Verteilte Systeme
(Betriebssysteme II)

Kapitel 5.1: Parallele Programmiermodelle

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

Programmiermodelle für das parallele Rechnen

- Klassifikation
- Grundlagen parallelisierender Compiler
- OpenMP (für shared memory)
- High Performance Fortran
- Message Passing mit MPI
- Message Passing mit PVM

Programmiermodelle für das parallele Rechnen

- Probleme beim parallelen Rechnen
  - Vielzahl nebenläufiger Ablaufeinheiten
  - komplexe Kommunikationsmuster
- Programmierung mit Hilfe von *Programmiermodellen*
- Trennung von Programmierung und Implementierung
  - mächtige und ausdrucksstarke Primitive erleichtern parallele Programmierung
- Modelle implementiert als *Middleware*
  - effiziente Implementierung auf unterschiedlicher paralleler Hardware

  → abstrakte parallele Maschine

Klassifizierung von Programmiermodellen

- Kriterien für Abstraktionsmaß
  - Zerlegung in Teilprobleme *(Decomposition)*
  - Zuordnung von Teilproblemen zu Prozessoren *(Mapping)*
  - Kommunikation *(Communication)*
  - Synchronisation *(Synchronization)*

Klassifizierung von Programmiermodellen

- Abstraktionsebenen (von hoch zu niedrig)
  5. Parallelität tritt im Programm nicht explizit auf
  4. Auszeichnung möglicher Parallelität im Programm
  3. explizite Zerlegung in Teilprobleme
  2. Zerlegung in Teilprobleme, Zuordnung zu Prozessoren
  1. Zerlegung in Teilprobleme, Zuordnung zu Prozessoren, Kommunikation der Teilprobleme
  0. alle Teilaspekte der Parallelisierung treten explizit im Programm auf

Beispiele für parallele und verteilte Programmiermodelle

Ebene	Programmiermodell
5 fully automatic	Parallelisierende Compiler, rein funktionale Programmierung (parallele Reduktion), Logik basierte Programmierung (AND und OR-Parallelität), Datenflusssprachen
4 exhibit	Parallelismus-Direktiven: „future“ Konstrukt in Multilisp, High Performance Fortran (HPF)
3 distribute	Bulk Synchronous Parallelism (BSP)
2 distribute+map	Tupelräume in Linda, Java Spaces
1 distribute + map + communicate	Aktive Nachrichten, Mentat Programming Language (MPL)
0 fully manual	Message Passing Interface (MPI)

Überblick: Ebenen der Parallelität

- **Externe Programmparallelität:** Es werden verschiedene Applikationen parallel ausgeführt.
- **Interne Programmparallelität:** Parallelität wird durch geeignete Programmierkonstrukte oder Tools **explizit** innerhalb eines Programms festgelegt.
- **Parallelität auf Instruktionsebene:** Einzelne Maschineninstruktionen werden parallel ausgeführt.
- **Parallelität auf Bit-Ebene:** Datenbits werden zu Datenworten zusammengefasst und parallel verarbeitet.

Überblick: Interne Programmparallelität

- Parallelität tritt innerhalb einer einzelnen Applikation auf.
  - Parallelität muss vom Programmierer explizit definiert werden.
  - Neue Fehlerquellen durch Nicht-Determinismus.
- Anwendungen:
  - Beschleunigung einer Berechnung
  - Abbildung externer paralleler Abläufe
- Für UMA / NUMA Architekturen wird auf dieser Ebene i.d.R. das **Multithreading** Programmiermodell verwendet.
- Für NORMA Architekturen wird i.d.R. das **Message Passing** Programmiermodell verwendet.
  - Message Passing kann auch für NUMA (sinnvoll) eingesetzt werden, da der Programmierer gezwungen wird, auf Lokalität der Daten zu achten.

Interne Programm-Parallelität

- Ziel: Parallelisierung durch Compiler
  - Parallelisierung von For-Schleifen möglich
  - alles Weitere ist schwierig bis unmöglich
- Probleme:
  - Compiler für konkrete Hardware vorhanden?
  - For-Schleife „schwer“ (lohnend) genug?
  - Abhängigkeiten innerhalb der Schleife überwindbar?
  - Compiler erprobt und nachweislich korrekt?

Parallelisierende Compiler (2000)

- Parallelizing compilers have become sophisticated and are now quite good at producing efficient, shared-variable parallel programs. This is especially true for scientific programs that have many loops and many time-consuming numerical calculations. However, it is far more difficult to produce good message-passing programs, because there are many choices for program structure and communication. Moreover, it is difficult to parallelize sophisticated sequential programs, such as multigrid and Barnes-Hut.

  – Gregory Andrews, Univ. Arizona [Andr00]

Parallelisierende Compiler (2005)

- Implicitly parallelizing compilers can help a little, but don't expect too much; they can't do nearly as good a job of parallelizing your sequential program as you could by turning it into an explicitly parallel and threaded version.
  - Herb Sutter, Microsoft [Sutt05]
- Program analysis is an imprecise discipline, and sufficiently complex programs are impossible for compilers to understand and restructure.
  - Herb Sutter, James Larus, Microsoft [SuLa05]

Dependence Analysis

- Gegeben Programm **{S1; S2}**
- **Daten-Abhängigkeit** (data dependence) of S2 on S1
  - S1 und S2 greifen auf gemeinsame Speicherstelle zu, sodass Reihenfolge {S1; S2} zwingend ist.
- **Flow dependence** S1 → S2
  - S1 schreibt ein x, das S2 liest
- **Antidependence** S1 ← S2
  - S2 schreibt ein x, das S1 liest
- **Output dependence** S2 ⇑ S1
  - S2 schreibt ein x, das S1 auch beschreibt.

Dependence Analysis

```
for [i = 1 to n] {
S1:     sum = 0.0;
        for [j = 1 to i - 1]
S2:             sum = sum + LU[ps[i], j] * x[j];
S3:     x[i] = b[ps[i]] – sum;
}
```

- Straight line dependencies
  - S1 → S2
  - S1 → S3 and S2 → S3
- Dependencies within loops
  - S2 → S2, “S2 carries a flow dependence via sum”
  - S3 → S2, via x[i]

Dependence Analysis

- **Einfach** für straight-line programs with scalar variables
- **Schwierig** für Schleifen und Array-Zugriffe
- **NP-hart** schon mit Schleifen und Array-Zugriffen nur durch lineare Funktionen, ohne Pointer

- → Man kann heute viel machen,
  - aber im Allgemeinen ist es hoffnungslos

- Interessante Entwicklung: Progr.-Sprache Fortress
  - SUN Microsystems gefördert von DARPA
  - “Fortress does for Fortran what Java did for C”
  - Standard: Schleifen-Parallelität, array bounds checks, virtual machine …

Goals of Loop Transformations

- Expose parallelism
  - Break dependencies
- Increase efficiency
  - different memory access pattern
  - better (larger) grain size

Popular Loop Transformations

➢ Loop interchange	Interchange inner and outer loop
➢ Privatization	Give each process a copy of a variable
➢ Scalar expansion	replace a scalar by an array
➢ Loop distribution	Split one loop into two separate ones
➢ Loop fusion	combine two loops into one
➢ Loop unrolling	unroll loop resulting in fewer iterations
➢ Strip mining	divide iterations of one loop into two nested loops
➢ Loop blocking (tiling)	Divide iteration space into rectangular blocks
➢ Loop skewing	Expose wavefront parallelism by new loop bounds

Privatization

```
for [i = 1 to n]
        for [j = 1 to n] {
                sum = 0.0;
                for [k = 1 to n]
                        sum = sum + a[i,k] * b[k,j];
                c[i,j] = sum;
        }
```

- Dependencies within loops
  - Each loop carries a flow dependence via sum
- Breaking dependencies for outer loops by privatization
  - give each loop (thread) a private copy of sum

Scalar expansion

```
for [i = 1 to n]
        for [j = 1 to n] {
                sum = 0.0;
                for [k = 1 to n]
                        sum = sum + a[i,k] * b[k,j];
                c[i,j] = sum;
        }
```

- Breaking dependencies for outer loops by scalar expansion
  - introduce array sum[1 : n]
  - parallelize outermost loop: use sum[i] instead of sum
    - compute rows of result matrix in parallel
  - parallelize second loop: use sum[j] instead of sum
    - compute columns of result matrix in parallel
  - parallelize both loops with array sum[1:n, 1:n]
    - compute elements of result matrix in parallel (use c[i,j] for sum[i,j])

Loop distribution / fusion

```
for [i = 1 to n]
        for [j = 1 to n]
                c[i,j] = 0.0;
for [i = 1 to n]
        for [j = 1 to n]
                for [k = 1 to n]
                        c[i,j] = c[i,j] + a[i,k] * b[k,j]
```

- Expose more parallelism by loop distribution
  - twice as many threads above
  - smaller grain size
- Increase grain size by loop fusion

Strip mining

```
for [i = 1 to n by 2]        // two rows in each strip, half as many iterations
        for [j = 1 to n]     // n columns in a strip
                for [k = 1 to n] {              // cover both rows in each strip
                    c[i,j] = c[i,j] + a[i,k] * b[k,j];          // first row
                    c[i+1,j] = c[i+1,j] + a[i+1,k] * b[k,j];  // second row
                }
```

- Increase grain size by strip mining
  - c[i,j] access also caches c[i+1,j] with column-major storage
  - b[k,j] cached once per inner loop execution

Parallelisierung – no quick fix

- I would be panicked if I were in industry. Now I'm forced into an approach that I haven't laid the groundwork for, it requires a lot more software leverage than the previous approaches, and the microprocessor manufacturers don't control the software business, so you've got a very difficult situation. It's far more important now to be engaging the universities and working on these problems […]. Unfortunately, we're not going to find a quick fix.

  John Hennessy (Stanford), David Patterson (Berkeley)

OpenMP

- Standardisiertes API zur Programmierung von Parallelrechnern mit gemeinsamem Adressraum.
  - Unterstützte Sprachen: C/C++ und Fortran.
- OpenMP befindet sich auf höherer Abstraktionsebene als die Thread APIs modernen Betriebssysteme.
  - POSIX Threads: Systemprogrammierer
  - OpenMP: Anwendungsprogrammierer
- OpenMP Standard definiert:
  - Compiler Direktiven
    `#pragma omp directive [clause list]`
    - **`directive`**: Name der Direktive
    - **`clause list`**: Liste von Klauseln
  - Bibliotheksaufrufe
  - Umgebungsvariablen

OpenMP: Die parallel Direktive

```
#pragma omp parallel [clause list]
  { ... /* parallel region */ ... }
```

- OpenMP Programme werden ab dem Auftreten einer **`parallel`** Direktive parallel ausgeführt.
  - Es wird eine Gruppe von Threads erzeugt welche jeweils den nachfolgenden Code-Block ausführen.
  - Der aufrufende Thead wird zum Master-Thread der Gruppe.
- Am Ende der parallelen Region findet eine Barrier-Synchronisation statt.
  - Master Thread fährt (sequentiell) fort, wenn alle Threads die letzte Instruktion der parallelen Region abgearbeitet haben.

Die parallel Direktive

- Festlegung der Anzahl der erzeugten Threads:
  - Statisch:
    - Klausel **`num_threads(integer expr)`**
  - dynamisch:
    - Systemaufruf: **`omp_set_num_threads(int num_threads)`**
    - Umgebungsvariable: **`OMP_NUM_THREADS`**

Die parallel Direktive: Zugriff auf Variablen des Master-Threads

- **`shared(variable list)`** Klausel
  - Alle Threads der Gruppe arbeiten auf den angegebenen Variablen des Master-Threads.
- **`private(variable list)`** Klausel
  - Jeder Thread arbeitet jeweils auf einer (privaten) Kopie der angegeben Variablen.
- **`firstprivate(variable list)`** Klausel
  - Wie **`private`**, zusätzlich werden die Kopien der Variablen mit den Werten des Master-Threads initialisiert.
- **`reduction(operator: variable list)`** Klausel
  - Alle Kopien der aufgelisteten (privaten) Variablen werden mittels des angegebenen skalaren Operators verknüpft.
  - Das Ergebnis wird den entsprechenden Variablen des Master-Threads zugewiesen.

Übersetzung: OpenMP nach POSIX Threads

```
int a, b;
main() {
   // serial segment
   #pragma omp parallel num_threads (8) private (a) shared (b)
   {
      // parallel segment
   }
   // rest of serial segment
}
```

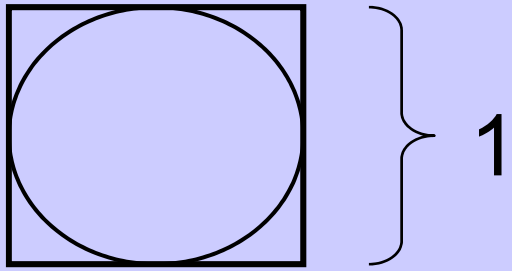
OpenMP Programm

```
int a, b;
main() {
   // serial segment

   for (i = 0; i < 8; i++)
       pthread_create (......., internal_thread_fn_name, ...);

   for (i = 0; i < 8; i++)
       pthread_join (.......);

   // rest of serial segment


}

void *internal_thread_fn_name (void *packaged_argument) {
   int a;

   // parallel segment

}
```

Erzeugter Pthreads Code

Beispiel: Näherungsverfahren zur Berechnung von PI

- Es wird zufällig eine große Anzahl von Punkten in einem Quadrat mit Kantenlänge 1 markiert in das ein Kreis mit Radius 0,5 einbeschrieben ist.
  - Die Fläche des Quadrats beträgt 1.
  - Die Fläche des Kreises beträgt π/4.



- Das Verhältnis der Anzahl der Punkte die im Kreis liegen zur Gesamtzahl der Punkte stellt ein Näherungswert für π/4 dar.

Beispiel: Näherungsverfahren zur Berechnung von PI


```
#pragma omp parallel default(private) shared(npoints)\
            reduction(+: sum) num_threads(8)
{
  num_threads = omp_get_num_threads();
  points_per_thread = npoints / num_threads;
  sum = 0;
  for (i=0; i<points_per_thread; i++) {
    rand_no_x = (double)rand()/(double)RAND_MAX;
    rand_no_y = (double)rand()/(double)RAND_MAX;
    if (((rand_no_x-0.5)*(rand_no_x-0.5) +
        (rand_no_y-0.5)*(rand_no_y-0.5)) < 0.25)
      sum++;
  }
}
```

Überblick: Work-Sharing Konstrukte

- Die **`parallel`** Direktive kann mit den folgenden Direktiven kombiniert werden, um den Threads einer Gruppe unterschiedliche Aufgaben zuzuweisen:
  - **`sections` Direktive:**
    Spezifikation verschiedener, unabhängiger Tasks.
  - **`for` Direktive:**
    Parallele Ausführung von Schleifeniterationen.
- Diese Direktiven werden ignoriert, falls keine **`parallel`** Direktive vorangestellt ist, d.h. die Ausführung erfolgt dann weiterhin sequentiell.

Die sections Direktive

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
    }
}
```

Die for Direktive

```
#pragma omp for [clause list]
    /* for loop */
```

- Mittels der **`for`** Direktive können Schleifeniterationen einer **`for`** Schleife auf einzelne Threads zur parallelen Ausführung verteilt werden.
- Anforderungen an **`for`** Schleife:
  - Schleifeniterationen müssen unabhängig sein
  - Schleife darf keine **`break`** Anweisung enthalten
  - Schleifenvariable muss vom Typ **`int`** sein
  - ...
- Mittels der **`schedule`** Klausel wird die Zuordnung der Schleifeniterationen zu den Threads gesteuert.

Beispielprogramm: Berechnung von Pi

```
#pragma omp parallel default(private) shared(npoints)\
            reduction(+: sum) num_threads(8)
{
  sum = 0;
  #pragma omp for schedule(static)
  for (i=0; i<npoints; i++) {
    rand_no_x = (double)rand()/(double)RAND_MAX;
    rand_no_y = (double)rand()/(double)RAND_MAX;
      if (((rand_no_x-0.5)*(rand_no_x-0.5) +
          (rand_no_y-0.5)*(rand_no_y-0.5)) < 0.25)
     sum++;
  }
}
```

Die Scheduling Klasse „static“

`schedule(static [, chunk_size])`

- Die Schleifeniterationen werden in Blöcke der Größe **`chunk_size`** aufgeteilt.
- Die Blöcke werden in einem round robin Verfahren den Threads (statisch) zugewiesen.
- Ist kein Wert für **`chunk_size`** angegeben, so wird für jeden Thread ein Block gebildet.

Die Scheduling Klasse „dynamic“

`schedule(dynamic [, chunk_size])`

- Die Schleifeniterationen werden in Blöcke der Größe **`chunk_size`** aufgeteilt.
- Die Chunks werden dynamisch freien Threads zugewiesen.
- Ist kein Wert für **`chunk_size`** angegeben, so wird für jede Iteration ein Block gebildet.
- Verwendung z.B. bei
  - Parallelrechnern mit unterschiedlich leistungsfähigen Prozessoren und/oder
  - unterschiedlicher Berechnungskomplexität einzelner Iterationen.

High Peformance Fortran (HPF)

- Datenparallele Programmierung
  - Grundprinzip: Operation parallel auf einzelne oder mehrere Elemente einer regulären Datenstruktur anwenden
- Generierung von SPMD-Programm
  - für Parallelrechner mit verteiltem Speicher
  - Programminstanz bearbeitet Teilmenge der gesamten Datenstruktur
  - *owner rules* Regel
    - Zuordnung von Datenelementen zu Prozessoren
    - Zugriff auf Datenelemente anderer Prozessoren → zusätzliche Kommunikation
- Erweiterungen im Vergleich zu Fortran90
  - Auszeichnung von Parallelität
  - Steuerung der Datenverteilung

HPF – Auszeichnung von Parallelität

- implizite Parallelität: Schleifen

```
do i = 1,m
  do j = 1,n
    A(i,j) = B(i,j) * C(i,j)
  enddo
enddo
```

- explizite Auszeichnung von unabhängigen Schleifendurchläufen mit `INDEPENDENT`-Direktive

```
!HPF$ INDEPENDENT
  do i = 1,n
    A(Index(i)) = B(i)
  enddo
```

HPF – Auszeichnung von Parallelität

- explizite Auszeichnung möglicher Parallelität:
  parallele Operatoren

```
real A(10,20)
real B(10,20)
logical L(10,20)
A = A + 1.0 ! Parallele Berechnung und Zuweisung
A = SQRT(A)
L = A .EQ. B
```

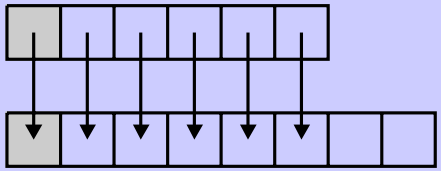
HPF – Datenverteilung

- Anforderungen
  - möglichst hohe Lokalität
  - wenige Kommunikationsaktionen
- Dreistufiges Modell
  - *Kollokation*:
    - `ALIGN`: Auszeichnen von Daten, die gleichartig auf Proz. zu verteilen sind
  - Verteilung auf abstrakte Prozessoren
    - `PROCESSORS`: Array virtueller Prozessoren
    - `DISTRIBUTE`: Verteilung der kollozierten Daten
      - `BLOCK(n)`: Blockweise Verteilung mit Blockgröße von n Elementen
      - `CYCLIC(n)`: Zyklische Verteilung mit Verschiebung um n Elemente
      - `*`: Keine Verteilung in dieser Dimension
  - Zuordnung virtuelle zu realen Prozessoren

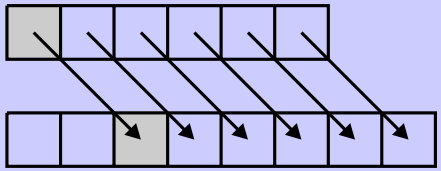
```
real A(10)
real B(10)
!HPF$ ALIGN B(:) WITH A(:)
```

```
!HPF$ PROCESSORS P(64)
!HPF$ PROCESSORS Q(8,8)
    real X(1024,1024)
!HPF$ DISTRIBUTE X(BLOCK,*) ONTO P
```

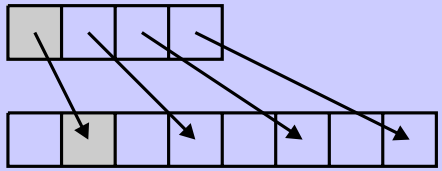
HPF – ALIGN Direktive



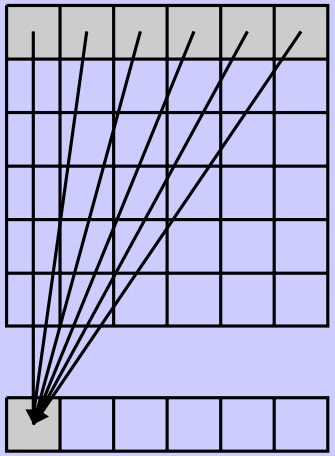
ALIGN A(I) WITH B(I)



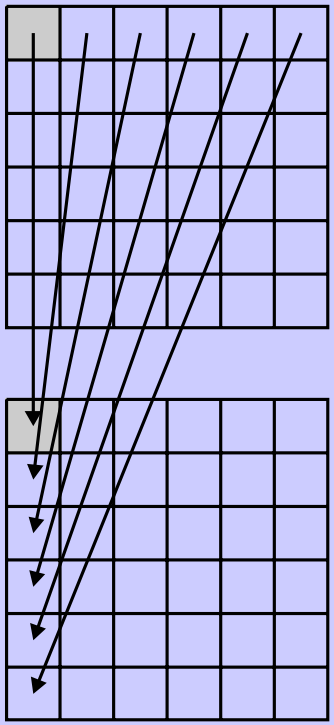
ALIGN A(I) WITH B(I+2)



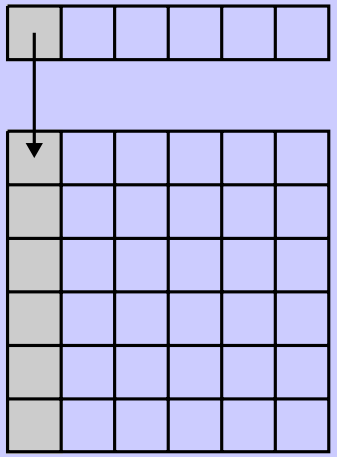
ALIGN A(I) WITH B(2*I)



ALIGN A(:,*) WITH B(:)

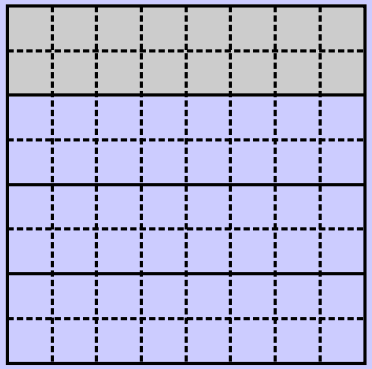


ALIGN A(I,J) WITH B(J,I)

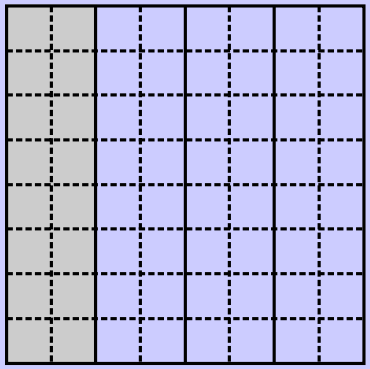


ALIGN A(:) WITH B(*,:)

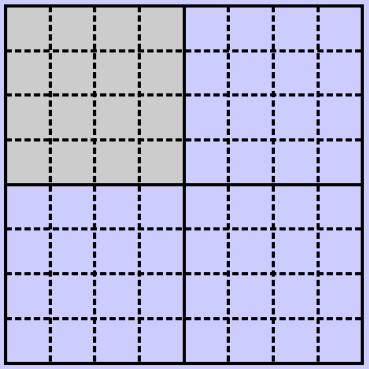
HPF – DISTRIBUTE Direktive



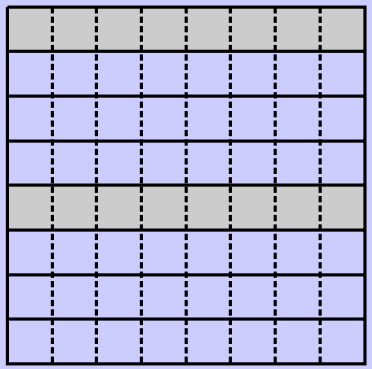
(BLOCK, *)



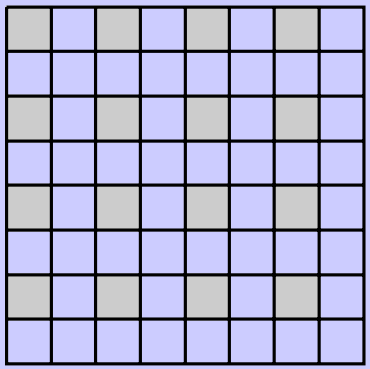
(*, BLOCK)



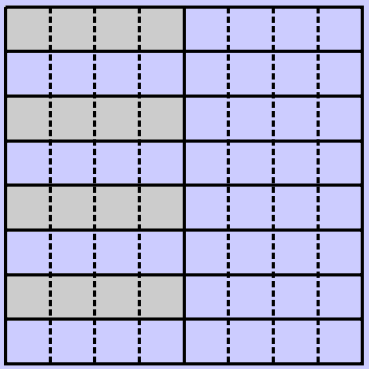
(BLOCK, BLOCK)



(CYCLIC, *)



(CYCLIC, CYCLIC)



(CYCLIC, BLOCK)

- Verschiedene mögliche Verteilungsschemata für ein Array der Größe 8x8 und 4 virtuellen Prozessoren
- Die dem ersten Prozessor zugeordneten Datenelemente sind grau dargestellt

Message Passing und MPI

- Menge von kooperierenden Prozessen, die Programme in sequentieller Programmiersprache ausführen
  - üblicherweise C/C++ oder Fortran
  - Programme enthalten Aufrufe von MPI Bibliotheksfunktionen
  - einzelne Prozesse können verschiedene Programme ausführen
- Message Passing Interface (MPI)
  - zur Kommunikation und Synchronisation verteilter Prozesse
  - bibliotheksbasierter Ansatz
  - keine vollständige Softwareinfrastruktur zur Erstellung von verteilten parallelen Programmen
  - z.B. fehlt Unterstützung für Prozessmanagement und Ein/Ausgabe
- am Beginn: Erzeugung einer festen Anzahl von Prozessen
  - ein Prozess je Prozessor
  - Identifizierung der Prozesse durch Integer-Wert

Message Passing und MPI

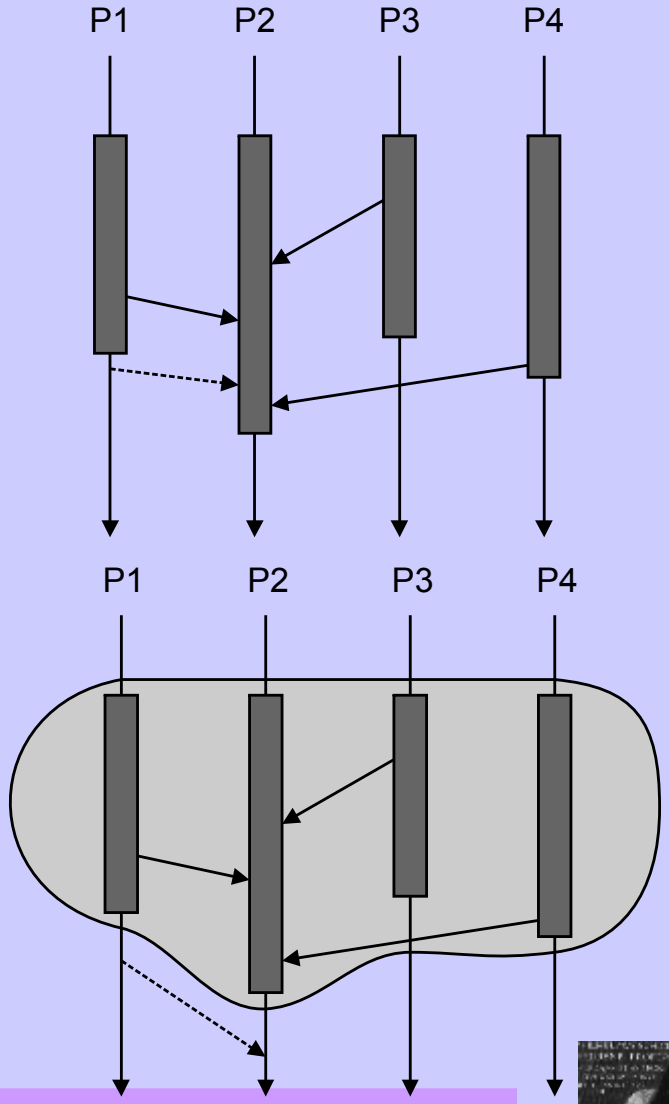
- verschiedene Kommunikationsmodelle vorgesehen
  - *Punkt zu Punkt Kommunikation*
    - Nachrichtenaustausch zwischen zwei bestimmten Prozessen
  - *asynchrone Kommunikation*
  - *Gruppenkommunikation*
    - Zusammenfassung mehrerer Prozesse zu einer Gruppe
    - globale Operationen innerhalb einer Gruppe
    - Verschicken von Broadcast Nachrichten
- MPI Standard
  - Interface Standard: von Herstellern verschieden implementiert
  - weit über hundert Funktionen definiert

MPI – modulare Programmierung

- Kommunikatoren
  - Kontext, in dem Nachrichten ausgetauscht werden
  - Kapselung interner Kommunikation eines MPI-basierten Programmmoduls
- Jede MPI-Funktion, die Kommunikation zwischen Prozessen realisiert, verlangt als Argument einen Kommunikator
  - Empfangsfunktion kann nur Nachricht empfangen, wenn diese im selben Kontext verschickt wurde
- `MPI_COMM_DUP`: dynamische Erzeugung eines neuen Kontextes
  - Argument: Kommunikator
  - Rückgabewert: Kommunikator, der dieselbe Gruppe von Prozessen bezeichnet, aber neuen Kontext aufspannt

MPI – modulare Programmierung

- vertikale Linien
  - zeitlicher Ablauf des Kontrollflusses eines Prozesses
- Pfeile
  - Austausch von Nachrichten
- graue Hinterlegung der Kontrollflusslinie
  - Prozess ist in einer Bibliotheksfunktion, die auch mit MPI realisiert sein soll
- Annahme
  - jeder Prozess führt dasselbe Programm aus
  - Ausführung der Bibliotheksfunktion in den einzelnen Prozessen dauert unterschiedlich lange
- Verwendung von Kontexten
  - Oben: Konflikt der Kommunikation außerhalb und innerhalb der Bibliothek
  - Unten: Modularisierung durch Kommunikator



MPI – Punkt-zu-Punkt-Kommunikation

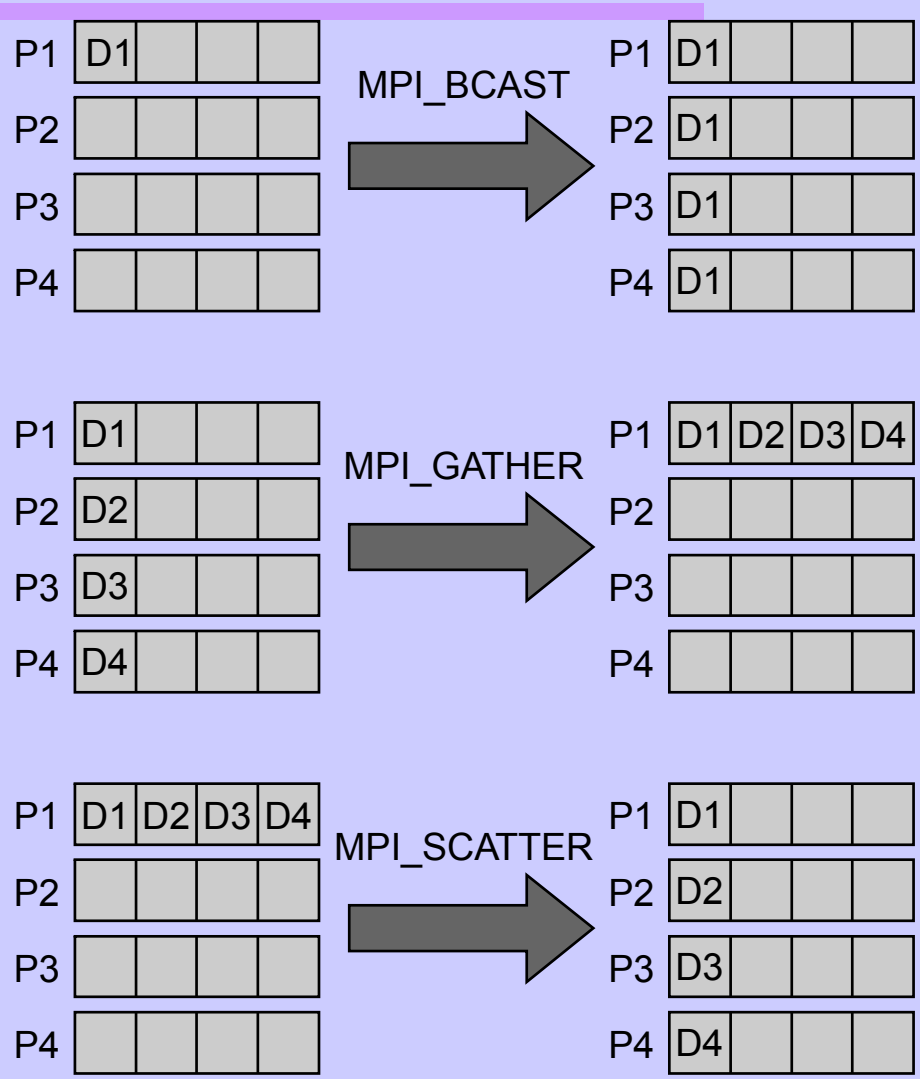
- `MPI_Init`
  - Initialisierung einer MPI Berechnung
  - Funktion muss zu Beginn einer Berechnung (bevor andere MPI Funktionen verwendet werden) genau einmal aufgerufen werden
- `MPI_Finalize`
  - Beenden einer MPI Berechnung.
  - Nach Aufruf kann kein weiterer Aufruf einer MPI Funktion erfolgen
- `MPI_Comm_Size`
  - Bestimmung der Anzahl der an der Berechnung beteiligten Prozesse
- `MPI_Comm_Rank`
  - Bestimmung der ID des eigenen Prozesses

MPI – Punkt-zu-Punkt-Kommunikation

- `MPI_Send`
  - Versenden einer Nachricht
  - Argumente: Datenpuffer, Länge und Datentyp der zu versendenden Daten, ID des Empfängers und ein so genanntes *Message Tag*
- `MPI_Recv`
  - Empfangen einer Nachricht
  - Argumente: Empfangspuffer, Senderadresse und Message Tag
  - Empfang nur solcher Nachrichten, die zur Absenderadresse und zum Message Tag passen
  - Ist keine entsprechende Nachricht vorhanden, blockiert der Aufruf
- `MPI_IProbe`
  - Test ob eine bestimmte Nachricht vorliegt
  - Argumente: ID des Senderprozesses und Message Tag
  - Rückgabewert: booleschen Wert, der angibt ob eine passende Nachricht vorliegt → asynchrone Kommunikation

MPI – globale Operationen

- `MPI_Barrier`
  - Synchronisation der Ausführung aller Prozesse einer Gruppe
  - Kein Prozess der Gruppe kehrt aus Funktion zurück, bevor nicht alle Prozesse der Gruppe sie aufgerufen haben
- `MPI_Bcast`
  - Verschicken von Broadcast Nachrichten
- `MPI_Gather`
  - Einsammeln von Daten
- `MPI_Scatter`
  - Verteilen von Daten
- `MPI_All_Reduce`
  - Auf Daten, die im Eingabepuffer aller Prozesse abgelegt sind, wird punktweise eine anzugebende Operation angewendet und das Ergebnis in die angegebenen Ausgabepuffer aller Prozesse der Gruppe geschrieben.

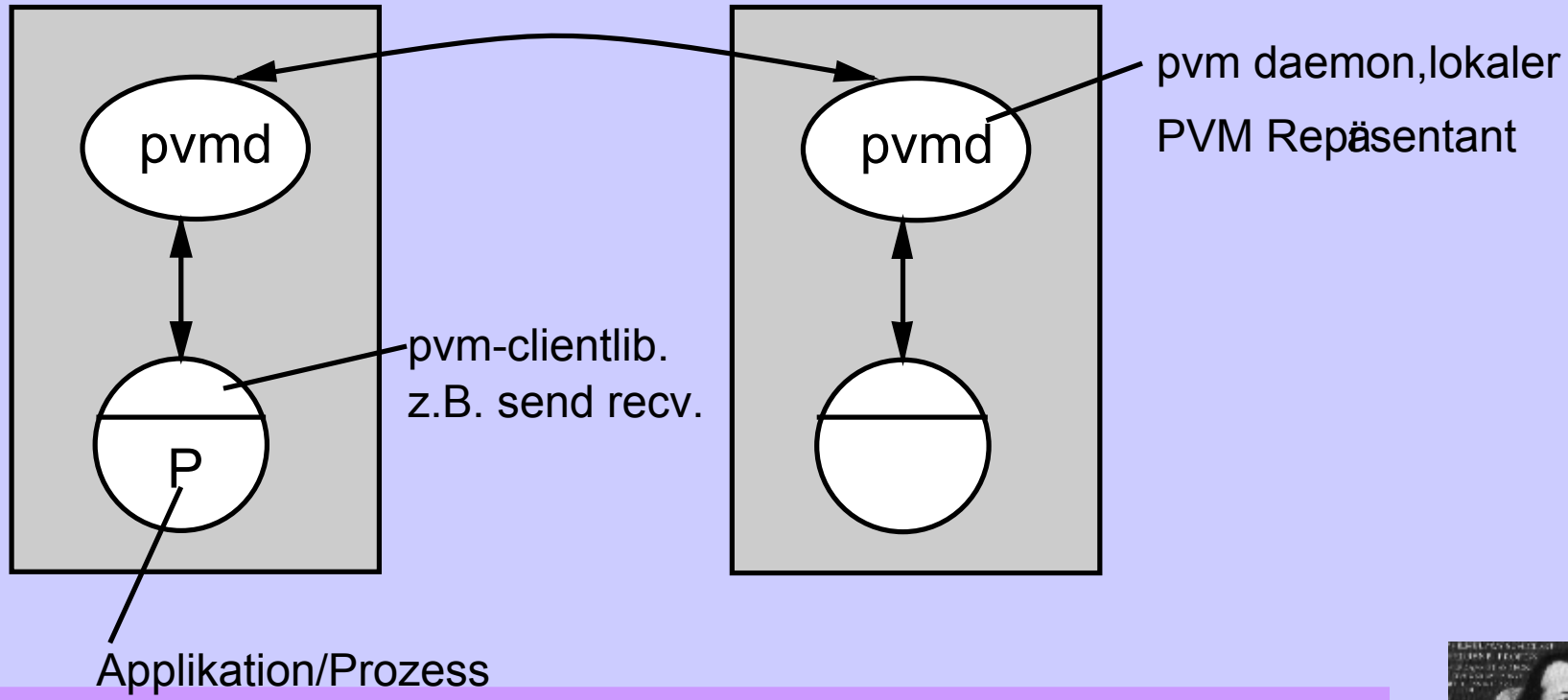


PVM

- Programmierumgebung zum verteilten Rechnen
  - viele interagierende, aber relativ unabhängige Komponenten
    - Nachrichtenaustausch
      - Broadcast
      - Unterstützung maschinenunabhängiger Formate
    - Start neuer Komponenten
      - synchron oder asynchron
      - unter der Bedingung, dass andere Prozesse gestartet oder terminiert wurden
      - unter der Bedingung, dass Daten eingetroffen sind
    - Synchronisation (Mutex, Barrier)
    - *virtual shared memory*
  - Komponenten in verschiedenen Sprachen können über PVM kommunizieren
- entwickelt von Sunderam, Geist und Dongarra (Emory U. und Oak Ridge Natl. Labs); public domain Implementierung
  - Interfaces zu MPI weiterentwickelt und standardisiert

PVM – Architektur

- PVM ist verteilt (ohne zentrale Instanz),
- dynamischer Prozessor-Pool
- unzuverlässiger Datagramm-Dienst ohne Nachrichtenordnung



PVM – Benutzerschnittstelle

- Beispiele für Komponenten
  - Gleichungslösen
  - Visualisierungskomponente
- Komponente ist ein Objekt-File,
  - wird als Benutzerprozess ausgeführt
- Alle Komponenten der Applikation werden in einem Beschreibungsfile zusammengestellt

```
Name      Location  Obj.                      File Arch.
factor    iPSC      /u0/host/factor1          ipsc
factor    msrsun    /usr/alg/math/factor      SUN3
factor    msrsun    /usr/alg4/math/factor     SUN4
...
factor2   iPSC      /u0/host/factor1          ipsc
```

PVM – Benutzerschnittstelle

- PVM-Prozess ist ausführende Instanz einer Komponente
  - Identifikation durch Komponenten-Namen und positive Instanz-Nummer (unabhängig vom Ausführungsort!)
  - erster Prozess manuell gestartet, dann mit `enroll` bei PVM angemeldet
  - weitere PVM-Prozesse mit `initiate` erzeugt

```
enroll ("startup");
for (i=0;i<10;i++)
   instance[i] = initiate("factor");
```

- `initiate`-Befehl
  - Argument: Name der Komponente
  - Rückgabewert: Instanz-Nummer des initiierten Prozesses

PVM – Benutzerschnittstelle

- Beispiel:
  - `entercomp("subproc","sequent&","/usr/a.out","seq");`
    - dynamische Platzierung von Prozessen
  - `initiate("subproc");`
    - statische Platzierung von Prozessen
  - `initiateP("factor","matmul",3);`
    - startet `factor` erst, nachdem Instanz 3 von `matmul` terminiert hat
  - `initiateD("chol","dataset 7");`
    - startet `chol`, nachdem der `dataset 7` event signalisiert wurde durch `ready("dataset 7");`
  - `terminate`
    - terminiert den Prozess
  - `waitprocess`
    - blockiert, bis ein bestimmter Prozess terminiert

PVM – Datentransfer und Barrier-Synchronisation

- Adressierung von Nachrichten
  - <Komponenten-Name, Instanznr.>
  - ortstransparent
- `receive` mit timeout
  - nach Zeit oder nach Anzahl anderer Nachrichten
- `broadcast`

```
/* Sending Process */
initsend(); // Initialize send buffer
putstring("Square root of");
putint(2);
putstring("is");
putfloat(1.414);
// Instance 4 of receiver, Msg. type 99
send("receiver",4,99);
/* Receiving Process */
char msg1[32], msg2[4];
int num; float sqnum;
recv(99); // Receive msg of type 99
getstring(msg1);
getint(&num);
getstring(msg2);
getfloat(&sqnum);
```

PVM – Datentransfer und Barrier-Synchronisation

- Eine Schranke (*barrier*) ist ein Synchronisationsmittel, um eine Menge von n Prozessen versammeln zu können.
- `barrier(n);`
  - blockiert die ersten n-1 Prozesse
  - n-ter Prozess hebt Blockade auf

```
barrier(k);
/* get start time */
/* perform computation */
/* get end time */
barrier(k);
/* compute global maximum of (end-start) */
```

PVM – Shared Memory und wechselseitiger Ausschluss

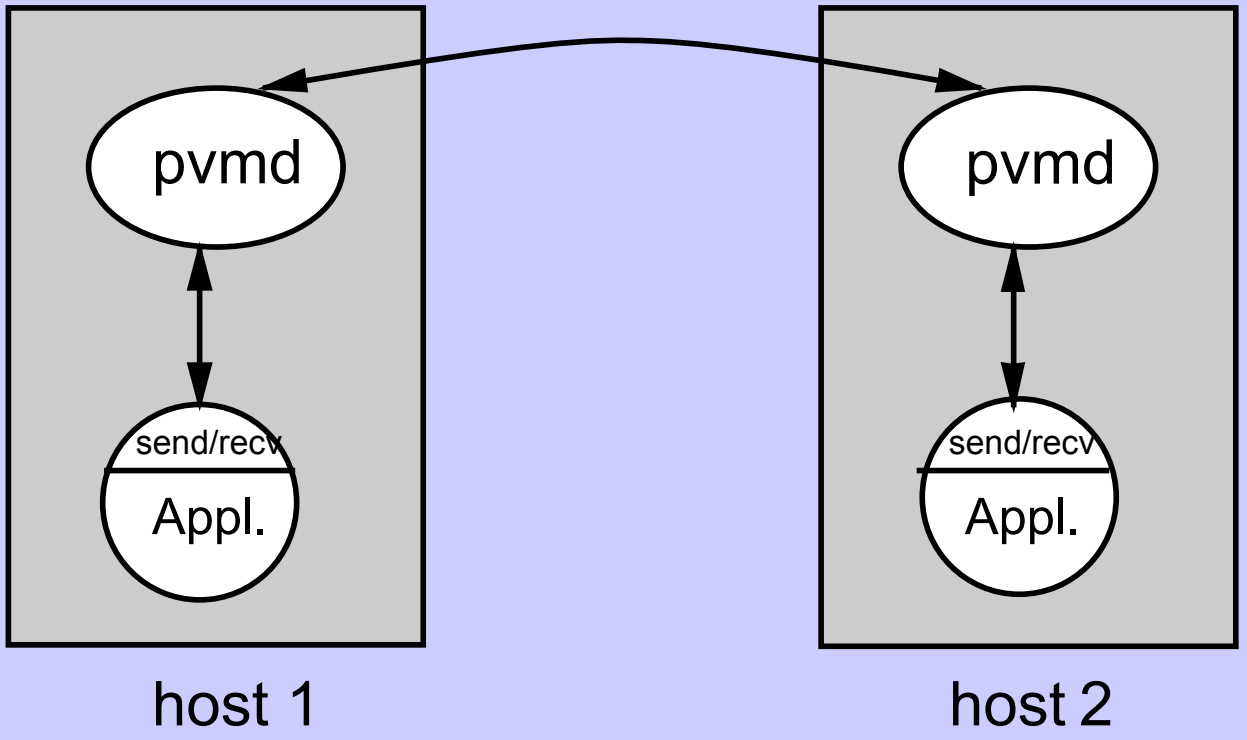
- PVM stellt virtual shared memory zur Verfügung
  - Konstrukte analog zu System V shared memory
- `shmget`
  - alloziert ein Segment
- `shmat`
  - ordnet ein Segment exklusiv einem Prozess zu
- getypte Varianten
  - shmatint und shmatfloat
- `shmdt`
  - hebt die Zuordnung zu einem Segment auf
- `shmfree`
  - deallokiert das Segment

PVM – Shared Memory und wechselseitiger Ausschluss

```
/* Process A */
if (shmget("matrix",1024)) error(); // Allocation failure
while (shmatfloat("matrix",fp,"RW",5));
for (i=0;i<256;i++) *fp++ = a[i]; // fill segment
shmdt("matrix");
/* Process B */
while (shmatfloat("matrix",fp,"R",5)); // Lock & map
for (i=0;i<256:i++) a[i] = *fp++;
shmdtfloat("matrix"); // Unlock & unmap
shmfree("matrix"); // Deallocate
```

- lock("resource-0",5);
  - Exklusivzugriff zu einer durch einen String benannten Ressource
    - 5 ist ein Timeout in sec.
    - erwirbt globales Lock auf eine Einheit mit Namen ”resource-0”

PVM – Kommunikation

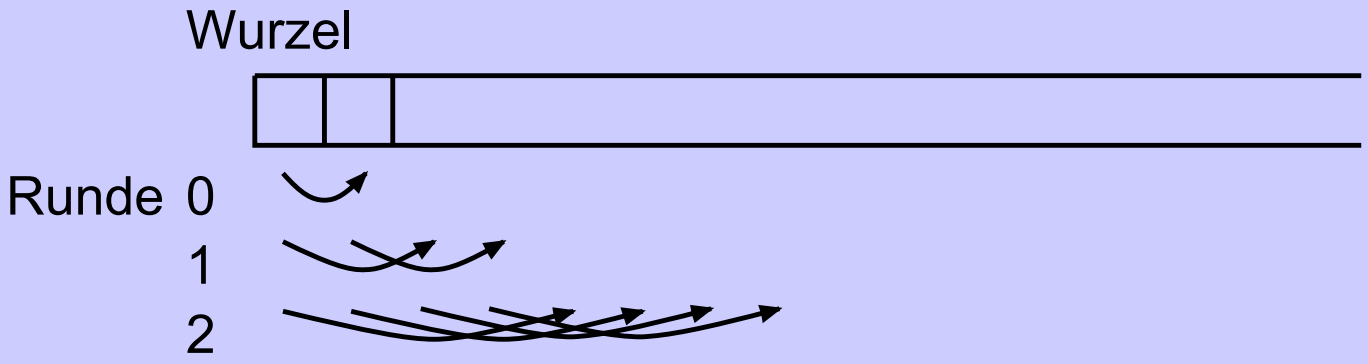


PVM – System-Entwurf und Implementierung

- Kommunikation über UDP mit Hilfe von wellknown-ports
- Kommunikation von Benutzer-Prozess zu Benutzer-Prozess
  - erste Kommunikation via pvmd bei Quelle und Ziel
  - Host und Port-Information wird an Kommunikation angehängt und von lokalen PVM send/recv Routinen gespeichert
    - Informationen erhält pvmd durch enroll
  - weitere Kommunikation unter direkter Verwendung der Host/Port-Adresse
  - Komponenten auf gleichem Host kommunizieren via UDP mit loopback
- Punkt-zu-Punkt-Kommunikation benutzt positives Acknowledgement-Schema mit Sequenznummern und Fragmentierungsinformation
  - Bei fehlendem Acknowledge wird die Nachricht eine gewisse Zeit wiederholt und danach die Komponente für tot erklärt

PVM – Broadcast

- PVM-Broadcast wird implementiert durch Punkt-zu-Punkt-Nachrichten mit **rekursivem Verdoppeln**.
  - Der Pool von Hosts ist mit 0 … p-1 durchnummeriert
  - Broadcast durch Wurzel initiiert
  - Teilnehmerzahl verdoppelt sich in jeder Runde
  - In Runde r übermittelt Prozessor i an Prozessor $(i+2^r)$ mod p
  - Prozessor j nimmt also ab Runde r_j teil
    - r_j = # signifikante Bits in (j – Wurzel) modulo p



PVM – Wechselseitiger Ausschluss

- Bekanntmachung der Lock-Anforderung per Broadcast
- kein Konflikt
  - Anforderer nimmt nach Ende des Broadcasts an, dass er Lock hat
  - Andere pvmd's speichern Tatsache in lokalen Tabellen
- Konflikt: A und B fordern gleiches Lock an
  - Annahme: Jede Kommunikation zwischen Prozessor-Paaren braucht gleich viel Zeit
  - B erhält Anforderung von A: Berechnung der Rundenanzahl, die A's Broadcast fortgeschritten ist
  - Teilnehmer mit höherer Rundenanzahl gewinnt
  - Gleichstand: Prozessor mit kleinerer Nummer gewinnt
  - andere Teilnehmer bestimmen lokal Gewinner und Verlierer
  - Zur Sicherheit: Austausch einer Bestätigung unter A und B
  - keine Übereinstimmung:
    - Höhere Nummer zieht zurück
    - Niedrigere Nummer schickt einen "reset lock" broadcast und fordert Lock neu an

PVM – Initiierung von Prozessen

- lokaler pvmd bestimmt mit Komponentenbeschreibungsdatei Menge möglicher Prozessoren (Hosts)
- Auswahlverfahren
  - Der nächste Kandidat wird round-robin bestimmt, basierend auf lokalen Initiierungen.
  - Hole vom Kandidaten eine Last-Messung ein
  - Ist die Messung kleiner als ein Schwellenwert, dann initiiere auf diesem Host.
  - Andernfalls führe das Verfahren erneut durch. Sind alle Hosts überlastet, wähle davon Host mit kleinster Last.
- lokaler pvmd schickt initiate-Aufforderung an den fernen pvmd
- ferner pvmd führt initiate aus und schickt an alle neue Instanz Nr. der Komponente
- Konfliktbehandlung wie bei Locks
  - Verlierer wählt daraufhin eine höhere Komponenten-Nr.

PVM – Implementierung von Shared Memory

- Erzeugung eines Files für jede shared memory Region
  - möglichst in einem Netzwerk-File-System
- File wird durch PVM Lock geschützt
- attach
  - File wird zum lokalen pvmd kopiert und von dort in den Adressraum abgebildet
- Nach Schreiben
  - File wird zurückkopiert
- Erzeugen, Zugriff, Rückgabe und Zerstören
  - Bekanntmachung durch broadcast