \ldots, a(5)$:

	$b(1)$	$b(2)$	$b(3)$	$b(4)$
for	4	7	9	11
forall	5	8	10	11
dopar	4	6	8	0

Funktionsparallelität

unabhängige Funktionseinheiten werden als Tasks dargestellt und die Abhängigkeit zwischen Tasks im Taskgraphen modelliert (siehe Abhängigkeitsgraph auf Instruktionsebene)

$\Rightarrow$ unabhängige Tasks können parallel abgearbeitet werden

Tasks werden Ausführungszeiten zugeordnet

Zuordnung von Tasks zu Prozessoren über

statisches oder dynamisches Scheduling

Spezifikation von Tasks in Programmen:

- explizit durch Definition von Tasks/Prozessen/Threads
- explizit durch Sprachkonstrukte
  (z.B. parallele Prozeduren in High Performance Fortran)

Explizite-implizite Darstellung von Parallelität

Wieviel Aufwand muss der Programmierer für die Parallelisierung treiben?

Unterschiedliche Abstufungen existieren

- **Implizite Parallelität**
  sequentielles Programm wird vom Compiler parallelisiert
- **Funktionale Programmiersprachen**
  parallele Berechnung von Funktionsargumenten
- **Explizite Parallelität mit impliziter Zerlegung**
  explizite Darstellung der Parallelität im Programm, aber keine Aufteilung in Tasks
- **Explizite Zerlegung**
  explizite Beschreibung von Tasks,
  aber keine explizite Zuordnung von Tasks zu Prozessoren
- **Explizite Zuordnung an Prozessoren**
  explizite Zuweisung von Tasks an Prozessoren, aber gemeinsamer Datenraum
- **Explizite Kommunikation + Synchronisation**
  explizite Realisierung der gesamten Parallelisierung

Wir betrachten meistens Vertreter der letzten Klasse

$\Rightarrow$ komplexeste Form der Parallelisierung

$\Rightarrow$ aber bisher auch noch effizientestes Vorgehen

Strukturierung paralleler Programme

Basisoperation bei expliziter paralleler Programmierung

Generierung von Tasks-Prozessen

- statische Generierung zum Programmstart
- dynamische Generierung während der Laufzeit

Übliche Operationen zur Taskgenerierung

- Fork-Join
- Spawn-Exit
- Parbegin-Parend

Unterschiedliche Organisationstrukturen für
parallele Tasks existieren

- **SPMD**
  identisches Programm existiert mehrfach
- **Master-Slave**
  Master-Prozess erzeugt Slaves und weist ihnen Arbeit zu
- **Pipelining**
  jeder Prozess liest Eingabe und erzeugt daraus Ausgabe, Ausgabe des i-ten Prozesses ist Eingabe des $i+1$-ten

Auswahl der Struktur je nach Aufgabenstellung

Kommunikation über gemeinsame Variablen

- Existenz eines gemeinsamen Speichers wird angenommen (virtuell oder real vorhanden)
- Unterscheidung zwischen privaten und gemeinsamen Variablen jedes Prozesses
- Koordinierter Zugriff auf gemeinsame Variablen erfordert Synchronisation
- Zugriffe werden sequentialisiert durch lock/unlock-Operationen, Semaphore oder Monitore
- Resultat darf nicht von der gewählten Sequentialisierung abhängen (deterministisches Verhalten)

Kommunikation über Nachrichtenaustausch

- Prozesse berücksichtigen explizit Kommunikationsoperationen ($\Rightarrow$ Kenntnis über Gesamt-/Teilstruktur des parallelen Programms in jedem Prozess vorhanden)
- Kommunikationsoperationen werden von Laufzeitbibliotheken zur Verfügung gestellt
  - Kommunikation beruht meist auf wenigen Basisoperationen
  - Bibliotheksroutinen abstrahieren von konkreter Architektur und Verteilung der Prozesse

Basiskommunikationsoperationen

Einzeltransfer:

Prozessor P_i schickt eine Nachricht an Prozessor P_j

Sender führt Senderoutine aus:

- ablegen der Nachricht im Sendepuffer
- Angabe der Empfangsadresse (bzw. Prozessornummer)
- je nach Semantik blockierendes oder nicht-blockierendes Senden

Empfänger führt Empfangsroutine aus

- auslesen der Nachricht aus Empfangspuffer
- evtl. danach verschicken einer Bestätigung

Aus Einzeltransfers können alle Kommunikationsoperationen prinzipiell zusammengesetzt werden

Einzel-Broadcast:

Ein Prozessor P_i schickt eine Nachricht an alle anderen Prozessoren

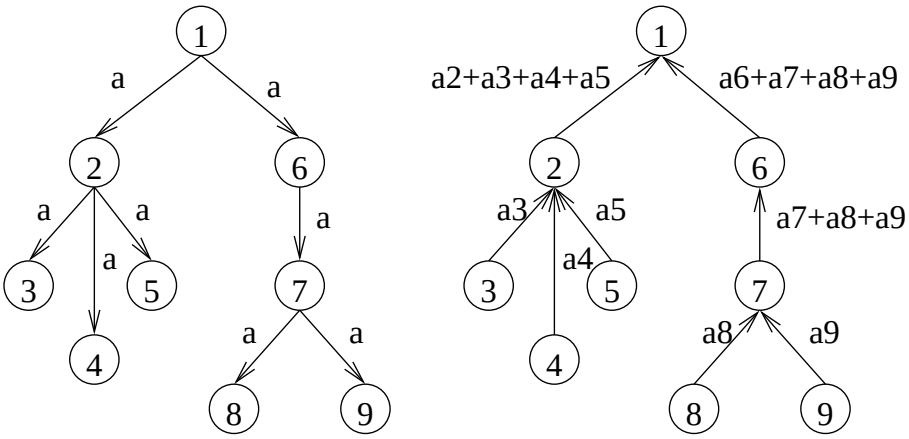
Einzel-Akkumulation:

Ein Prozessor P_i empfängt von jedem anderen Prozessor P_j eine Nachricht eines festen Typs

Nachrichten werden auf dem Weg zum Empfänger verschmolzen (Reduktion)

Einzel-Broadcast und Einzel-Akkumulation können mit Hilfe von Spannbäumen beschrieben werden

Beispiele (Reduktionsoperation durch + bezeichnet)



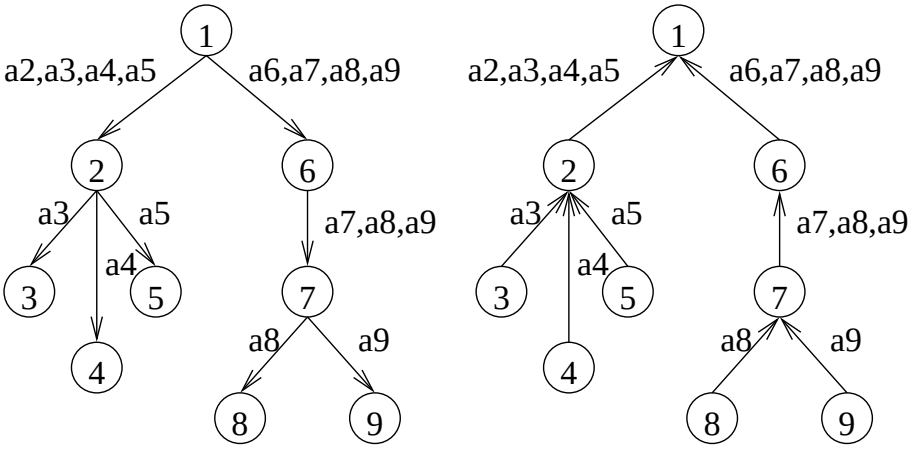
Gather:

Ein Prozessor P_i erhält ein individuelles Paket von allen anderen Prozessoren (Pakete können nicht verschmolzen werden)

Scatter:

Ein Prozessor P_i verschickt eine individuelle Nachricht an alle anderen Prozessoren

Beispiele: (ai,aj bedeutet, dass beide Nachrichten individuell übertragen werden)



Multi-Broadcast:

Alle Prozessoren führen gleichzeitig einen Broadcast aus

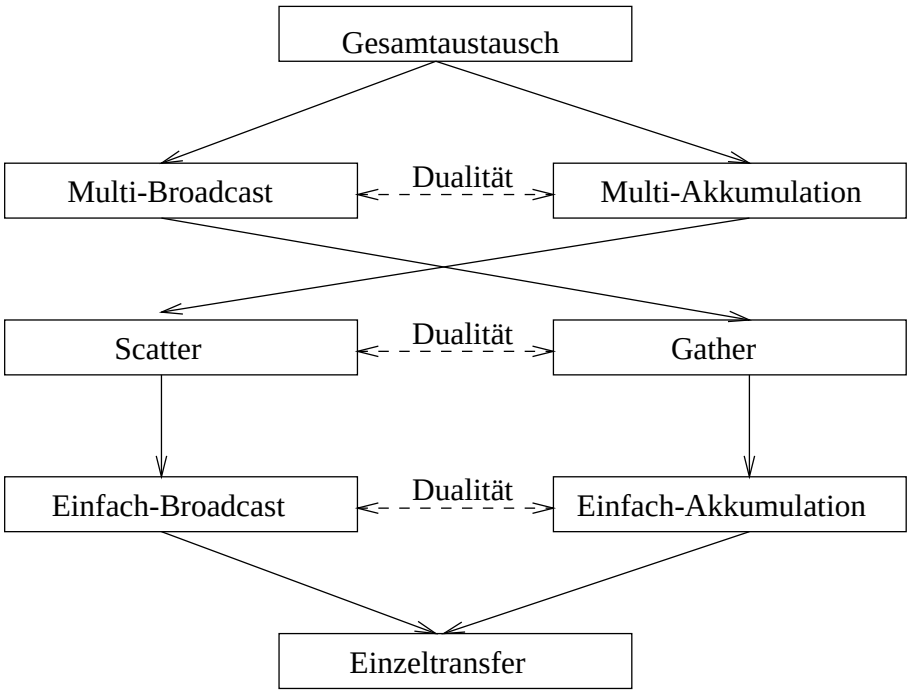
Multi-Akkumulation:

Alle Prozessoren führen gleichzeitig eine Akkumulation aus

Gesamtaustausch:

Jeder Prozessor sendet jedem anderen je eine individuelle Nachricht

Hierarchie der Kommunikationsprobleme



- Algorithmus für eine Operation, die höher in der Hierarchie ist, kann auch für Operation verwendet werden, die niedriger in der Hierarchie liegt
- Algorithmen für duale Probleme können leicht ineinander überführt werden

Parallele Matrix-Vektor-Multiplikation

Basisoperationen vieler technischer Anwendungen

Parallele Realisierung mit Hilfe der vorgestellten Basiskommunikationsoperationen

Notationen:

- $A \in \mathbb{R}^{n \times m}$ Matrix
- $b \in \mathbb{R}^{m}$ Vektor
- $c = A \cdot b \in \mathbb{R}^{n}$

Indizes laufen von 1 bis n bzw. m

$$c(i) = \sum_{j=1}^{m} A(i,j) \cdot b(j)$$

Mögliche Realisierungen

(C-Notation $\Rightarrow$ Indizes laufen von 0 bis $n-1$ bzw. $m-1$)

Version 1

```
for (i = 0; i < n; i++)
  c[i] = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    c[i] = c[i] + A[i][j] * b[j];
```

Version 2

```
for (i = 0; i < n; i++)
  c[i] = 0;
for (j = 0; m; j++)
  for (i = 0; i < n; i++)
    c[i] = c[i] + A[i][j] * b[j];
```

Version 1 berechnet das Vektor-Matrix-Produkt

durch n innere Produkte

Sei $A_{i\bullet}$ i-te Zeile von A, dann ist

$$A \cdot b = \begin{pmatrix} (A_{1\bullet}, b) \\ \vdots \\ \vdots \\ (A_{n\bullet}, b) \end{pmatrix}$$

mit $(x, y) = \sum_{j=1}^{m} x(j) \cdot y(j)$ (inneres Produkt)

Version 2 berechnet das Vektor-Matrix-Produkt

als Linearkombination der Spalten

Sei $A_{\bullet i}$ i-te Spalte von A, dann ist

$$A \cdot b = \sum_{j=1}^{m} b(j) \cdot A_{\bullet j}$$

Sequentielle Realisierung beider Varianten sind ähnlich

- da Schleifen auf Grund der Datenunabhängigkeit vertauscht werden dürfen
- unterschiedliche Laufzeiten können bei großen Matrizen durch unterschiedliche Zugriffe auf Hintergrundspeicher auftreten

Parallele Implementierungen unterscheiden sich deutlich

(sei p Anzahl Prozessoren)

- bei Version 1 werden die n inneren Produkte so auf die Prozessoren verteilt, dass jeder Prozessor ungefähr n/p innere Produkt berechnet
- bei Version 2 werden die m Linearkombinationen so auf die Prozessoren verteilt, dass jeder Prozessor ungefähr m/p Linearkombinationen berechnet

Detaillierte Betrachtung der Algorithmen:

Annahmen:

- Matrix-Vektor-Multiplikation ist Teil eines iterativen Prozesses (d.h. mehrere Iterationen)
- A ist eine quadratische Matrix es gilt $n = m$ und es wird das Iterationsschema $b^k = A \cdot b^{k-1}$ berechnet
- Daten werden am Anfang einmal verteilt (ausgehend von einem Master-Prozess im System)
- Daten müssen nach jeder Iteration so verteilt werden, dass nächste Iteration beginnen kann
- am Ende der Iteration wird der Ergebnisvektor zum Master übertragen und von dort ausgegeben

Version 1

Prozessor erhält

- Zeilen der Matrix deren innere Produkte er berechnen soll
  (Scatter Operation mit Nachrichtengröße $n \cdot m/p$)
- Kopie des Vektors b
  (Einfacher Broadcast mit Nachrichtengröße m)

Annahme: blockweise Zuteilung der Zeilen zu Prozessoren
(Prozessor i berechnet Zeilen $n/p \cdot (i-1)+1, \ldots, n/p \cdot i$)

Lokale Datenstrukturen und Konstante
(Annahme n/p ganzzahlig):

- $A\_local[i][j]$ der Größe $n/p \times m$
- $c\_local[i]$ der Größe n/p
- $n\_local = n/p$

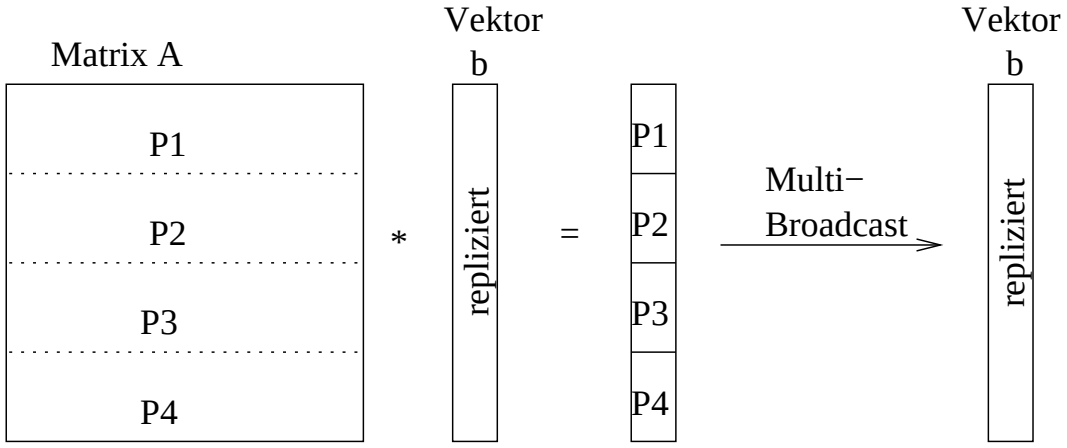
Lokaler Algorithmus auf Prozessor p

for $(i = 0;\ i < n\_local;\ i++)$
  $c\_local[i] = 0;$
for $(i = 0;\ i < n\_local;\ i++)$
  for $(j = 0;\ j < m;\ j++)$
    $c\_local[i] = c\_local[i] + A\_local[i][j] * b[j];$
multi_broadcast$(c_local,\ n_local,\ b)$;

Abarbeitung des Programms im SPMD Stil

Kommunikationsoperation multi_broadcast

- führt Synchronisation herbei
- sorgt dafür, dass berechneter Vektor c anschließend als Iterationsvektor b verfügbar ist



Nach Ende der Iterationen Gather-Operation des Masters zum Erhalt des Resultatvektors (Nachrichtengröße n/p)

Realisierung auf Rechner mit geteiltem Adressraum:

- Synchronisationspunkt statt Multi-Broadcast
- ansonsten unveränderter Algorithmus

Version 2

Prozessor erhält

- Spalten der Matrix mit denen Linearkombinationen berechnet werden
  (Scatter Operation mit Nachrichtengröße $n \cdot m/p$)
- Teil des Vektors b (Scatter mit Nachrichtengröße m/p)

Annahme: blockweise Zuteilung der Spalten zu Prozessoren
(Prozessor i berechnet Zeilen $m/p \cdot (i-1)+1, \ldots, m/p \cdot i$)
Lokale Datenstrukturen und Konstante
(Annahme n/p ganzzahlig):

- $A\_local[i][j]$ der Größe $n \times m/p$
- $b\_local[i]$ der Größe m/p
- $n\_local = m/p$

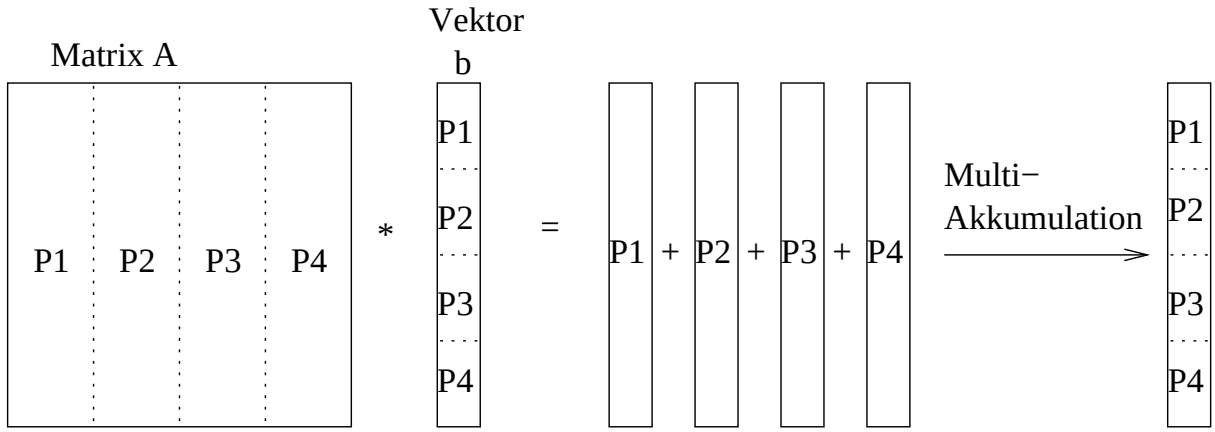
Lokaler Algorithmus auf Prozessor p

for $(i = 0;\ i < n;\ i++)$
  $c[i] = 0;$
for $(j = 0;\ j < m\_local;\ j++)$
  for $(i = 0;\ i < n;\ i++)$
    $c[i] = c[i] + A\_local[i][j] * b\_local[j];$
single_accumulation$(c,\ m_local,\ b_local,\ ADD)$;

Abarbeitung des Programms im SPMD Stil

Kommunikationsoperation single_accumulation

- führt Synchronisation herbei
- sorgt dafür, dass berechneter Teilvektor von c, der zu Spalten gehört die auf dem Prozessor berechnet werden, anschließend als Iterationsvektor b_local verfügbar ist



Nach Ende der Iterationen einfache Akkumulation des Masters zum Erhalt des Resultatvektors (Nachrichtengröße n)

Realisierung auf Rechner mit geteiltem Adressraum nicht empfehlenswert, da Prozessoren gleichzeitig auf c schreibend zugreifen (Zugriffskonflikt)

3.2 Message Passing Programmierung

Abstraktion eines Parallelrechners mit verteiltem Speicher

i.d.R. ohne explizite Sicht auf die Topologie

(⇒größere Portabilität der Programme)

Struktur eines message-passing Programms:

- Prozesse mit lokalen Daten
- Programmstrukturen
  - MPMD (multiple program, multiple data)
  - SPMD (single program, multiple data)
- Nachrichtenaustausch zwischen Prozessen

⇒ expliziter Aufruf von Kommunikationsroutinen

(für Sendung und Empfang von Daten!)

Mögliche Realisierungen

- direkte Programmierung der Kommunikationsoperationen
- Verwendung von Standardbibliotheken
  - Kommunikationsroutinen für unterschiedliche Architekturen (⇒ portable Programme)
  - Routinen für Basiskommunikationsoperationen (Punkt-zu-Punkt, Broadcast)

Hier behandelt MPI

Folien separat verfügbar siehe Web-Seite

3.3 Programmierung mit gemeinsamen Variablen

Programmierung mit gemeinsamen Variablen

Prozessmodell:

- unabhängige Programmstücke greifen auf gemeinsamen Adressraum zu
- Prozesse (oder Threads) können Speicherzellen im gemeinsamen Adressraum lesen oder schreiben
- Kommunikation durch Lesen und Schreiben von Variablen im gemeinsamen Adressraum (keine direkten Kommunikationsoperationen notwendig)
- Synchronisationsoperationen beim Zugriff auf gemeinsamen Adressraum sind notwendig

Erste Realisierungen dieses Prozessmodells im Bereich der Mehrbenutzerbetriebssysteme

Ziel:

Natürliches Programmdesign für Systeme kommunizierender Komponenten (übliches Paradigma vieler Anwendungen)

$\Rightarrow$ Threadkonzept in JAVA

Wir betrachten Pthreads

- POSIX Standard für Unix Betriebssysteme
- frei verfügbar für viele Systeme (Linux, Windows)

Begriffsklärungen und Motivation

Heutige Betriebssysteme sind Multitasking-Systeme d.h.

- mehrere Prozesse können gleichzeitig aktiv sein
- CPU wird nach dem Time-Sharing Verfahren an Prozesse vergeben
- CPU-Scheduler steuert die Ausführung der Prozesse

Prozess

- ein Ausführung befindliches Programm
- mit den zugehörigen Daten
- und dem Kontext

Prozesswechsel erfordert Kontextwechsel

$\Rightarrow$ relativ hoher Aufwand

Vereinfachung durch Verwendung von Threads

- Threads beschreiben Kontrollflüsse in einem Prozess
- Threads teilen sich einen Adressraum
- Threads besitzen lokalen Datenstack und u.U. lokale Variablen

$\Rightarrow$ effiziente Erzeugung und Verwaltung aber
keine Abgrenzung zwischen Threads eines Prozesses

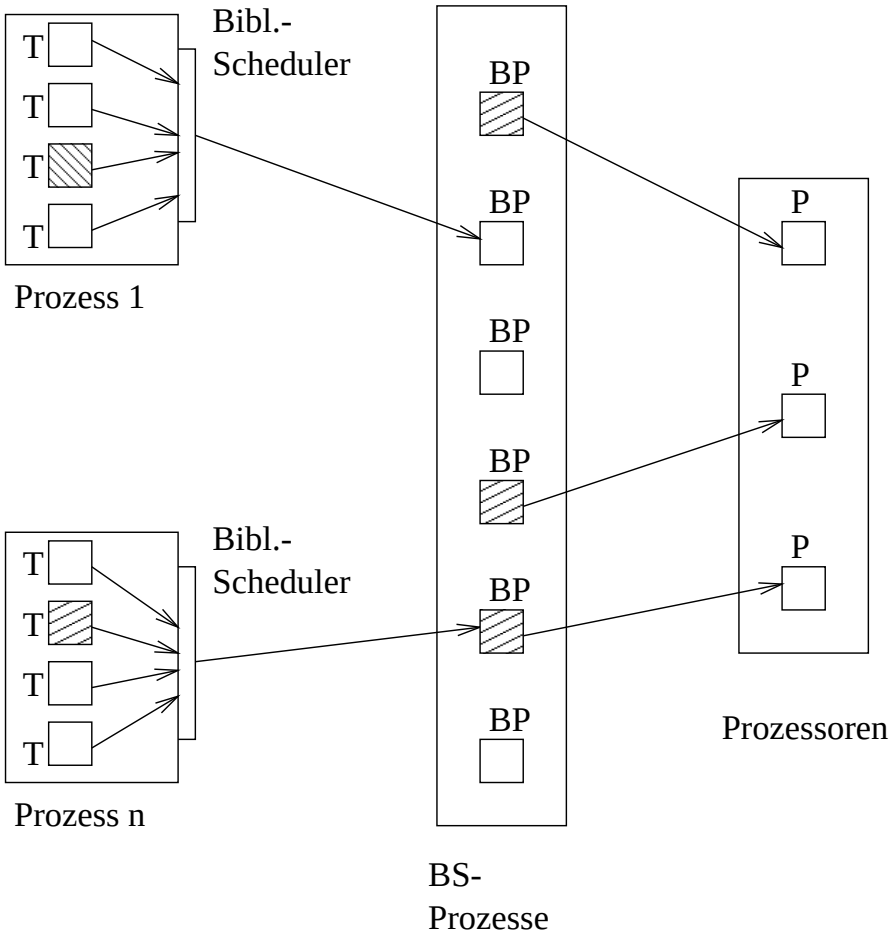
Realisierung von Threads

- auf Benutzerebene ohne Beteiligung des Betriebssystems
- auf Betriebssystemebene

Ausführungsmodelle für Threads

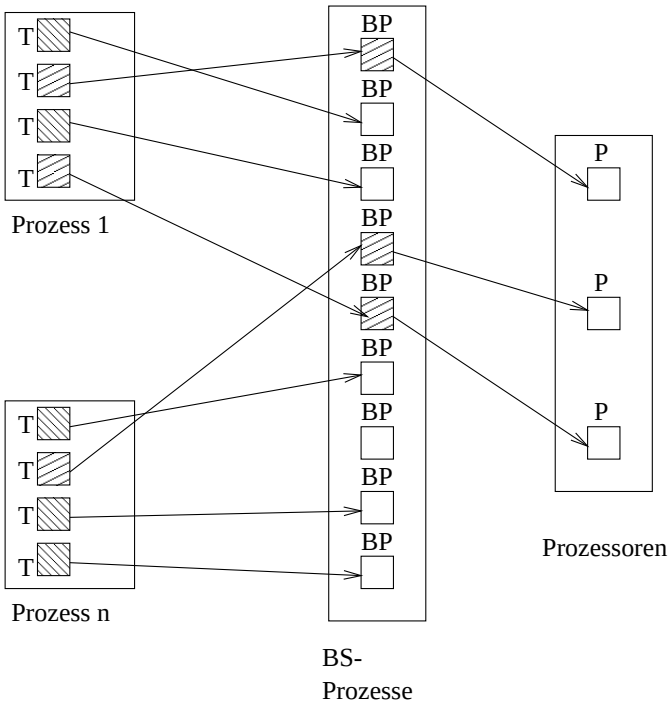
Falls BS Threadverwaltung nicht unterstützt

N:1 Abbildung

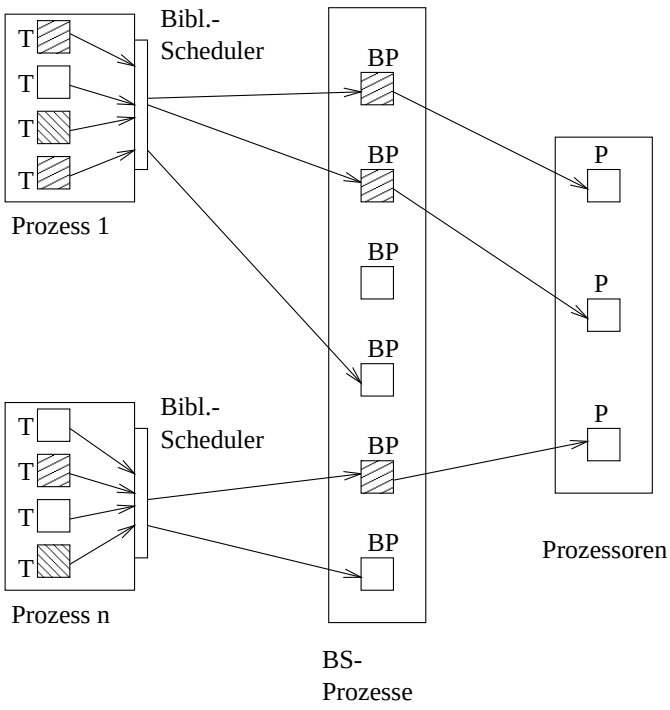


Falls BS Threadverwaltung unterstützt

1:1 Abbildung



N:M Abbildung



Programmiermodell für Pthreads

Wir nehmen an, dass N:M Abbildung gewählt wird
(Allgemeinste Form, die in Zukunft wohl von vielen BS unterstützt wird!)

Aufgabe des Benutzers:

- Zerlegung des Programms in Benutzer-Threads, die konkurrierend ausgeführt werden
- Sicherstellung der Synchronisation und Kommunikation zwischen Benutzer-Threads
- Bibliotheks-Scheduler bildet Benutzer-Threads auf BS-Threads ab
- BS-Scheduler bildet BS-Threads auf Prozessoren ab

Benutzer hat

- kaum Einflussmöglichkeiten auf Bibliotheks-Scheduler
- keine Einflussmöglichkeit auf BS-Scheduler

$\Rightarrow$ Mapping der Threads auf Prozessoren erfolgt automatisch

Vorteil: Vereinfachte Programmierung

Nachteil: Keine Einflussmöglichkeit durch Programmierer

Mögliche (und übliche) Modelle der Zusammenarbeit zwischen Threads:

Master-Slave-Modell:

- ein Master-Thread steuert das Programm
- Slave-Threads führen Berechnung durch
- Master-Thread startet Slaves, versorgt sie mit Daten und sammelt Ergebnisse

Worker-Modell:

- Programm besteht aus Menge von gleichberechtigter Worker-Threads
- Threads koordinieren sich zur Durchführung der Berechnungen
- jeder Worker-Thread kann bei Bedarf neue Worker erzeugen

Pipeline-Modell:

- Ein-/Ausgabebeziehung zwischen beteiligten Threads
- Thread i benötigt Daten von $i-1$ für seine Berechnungen
- üblicherweise nur für relativ kleine Zahl von Threads realisierbar

Pthreads Standard

Standard wird üblicherweise in C realisiert

Namenskonvention:

`pthread_[<object>]_<operation>`

- `object` bezeichnet Objekt, auf dem Operation ausgeführt wird
- `operation` die zugehörige Operation

Beispiele

- `pthread_mutex_init()` initialisiert eine Mutex Variable
- `pthread_create()` generiert einen neuen Thread (Objektbezeichnung thread wird weggelassen)

Datentypen

Datentyp	Bedeutung
pthread_t	Thread-Name
pthread_mutex	Mutex-Variable
pthread_cond_t	Bedingungsvariable
pthread_key_t	Zugriffsschlüssel
pthread_attr_t	Thread-Attribut
pthread_mutexattr_t	Mutex-Attribut
pthread_condattr_t	Bedingungsvariablen-Attr.
pthread_one_t	Kontrollkontext

Erzeugen und Verwalten von Threads

Nach dem Start eines Pthread-Programms ist
der Haupt-Thread aktiv, der die main()-Funktion
des Programms ausführt
Erzeugung von neuen Threads durch

```
int pthread_create   (pthread_t *thread,
                      const pthread_attr_t *attr,
                      void *(*start_routine) (void *),
                      void *arg)
```

- `pthread_t` liefert Verweis auf erzeugten Thread
- `pthread_attr_t` Verweis auf Attributobjekt für den neuen Thread
- `start_routine` Routine, dei der neue Thread nach seinem Start ausführt
- `arg` Argument der Startroutine
- Anzahl der erzeugbaren Threads in der Regel beschränkt (64 laut Standard)

Beendigung eines Threads durch Funktion
`void pthread_exit (void *valuep)`
`valuep` enthält den Rückgabewert des threads
Funktion wird immer implizit aufgerufen, wenn ein Thread seine Startroutine beendet

Koordination von Threads

Warten auf Beendigung eines anderen Threads

```
int pthread_join (pthread_t thread, void **valuep)
```

- `thread` bezeichnet Thread auf den gewartet wird
- `valuep` Rückgabewert des Threads auf den gewartet wurde

Falls mehrere Threads auf einen warten,

- ist nur einer erfolgreich
- alle anderen erhalten Fehlercode

Laufzeitsystem speichert Datenstruktur beendeter Threads, um eine spätere Join-Operation zu ermöglichen
Explizite Freigabe der Datenstruktur durch

- Aufruf von `pthread_join` oder
- Aufruf von `pthread_detach`
  (dann kein join mit diesem Thread mehr möglich!)

Beispiel Matrix-Multiplikation

```
#include <pthread.h>
typedef struct {
   int size, row, column ;
  double (*MA)[8], (*MB)[8], (*MC)[8] ;
} matrix_type_t ;
void thread_mult(matrix_type_t *work) {
  int i, row = work->row, column = work->column;
  work->MC[row][column] = 0.0 ;
  for (i=0; i < work->size; i++)
    work->MC[row][column] += work->MA[row][i] * work->MB[i][column] ;
}
int main() {
  int row, column, size = 8 ;
  double MA[8][8], MB[8][8], MC[8][8] ;
  matrix_type_t *work ;
  pthread_t thread[8][8] ;
  for (row=0; row<size; row++) {
    for (column=0; column<size; column++){
      work = (matrix_type_t *) malloc (sizeof (matrix_type_t)) ;
      work->size = size ; work->row = row ; work->column = column;
      work->MA = MA; work->MB = MB ; work->MC = MC ;
      pthread_create(&(thread[row][column]), NULL,
                     (void *) thread_mult, (void *) work) ;
  } }
  for (row=0; row<size; row++) {
    for (column=0; column<size; column++) {
      pthread_join (thread[row][column], NULL) ;
} } }
```

Behandlung zeitkritischer Abläufe

Problem durch gemeinsamen Adressraum der Threads

$\Rightarrow$ Zugriffskonflikte

Mutex-Variablen

Datentyp der Thread-Bibliothek zur Realisierung des wechselseitigen Auschlusses

Zustände einer Mutex Variable

- gesperrt (locked)
- ungesperrt (unlocked)

Zugriff auf einen geschützten Bereich erfordert:

Sperren der Mutex-Variable

- falls erfolgreich wird aufrufender Thread Eigentümer der Mutex-Variable
- falls nicht erfolgreich wird Thread suspendiert bis Mutex wieder frei

Nach Abarbeitung des kritischen Bereichs:

Freigeben der Mutex Variable

- ein evtl. auf die Mutex Variable wartender Thread bekommt anschließend Zugriff

Nachteile des Vorgehens:

- jeder Thread muss sicherstellen, dass erst Mutex gesetzt wird und dann auf den kritischen Bereich zugegriffen wird (ansonsten kein exklusiver Zugriff garantiert)
- jeder Thread muss sicherstellen, dass Mutex wieder freigegeben wird (ansonsten Deadlockgefahr)

Funktionen zur Behandlung von Mutex Variablen:

`int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr)`
initialisieren eines neuen Mutex

`int pthread_mutex_destroy (pthread_mutex_t *mutex)`
löschen einer Mutex Variable

`int pthread_mutex_lock (pthread_mutex_t *mutex)`
sperren der Mutex Variable

`int pthread_mutex_trylock (pthread_mutx_t *mutex)`
sperren der Mutex-Variable, falls diese "frei ist"
ansonsten Fehlermeldung an aufrufenden Thread, der aber nicht suspendiert wird

`int pthread_mutex_unlock (pthread_mutex_t *mutex)`
entsperren der Mutex Variable

Bedingungsvariablen

Datenstruktur, die es Threads erlaubt, auf beliebige Bedingungen zu warten (ohne zwischen durch aktiv zu werden und die Bedingung zu testen)

$\Rightarrow$ kein CPU-Zeitverbrauch während der Wartezeit

Bedingungsvariablen haben Typ `pthread_cond_t`

und ermöglichen durch Funktionen zum

Warten und Signalisieren

sehr allgemeine Synchronisation von Threads

Funktion zur Initialisierung

```
int pthread_cond_init (pthread_cond_t *cond,
                       const pthread_condattr_t *attr)
```

Bedinungsvariablen werden eindeutig mit Mutex-Variablen assoziiert

- alle auf eine Bedingungsvariable wartenden Threads müssen den selben Mutex benutzen
- verschiedene Bedingungsvariablen dürfen aber den selben Mutex benutzen

Funktion zum warten auf eine Bedingung

```
int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex)
```

Typische Verwendung

```
pthread_mutex_lock(&mutex) ;
while (!  Bedingung)
   pthread_cond_wait(&cond, &mutex) ;
pthread_mutex_unlock(&mutex) ;
```

Ablauf:

- falls Bedingung erfüllt, wird nicht gewartet und die Mutex-Variable wieder freigegeben
- falls Bedingung nicht erfüllt, wird die Mutex Variable implizit freigegeben und der Prozess bis zu einer Signalisierungsoperation suspendiert
- Mutex Variable ist notwendig, um sicherzustellen, dass Bedingung sich nach Abprüfen in der while Schleife nicht ändert

Zum Aufwecken von Prozessen, die auf eine Bedingungsvariable warten, existieren die Funktionen

```
pthread_cond_signal(pthread_cond_t *cond)
```

- Funktion weckt einen auf die Bedingungsvariable wartenden Prozess auf
- Auswahl nach Priorität und Schedulingstrategie
- falls kein Prozess wartete, kein Effekt

```
pthread_cond_broadcast (pthread_cond_t *cond)
```

- weckt alle auf die Bedingungsvariable wartenden Prozesse auf
- davon bekommt aber nur ein Prozess die Kontrolle über die Mutex Variable
- falls kein Prozess wartet, kein Effekt

Die Funktion

```
int pthread_cond_timewait
      (pthread_cond_t *cond, pthread_mutex_t *mutex,
       const struct timespec *time)
```

wartet bis

- zu einer Signalisierung über die Bedingungsvariable oder
- bis zum Ablauf der in time spezifizierten Zeit

im zweiten Fall Rückgabe der Fehlermeldung ETIMEOUT

Komplexe Synchronisationsprobleme können durch
Kombination von Mutex- und
Bedingungsvariablen gelöst werden

⇒ Darstellungs- und Sprachebene relativ elementar und
damit fehleranfällig!

Steuerung und Attribute von Threads

Bei der Generierung von Threads können diesen Attribute zugewiesen werden

Falls in `pthread_create` der Wert `NULL` für Attribute verwendet wird, so wird ein (bibliotheksabhängige) Standardeinstellung der Attribute gewählt

Dazu muss ein Objekt vom Typ pthread_attr_t mit der Funktion

```
int pthread_attr_init (pthread_attr_t *attr)
```

erzeugt werden

Objekt enthält Default-Werte der Parameter

Verschieden Parameter können gesetzt werden

Rückgabewert

- Standard `PTHREAD_CREATE_JOINABLE`, Rückgabewert wird nach Beendigung gespeichert
- `PTHREAD_CREATE_DETACHED` sorgt dafür, dass kein Rückgabewert zurückgegeben wird

Auslesen und Setzen des Wertes durch

```
int pthread_attr_getdetachstate
    (const pthread_attr_t *attr, int *detachstate)
int pthread_attr_setdetachstate
    (const pthread_attr_t *attr, int detachstate)
```

Stackeigenschaften

Größe des lokalen Stacks eines Threads kann mittels der Funktionen

```
int pthread_attr_getstacksize
    (const pthread_attr_t *attr, size_t *stacksize)

int pthread_attr_setstacksize
    (const pthread_attr_t *attr, size_t stacksize)
```

abgefragt und gesetzt werden

Abbruch von Threads

```
int pthread_cancel (pthread_t thread)
```

kündigt einen Abbruch an

Abbruch wird erst wirksam,

wenn ein Abbruchpunkt erreicht wird

Abbruchpunkte sind

- alle Synchronisationspunkte
  (pthread_join(), pthread_cond_wait(), open(), ..)
- durch die Funktion int pthread_testcancel() eingefügt

Ein Thread kann seinen eigenen Abbruch verhindern, indem seinen Zustand auf nicht unterbrechbar setzt mittels

```
int pthread_setcancelstate (int state, int *oldstate)
```

mit `state = PTHREAD_CANCEL_DISABLE`

Scheduling von Threads

- Beeinflussung der Abbildung von Benutzer-Threads auf Betriebssystem-Threads durch Scheduling Attribute
- Scheduling-Priorität bestimmt wie bevorzugt ein Thread behandelt wird
- Priorität kann zwischen minimalem und maximalem Wert liegen

Funktionen

```
int sched_get_priority_min (int policy)
int sched_get_priority_max (int policy)
```

liefern die Werte

Auswirkung der Priorität hängt von der Scheduling-Strategie (policy) ab

- Für jede Priorität existiert eine separate Bereitschaftswarteschlange
- Scheduler wählt Thread aus Warteschlange mit höchster Priorität nach Scheduling Strategie dieser Warteschlange
- Threads mit niedriger Priorität werden von der Bearbeitung ausgeschlossen, wenn permanent Threads hoher Priorität ausführbar sind

Mögliche Scheduling Strategien

SCHED_FIFO: Ausführung des ersten Threads aus der Warteschlange, bis dieser anhält oder durch einen Thread höherer Priorität unterbrochen wird

SCHED_RR: Threads aus einer Warteschlange werden nach dem Zeitscheibenverfahren abgearbeitet

SCHED_OTHER: spezielle Strategie an das Scheduling des jeweiligen Betriebssystems angepasst

Funktionen

```
int pthread_attr_getschedpolicy
    (const pthread_attr_t *attr, int *schedpolicy)
int pthread_attr_setschedpolicy
    (pthread_attr_t *attr, int schedpolicy)
```

zum Abfragen und Setzen der Scheduling Strategie

Scheduling-Bereich:

- lokales Scheduling nur der Threads eines Prozesses
- globales Scheduling der Threads aller Prozesse (nicht in allen Pthread Realisierungen vorhanden)

Üblicherweise wird lokales Scheduling verwendet, da diese ohne den BS-Scheduler auskommt

Datenhaltung in Threads

- alle Threads haben gemeinsamen Adressraum
- globale und dynamisch generierte Variablen sind von allen Threads zugreifbar
- innerhalb einer Funktion deklarierte Variablen sind nur von einem Thread zugreifbar

Nachteil: lokale Variablen gehen verloren, wenn eine Funktion verlassen wird

Lösungsansatz in Pthreads sind **Schlüssel**

- Schlüssel werden prozessglobal verwaltet
- Thread kann auf einen Schlüssel zugreifen und Daten zuweisen
- jeder Thread liest seine Daten beim Lesen des Schlüssels

Generierung eines Schlüssel

```
int pthread_key_create (pthread_key_t *key,
    void (*destructor) (void *))
```

key muss globale oder dynamisch allokierte Variable sein

```
int pthread_setspecific (pthread_key_t key, void *valu
```

weist dem Schlüssel key den Wert value zu

Üblicherweise ist `value` die Adresse eines dynamisch allokierten Datenobjekts (und keine lokale Variable!)

```
void *pthread_getspecific (pthread_key_t key)
```

liefert den gespeicherten Wert zurück

3.4 Laufzeitanalyse paralleler Programme

Wesentlicher Grund für den Einsatz von Parallelrechnern: Reduktion der Laufzeit von Anwendungsprogrammen

$\Rightarrow$ Laufzeitanalyse entscheidender Aspekt der

Parallelverarbeitung

Faktoren, die die Laufzeit beeinflussen:

- Architektur des Parallelrechners
- verwendeter Compiler
- verwendetes Betriebssystem
- parallele Programmierumgebung
- Lokalitätseigenschaften des verwendeten Algorithmus

Unterschiedliche Ansätze zur

Leistungsbewertung und -messung

Leider kein vollständiger Ansatz für parallele Programme vorhanden

Hier betrachtet

- Maße für sequentielle Programme
- spezifische Maße für parallele Programme
- einfache Modelle zur Modellierung und Analyse

Leistungsmaße sequentieller Systeme

Maße aus Sicht des Benutzers-Betreibers

- Antwortzeit: Zeit zwischen Start und Terminierung eines Programms
- Durchsatz: Anzahl Arbeitseinheiten pro Zeiteinheit

Maße des Prozessors:

- Taktrate (1/Zykluszeit)
- CPI (Clock Cycles per Instruction)
- MIPS Rate (Million Instructions per Second)
- MFLOPS (Million Floating-point Operations per Second)

Leistungsmaße des Prozessor reichen i.d.R. nicht aus, um Leistung des Systems zu bestimmen!

Leistung resultiert aus dem komplexen Zusammenspiel
zwischen Hardware und Software

$\Rightarrow$ Leistungsmessung/-vergleich oft über Messung
der Laufzeit spezieller Programme (Benchmarks)

Bekannte Benchmark-Sammlungen: **SPEC-Benchmarks**

- Zahlreiche Benchmarks für verschiedenste Anwendungen
- Bekannte Leistungsmaße SPECint2000 und SPECfp2000

Ermittlung der Leistungsmaße:

- mehrfache Ausführung der Benchmark-Programme
- Bestimmung der mittleren Ausführungszeit
- Normalisierung der gemessenen Ausführungszeit im Vergleich zu den Werten einer Referenzmaschine
- Bestimmung des geometrischen Mittelwerts der normalisierten Ausführungszeiten über alle Benchmark-Programme liefert SPEC-Leistungsmaß

Numerische Programme aus der Netlib

- besonders Linpack Routinen zum Vergleich numerischer Leistungsfähigkeit
- Vorgehen wie bei SPEC Benchmarks

Parallele Benchmarks

- Ansatz der Leistungsmessung im Prinzip auf parallele Programme übertragbar
- durch zusätzliche Freiheitsgrade treten viele praktische Probleme auf
- Gauss-Elimination aus Linpack ist ein üblicher Benchmark für Parallelrechner (siehe TOP 500 Liste)

Leistungsmaße paralleler Systeme

$T_p(n)$ **Laufzeit** eines parallelen Programms auf einem System mit p Prozessoren für ein Problem der Größe n

Laufzeit setzt sich zusammen aus

- Zeit für lokale Berechnungen
- Zeit für Datenaustausch durch Ausführung von Kommunikationsoperationen
- Wartezeiten auf Ergebnisse anderer Prozesse
- Zeit zur Synchronisation

Vergleich paralleler zur sequentieller Laufzeit:

$T^*(n)$ Laufzeit des sequentiellen Programms, definiert als

- Laufzeit des "besten" sequentiellen Algorithmus
- Laufzeit eines "üblichen" sequentiellen Algorithmus
- Laufzeit $T_1(n)$ (Vorsicht kann zu nicht realistischen Ergebnissen führen)

Kosten des parallelen Programms: $C_p(n) = p \cdot T_p(n)$

Paralleles Programm ist kostenoptimal, wenn

$$C_p(n) = T^*(n) \text{ gilt}$$

Bei asymptotischer Betrachtung $T^*(n)/C_p(n) = \Theta(1)$

Speedup zum Vergleich paralleler und sequentieller Programme

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

relativer Geschwindigkeitszuwachs durch Verwendung von p Prozessoren

Parallele Effizienz

$$E_p(n) = \frac{S_p(n)}{p}$$

Für theoretische Analysen gilt immer $S_p(n) \leq p$

Falls $S_p(n) < p$ könnte man (theoretisch) einen sequentiellen Algorithmus mit Laufzeit $p \cdot T^*(n)$ realisieren!

Algorithmus führt nacheinander jeweils eine Operation der p Prozessoren aus

$\Rightarrow$ Laufzeit $\leq p \cdot T^*(n)$

Praktisch kann durchaus $S_p(n) > p$ beobachtet werden durch

- größere Speicher
- Cacheeffekte
- hohe Varianz bei randomisierten Algorithmen

In der Praxis wird aber meist $p > S_p(n)$ beobachtet

Einige Gesetze bzgl. erreichbarer Leistung:

Amdahlsches Gesetz:

sei f Anteil des Programms, welcher sequentiell ausgeführt werden muss

Im besten Fall gilt :

$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} \cdot T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$

z.B. 20% sequentieller Anteil $\Rightarrow$ Speedup ≤ 5

Wichtiger Aspekt paralleler Programme **Skalierbarkeit**

Leistungsgewinn proportional zur Prozessorzahl (i.d.R. bei steigender Problemgröße)

Gustafson-Gesetz

- τ_f Ausführungszeit sequentieller Anteil (unabhängig von Problemgröße)
- $\tau_v(n,p)$ Ausführungszeit paralleler Anteil mit p Prozessoren bei Problemgröße n

$S_p(n) = \frac{\tau_f + \tau_v(n,1)}{\tau_f + \tau_v(n,p)}$

falls sich paralleler Anteil perfekt parallelisieren lässt gilt

$\tau_v(n,p) = (T^*(n) - \tau_f)/p$ und damit auch

$$S_p(n) = \frac{\tau_f + T^*(n) - \tau_f}{\tau_f + (T^*(n) - \tau_f)/p} = \frac{\frac{\tau_f}{T^*(n) - \tau_f} + 1}{\frac{\tau_f}{T^*(n) - \tau_f} + \frac{1}{p}}$$

Damit ist $\lim_{n\to\infty} S_p(n) = p$,

wenn $T^*(n)$ streng monoton in n steigt

Overhead bei Verwendung von p Prozessoren

$$H_p(n) = p \cdot T_p(n) - T^*(n)$$

für die Effizienz gilt dann

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

Auch wenn $\lim_{n\to\infty} S_p(n) = p$ gilt, ist immer noch interessant, wie n sich bei wachsendem p verhält

Dies drückt die **Isoeffizienzfunktion** aus

Sei E die geforderte Effizienz, dann drückt die Isoeffizienzfunktion das Verhältnis von p und n aus, so dass gilt

$$T^*(n) = \frac{E}{1-E} \cdot H_p(n)$$

Ideal: konstantes Verhältnis, real oft nur lineares Verhältnis

siehe auch Verhältnis $Nmax$ - $N1/2$ bei Supercomputern

Laufzeiten paralleler Programme ohne Kommunikationszeiten

Zur Darstellung von Abhängigkeiten

Präzendenzgraphen als Programmmodell

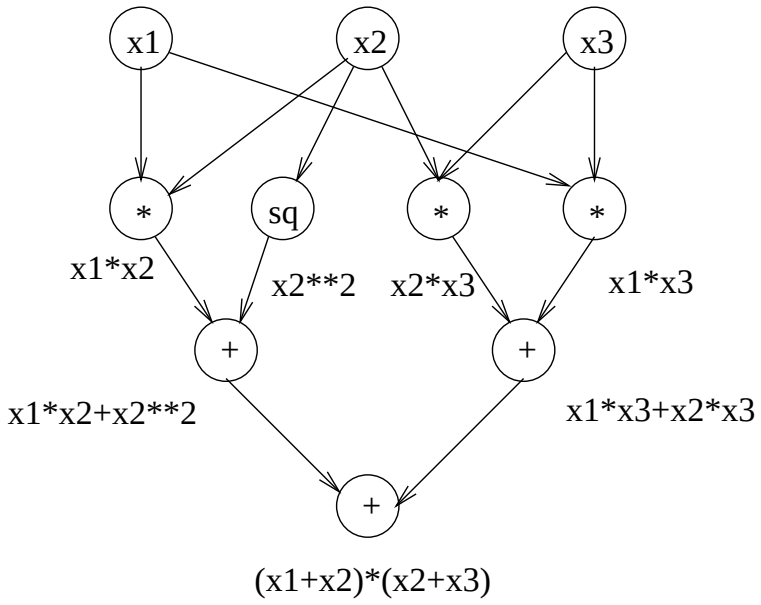
$G = (N, A)$ sei ein gerichteter azyklischer Graph

- Knoten beschreiben Operationen oder Tasks
- Kanten beschreiben Abhängigkeiten (Daten - Instruktionen)

Ähnliche Definition wie Abhängigkeitsgraphen auf Instruktionsebene (siehe Folie 8)

Hier aber keine Festlegung der Granularität!

Beispiel



Einige Notationen

- Eingangsgrad: Anzahl Kanten, die in einem Knoten enden
- Eingangsknoten: Knoten ohne Eingangskanten
- Ausgangsknoten: Knoten ohne Ausgangskanten
- N_0 Menge aller Knoten, die keine Eingangsknoten sind
- Tiefe: Länge des längsten Pfades von einem Eingangsknoten zu einem Ausgangsknoten (= Anzahl Kanten)

Interpretation der Knoten

- Eingangsknoten beschreiben Daten
- Alle weiteren Knoten beschreiben Operationen

Präzendenzgraphen sind nur zur Darstellung
sequentieller Abläufe geeignet!

Darstellung von Schleifen durch Entfaltung

for $(i = 0; i <= n; i++)$

$x[i] = y[i] + z[i]$;

wird zu

$x[0] = y[0] + z[0]$;

$\vdots$

$x[n] = y[n] + z[n]$;

Analyse von Programmen mittels Präzedenzgraphen:
vereinfachende Annahmen

- alle Operationen dauern eine Zeiteinheit
  (nur bei elementaren Operationen gerechtfertigt)
- Operationen der Eingangsknoten benötigen keine Zeit
- Kommunikationzeiten bleiben unberücksichtigt

$\Rightarrow$ Resultate beschreiben den "best case"

Präzedenzgraph wird mit p Prozessoren abgearbeitet
$\Rightarrow$ Abbildung der Knoten aus N_0 auf die Prozessoren
Jeder Knoten muss von einem Prozessor bearbeitet werden (keine Parallelisierbarkeit bzgl. einzelner Knoten)

- P_i Prozessor, der Knoten i bearbeitet
- $t_i \in \mathbb{N}$ Zeit zu der Knoten i bearbeitet wird

Es muss gelten

- Für alle $i, j \in N_0$ mit $i \neq j$ und $t_i = t_j$ ist $P_i \neq P_j$
- Falls $(i, j) \in A$, dann gilt $t_j \geq t_i + 1$

Eine Menge $\{(i, P_i, t_i) | i \in N_0\}$, die die Bedingungen erfüllt heißt **Schedule**

Falls Kommunikationszeiten berücksichtigt werden, muss die zweite Bedingung verschärft werden zu

Falls $(i, j) \in A$, dann gilt

- $t_j \geq t_i + 1$ falls $P_i = P_j$
- $t_j \geq t_i + 1 + \tau(P_i, P_j)$ sonst
  ($\tau(P_i, P_j)$ Kommunikationszeit von P_i nach P_j)

Vereinfachende Annahme im Folgenden:

Kommunikationszeiten vernachlässigbar

Laufzeit eines Schedules: $\max_{i \in N_0} t_i$

Definitionen

- T_p Minimum der Laufzeiten aller Schedules mit p Prozessoren
- $T_\infty = \min_{p \geq 1} T_p$

Es gilt

- T_p monoton nicht steigende Funktion
- $T_p = T_\infty$ für $p \geq p^*$

Bestimmung der Laufzeiten

- T_1 entspricht Anzahl Knoten also $|N_0|$
- T_∞ entspricht der Tiefe D des Präzedenzgraphen
- T_p es gilt $T_1 \geq T_p \geq T_\infty$ exakte Bestimmung NP-hart

Schranken für T_p:

- Sei $c \in \mathbb{N}$ und $q = c \cdot p$, dann gilt $T_p \leq c \cdot T_q$
- Für $p \geq 1$ gilt $T_p < T_\infty + \frac{T_1}{p}$
- Falls $p \geq \frac{T_1}{T_\infty}$ gilt $T_p < 2 \cdot T_\infty$
- Falls $p \leq \frac{T_1}{T_\infty}$ gilt $\frac{T_1}{p} \leq T_p < 2 \cdot \frac{T_1}{p}$

Aus den letzten beiden Punkten folgt:

$\Omega(T_1/T_\infty)$ Prozessoren reichen aus, um bis auf einen konstanten Faktor die optimale Laufzeit zu erreichen

($f(n) \in \Omega(g(n)) \Rightarrow g(n) \leq c \cdot f(n)$ für $n \geq n_0$ und $c \in \mathbb{R}_+$ bzw. $g(n) \in O(f(n))$)

Konstruktion eines solchen Schedules für p Prozessoren

- Konstruiere ein Schedule $|N_0|$ Prozessoren indem jedem Knoten ein Prozessor zugewiesen wird
- Generiere Schedule für $p \approx T_1/T_\infty$ Prozessoren, indem gleichzeitige Operationen im Schedule mit beliebig vielen Prozessoren auf p Prozessoren verteilt werden

Dieses Schedule benötigt $T_p < T_\infty + \frac{T_1}{p}$ Zeiteinheiten

Bisher untersucht:

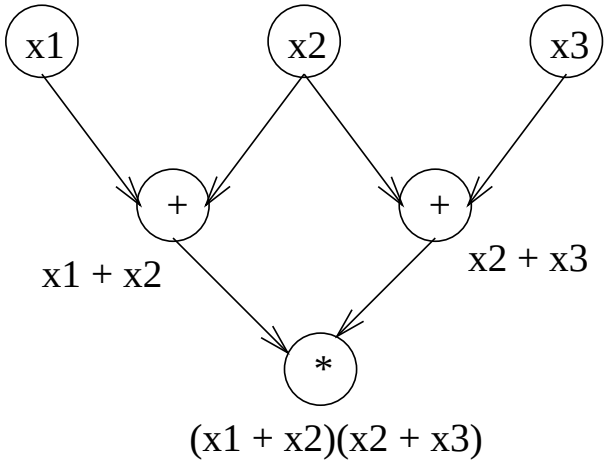
Schedule für einen gegebenen Präzedenzgraphen

Für ein Problem existieren aber oft

mehrere Präzedenzgraphen

Präzedenzgraph auf Folie 56 liefert $T_1 = 7$ und $T_\infty = 3$

Folgender Präzedenzgraph liefert $T_1 = 3$ und $T_\infty = 2$

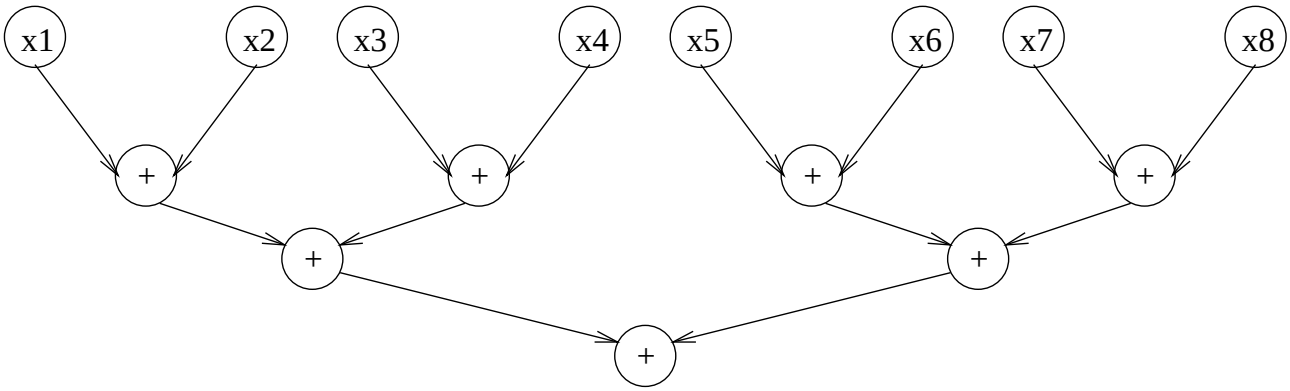


Auswahl eines optimalen Präzedenzgraphen (oder optimalen Präzedenzgraphen für eine feste Prozessorzahl) entspricht Entwurf eines optimalen Algorithmus!

Also im allgemeinen keine lösbare Aufgabe
(zumindest falls wirklich ein Optimum gesucht wird)

Einige allgemeine Beispiele

Skalaraddition



Sequentieller Algorithmus benötigt $n-1$ Schritte also $T^*(n) = n-1$

Paralleler Algorithmus mit

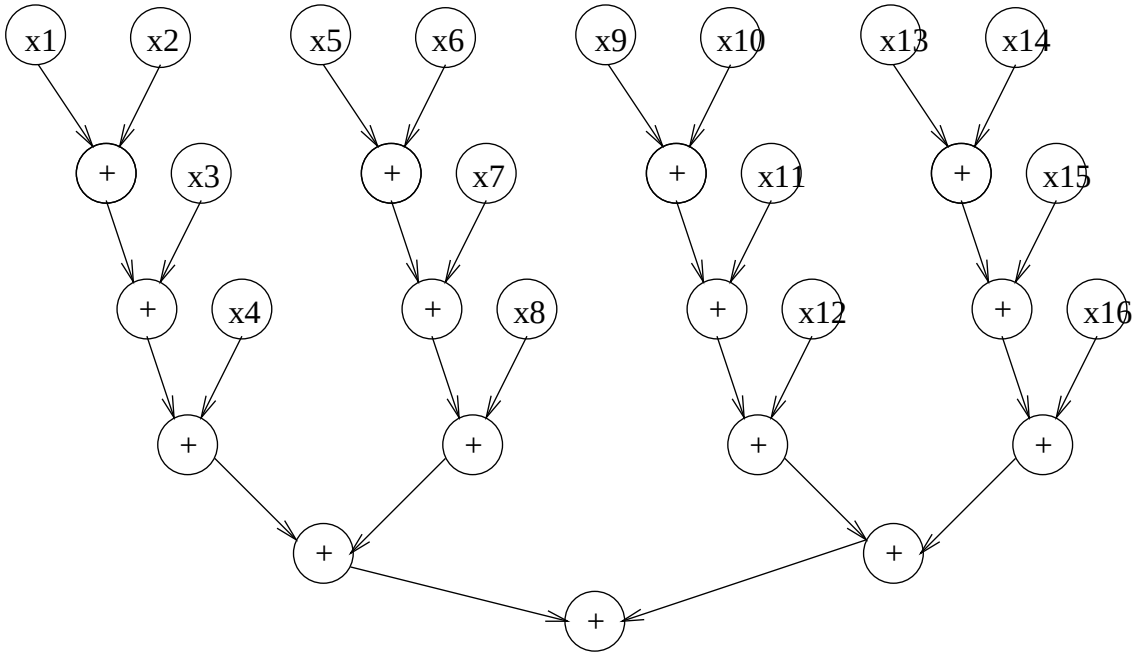
$\lfloor n/2 \rfloor$ Prozessoren und Laufzeit $\lceil \log n \rceil$

Speedup $S_{n/2} \approx \frac{n}{\log n}$

Für die Effizienz gilt $E_{n/2}(n) = \frac{n-1}{\lfloor n/2 \rfloor \cdot \lceil \log n \rceil}$

Für $n \to \infty$ konvergiert die Effizienz gegen 0

Alternativer Präzedenzgraph



Paralleler Algorithmus mit

$\lfloor n/\log n \rfloor$ Prozessoren und Laufzeit $\approx 2 \cdot \lceil \log n \rceil$

Effizienz $E_{n/\log n}(n) \approx \frac{n-1}{2 \cdot \lceil \log n \rceil \cdot \lfloor n/\log n \rfloor} \approx \frac{1}{2}$

es werden T_1/T_∞ Prozessoren eingesetzt!

Herleitung der Isoeffizienzfunktion:

Sei $n = 2^x$ und $p = 2^y$ $(0 < y < x)$

Zwei-Phasen Algorithmus (als Verallgemeinerung)

1. Phase: jeder Prozessor addiert $2^x/2^y = 2^{x-y}$ Elemente in $2^{x-y} - 1$ Zeiteinheiten
2. Phase: Addition der 2^y Teilsummen in y Schritten

Für $y = x - 1$ folgte Algorithmus 1

für $y = x - \log x$ Algorithmus 2

Werte des Algorithmus

- Laufzeit $T_p(n) = T_{2^y}(n) = 2^{x-y} + y - 1$
- Speedup $S_{2^y}(n) = \frac{2^x - 1}{2^{x-y}+y-1}$
- Effizienz $E_{2^y}(n) = \frac{2^x-1}{2^y \cdot (2^{x-y}+y-1)} \approx \frac{2^{x-y}}{(2^{x-y}+y)}$
- Overhead $H_{2^y}(n) = 2^y \cdot (y-1) + 1 \approx y \cdot 2^y$

Für feste Effizienz E gilt die Gleichung

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

Isoeffizienzfunktion $\Theta(p \log p)$

Bei Vergrößerung der Prozessorzahl von p nach p'

muss n um den Faktor $\frac{p' \log p'}{p \log p}$ wachsen,

um konstante Effizienz zu garantieren

Innere-Produkte

$$(x, y) = \sum_{i=1}^{n} x(i) \cdot y(i)$$

Algorithmus 1:

n Prozessoren und $\lceil \log n \rceil + 1$ Schritte

- jeder Prozessor multipliziert zwei Werte
- Addition der Werte mit Algorithmus 1 zur Summation in $\log n$ Schritten

Speedup $S_n(n) = \frac{n+(n-1)}{\lceil \log n \rceil + 1} \approx \frac{2n}{\log n}$

Effizienz $E_n(n) \approx \frac{2}{\log n}$

Algorithmus 2:

$n/\log n$ Prozessoren bilden die n Produkt in $\log n$ Schritten

- Addition der Werte mit Algorithmus 2 zur Summation in $2 \log n$ Schritten

Speedup $S_{n/\log n}(n) \approx \frac{2n}{3 \log n}$

Effizienz $E_{n/\log n}(n) \approx \frac{2}{3}$

Matrix-Multiplikation

Multiplikation $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{n \times l}$

Algorithmus 1:

Rückführung auf $m \cdot l$ innere Produkte

n-dimensionaler Vektoren

Verwendung von Algorithmus 1 zur Bildung innerer Produkte

mit $O(n \cdot l \cdot m)$ Prozessoren

Algorithmus 2:

Rückführung auf Algorithmus 2 zur Bildung innerer Produkte

Für $n = m = l$ (quadratische Matrizen) folgt ein Algorithmus mit $n^3 / \log n$ Prozessoren in $2 \log n$ Schritten

Realistischer für diese Problemklasse

Algorithmus mit $\Theta(n^2)$ Prozessoren und Laufzeit $\Theta(n)$

Algorithmus mit $\Theta(n)$ Prozessoren und Laufzeit $\Theta(n^2)$

in beiden Fällen Effizienz in $\Theta(1)$

Aussagen zur Analyse mit Präzedenzgraphen

Laufzeitanalyse auf dieser Ebene liefert "best case" Resultate, die sich bei Berücksichtigung der Kommunikationzeiten verschlechtern

Ziel:

Realisierung der Algorithmen, so dass das asymptotische Verhalten durch die Kommunikationzeiten nicht beeinflusst wird

Bestimmung von Kommunikationszeiten

Analyse der Laufzeiten der Basiskommunikationsprobleme auf unterschiedlichen Architekturen

Hier untersucht: Laufzeit in Abhängigkeit der Topologie

Annahmen:

- Verbindungen bidirektional, zu jedem Zeitpunkt kann in jede Richtung eine Nachricht unterwegs sein
- Knoten können auf auslaufenden Leitungen simultan senden und von ankommenden Leitungen simultan empfangen
- Nachrichten werden ohne Unterbrechung übertragen
- Übertragungszeit eine Nachricht der Länge m: $TT(m) = t_S + m \cdot t_B$ (für alle Leitungen identisch)
- Store-and-Forward Routing, Nachricht durchläuft Pfad von Quelle zu Senke $\Rightarrow$
  Länge des Pfades entspricht Anzahl benötigter Schritte

Ziel Ableitung von asymptotischen Aussagen in Abhängigkeit von der Prozessorzahl p: Ausführungszeit in $\Theta(g(p))$:

d.h. $\exists c_1, c_2 : 0 < c_1 < c_2$ und $p_0 \in \mathbb{N}$, so dass für die Kommunikationszeit $f(p)$ gilt:

$c_1 \cdot g(p) \leq f(p) \leq c_2 \cdot g(p)$ falls $p \geq p_0$

Als Beispiel Herleitung der Werte für ein lineares Feld:

Einzel-Broadcast

- Paket wird vom Initiator an seine Nachbarn weitergeleitet
- Nachbarn leiten Paket zu ihren Nachbarn weiter

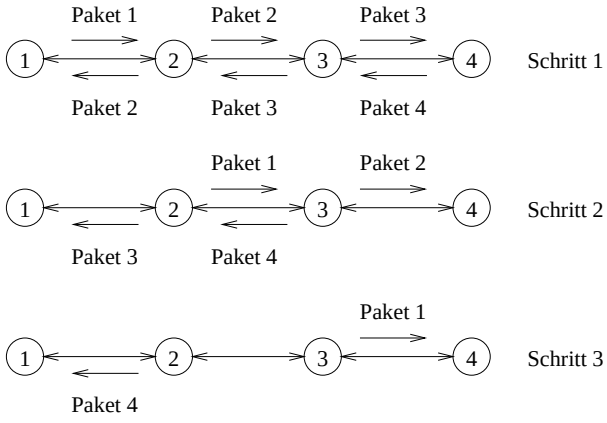
Aufwand zwischen $\lfloor p/2 \rfloor$ und $p-1$ Schritte $(\Theta(p))$

in ähnlicher Form Einzel-Akkumulation

Allgemeiner Broadcast

- 1. Schritt: jeder Knoten schickt sein Paket an seine/seinen Nachbarn
- k-ter Schritt $(2 \leq k < p)$:
  Knoten i $(p > i \geq k)$ schickt das Paket von Knoten $i-k+1$ an seinen rechten Nachbar $i+1$
  Knoten i $(2 \leq i \leq p-k+1)$ schickt das Paket von Knoten $i+k-1$ an seinen linken Nachbar $i-1$

Aufwand $p-1$ Schritte $(\Theta(p))$



in ähnlicher Form Allgemeine-Akkumulation

Scatter und Gather Operation:

als Spezialisierung des Allgemeinen Broadcast bzw. der Allgemeinen Akkumulation realisierbar

Gesamtaustausch

- Realisierung durch p sequentielle Scatter Operationen

Gesamtaufwand $\Theta(p^2)$

Asymptotisch optimal, da

- minimale Laufzeit durch maximale Nachrichtenzahl in eine Richtung über eine Leitung festgelegt wird
- Nachrichtenzahl über die "mittlere" Kante entspricht $p^2/4$ pro Richtung

Aufwand der Kommunikationsprobleme
auf anderen Topologien

Operation	Baum	d-dim. Gitter	Hyperwürfel
Einzel-Broadcast	$\Theta(\log p)$	$\Theta(p^{1/d})$	$\Theta(\log p)$
Scatter	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Allg.-Broadcast	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Austausch	$\Theta(p^2)$	$\Theta(p^{(d+1)/d})$	$\Theta(p)$

Für Details siehe Rauber/Rünger Kap. 6.3.1

Parallele innere Produkte

$(x, y) = \sum_{i=1}^{n} x(i) \cdot y(i)$

Annahmen

- $n = r \cdot p$ mit $r \in \mathbb{N}$
- Vektoren blockweise auf den Prozessoren verteilt
- Prozessor k berechnet

$$c_k = \sum_{j=r\cdot(k-1)+1}^{r\cdot k} x(j) \cdot y(j)$$

- Ergebnisse werden per Akkumulation in einem Prozessor gesammelt
- Übertragungszeit eines Wertes zwischen verbundenen Knoten t_S
- Berechnungszeit einer Operation t_A

Realisierung auf einem linearen Feld

mittlerer Knoten sammelt Ergebnisse

- Zeit zur Bildung lokaler Summen $\approx \frac{2 \cdot n}{p} \cdot t_A$
- Zeit für Akkumulationoperation $\approx \frac{p}{2} \cdot (t_A + t_S)$

Damit gilt

$$T_p(n) = \frac{2 \cdot n}{p} \cdot t_A + \frac{p}{2} \cdot (t_A + t_S)$$

Ziel: minimale Laufzeit für eine gegebene Problemgröße

$\Rightarrow$ Bestimmung der optimalen Prozessorzahl

$\Rightarrow$ Darstellung der Laufzeit als Funktion von p:

$$T(p) \equiv T_p(n)$$

Optimum wird erreicht bei

$T'(p) = -\frac{2 \cdot n \cdot t_A}{p^2} + \frac{t_A \cdot t_S}{2} = 0$

$\Rightarrow p = \sqrt{\frac{4 \cdot n \cdot t_A}{t_A + t_S}}$

(zweite Ableitung > 0 dadurch Minimum!)

Einige Beobachtungen

- Asymptotische Laufzeit für große n: $\Theta(\sqrt{n})$
- Optimale Prozessorzahl wächst mit $\sqrt{n}$
- Für $t_S > (4 \cdot n - 1) \cdot t_A$ ist eine sequentielle Realisierung am effizientesten

Realisierung auf einem Hyperwürfel

- Beliebiger Knoten sammelt die Ergebnisse
- Berechnungszeit wie beim linearen Feld
- Zeit für die Akkumulationsoperation $\approx \log p \cdot (t_A + t_S)$

Damit gilt

$$T_p(n) = \frac{2 \cdot n}{p} \cdot t_A + \log p \cdot (t_A + t_S)$$

Optimale Prozessorzahl

$$T'(p) = -\frac{2 \cdot n \cdot t_A}{p^2} + \frac{t_A \cdot t_S}{p \cdot \ln 2} = 0$$

$$\Rightarrow p \approx \frac{2 \cdot \ln 2 \cdot n \cdot t_A}{t_A + t_S}$$

- Optimale asymptotische Laufzeit $\Theta(\log n)$
- Optimale Prozessorzahl wächst linear in n

Parallele Matrix-Vektor-Multiplikation

Berechnung des Produkts einer $n \times n$ Matrix mit einem Vektor

$$c(i) = \sum_{j=1}^{n} A(i,j) \cdot b(j)$$

Möglichkeiten der Verteilung

- spaltenweise mit Allgemeiner-Akkumulation zur Sammlung der Werte
- zeilenweise mit Allgemeinem-Broadcast zur Verteilung der Werte

siehe Folie 22

In beiden Fällen

- Berechnungsaufwand $\frac{2 \cdot n^2}{p}$ Operationen pro Prozessor
- da Allgemeiner-Broadcast und Allgemeine-Akkumulation duale Operationen auch keine Unterschiede bei asymptotischen Kommunikationszeiten
- Übertragungszeit einer Nachricht mit r Werten zwischen verbundenen Knoten $t_S + r \cdot t_B$
- Berechnungszeit einer Operation t_A

Realisierung auf einem linearen Feld

$$T_p(n) = \frac{2 \cdot n^2}{p} \cdot t_A + p \cdot \left(t_S + \frac{n}{p} \cdot t_B \right)$$

$$= \frac{2 \cdot n^2}{p} \cdot t_A + p \cdot t_S + n \cdot t_B$$

Zur Ermittlung der optimalen Prozessorzahl sei wieder

$T(p) \equiv T_p(n) : T'(p) = -\frac{2 \cdot n^2 \cdot t_A}{p^2} + t_S$

und damit lautet die optimale Prozessorzahl

$$p = n \cdot \sqrt{\frac{2 \cdot t_A}{t_S}}$$

- Optimale Prozessorzahl linear in n
- Asymptotische Laufzeit $\Theta(n)$
- Asymptotische Effizienz $\Theta(1)$ (optimaler Wert)

Realisierung auf einem Hyperwürfel

Aufwand einer Allgemeinen-Broadcast:

$p/\log p$ Schritte mit Aufwand $t_S + t_B \cdot n/p$ pro Schritt

$$T_p(n) = \frac{2 \cdot n^2}{p} \cdot t_A + \frac{p}{\log p} \cdot \left(t_S + \frac{n}{p} \cdot t_B \right)$$

$$= \frac{2 \cdot n^2}{p} \cdot t_A + \frac{p}{\log p} \cdot t_S + \frac{n \cdot t_B}{\log p}$$

- Optimale Prozessorzahl nicht mehr analytisch berechenbar
- Bei Wahl von $O(n^2)$ Prozessoren Laufzeit $O(\log n)$