3. Parallele Programmierung

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

⇒ 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 Laufzeitanalyse paralleler Programme

3.3 Programmierung mit gemeinsamen Variablen (Pthreads)

3.4 Message Passing Programmierung

3.4.1 PVM
3.4.2 MPI

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

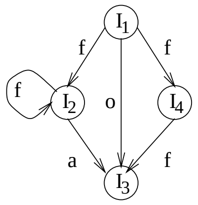
Fluss-Abh.	Anti-Abh.	Ausgabe-Abh.
$I_1 : R_1 \leftarrow R_2 + R_3$	$I_1 : R_1 \leftarrow R_2 + R_3$	$I_1 : R_1 \leftarrow R_2 + R_3$
$I_2 : R_5 \leftarrow R_1 + R_4$	$I_2 : R_2 \leftarrow R_4 + R_5$	$I_2 : R_1 \leftarrow R_4 + R_5$

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 technische/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:

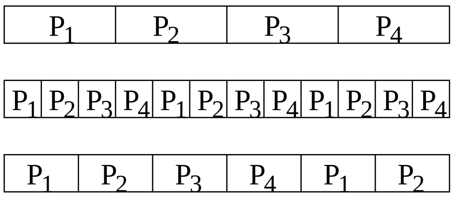
Prozessoren erhalten benachbarte Feldelemente, so dass sich die Zahl der Elemente pro Prozessor sich um maximal 1 unterscheidet

zyklische Verteilung:

Feldelemente werden reihum an die Prozessoren verteilt

blockzyklische Verteilung:

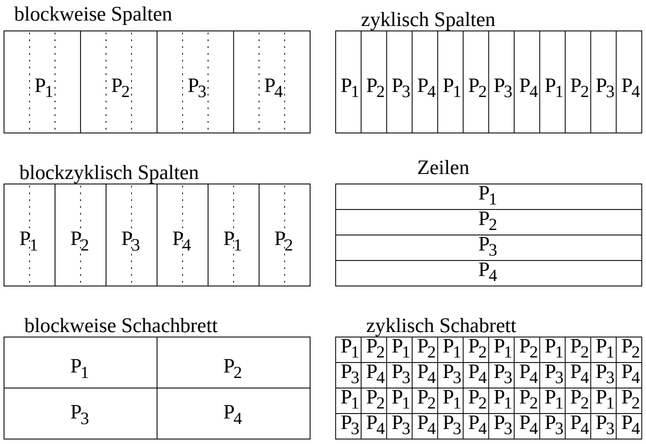
Kombination der vorherigen, Blöcke vorgegebener Größe werden zyklisch verteilt



Verteilung zweidimensionaler Felder (Matrizen):

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

Beispiele:



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 $\mathit{forall}\ (i = 1 : n)$

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

$\mathit{endforall}$

entspricht der Vektoranweisung

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

⇒ parallele Abarbeitung aller Schleifenrümpfe
sequentielle Abarbeitung innerhalb eines
Schleifenrumpfes

Alternative dopar - Schleifen

⇒ 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)

⇒ 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

⇒ komplexeste Form der Parallelisierung

⇒ 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

Informationsaustausch

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

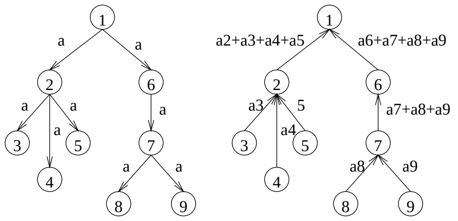
Einzel-Akkumulation:

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)



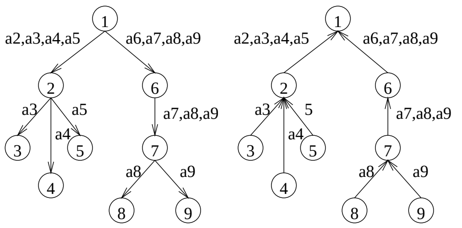
Gather:

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

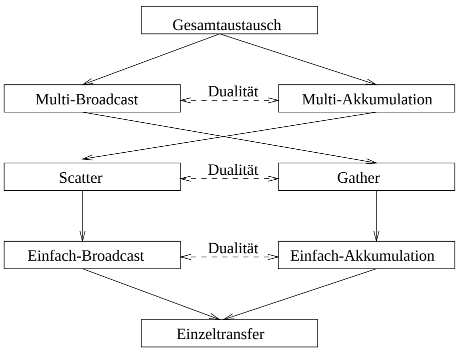
Multi-Akkumulation:

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	Version 2
for ($i = 0$; $i < n$; $i++$)	*for* ($i = 0$; $i < n$; $i++$)
$c[i] = 0$;	$c[i] = 0$;
for ($i = 0$; $i < n$; $i++$)	*for* ($j = 0$; m; $j++$)
for ($j = 0$; $j < m$; $j++$)	*for* ($i = 0$; $i < n$; $i++$)
$c[i] = c[i] + A[i,j] * b[j]$;	$c[i] = c[i] + A[i,j] * b[j]$;

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 n Linearkombinationen so auf die Prozessoren verteilt, dass jeder Prozessor ungefähr n/p Linearkombinationen berechnet

<u>Detaillierte Betrachtung der Algorithmen</u>:

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

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
- $n\_local = n/p$

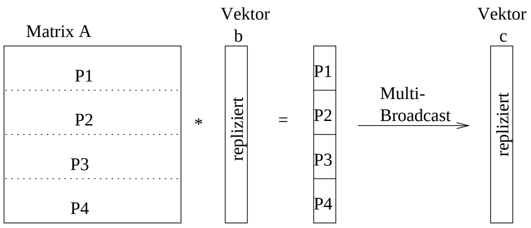
Lokaler Algorithmus auf Prozessor p

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
- $n\_local = m/p$

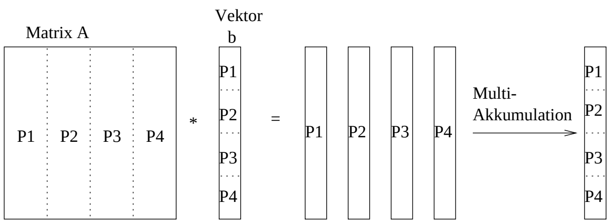
Lokaler Algorithmus auf Prozessor p

for $(i = 0;\ i < n;\ i++)$
  $c[i] = 0;$
for $(j = 0;\ i < m\_local;\ j++)$
  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 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
⇒ Leistungsmessung/-vergleich oft über Messung der Laufzeit spezieller Programme (Benchmarks)

Bekannte Benchmark-Sammlungen:

SPEC-Benchmarks

- Zahlreiche Benchmarks für verschiedenste Anwendungen
- Bekannte Leistungsmaße SPECint95 und SPECfp95

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 $p \cdot S_p(n)$ realisieren!

Algorithmus führt nacheinander jeweils eine Operation der p Prozessoren aus
$\Rightarrow$ Laufzeit $\leq p \cdot T_p(n)$

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

$$\tau_f(n, p) = (T^*(n) - \tau_f)/p$$

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

Modellierung paralleler Laufzeiten

Laufzeiten paralleler Programme ohne Kommunikationszeiten

Zur Darstellung von Abhängigkeiten

Präzendenzgraphen als Programmmodell

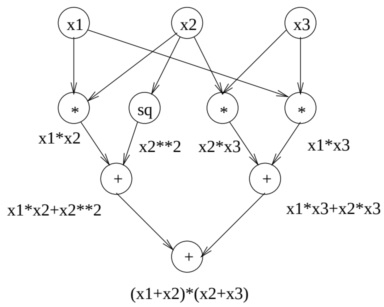
$G = (N, A)$ sei ein gerichteter azyklischer Graph

- 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 Schliefen 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 Prozessor 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:
<u>Kommunikationszeiten vernachlässigbar</u>

Laufzeit eines Schedules: $\max_{i \in N_0} t_i$
<u>Definitionen</u>

- T_p Minimum über die 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 $T_p \leq T_\infty + \frac{T_1}{p}$
- 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 $T_p \leq T_\infty + \frac{T_1}{p}$ Zeiteinheiten

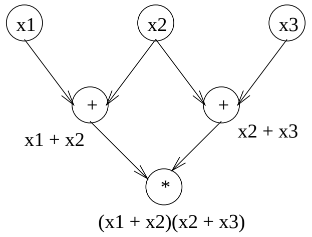
Bisher untersucht:

Schedule für einen gegebenen Präzedenzgraphen

Für ein Problem existieren aber oft
mehrere Präzedenzgraphen

Präzedenzgraph auf Folie 39 liefert $T_1 = 7$ und $T_\infty = 3$

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

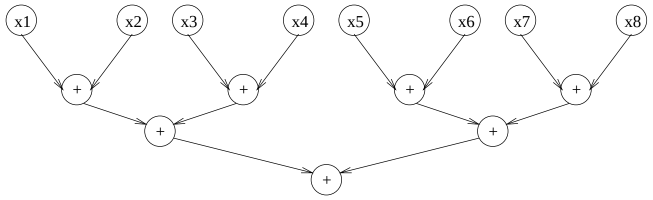


Auswahl eines optimalen Präzedenzgraphen (oder optimalen Präzedenzgraphen für eine feste Prozessorzahl) entspricht Entwurf eines optimalen Algorithmus!

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

<u>Einige allgemeine Beispiele</u>

Skalaraddition



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

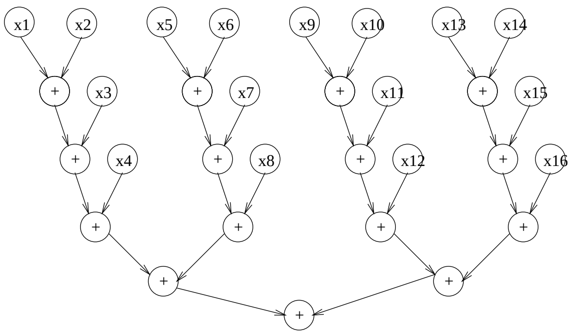
Paralleler Algorithmus mit
$\lfloor n/2 \rfloor$ Prozessoren und Laufzeit $\lceil \log n \rceil$

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!

<u>Herleitung der Isoeffizienzfunktion</u>:

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

<u>Weitere Algorithmen</u>

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

Realistsicher 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 allen auslaufenden Leitungen simultan senden
- Knoten können von allen 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

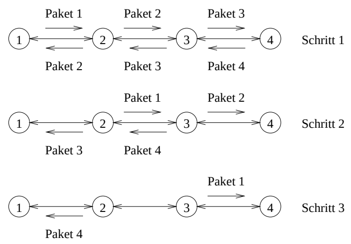
Aufwand zwischen $\lfloor p/2 \rfloor$ und $p-1$ Schritte ($\Theta(p)$)

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			
$\Theta(p^2)$			
$\Theta(p^{(d+1)/d})$			
$\Theta(p)$			

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
⇒ Bestimmung der optimalen Prozessorzahl
⇒ 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

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

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

$$p = n \cdot \sqrt{\frac{2 \cdot \alpha}{\beta}}$$

- 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

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

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

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)

⇒ 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
⇒ 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

⇒ effiziente Erzeugung und Verwaltung aber
keine Abgrenzung zwischen Threads eines Prozesses

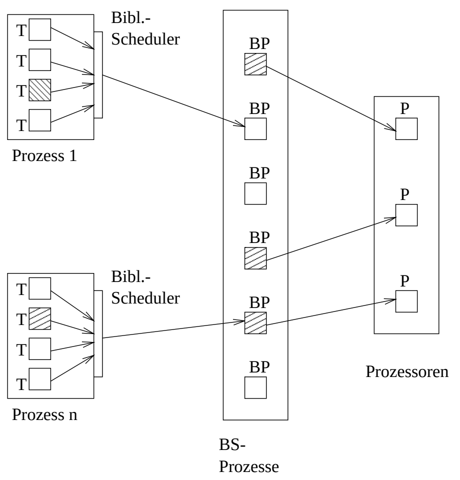
Realisierung von Threads

- auf Benutzerebene ohne Beteiligung des Betriebssystems
- auf Betriebssystemebene

Ausführungsmodelle für Threads

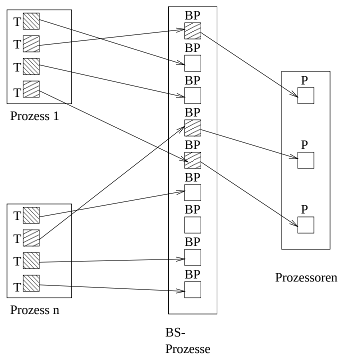
Falls BS Threadverwaltung nicht unterstützt

N:1 Abbildung

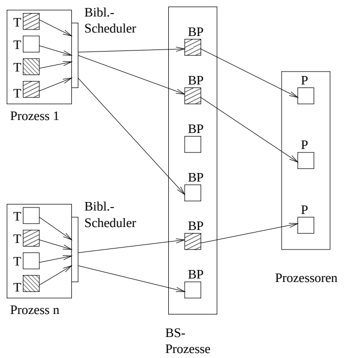


Falls BS Threadverwaltung unterstützt

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 Synchronistion 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_t`	Mutex-Variable
`pthread_cond_t`	Bedingungsvariable
`pthread_key_t`	Zugriffsschlüssel
`pthread_attr_t`	Thread-Attribut
`pthread_mutrexattr_t`	Mutex-Attribut
`pthread_condattr_t`	Bedingungsvariablen-Attr.
`pthread_once_t`	

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 sein 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] ;
} matrix_type_t ;

double thread_mult(matrix_type_t *work) {
  int i, row = work->row, column = work->column;
  double result = 0.0 ;
  for (i=0; i < work->size; i++)
    result += work->MA[row][i] * work->MB[i][column] ;
  return(result) ;
}

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 ;
      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], (void **) &(MC[row][column])) ;
} } }
```


Behandlung zeitkritischer Abläufe

Problem durch gemeinsamen Adressraum der Threads
⇒ 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

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 Thread erlaubt auf beliebige Bedingung 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 Pirorität 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
- jeder 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 *value)
```

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 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 (sehr aufwendig)
- Verwendung von Standardbibliotheken
  - Kommunikationsroutinen für unterschiedliche Architekturen (⇒ portable Programme)

  - Routinen für Basiskommunikationsoperationen (Punkt-zu-Punkt, Broadcast)

Hier behandelt

3.4.1 PVM

3.4.2 MPI

3.4.1 PVM

PVM (Parallel Virtual Machine)

Softwarepaket zur Realisierung

- einer einheitlichen Programmierumgebung für eine Ansammlung von (heterogenen) Rechnern und
- dem Aufbau einer virtuellen parallelen Maschine
- mit Anbindung an C, C++ und Fortran

Entwicklung seit 1989 an verschiedenen Universitäten +
Oak Ridge Labs. als Grundlagenforschungsprojekt!

Übliche Konfiguration

- PCs oder Workstations über LAN verbunden
- auch für viele Parallelrechner mit verteiltem Speicher verfügbar

Kommunikationsbasis oft UDP/TCP aber auch andere Realisierungen existieren (z.B. direkt auf dem Data-Link Layer ohne Transportschicht)

PVM bietet eine uniforme Sicht auf eine u.U. extrem
heterogene Umgebung

PVM-Programm besteht aus

- Anzahl von Tasks
  - die Daten austauschen können
  - sich synchronisieren können
- Tasks können zur Laufzeit erzeugt werden
  - jeder Tasks kann neue Tasks erzeugen und
  - von ihm erzeugte Tasks abbrechen
- Tasks laufen auf einer virtuellen Maschine, die
  - aus einem oder mehreren Rechnern besteht
  - während der Laufzeit erweitert werden kann
- Tasks
  - werden über TID (Task Identifier) angesprochen
    PVM vergibt eindeutige Nummer für jeden Task
  - können zu Gruppen zusammengefasst werden
    * jeder Tasks einer Gruppe enthält eine eindeutige Nummer in der Gruppe
    * ein Task kann zu mehreren Gruppen gehören
- Tasks kommunizieren

  - mit Hilfe von PVM-Kommunikationsoperationen
  - auch, wenn sie auf Rechnern mit unterschiedlichen Datenformaten laufen

Aufbau einer virtuellen Maschine
Start der PVM-Konsole mit den Kommandos

- `add <name>` fügt Rechner zur virtuellen Maschine hinzu
- `delete <name>` entfernt Rechner aus der virtuellen Maschine
- `conf` listet die aktuelle virtuelle Maschine auf
- `quit` Konsole wird abgebrochen PVM läuft weiter
- `halt` PVM und alle PVM-Prozesse werden abgebrochen
- ...

andere Möglichkeiten des Aufbaus der virtuellen Maschine

- XPVM als graphisches front-end
- aus dem Programm heraus

```
int pvm_addhost
    (char **hosts, int nhosts, int info) ;
```

fügt `nhosts` Rechner mit den in `hosts` angegebenen Namen hinzu

```
int pvm_delhost
    (char **hosts, int nhosts, int info) ;
```


entfernt `nhosts` Rechner

```
int pvm_config (int *nhost, int *narch,
        struct pvmhostinfo **hostp);
```

liefert die Konfiguration

Information über einen Rechner

```
struct pvmhostinfo (int hi_tid, char *hi_name,
                    char *hi_arch, int hi_speed);
```

wobei

- `hi_tid` TID des PVM-Dämons auf der Maschine
- `hi_name` Name der Maschine
- `hi_arch` Architektur
- `hi_speed` Geschwindigkeit

(letzten beiden Informationen werden in der .rhosts Datei vom Benutzer angegeben und nicht geprüft!)

Starten von PVM-Programmen

- aus PVM heraus `pvm % spawn -> program_name`
- aus XPVM mittels Menü
- aus einem C-Programm
  `int pvm_spawn(char *task, char *argv, int flag, *where, int ntask, int *tids) ;`
  - `task` Programmname
  - `argv` Argumentstring
  - `flag` wo und wie gestartet wird z.B. `PvmTaksDefault` auf von PVM gewählter Maschine, `PvmTaskHost` auf vorgegebenem Rechner

  - `where` Hostname
  - `ntask` Anzahl Kopien
  - `tids` IDs der gestarteten Kopien
  - Rückgabewert: Anzahl gestarteter Kopien oder (negativer) Fehlercode

Übersicht über laufende Tasks mittels

```
int pvm_tasks(int which, int *ntask,
          struct taskinfo **taskp) ;
```

- `which` = 0 liefert Informationen über alle Tasks der virtuellen Maschine
- `ntask` Anzahl Tasks
- `taskp` Informationen über die einzelnen Tasks
- Rückgabewert Status des Funktionsaufrufs, < 0 bedeutet Fehler

Struktur mit Taskinformationen

```
struct taskinfo {
  int ti_tid ; /* ID des Tasks */
  int ti_ptid; /* ID Eltertask */
  int ti_host; /* ID Host */
  int ti_flag; /* Taskstatuts */
  char *ti_a_out;
          /* Name des ausgeführten Programms */
  int ti_pid; /* Prozessnummer */
}
```


Kommunikation in PVM mittels vorhandener Kommunikationsoperationen

<u>Schritte einer einfachen Sendung</u>

1) `int pvm_initsend(int encoding) ;`

- `enconding` spezifiziert das verwendete Datenformat:
  `PvmDataDefault` PVM-Datenformat
  `PvmDataRaw` keine Codierung
  `PvmDataInPlace` Sendung aus Nutzerbereich
- Rückgabewert Nummer des Sendepuffers

Funktion bereitet Sendepuffer vor

2) Packanweisungen für unterschiedliche Datentypen

Beispiele `int` und `string`

2a) `int pvm_pkint (int *val, int count, int stride);`

- `val` Datenbereich, der verschickt wird
- `count` Anzahl Elemente (hier vom Typ `int`)
- `stride` Schrittweite mit der Elemente entnommen werden (es können `pvm_pk*` Anweisungen gemischt werden)
- Rückgabewert Status der Routine (< 0 Fehler)

2b) `int pvm_pkstr (char *sp) ;`

- `sp` String, der durch den `NULL character` begrenzt wird

3) `int pvm_send (int dest, int tag) ;`

- `dest` ID Zieltask
- `tag` Identifikator der Nachricht (≥ 0)
- Rückgabewert Status der Routine (< 0 Fehler)

Asynchrones (!) Senden der Nachricht

Empfangen einer Nachricht

Schritte eines einfachen Nachrichtenempfangs

1) `int pvm_recv(int tid, int tag) ;`

- `tid` ID Senderprozess, -1 beliebiger Sender
- `tag` Identifikation der Nachricht, -1 beliebig
- Rückgabewert ID des Empfangspuffers (< 0 Fehler)

blockierendes Empfangen einer Nachricht

2) Entpackanweisungen für unterschiedliche Datentypen

Entpackanweisungen müssen in gleicher Reihenfolge wie Pack-anweisungen aufgerufen werden!

Beispiele: `int` und `string`

2a) `int pvm_upkint (int *ip, int nitem, int stride);`

- Parameter analog zu `pvm_pkint`

2b) `int pvm_upkstr (char *sp) ;`

- Parameter analog zu `pvm_pkstr`

Weitere Kommunikationsoperationen

Sendeoperationen

`int pvm_getsbuf (void) ;`

- Rückgabewert ID des aktiven Sendepuffers
  (0 falls keiner aktiv)

`int pvm_mcast (int *tids, int ntask, int tag);`

- `tids` Liste von Taks IDs
- `ntask` Anzahl Task IDs
- `tag` Identifikator der Nachricht

Broadcast Operation ⇒
an aller Tasks aus `tids` wird die Nachricht im Sendepuffer gesendet

`int pvm_sendsig (int tid, int signum) ;`

- `tid` ID des Empfängers
- `signum` Signalnummer

Sendung des Signals `signum` an den Empfänger


```
int pvm_psend (int dest, int msgtag, char *buf,
               int len, int datatype) ;
```

- `dest` ID des Empfängers
- `msgtag` Identifikator der Nachricht
- `buf` Zeiger auf den zu sendenden Datenbereich
- `len` Länge des Datenbereichs
  (in Anzahl zu sendende Elemente)
- `datatype` Identifikator des Datentyps, vorhanden sind u.a. :
  - PVM_STR string
  - PVM_BYTE byte
  - PVM_SHORT short
  - PVM_INT integer
  - PVM_FLOAT real
  - PVM_DOUBLE double
  - PVM_CPLX complex
  - PVM_DCPLX double complex
  - ...

Kombination von Packen und Senden bei Sendung von Nachrichten, die aus einem Array von Daten eines Typs bestehen

Empfangsoperationen

```
int pvm_bufinfo (int bufid, int *bytes,
             int *msgtag, int *tid) ;
```

- `bufid` ID des Empfangspuffers
- `bytes` Länge der empfangenen Nachricht in Bytes
- `msgtag` Identifikator der empfangenen Nachricht
- `tid` ID Sendeprozess

Informationen über die empfangene Nachricht

```
int pvm_getrbuf (void) ;
```

- Rückgabewert ID des aktiven Empfangspuffers (0 falls keiner aktiv)

```
int pvm_trecv(int tid, int tag,
             struct timeval * tmout) ;
```

- `tid` ID Senderprozess, -1 beliebiger Sender
- `tag` Identifikation der Nachricht, -1 beliebig
- `tmout` Angabe eines Zeitintervalls für den Timeout

- Rückgabewert ID des Empfangspuffers (0 keine Nachricht vorhanden,<0 Fehler)

Empfangen einer Nachricht mit Timeout


```
int pvm_nrecv(int tid, int tag) ;
```

- `tid` ID Senderprozess, -1 beliebiger Sender
- `tag` Identifikation der Nachricht, -1 beliebig
- Rückgabewert ID des Empfangspuffers (0 keine Nachricht vorhanden,<0 Fehler)

nicht-blockierendes Empfangen einer Nachricht
(bisheriger Empfangspuffer wird überschrieben!)

```
int pvm_prob(int tid, int tag) ;
```

- `tid` ID Senderprozess, -1 beliebiger Sender
- `tag` Identifikation der Nachricht, -1 beliebig
- Rückgabewert ID des Empfangspuffers (0 keine Nachricht vorhanden,<0 Fehler)

Testen, ob eine Nachricht empfangen wurde

Im Gegensatz zum nicht-blockierenden Empfangen, kann mehrfach getestet werden!

Sonstige Operationen

`int pvm_freebuf (int bufid) ;`

- `bufid` ID eines Sende- oder Empfangspuffers

Freigabe eines Puffers, geschieht implizit beim Auslesen von Empfangspuffern und Senden aus Sendepuffern

`int pvm_mkbuf(int encoding) ;`

Generiert einen neuen Sendepuffer mit Nachrichtenkodierung wie in `encoding` angegeben

```
int pvm_notify (int what, int msgtag, int cnt,
                int *tids) ;
```

- `what` Ereignis, welches die Benachrichtigung auslöst (`PvmTaksExit`, `PvmHostDelete`, `PvmHostAdd`)
- `mstag` Identifikator der Nachricht
- `cnt` Anzahl Tasks, die benachrichtigt werden
- `tid` Taskliste der Länge `cnt`

Benachrichtigung von Tasks bei Eintritt eines Ereignisses auf PVM-Ebene

Prozessgruppen

PVM erlaubt die Generierung von Prozessgruppen

Tasks können

- sich Prozessgruppen anschließen
- mehreren Prozessgruppen angehören
- Prozessgruppen verlassen
- Kommunikationsoperationen innerhalb einer Gruppe ausführen

Funktionen zur Gruppenverwaltung

`int pvm_joingroup (char *group) ;`

- group Name der Gruppe
- Rückgabewert ID des Tasks in der Gruppe

Hinzufügen des aufrufenden Tasks zur Gruppe group

`int pvm_getinst (char *group, int tid) ;`

- group Name der Gruppe
- `tid` ID des Tasks

- Rückgabewert ID des Tasks mit ID `tid` in der Gruppe

Zugriff auf Prozessnummer eines Tasks

`int pvm_gettid (char *group, int inum) ;`

- group Name der Gruppe
- `inum` ID in der Gruppe
- Rückgabewert ID des Tasks mit Gruppen-ID `inum` (< 0, falls kein Task mit Gruppen-ID `inum` vorhanden)

`int pvm_gsize (char *group) ;`

- group Name der Gruppe
- Rückgabewert Anzahl Tasks in der Gruppe

`int pvm_lvgroup (char *group) ;`

- group Name der Gruppe

Verlassen der Gruppe group

`int pvm_barrier` (char *group, int count) ;

- group Name der Gruppe
- `count` Anzahl Gruppenmitglieder, die an der Barriere synchronisieren müssen (sollen alle Tasks der Gruppe synchronisieren wähle `count`$= -1$)

Blockieren der aufrufenden Tasks bis count Gruppenmitglieder die Funktion aufgerufen haben


```
int pvm_bcast (char *group, int msgtag) ;
```

- group Name der Gruppe
- msgtag Identifikator der Nachricht

Broadcast einer Nachricht innerhalb der Gruppe

Kann auch von Nichtmitgliedern aufgerufen werden

Broadcast berücksichtigt nicht, dass Tasks während der Operation die Gruppe verlassen oder der Gruppe betreten können (asynchrones Verhalten!)

Falls möglich, basiert die Operation auf Broadcast-Operationen der verwendeten Rechnerarchitektur


```
int pvm_reduce (void (*func) (), void *data,
                int count, int datatype,
                int msgtag,char *group, int root);
```

- func Funktion, die auf den globalen Daten ausgeführt werden soll (vordefiniert sind: PvmMin, PvmMax, PvmSum, PvmProduct)
- data Zeiger auf den lokalen Datenbereich (Daten von root werden mit dem Ergebnis überschrieben)
- count Anzahl Elemente im Array data
- datatype Elementtyp
- msgtag Identifikator der Nachricht
- group Name der Gruppe, in der die Akkumulation ausgeführt wird
- root Gruppen ID des Wurzelelements, bei dem das Ergebnis gespeichert wird

Durchführung einer einfachen Akkumulation

Funktion muss von allen Mitgliedern aufgerufen werden

Funktion blockiert nicht, d.h. Tasks sollten während der Operation die Gruppe weder verlassen noch betreten

Ein einfaches Beispiel

(verteilte Realisierung von "Hello world")

```
#include <stdio.h>
#include "pvm3.h"

main()
{
  int cc, tid;
  char buf[100];

   printf("i'm t%x\n", pvm_mytid());
   cc = pvm_spawn
         ("hello_other", (char**)0, 0, "", 1, &tid);
   if (cc == 1) {
    cc = pvm_recv(-1, -1);
    pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
    pvm_upkstr(buf);
    printf("from t%x:  %s\n", tid, buf);
   }
   else
    printf("can't start hello_other\n");
   pvm_exit();
```



```
   exit(0);
}
```


Vom Hauptprogramm aus wird folgender Task gestartet

```
#include "pvm3.h"

main() {
  int ptid;
  char buf[100];

  ptid = pvm_parent();
  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, 1);
  pvm_exit();
  exit(0);
}
```

Bei dieser Realisierung wird der Task auf einem beliebigen Rechner der virtuellen Maschine gestartet

3.4.2 MPI

Message Passing Interface (MPI)

- Standard für *message passing* in verteilten Systemen
- Entwicklung seit 1992
  (inzwischen Version MPI-2 seit 1997)
- ca. 60 Entwickler aus 40 Organisationen unter Federführung des ARPA und NSF

MPI definiert im Gegensatz zu PVM einen Standard und ist keine Softwarebibliothek!
Inzwischen existieren mehrere Implementierungen

- LAM (local area multicomputer) aus Ohio
- mpich von Argonne Labs
- CHIMP (common high-level interface to message passing)

Wir benutzen LAM

- frei verfügbar (siehe Web Seite)
- Teil der Suse Linux Distribution

MPI Programm besteht aus Prozessen,
die Nachrichten austauschen

Zahl und Type der Prozesse werden beim Programmstart festgelegt (keine dynamische Prozessgenerierung!)

Anbindungen an C, C++ und Fortran existieren

Wir betrachten C Anbindung

Wie bei PVM wird für MPI zuerst
eine virtuelle Maschine erzeugt

```
lamboot -v <hostfile>
```

dann können die Programme gestartet werden

```
mpirun -np 4 <pname> <arg>
```

startet 4 Replikationen des Programms `pname` mit den Argumenten `arg`

Funktionen zur Prozessverwaltung

```
int MPI_Init (int *argc, char ***argv) ;
```

initialisiert MPI und muss als erste MPI Funktion aufgerufen werden

```
int MPI_Finalize(void) ;
```

beendet die Anbindung an MPI, danach kann keine MPI Funktion mehr aufgerufen werden

```
int MPI_Com_rank(MPI_com comm, int *rank) ;
```

- comm Angabe des Gruppentyps
  (`MPI_COMM_WORLD` beschreibt alle MPI Prozesse)

- `rank` Rangposition des aufrufenden Prozesses

Ausgabe des Rangs des aufrufenden Prozesses in einer Gruppe (Rang liegt zwischen 0 und Anzahl Prozesse-1)


```
int MPI_Com_size(MPI_comm comm, int *size) ;
```

- comm Angabe des Gruppentyps
- `size` Anzahl Prozesse in der Gruppe

Ein einfaches MPI Programm

```
#include <stdio.h>
#include “mpi.h”
int main (int argc, char **argv)
{
  int rank, size ;
  MPI_Init (&argc, &argv) ;
  MPI_Com_size (MPI_COMM_WORLD, &size) ;
  MPI_Com_rank(MPI_COMM_WORLD, &rank) ;
  printf(“Rank %i of %i\n”, rank, size) ;
  MPI_Finalize() ;
  return 0 ;
}
```

Kommunikationsoperationen

Kommunikation in MPI kann unterschiedliche Semantik haben:

aus Sicht eines Prozesses

- blockierend: Kontrolle wird zurückgegeben nachdem alle lokalen Zustandsänderungen ausgeführt wurden
- nicht-blockierend: Kontrolle wird zurückgegeben bevor alle lokalen Zustandsänderungen durchgeführt wurden

aus Sicht des Systems

- synchron: Sender und Empfänger nehmen gleichzeitig an der Kommunikation teil
- asynchron: Sender kann Daten versenden, ohne dass Empfänger bereit ist, Daten zu empfangen

Einzeltransferoperationen

MPI Kommunikation über Kommunikatoren

(Kommunikator = Menge von Prozessen,
die untereinander Nachrichten austauschen)

Erst einmal betrachten wir Kommunikator `MPI_COMM_WORLD` d.h. alle Prozesse des Programms


```
int MPI_Send (void *smessage, int count,
              MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm) ;
```

- `smessage` Sendepuffer
- `count` Anzahl zu sendende Elemente
- `datatype` Typ der zu sendenden Elemente, alle Elemente haben den gleichen Typ
- `dest` Nummer des Zielprozesses
- `tag` Identifikator der Nachricht
- `comm` Kommunikator (hier `MPI_COMM_WORLD`)

```
int MPI_Recv (void *rmessage, int count,
              MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm,
              MPI_Status *status) ;
```

- `rmessage` Empfangspuffer
- `count` Obergrenze Anzahl empfangbarer Elemente
- `datatype` Typ der zu empfangenden Elemente
- `dest` Nummer des Sendeprozesses

- `tag` Identifikator der Nachricht
- `comm` Kommunikator
- `status` Information über die tatsächlich empfangene Nachricht

Einstellungen zum Empfang einer beliebigen Nachricht
source = MPI_ANY_SOURCE
tag = MPI_ANY_TAG

Statusinformationen nach Aufruf der Funktion

- `status.MPI_SOURCE` Sender der Nachricht
- `status.MPI_TAG` Identifikator der Nachricht
- `status.MPI_ERROR` Fehlercode

Ermittlung der Größe der empfangenen Nachricht durch

```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count_ptr)
```

- `status` wurde vorher durch Aufruf von MPI_Recv erzeugt
- `count_ptr` enthält Anzahl Elemente in der Nachricht

Test, ob überhaupt eine Nachricht vorhanden durch

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

Parameter wie bei MPI_Recev(),
`status` enthält benötigte Informationen

blockierende Operation, d.h. Rückkehr erst, wenn entsprechende Nachricht vorhanden!

Vorteil gegenüber direkter Verwendung von `MPI_Recv`: Puffer wird erst nach Empfang der Nachricht benötigt

MPI Standarddatentypen

MPI-Datentyp	
C-Datentyp	
MPI_CHAR	
signed char	
MPI_SHORT	
signed short int	
MPI INT	
signed int	
MPI_LONG	
signed long int	
MPI_UNSIGNED_CHAR	
unsigned char	
MPI_UNSIGNED_SHORT	

`MPI_PACKED` und `MPI_BYTE` ohne Entsprechung in C

Darüber hinaus können zusammengesetzte Typen definiert werden!

Realisierung eines Nachrichtentransfers

1. Daten werden aus dem Sendepuffer in den Systempuffer kopiert und Nachricht wird generiert
2. Daten werden vom Sender zum Empfänger geschickt
3. Daten werden vom Systempuffer in den Empfangspuffer kopiert

Vorgestellte Operationen realisieren
blockierende und asynchrone Kommunikation

- MPI_Send blockiert, bis Sendepuffer wieder verfügbar ist
- MPI_Receiv blockiert bis Empfangspuffer die angegebene Nachricht enthält

⇒ tatsächliches Verhalten abhängig von MPI Implementierung

- Benutzung von Systempuffern
- Größe der Systempuffer

⇒ Gefahr von Deadlocks und unterschiedlichem Verhalten auf unterschiedlichen Architekturen

MPI stellt sicher, dass

- Nachrichten sich nicht überholen

Sichere Implementierung eines MPI Programms setzt voraus, dass

- Programm ohne Systempuffer korrekt arbeiten würde!

- Programm mit Systempuffern ein deterministisches Verhalten hat

Beispielprogramm

```
#include <stdio.h>
#include <string.h>
#include ‘‘mpi.h’’
int main (int argc, char *argv[]) {
int my_rank, p, source, dest, tag=0;
char msg[20] ;
MPI_status status ;
MPI_init (&argc, &argv) ;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank) ;
MPI_Comm_size (MPI_COMM_WORLD, &p) ;
if (my_rank == 0) {
  strcpy(msg, ‘‘Hello world’’) ;
  MPI_Send (msg, strlen(msg)+1, MPI_CHAR, 1,
            tag, MPI_COMM_WORLD) ;
}
if (my_rank == 1) {
  MPI_Recv(msg, 20, MPI_CHAR, 0, tag,
            MPI_COMM_WORLD, &status) ;
  printf(‘‘%s \n’’, msg) ;
```



```
}
MPI_FINALIZE () ;
}
```

Funktion zum Senden und Empfangen

```
int MPI_Sendrecv (void *smessage, int sendcount,
             MPI_Datatype sendtype, int dest,
             int sendtag, void *rmessage,
             int recvcount, MPI_Datatype recvtype,
             int source, int recvtag,
             MPI_Comm comm, MPI_status *status);
```

- `s-/rmessage` Sende-/Empfangspuffer
- `send-/recevcount` Anzahl zu sendender/maximal zu empfangender Elemente
- `send-/recvtype` Typen
- `dest/source` Nummer des Start-/Zielprozesses
- `send-/recvtag` Identifikatoren der Nachrichten
- comm Kommunikator (hier `MPI_COMM_WORLD`)
- `status` Fehlercode

Sende- und Empfangspuffer müssen verschieden sein

$\Rightarrow$

auch in Systemen ohne Systempuffer sorgt das Laufzeitsystem für einen verklemmungsfreien Ablauf!

Funktion ist beendet, falls Sendepuffer freigegeben und Nachricht empfangen!

Bisherige Kommunikationsoperationen blockierend

$\Rightarrow$ Operation erst abgeschlossen, wenn Sende-/Empfangspuffer wieder benutzt werden können

$\Rightarrow$ u.U. schlechte Ressourcenauslastung

Alternative: Nicht-blockierende Operationen

```
int MPI_Isend (void *smessage, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm
             MPI_Request *request) ;
```

- `request` interne Datenstruktur
- übrige Parameter wie bei `MPI_Send`

Kontrolle kehrt sofort zurück, ohne dass Sendepuffer frei sein muss

```
int MPI_Irecv (void *rmessage, int count,
             MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm,
             MPI_Request *request) ;
```

- `request` interne Datenstruktur
- übrige Parameter wie bei `MPI_Recv`

Kontrolle kehrt sofort zurück, ohne dass Empfangspuffer gefüllt!

$\Rightarrow$Vorsicht beim Schreiben und Lesen der Puffer!!

Falls Zugriff auf Puffer erfolgen soll, erst Aufruf von

```
int MPI_Wait (MPI_Request *request,
               MPI_Status *status);
```

- `request` Operation, auf die sich Funktion bezieht
- `status` wie bei `MPI_Recv`

Funktion blockiert,
bis die zugehörige Kommunikation ausgeführt
(d.h. Puffer wieder beschrieben bzw. gelesen werden kann)

Nicht blockierende Alternative

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

- `request` Operation, auf die sich Funktion bezieht
- `flag true`, falls Kommunikationsoperation beendet, `false` sonst
- `status` wie bei `MPI_Recv`

Funktion blockiert, bis die zugehörige Kommunikation ausgeführt (d.h. Puffer wieder zugreifbar ist)

Übertragungsmodi für Sendeoperationen

- **Standardmodus** Laufzeitsystem legt fest, ob und welche Nachrichten im Systempuffer zwischengespeichert werden
  - Verhalten damit abhängig vom Laufzeitsystem
  - bisher vorgestellte Operationen benutzen diesen Modus
- **Synchroner Modus** Kontrolle kehrt erst zurück, wenn zugehörige Empfangsoperation gestartet wurde
  - Funktionen `MPI_Ssend ()` und `MPI_Issend ()`
    (hat hier nur Auswirkungen auf `MPI_Wait ()`)
- **Puffermodus** MPI stellt Puffer zur Verfügung, in dem abgesendete Nachrichten zwischengespeichert werden
  - Allokation von Pufferplatz durch `MPI_Buffer_attach (void *buffer, int buffersize) ;`
  - Fehler bei Pufferüberlauf
  - Funktionen `MPI_Bsend()` und `MPI_Ibsend()`

Empfangsoperationen nur im Standardmodus

Globale Kommunikationsoperationen

Kommunikationsoperationen, die alle oder einen Teil der Prozesse eines Programms betreffen

In MPI *collective communication* Operationen

Kommunikationsoperation	
MPI-Funktion	
Broadcast	
`MPI_Bcast()`	
Akkumulation	
`MPI_Reduce()`	
Gather	
`MPI_Gather()`	
Scatter	
`MPI_Scatter()`	
Multi-Broadcast	
`MPI_Allgather()`	
Multi_accumulation	
`MPI_Allreduce()`	
Globaler Austausch	
`MPI_Alltoall()`	

Alle globalen Kommunikationsoperationen in MPI sind

- blockierend,
- benutzen den Standardmodus zur Übertragung,
- müssen als globale Operationen von allen Prozessen aufgerufen werden

Einige Beispiele

```
int MPI_Bcast (void *message, int count,
               MPI_Datatype datatype, int root,
               MPI_Comm comm) ;
```

- `root` Initiatorprozess
- übrige Parameter wie bei anderen Kommunikationsoperationen

`message` ist für root Sendepuffer und
für alle anderen Prozesse Empfangspuffer

`MPI_Bcast` ist eine kollektive Operation, d.h.

- alle Prozesse müssen die Funktion mit identischen Argumenten aufrufen
- Broadcast-Nachrichten können nicht per `MPI_Recv()` empfangen werden
- Broadcast-Nachrichten sind ununterscheidbar

Multi-Akkumulationsoperation

MPI stellt nur eine eingeschränkte Variante zur Verfügung, da alle Prozesse die gleichen Daten benutzen

Skizze des Ablaufs

$$P_0 : x_0$$

$$P_1 : x_0 + x_1 + \ldots + x_{p-1}$$

$$P_1 : x_1$$

$$\text{Akk.}(+)$$

$$P_2 : x_0 + x_1 + \ldots + x_{p-1}$$

$$\vdots$$

$$\Longrightarrow$$

$$\vdots$$

$$p_{p-1} : x_{p-1}$$

$$P_{p-1} : x_0 + x_1 + \ldots + x_{p-1}$$

```
int MPI_Allreduce (void *sbuf, void *rbuf,
                   int count, MPI_Datatype datatype,
```



```
                   MPI_Op op, MPI_Comm comm) ;
```

- `sbuf`, `rbuf` Sende- und Empfangspuffer
- op MPI Operation zur Akkumulation
- übrige Parameter wie bei den anderen Operationen

In `sbuf` stellt jeder Prozess seine Daten zur Verfügung

in `rbuf` steht am Ende der Operation das Ergebnis

Mögliche Reduktionoperationen (Auswahl)

Operation	
Darstellung	
MPI_MAX	
Maximum	
MPI_MIN	
Minimum	
MPI_PROD	
Produkt	
MPI_SUM	
Summe	

Anwendungsbeispiel

Vektor-Matrix-Multiplikation zur Berechnung von $Ab = c$ mit spaltenweiser Speicherung der Matrix A

```
int m, local_m, n, p ;
float a[MAX_N][MAX_LOC_M], local_b[MAX_LOC_M] ;
float c[MAX_N], sum[MAX_N] ;
local_m = m/p ;
for (i = 0; i < n; i++) {
  sum[i] = 0 ;
  for (j = 0; j < local_m; j++)
   sum[i] = sum[i] + a[i][j] * local_b[j] ;
}
MPI_Allreduce
   (sum, c, n,MPI_FLOAT, MPI_SUM, comm) ;
```

Jeder Prozessor hat anschließend eine vollständige Kopie des Lösungsvektor in c gespeichert

Gesamtaustausch

```
int MPI_Alltoall (void *sbuf, int scount,
              MPI_Datatype stype, void *rbuf,
              int rcount, MPI_Datatype rtype,
              MPI_Comm comm) ;
```

- `sbuf, rbuf` Sende- und Empfangspuffer
- `stype, rtype` MPI Datentypen
- `scount, rcount` Anzahl Elemente die pro Prozess gesendet/empfangen werden

Alle Prozesse müssen Operation mit identischen Datentypen und identischer Anzahl Elemente aufrufen

Größe der Puffer `scount * p * sizeof(stype)`
wobei p = Anzahl Prozesse

Semantik der Operation entspricht

p Sendeoperationen $(i = 0, \dots, p-1)$

```
MPI_Send(sbuf + i * scount * sizeof(stype),
              scount, stype, i, ...) ;
```

und anschließend p Empfangsoperationen

```
MPI_Recv(rbuf + i * rcount * sizeof(rtype),
              rcount, rtype, i, ...) ;
```


Für Datenblöcke unterschiedlicher Größe existiert
eine Vektorvariante

Deadlocks in MPI

MPI standard legt keine konkrete Implementierung der Funktionen fest

- zugesichert wird eine FIFO Übertragung von Nachrichten zwischen zwei Prozessen
- inwieweit Systempuffer zur Zwischenspeicherung benutzt werden, ist nicht im Standard festgelegt

⇒ Verhalten implementierungsabhängig
insbesondere Gefahr von Deadlocks

<u>Beispiele für das Auftreten eines Deadlocks</u>

Reihenfolge kollektiver Kommunikationsoperationen

```
switch (my_rank) {
case 0:  MPI_Bcast(buf1, count, type, 0, comm) ;
       MPI_Bcast(buf2, count, type, 1, comm) ;
case 1:  MPI_Bcast(buf2, count, type, 1, comm) ;
       MPI_Bcast(buf1, count, type, 0, comm) ;
}
```

<u>Gefahren</u>

1. Laufzeitsystem könnte Broadcast-Operationen mit

unterschiedlicher Wurzel aufeinander beziehen
⇒ Laufzeitfehler

2. Laufzeitsystem könnte Broadcast-Operationen mit gleicher Wurzel aufeinander beziehen ⇒ ohne Systempuffer synchronisieren die Prozesse und es entsteht ein Deadlock

Mischen von globaler und lokaler Kommunikation

```
switch (my_rank) {
case 0:  MPI_Bcast(buf1, count, type, 0, comm) ;
      MPI_Send
          (buf2, count, type, 1, tag, comm) ;
      break ;
case 1:  MPI_Recv(buf2, count, type,
          MPI_ANY_SOURCE, tag, comm, &status) ;
        MPI_Bcast(buf1, count, type, 0, comm) ;
        MPI_Recv(buf2, count, type,
          MPI_ANY_SOURCE, tag, comm, &status) ;
        break ;
case 2:  MPI_Send
          (buf2, count, type, 1, tag, comm) ;
      MPI_Bcast(buf1, count, type, 0, comm) ;
      break ;
}
```

1. Prozess 0 führt einen Broadcast aus und sendet anschließend eine Nachricht an Prozess 1
2. Prozess 1 empfängt eine Nachricht, führt einen Broadcast aus und empfängt anschließend eine weitere Nachricht

3. Prozess 2 sendet eine Nachricht an 1 und führt anschließend einen Broadcast aus

Mögliche Abarbeitungsreihenfolgen

Prozess 1 empfängt erst die Nachricht von Prozess 2

Prozess 0

Prozess 1

Prozess 2

MPI_Recv()

$\Leftarrow$

MPI_Send()

MPI_Bcast()

MPI_Bcast()

MPI_Bcast()

MPI_Send()

$\Rightarrow$

MPI_Recv()

mit und ohne Systempuffer möglich

Prozess 1 empfängt erst die Nachricht von Prozess 0

Prozess 0

Prozess 1

Prozess 2

MPI_Bcast()

MPI_Send()

$\Rightarrow$

MPI_Recv()

MPI_Bcast()

nur mit Systempuffer möglich

⇒ Systempuffer verhindern u.U. Deadlocks,
können aber auch zu nichtdeterministischem
Verhalten führen

Unbedingte Synchronisation von Prozessen durch

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

alle durch comm spezifizierten Prozesse werden blockiert, bis alle Prozesse aus comm die Funktion aufgerufen haben

Prozessgruppen

MPI erlaubt Bildung von

Prozessgruppen := geordnete Menge von Prozessen

- Jeder Prozess hat eindeutige Nummer (rank) in der Gruppe (Nummerierung $0, \ldots, p-1$)
- Prozesse können mehreren Gruppen angehören
- Gruppen definieren Kommunikationsgebiete
- Zu jeder Gruppe gehört ein Kommunikator (opaker Datentyp)
- Generierung von Prozessgruppen durch Mengeoperationen

Vordefinierte Gruppe

MPI_GROUP_EMTPTY leere Gruppe

Erzeugung einer Gruppe zu einem Kommunikator

```
int MPI_Comm_Group
            (MPI_Comm comm, MPI_Group *group) ;
```

liefert Gruppe zum Kommunikator

MPI_Comm_Group
(MPI_COMM_WORLD, &MPI_Group_All) ;

erzeugt Gruppe aller Prozesse in MPI_Group_All

Basisoperationen zur Erzeugung von Gruppen

Mengenoperationen

```
int MPI_Group_union (MPI_Group group1 ,
                     MPI_Group group2,
                     MPI_Group *new_group) ;
```

Gruppe mit allen Elementen aus group1 gefolgt von allen Elementen aus group2, die nicht in group1 sind

```
int MPI_Group_intersection (MPI_Group group1 ,
                            MPI_Group group2,
                            MPI_Group *new_group) ;
```

Gruppe mit allen Elementen aus group1 und group2,
geordnet wie in group1

```
int MPI_Group_difference (MPI_Group group1 ,
                          MPI_Group group2,
                          MPI_Group *new_group) ;
```

Gruppe mit allen Elementen aus group1,
die nicht in group2 sind

Ein- und Ausschließen von Elementen

```
int MPI_Group_excl (MPI_Group group, int p,
            int *ranks, MPI_Group *new_group) ;
```

löscht die Prozesse rank[0],...,rank[p-1] aus group


```
int MPI_Group_incl (MPI_Group group, int p,
            int *ranks, MPI_Group *new_group) ;
```

Aufnahme der Prozesse mit rank[0],...,rank[p-1] aus group in new_group

Abfrage von Gruppeninformation

```
int MPI_Group_size
              (MPI_Group group, int *size) ;
```

liefert die Anzahl der Prozesse in group

```
int MPI_Group_rank
              (MPI_Group group, int *size) ;
```

liefert die Nummer des aufrufenden Prozesses in group (MPI_UNDEFINED, falls Prozess nicht in group)

Kommunikatoren

```
int MPI_Comm_create (MPI_Comm comm ,
                     MPI_Group group,
                     MPI_Comm *new_comm) ;
```

- liefert einen neuen Kommunikator für group
- group muss Teilmenge der Gruppe sein, die durch comm beschrieben wird

```
int MPI_Comm_dup (MPI_Comm comm ,
                  MPI_Comm *new_comm) ;
```

- dupliziert Kommunikator com

- Anwendung Trennung von interner und externer Kommunikation (z.B. bei Bibliotheken)

Prozesstopologien

MPI erlaubt die Definition von virtuellen Prozesstopologien

Zur Unterstützung bestimmter Algorithmen (z.B. auf Gitterstrukturen)

Vorteil: Adressierung eines Prozesses bzgl. der logischen Struktur des Algorithmus

Generierung eines virtuellen Gitters

```
int MPI_Cart_create (MPI_Comm comm , int ndims
                 int *dims, int *periods,
                 int reorder, MPI_Comm *new_comm);
```

- comm Kommunikator ohne Topologie
- ndims Dimension des Gitters
- dims[i] Prozessoren in Dimension i (Produkt der Einträge muss Anzahl Prozessen entsprechen)
- periods[i] = true zyklische Verbindung der Elemente in dieser Dimension, ansonsten keine zyklische Verbindung
- reorder = true Laufzeitsystem darf Prozesse umordnen

- new_comm Kommunikator für die virtuelle Topologie

Erzeugen eines Kommunikators für ein virtuelles Gitter

Beispiel

Sei

- comm Kommunikator mit 12 Prozessen
- dims[0] = 3; dims[1] = 4 ;
- period[0] = period[1] = 0 ;
- reorder = 0 ;

```
MPI_Cart_create (comm, 2, dims, period,
                 reorder, &new_comm) ;
```

Erzeugt die folgende Struktur:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

$\Rightarrow$ Prozess hat zwei Adressen

- logische Nummer in der Prozessgruppe
- kartesische Koordinaten im Gitter

Funktionen zur Übersetzung von kartesischen Koordinaten in logische Nummern (und umgekehrt)

```
int MPI_Cart_rank (MPI_Comm comm , int *coords
                   int *rank);
int MPI_Cart_coords (MPI_Comm comm , int rank
                     int maxdims, int *coords);
```

Ziel der Definition von Topologien:

Vereinfachung der Kommunikation

```
int MPI_Cart_shift (MPI_Comm comm , int dir,
                    int displ, int *rank_src,
                    int *rank_dest);
```

- comm Kommunikator der Topologie
- dir Dimension in der Nachbar ermittelt werden soll
- displ Abstand des Nachbarn (positiv aufsteigende Adresse, negativ absteigende Adresse)
- rank_src Prozess für den aufrufender Prozess Nachbar ist
- rank_dest Prozess der Nachbar des aufrufenden Prozesses ist

Bestimmung der Prozessnummern von Nachbarprozessen


```
int MPI_Cart_sub (MPI_Comm comm,
                  int *remain_dims,
                  MPI_Comm *new_comm);
```

- comm Kommunikator eines Gitters
- remains_dim[i] = true Dimension i bleibt im Untergitter erhalten, false Dimension i wird im Untergitter nicht verwendet
- new_comm Kommunikator des Untergitters, das den aufrufenden Prozess enthält

Unterteilung eines Gitters in Untergitter niedrigerer Dimension

Verwaltungsfunktionen

```
double MPI_Wtime(void) ;
```

Messung der (absoluten) Zeit seit einem Punkt in der Vergangenheit

```
zeit = MPI_Wtime() ;
part_to_measure() ;
zeit = MPI_Wtime() - zeit ;
```

liefert in zeit die absolute Zeit, die für die Ausführung von part_to_measure() benötigt wird


```
double MPI_Wtick(void) ;
```

Auflösung der Zeit zwischen zwei Takten in Sekunden

```
int MPI_Abort(MPI_Comm, int error_code) ;
```

beendet alle Prozesse des Kommunikators comm