Parallele Programmierung in Java

Prof. Dr. Uwe Kastens

Sommersemester 2000

Ziele und Durchführung

Ziele:

Die Studierenden sollen

- grundlegende **Konzepte** und höhere Paradigmen zum **Entwurf paralleler Programme**,
- **systematische Methoden** zur Entwicklung paralleler Programme und
- Techniken zur **parallelen Programmierung in Java**

erlernen und **praktisch erproben**.

Durchführung:

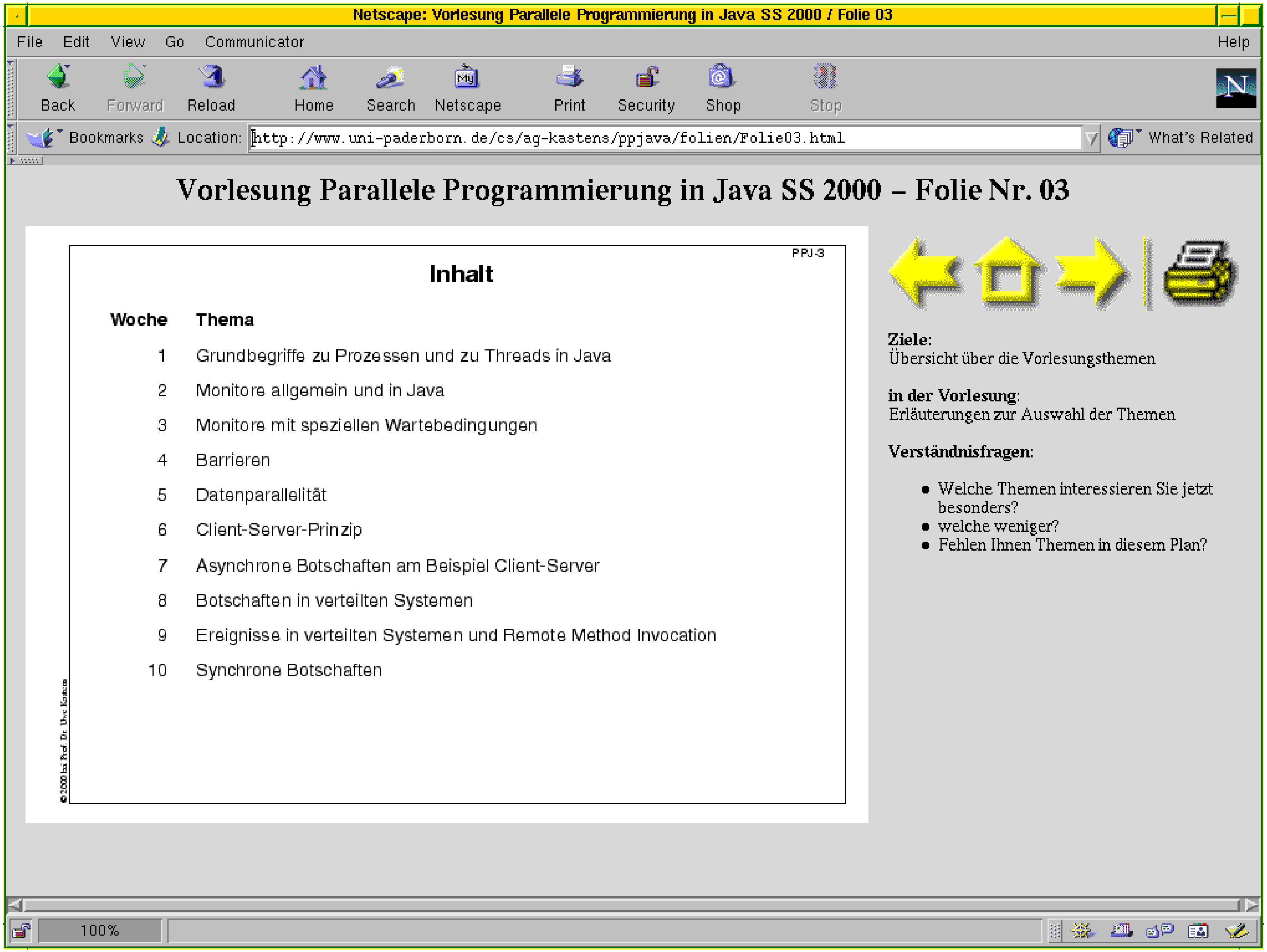
- Vorlesung und Übung an zusammenhängendem Termin
- „fliegender Wechsel" zwischen Vorlesung, Kleingruppenarbeit und Rechnerübung
- Hausaufgaben, um Themen nachzuarbeiten oder vorzubereiten

Inhalt

Voraussetzungen

Thema	**z. B. aus der Vorlesung**
praktische Programmiererfahrung mit Java	Software-Entwicklung (SWE)
Grundbegriffe der parallelen Programmierung:	Konzepte und Methoden der Systemsoftware (KMS), SWE
Prozess, Nebenläufigkeit, Parallelität,	KMS 2-5, 2-9,
verzahnte Ausführung,	KMS 2-10
Adressräume, Threads, Prozesszustände	KMS 2-11, 2-12, 2-18
Monitor	KMS 3-37 bis 3-41
Prozess, Nebenläufigkeit, Parallelität,	SWE-131
Threads	SWE-133 bis 137
Synchronisation, Monitore in Java	SWE-142 bis 150

Elektronisches Vorlesungsmaterial im Web



Literatur

Skript zur Vorlesung „**Parallele Programmierung in Java**“ SS 2000 (wächst mit):
http://www.upb.de/cs/ag-kastens/ppjava

Skript zur Vorlesung „Software-Entwicklung I + II“ WS, SS 1998/1999:
http://www.upb.de/cs/ag-kastens/swei

Skript zur Vorlesung „Konzepte und Methoden der Systemsoftware“ SS 1999:
http://www.upb.de/cs/heiss/lehre/kms/skript.html

Gregory R. Andrews: **Concurrent Programming**, Benjamin Cummings, 1991

Scott Oaks, Henry Wong: **Java Threads**, 2nd ed., O‘Reilly, 1999

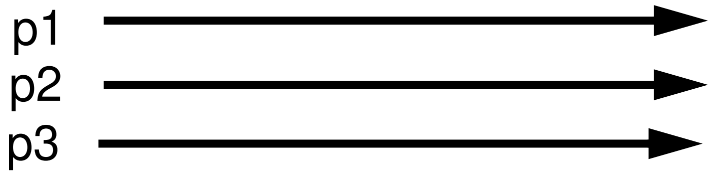
Jim Farley: **Java Distributed Computing**, O‘Reilly, 1998

Doug Lea: **Concurrent Programming in Java**, Addison-Wesley, 1996

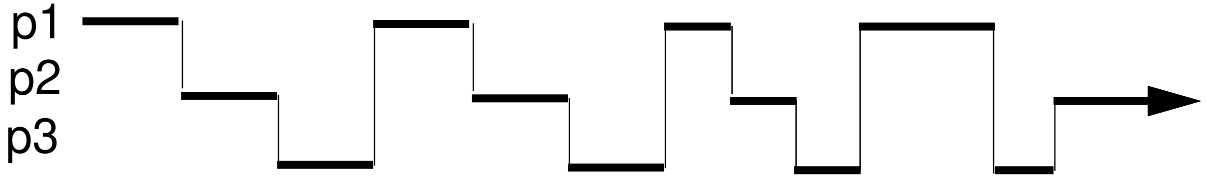
Parallele Prozesse

Prozess:
Ausführung eines sequentiellen Programmstückes in dem zugeordneten Speicher (Adressraum). Veränderlicher Zustand: Speicherinhalt und Programmposition.

parallele Prozesse:
mehrere Prozesse, die gleichzeitig auf mehreren Prozessoren ausgeführt werden.



verzahnte Prozesse:
mehrere Prozesse, die stückweise abwechselnd auf einem Prozessor ausgeführt werden.
Prozessumschaltung durch die Prozessverwaltung oder durch die Prozesse selbst.



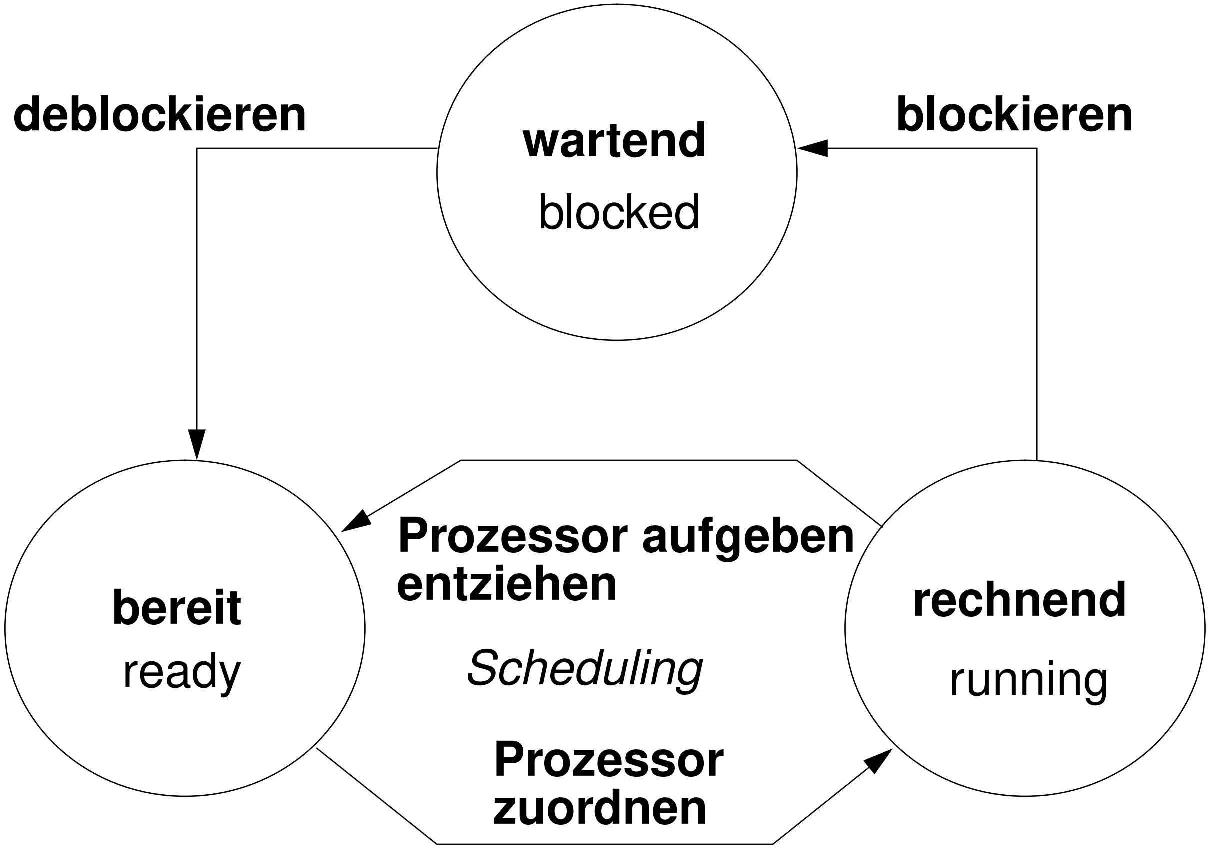
Verzahnte Ausführung kann parallele Ausführung simulieren.
Häufiger Wechsel vermittelt den Eindruck, daß alle Prozesse gleichmäßig fortschreiten.

nebenläufige Prozesse:
Prozesse, die parallel oder verzahnt ausgeführt werden können.

Anwendungen Paralleler Prozesse

- **Benutzungsoberflächen**:
  Die Ereignisse werden von einem speziellen Systemprozeß weitergegeben.
  Aufwendige Berechnungen sollten nebenläufig programmiert werden, damit die Bedienung der Oberfläche nicht blockiert wird.
- **Simulation** realer Abläufe:
  z. B. Produktion in einer Fabrik
- **Animation**:
  Veranschaulichung von Abläufen, Algorithmen; Spiele
- **Steuerung** von Geräten:
  Prozesse im Rechner überwachen und steuern externe Geräte, z. B. Montage-Roboter
- **Leistungssteigerung** durch Parallelrechner:
  mehrere Prozesse bearbeiten gemeinsam die gleiche Aufgabe,
  z. B. paralleles Sortieren großer Datenmengen.

Prozesszustände und ihre Übergänge



siehe KMS 2-17, 2-18

Threads (lightweight processes, Leichtgewichtsprozesse):
Prozesse, die parallel oder verzahnt im gemeinsamen Speicher (Adressraum) ablaufen.
Die Prozessumschaltung ist besonders einfach und schnell.

Java Threads erzeugen - Technik Runnable implementieren

Prozesse, Threads in Java:

nebenläufig im **gemeinsamen Speicher** des Programms (o. Applets)

Objekte von Klassen mit bestimmten Eigenschaften

Erste Technik: Eine **Benutzerklasse implementiert das Interface `Runnable`**:

```
class Aufgabe implements Runnable
{  ...
   public void run ()
   {...}
   public Aufgabe (...) {...}
}
```

`public void run ()` – vom Interface geforderte Methode `run`

`{...}` – das als Prozess auszuführende Programmstück

`public Aufgabe (...) {...}` – Konstruktor

Der Prozess wird als **Objekt der vordefinierten Klasse Thread** erzeugt:

```
Thread auftrag = new Thread (new Aufgabe (...));
```

Erst folgender Aufruf startet dann den Prozess:

`auftrag.start();` Der neue Prozess beginnt, neben dem hier aktiven zu laufen.

Diese Technik (das Interface `Runnable` implementieren) sollte man anwenden, wenn

- der **abgespaltene Prozess nicht weiter beeinflußt** werden muss; also einen Auftrag erledigt (Methode `run`) und dann terminiert, oder
- die Benutzerklasse als Unterklasse einer anderen Klasse definiert werden soll.

Java Threads erzeugen - Technik Unterklasse von Thread

Zweite Technik:
Die Benutzerklasse wird als **Unterklasse der vordefinierten Klasse `Thread`** definiert:

```
class DigiClock extends Thread
{ ...
   public void run ()
   {...}
   DigiClock (...) {...}
}
```

- `public void run ()`: überschreibt die `Thread`-Methode `run`
- `{...}`: das als Prozess auszuführende Programmstück
- `DigiClock (...) {...}`: Konstruktor

Der Prozess wird als Objekt der Benutzerklasse erzeugt (es ist auch ein `Thread`-Objekt):

```
Thread clock = new DigiClock (...);
```

Erst folgender Aufruf startet dann den Prozess:

```
clock.start();
```

der neue Prozess beginnt neben dem hier aktiven zu laufen

Diese Technik (Unterklasse von Thread) sollte man anwenden, wenn der abgespaltene Prozess weiter beeinflußt werden soll;
also weitere Methoden der Benutzerklasse definiert und von aussen aufgerufen werden,
z. B. zum vorübergehenden Anhalten oder endgültigem Terminieren.

Wichtige Methoden der Klasse `Thread`

`public void run ();`

wird überschrieben mit der Methode, die die auszuführenden Anweisungen enthält

`public void start ();`

startet die Ausführung des Prozesses

`public void suspend ();`

hält den angegebenen Prozess an: **`clock.suspend();`**

`public void resume ();`

setzt den angegebenen Prozess fort: **`clock.resume();`**

`public void join () throws InterruptedException;`

der aufrufende Prozess wartet bis der angegebene Prozess terminiert ist:

```
try { auftrag.join(); } catch (Exception e){}
```

`public static void sleep (long millisec) throws InterruptedException;`

der aufrufende Prozess wartet mindestens die in Millisekunden angegebene Zeit:

```
try { Thread.sleep (1000); } catch (Exception e){}
```

`public final void stop () throws SecurityException;`

nicht benutzen! Terminiert den Prozess u. U. in einem inkonsistenten Zustand

Beispiel: Digitale Uhr als Prozess im Applet (1)

Der Prozess soll in jeder Sekunde Datum und Uhrzeit als Text aktualisiert anzeigen.


```
class DigiClock extends Thread
{  public void run ()
   {  while (running)                                     iterieren bis von außen terminiert wird
      {  line.setText(new Date().toString());                                  Datum schreiben
         try { sleep (1000); } catch (Exception ex) {}                                   Pause
      }
   }
                                              Methode, die den Prozess von außen terminiert:
   public void stopIt () { running = false; }
   private boolean running = true;                                           Zustandsvariable

   public DigiClock (Label t) {line = t;}   Label zum Beschreiben übergeben
   private Label line;
}
```

Prozess als Unterklasse von **`Thread`**, weil er

- durch Aufruf von **`stopIt`** terminiert,
- durch weitere **`Thread`**-Methoden unterbrochen werden soll,
- eine andere Oberklasse nicht benötigt.

Beispiel: Digitale Uhr als Prozess im Applet (2)

Der Prozess wird in der **`init`**-Methode der **`Applet`**-Unterklasse erzeugt:

```
public class DigiApp extends Applet
{  public void init ()
   {  Label clockText = new Label ("--------------------------------");
      add (clockText);

      clock = new DigiClock (clockText);                  Prozess erzeugen
      clock.start();                                      Prozess starten
   }

   public void start ()   { clock.resume(); }            Prozess fortsetzen
   public void stop ()    { clock.suspend(); }           Prozess anhalten
   public void destroy () { clock.stopIt(); }            Prozess terminieren

   private DigiClock clock;
}
```

Prozesse, die in einem Applet gestartet werden,

- sollen angehalten werden (**`suspend`**, **`resume`**), solange das Applet nicht sichtbar ist (**`stop`**, **`start`**),
- müssen terminiert werden (**`stopIt`**), wenn das Applet entladen wird (**`destroy`**).

Andernfalls belasten Sie den Rechner, obwohl sie nicht mehr sichtbar sind.

Verzahnung als abstraktes Ausführungsmodell

Zwischen Prozessen, die nicht warten, kann prinzipiell zu **beliebigen Zeitpunkten** umgeschaltet werden. Eine **Scheduling Strategie** schränkt die Beliebigkeit ein.

Beispiel für unterschiedliche Ergebnisse bei unterschiedlicher Verzahnung:
Zwei Prozesse operieren auf einer gemeinsamen Variable:

```
                     konto = 50;

                          a            b              c
                    ___________   ___________    ___________
Prozess1:  t1 = konto; t1 = t1 + 10; konto = t1;

Prozess2:  t2 = konto; t2 = t2 - 5;  konto = t2;
                    ___________   ___________    ___________
                          d            e              f
```

Mit atomaren Zuweisungen *a* - *f* liefert das Programm die gleiche Vielfalt von Ergebnissen wie mit den atomaren Zusammenfassung *<a,b>* und *<d, e>* oder *<b, c>* und *<e, f>*,
denn in *b* und *e* kommen keine globalen Variablen vor.

Atomare Aktionen

Atomare Aktion: Folge von (einer oder mehreren) Operationen, deren interne Zustände nicht beobachtet werden können, weil

- es eine **unteilbare Maschineninstruktion** ist,
- weil globale Variablen nur einmal zugegriffen werden (**AMO**),
- weil durch **Synchronisation** die Verzahnung mit anderen Prozessen eingeschränkt wird.

At-most-once Eigenschaft (AMO):

- **Ausdruck A:**
  A hat höchstens eine Variable v, die von einem anderen Prozess geschrieben wird, und
  v kommt nur einmal in A vor.
- **Zweisung x := A :**
  A ist AMOund x wird nicht von einem anderen Prozess gelesen oder
  x wird von einem anderen Prozess gelesen aber A hat nur Prozess-lokale Variable.
- **Anweisungsfolge:**
  eine Anweisung ist AMO und alle anderen enthalten nur Prozess-lokale Variablen.

Interferenz zwischen Prozessen

Für jede **Aussage C über beobachtbare Zustände** eines Prozesses p muss Nicht-Interferenz NI (Z, C) mit jeder **Zuweisung Z** jedes anderen Prozesses q gelten:

p: {C1} <... atomare Aktion...> {C2} < ... > {C3} < ... > {C4}

NI(Z, C)

q: x = ... y = ...

Nicht-Interferenz NI (Z, C) zwischen Zuweisung Z: x = e in q und Aussage C über p gilt, wenn man schließen kann

$$\{ C \wedge pre(Z) \} \quad Z \quad \{ C \}$$

d. H. die Ausführung von **Z lässt C invariant**, falls die schwächste Vorbedingung von Z pre(Z) überhaupt zulässt, dass Z ausgeführt wird, wenn C gilt.

Kommunikation und Synchronisation paralleler Prozesse

Kommunikation zwischen parallelen Prozessen: Datenaustausch durch

- Benutzung gemeinsamer, globaler **Variable**
  nur im Programmiermodell **gemeinsamer Speicher**
- **Botschaften** in Programmiermodellen **verteilter** oder gemeinsamer **Speicher**
  synchrone Botschaften: Sender wartet auf Empfänger (Sprachen: CSP, Occam, Ada, SR)
  asynchrone Botschaften: Sender wartet nicht auf Empfänger (Sprachen: SR)

Synchronisation paralleler Prozesse:

- **gegenseitiger Ausschluss (mutual exclusion):**
  bestimmte Anweisungsfolgen (kritische Abschnitte) dürfen nicht gleichzeitig von mehreren Prozessen ausgeführt werden.
- **Bedingungssynchronisation (condition synchronization):**
  ein Prozess wartet, bis eine bestimmte Bedingung durch einen anderen Prozess erfüllt wird.

Sprachkonstrukte zur Synchronisation:
Semaphore, Monitore, Bedingungsvariable (Programmiermodell gemeinsamer Speicher),
Botschaften (siehe oben)

Deadlock (Verklemmung):
Einige Prozesse warten wechselweise aufeinander, sind dadurch endgültig blockiert.

Monitor - allgemeines Prinzip

Monitor [T. Hoare 1974, P. Brinch Hansen 1975]

- Programmmodul im Programmiermodell „gemeinsamer Spreicher“, kapselt Daten mit Operationen darauf
- Prozesse rufen Operationen (**Entry Prozeduren**) auf; der Monitor ist **passiv**
- Monitor garantiert **gegenseitigen Ausschluss für die Aufrufe der Entry Prozeduren:** zu jedem Zeitpunkt ist höchstens ein Prozess im Monitor.
- **Bedingungssynchronisation** durch **Bedingungsvariable**; werden lokal im Monitor benutzt.

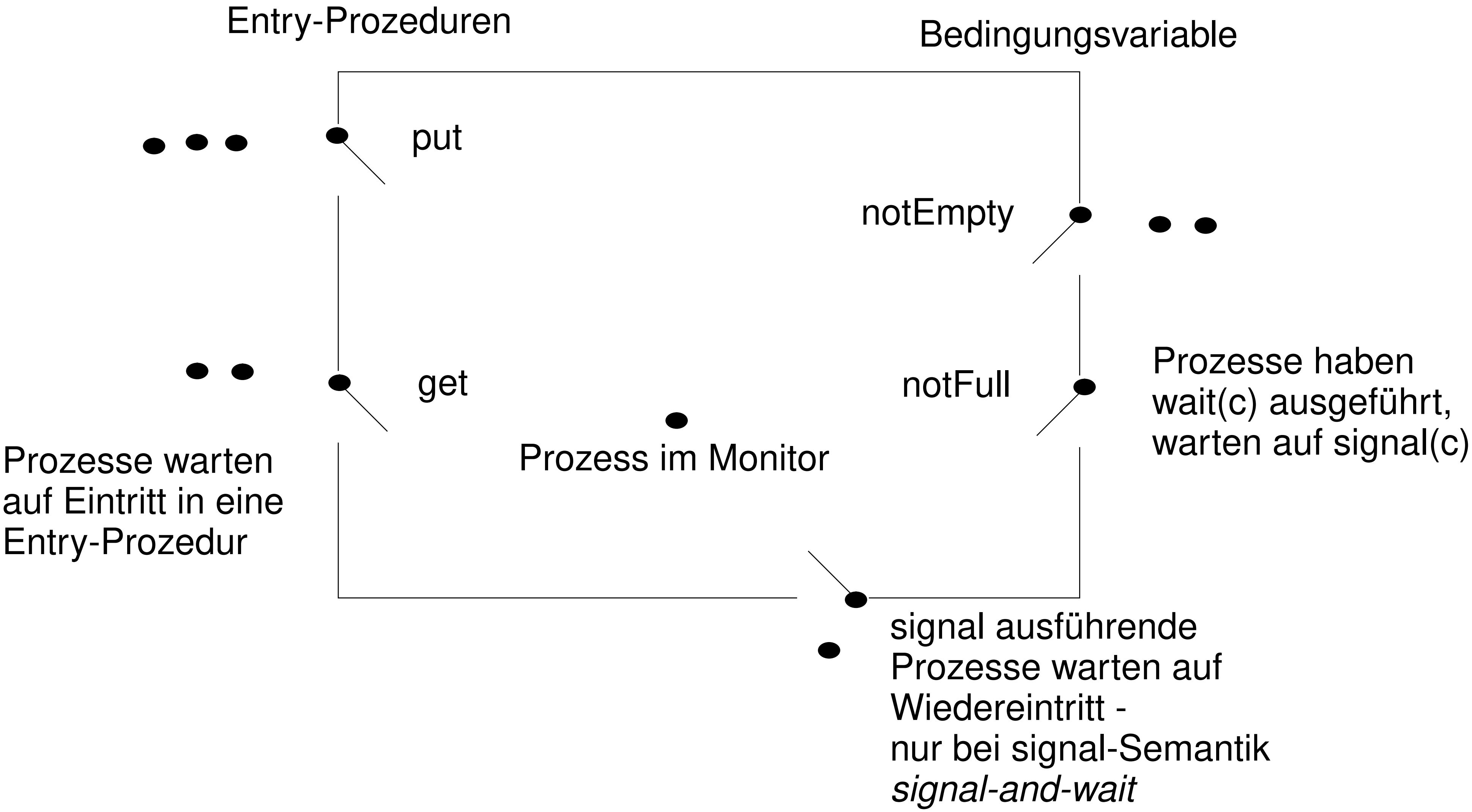
Bedingungsvariable c mit 2 Operationen darauf:

- **wait (c)** ausführender Prozess gibt den Monitor frei,
  wartet in einer Warteschlange zu c auf ein nachfolgendes signal(c),
  setzt dann im Monitor fort.
- **signal(c):** ausführender Prozess gibt **einen** beliebigen der auf c wartenden Prozesse frei;
  setzt dann im Monitor fort (signal-and-continue);
  hat keinen Effekt, wenn keiner auf c wartet.

Beispiel: beschränkter Puffer

```
monitor Buffer
   buff: Queue (k);
   notFull, notEmpty: Condition;    2 Bedingungsvariable: Zustand des Puffers

   entry put (d: Data)
      do length(buf) = k -> wait (notFull); od;
      enqueue (buf, d);
      signal (notEmpty);
   end;

   entry get (var d: Data) x: Data;
      do length (buf) = 0 -> wait (notEmpty); od;
      x := front (buf); dequeue (buf);
      signal (notFull);
   end;
end;

process Producer (i: 1..n) d: Data;
   loop d := produce(); Buffer.put(d); end;
end;

process Consumer (i: 1..m) d: Data;
   loop Buffer.get(d); consume(d); end;
end;
```

Synchronisation im Monitor



Signal-Wait Varianten

Prozesse in Konkurrenz um den Monitor:

- durch wait(c) blockierter Prozess,
- signal(c) ausführender Prozess, der ihn freigibt,
- auf Entry-Prozeduren wartende Prozesse.

signal-and-continue Semantik:
Der signal ausführende Prozess setzt sofort im Monitor fort.
Der freigegebene Prozess muss warten bis der Monitor frei ist.

Er muss die Wartebedingung erneut prüfen - auch wenn sie bei signal galt:

```
do length(buf) = k -> wait(notFull); od;
```

signal-and-wait Semantik:
Der signal ausführende Prozess muss warten bis der Monitor frei ist.
Variante **signal-and-urgent-wait**:
signal-Prozess hat höhere Priorität als Eintritt in Entry-Prozeduren

signal-and-exit Semantik:
Der signal ausführende Prozess verlässt den Monitor;
der freigegebene Prozess kann sofort im Monitor fortsetzen.

Monitore in Java: gegenseitiger Ausschluss

Objekte jeder Klasse können als **Monitor** verwendet werden.

Entry Prozeduren:

Methoden einer Klasse, die kritische Abschnitte auf Objektvariablen implementieren, können als `synchronized` gekennzeichnet werden:

```
class Buffer
{  synchronized public void put (Data d) {...}
   synchronized public Data get () {...}
   ...
   private Queue buf;
}
```

Aufrufe von `synchronized` Methoden von mehreren Prozessen für dasselbe Objekt werden unter **gegenseitigem Ausschluß** ausgeführt.
Synchronisation durch eine interne Synchronisationsvariable des Objektes (lock).

Nicht-`synchronized` Methoden können zu beliebigen Zeitpunkten aufgerufen werden.

Es gibt auch `synchronized` **Klassenmethoden**: sie werden in Bezug auf die Klasse unter gegenseitigem Ausschluss aufgerufen.

Mit `synchronized` **Blöcken** kann man einen kritischen Abschnitt bezüglich eines beliebigen Objektes unter gegenseitigen Ausschluss stellen.

Monitore inJava: Bedingungssynchronisation

Es gibt nur eine einzige Menge mit **`wait`** blockierter Prozesse,
keine deklarierbaren Bedingungsvariable.

Operationen zur Bedingungssynchronisation
müssen aus **`synchronized`** Methoden aufgerufen werden:

- **`wait()`** **blockiert** den aufrufenden Prozess,
gibt das Monitorobjekt frei und
wartet in einer unspezifischen Prozessmenge zu dem Objekt.

- **`notifyAll()`** gibt **alle** Prozesse frei, die mit **`wait`** auf das Objekt warten;
sie konkurrieren dann um den Monitor;
der aufrufende Prozess setzt im Monitor fort (signal-and-continue Semantik).

- **`notify()`** gibt **einen beliebigen** der Prozesse frei, die mit **`wait`** auf das Objekt warten;
der aufrufende Prozess setzt im Monitor fort (signal-and-continue Semantik).
Ist nur verwendbar, wenn alle Prozesse auf dieselbe Bedingung warten.

In Schleife warten, weil bei **signal-and-continue**-Semantik
nach **`notify, notifyAll`** die **Wartebedingung verändert** werden könnte:

```
while (!notFull) try { wait(); } catch (InterruptedException e) {}
```

Monitor-Klasse für Beschränkte Puffer

```
class Buffer
{  private Queue buf;          // Schlange der Länge n zur Aufnahme der Elemente
   public Buffer (int n) {buf = new Queue(n); }

   synchronized public void put (Object elem)
   {                           // ein Produzenten-Prozeß versucht, ein Element zu liefern
      while (buf.isFull()) // warten bis der Puffer nicht voll ist
         try {wait();} catch (InterruptedException e) {}
      buf.enqueue (elem);  // Wartebedingung der get-Methode hat sich verändert
      notifyAll();             // jeder blockierte Prozeß prüft seine Wartebedingung
   }
   synchronized public Object get ()
   {                         // ein Konsumenten-Prozeß versucht, ein Element zu nehmen
      while (buf.isEmpty())                  // warten bis der Puffer nicht leer ist
         try {wait();} catch (InterruptedException e) {}
      Object elem = buf.first();
      buf.dequeue();           // Wartebedingung der put-Methode hat sich verändert
      notifyAll();             // jeder blockierte Prozeß prüft seine Wartebedingung
      return elem;
   }
}
```

Monitor-Invariante

Eine **Monitor-Invariante (MI)** beschreibt **zulässige Zustände des Monitors**.

Sie gilt, immer wenn ein Prozess den Monitor (wieder) betritt:

- nach der Initialisierung (*),
- am Anfang und am Ende (*) von Entry-Prozeduren,
- vor (*) und nach wait-Aufrufen

Beispiel zum beschränkten Puffer:

MI: 0 <= buf.length() <= n

Für die Stellen (*) muss man zeigen, dass MI gilt.
Dabei kann man voraussetzen, dass MI an den anderen Stellen gilt.

Systematischer Entwurf von Monitoren

1. **Monitor-Zustand** definieren und **Entry-Prozeduren ohne Synchronisation** entwerfen.
   z. B. Beschränkter Puffer mit put und get

2. **Monitor-Invariante** formulieren
   z. B.: MI: 0 <= buf.length() <= n

3. **Wartebedingungen** formulieren und **Bedingungssynchronisation** einfügen
   ```
   while (!Cond) try { wait(); } catch (InterruptedException e) {}
   ```
   an den Programmstellen, die Cond && MI voraussetzen müssen, um MI nicht zu verletzen,
   z. B. vor { buf.length()<n && MI } buf.enqueue(); { MI }

   Eigentlich eine Bedingungsvariable c für jede Bedingung Cond - aber es gibt keine in Java.

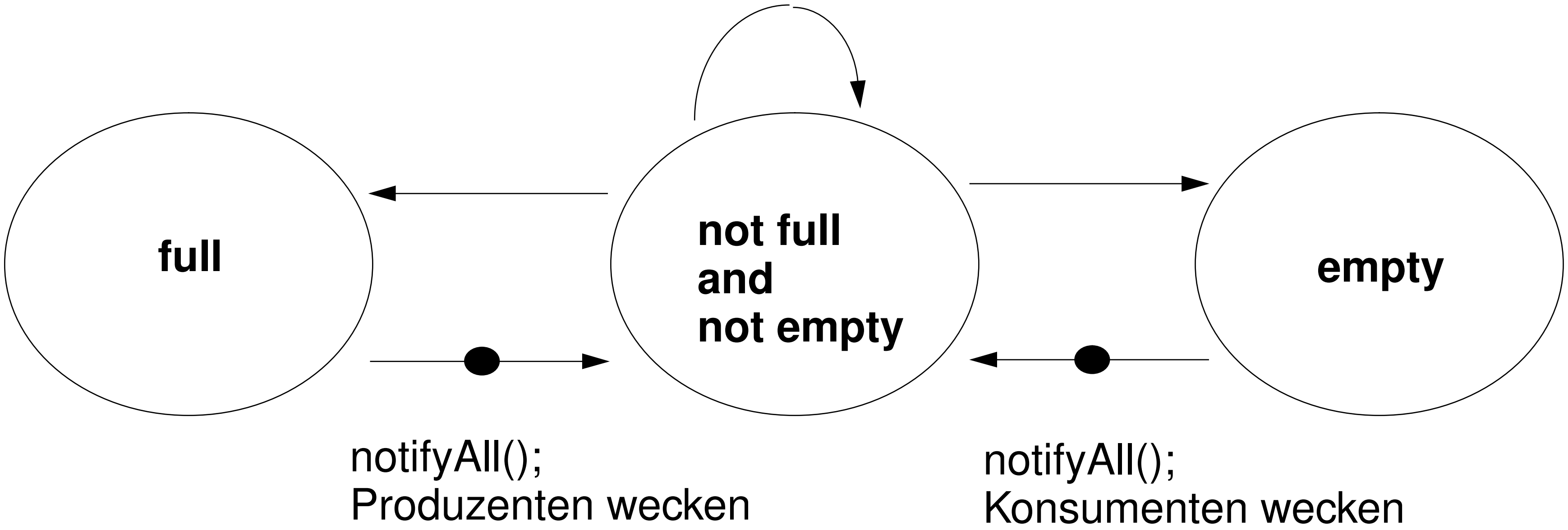
4. **Wecken der Prozesse** einfügen:
   überall wo der Zustand so geändert wird, dass eine Wartebedingung erfüllt sein könnte:
   signal(c)   bzw in Java:
   ```
   notifyAll();
   ```
   z. B. nach `buf.dequeue(); notifyAll();`

5. Ggf. überflüssige Aufrufe von `notifyAll` eliminieren. (PPJ-28)
   Zuviele Aufrufe schaden nicht der Korrektheit - nur der Effizienz.
   Zuwenige Aufrufe können Deadlocks verursachen.

Relevante Zustandsänderungen

Nicht bei jeder Änderung einer Zustandsvariablen Prozesse wecken sondern nur bei relevanten Zustandsänderungen: **irgendeine Wartebedingung könnte wahr werden**.

z. B. im Beschränkten Puffer nicht bei jedem Aufruf von `put` und `get` `notifyAll` aufrufen, sondern **vergröberte Zustände**:



Schema: Gleichartige Ressourcen vergeben

Ein **Monitor** verwaltet eine Menge von k >= 1 **gleichartigen Ressourcen**.
Prozesse fordern jeweils n Ressourcen an, 1<=n<=k, und geben sie nach Gebrauch zurück.

Monitor-Invariante: Die Summe der freien und vergebenen Ressourcen ist k.
MI: avail + inUse = k and 0<= avail and 0<= inUse

Wartebedingung für das Anforden: Sind n Ressourcen verfügbar? avail >= n

Beispiele:

- Verleih von Fahrrädern an Ausflugsgruppen
- Vergabe von Speicherblöcken an Prozesse
- Taxi-Vermietung, immer n = 1

auch **abstrakte Ressourcen**:
z. B. das Recht, eine Brücke mit n Tonnen Gewicht zu befahren

Je nach Aufgabenstellung werden

- die **Ressourcen nur gezählt** oder
- ihre **Identität** (Nummern der Taxis, Speicherblöcke) in einer Datenstruktur verwaltet.

Sonderfall n = 1: Alle Prozesse warten auf die gleiche Bedingung: avail >= 1;
also **`notifyAll()`** ersetzen durch **`notify()`**.

Monitor zur Vergabe von Ressourcen

Ein **Monitor** verwaltet eine Menge von k >= 1 **gleichartigen Ressourcen**.
Prozesse fordern jeweils n Ressourcen an, 1<=n<=k, und geben sie nach Gebrauch zurück.

Annahme: Prozess gibt nicht mehr zurück als er bekommen hat => vereinfachte Invariante:

```
class Resources
{  private int avail;                                  // Invariante: avail >= 0

   public Resources (int k) { avail = k; }

   synchronized public void getElems (int n)           // n Elemente abgeben
   {  while (avail<n)                                  // negierte Wartebedingung
         try { wait(); } catch (InterruptedException e) {}
      avail -= n;
   }

   synchronized public void putElems (int n)           // n Elemente zurückgeben
   {  avail += n;                                      // wegen Annahme ist Warten nicht nötig
      notifyAll();                                     // nicht notify()!
   }
}
```

Prozesse und Hauptprogramm zu Ressourcen-Monitor

```
import java.util.Random;

class Client extends Thread
{  private Resources mon; private Random rand;
   private int ident, rounds, maximum;

   public Client (Resources m, int id, int rd, int max)
   {  mon = m; ident = id; rounds = rd; maximum = max;
      rand = new Random();                  // Zufallszahlengenerator bestimmt die
   }                                        // in jeder Runde angeforderten Elemente

   public void run ()                       // und die Wartezeit bis zur Rückgabe
   {  while (rounds > 0)
      {  int m = Math.abs(rand.nextInt()) % maximum + 1;
         mon.getElems (m);
         try { sleep (Math.abs(rand.nextInt()) % 1000 + 1); }
            catch (InterruptedException e) {}
         mon.putElems (m);
         rounds--;
      }
   }
}
```

```
public class TestResource
{  public static void main (String[] args)
   {  int avail = 20;
      Resources mon = new Resources (avail);
      for (int i=0; i<5; i++)
         new Client (mon, i, 4, avail).start();
   }
}
```

Leser/Schreiber-Schema

Ein Monitor verwaltet lesende und schreibende Zugriffe von Prozessen auf eine Datenbasis.

Entwurfsschritte:

1. **Zustand**: Zahl der gerade lesenden (nr) und schreibenden (nw) Prozesse
   Operationen: requestRead, releaseRead, requestWrite, releaseWrite

2. **Monitor-Invariante** RW: (nr == 0 or nw == 0) and nw <= 1

3. **Wartebedingungen** vor Erhöhen der Zähler:
   in requestRead: { nw == 0 and RW } nr++; { RW }
   in requestWrite: { nr == 0 and nw == 0 and RW } nw++; { RW }

   verschiedene, überlappende Wartebedingungen - typisch für Leser/Schreiber-Schema

4. **Prozesse wecken** nach Erniedrigen der Zähler:
   in releaseRead: nr--; notifyAll();
   in releaseWrite: nw--; notifyAll();

5. **nur relevante Zustandsänderungen**:
   in releaseRead: nr--; if (nr==0) notify();
   Wecken eines wartenden Schreiber genügt.

Leser/Schreiber-Monitor

```
class ReaderWriter
{  private int nr = 0, nw = 0;
                  // Monitor-Invariante RW: (nr == 0 || nw == 0) && nw <= 1
   synchronized public void requestRead ()
   {  while (nw > 0)                                    // negierte Wartebedingung
         try { wait(); } catch (InterruptedException e) {}
      nr++;
   }
   synchronized public void releaseRead ()
   {  nr--;
      if (nr == 0) notify ();                        // einen Schreiber wecken genügt
   }

   synchronized public void requestWrite ()
   {  while (nr > 0 || nw > 0)                          // negierte Wartebedingung
         try { wait(); } catch (InterruptedException e) {}
      nw++;
   }
   synchronized public void releaseWrite ()
   {  nw--;
      notifyAll ();          // 1 Schreiber und alle Leser zu wecken würde genügen!
   }
}
```

Schema: Begegnungen von Prozessen

Prozesse durchlaufen **verschiedene Stationen** und interagieren dabei.
Ein Monitor organisiert die **Begegnungen in der vorgeschriebenen Abfolge**.

Entwurfsmethode:

Zustände durch **Zählvariable** beschreiben;
zulässige Zustände durch **Invariante** über Zählvariable charakterisieren;
daraus **Wartebedingungen** der Monitor-Operationen herleiten;
Zählvariable durch binäre **Variable substituieren**.

Beispiel **Sleeping Barber**:

In einem verschlafenen Ort im südlichen Ostwestfalen wartet ein Friseur in seinem Laden auf Kundschaft. Wenn ein Kunde im Frisierstuhl platzgenommen hat, schneidet er ihm die Haare. Wenn er den Haarschnitt beendet hat, kassiert er und verabschiedet den Kunden.

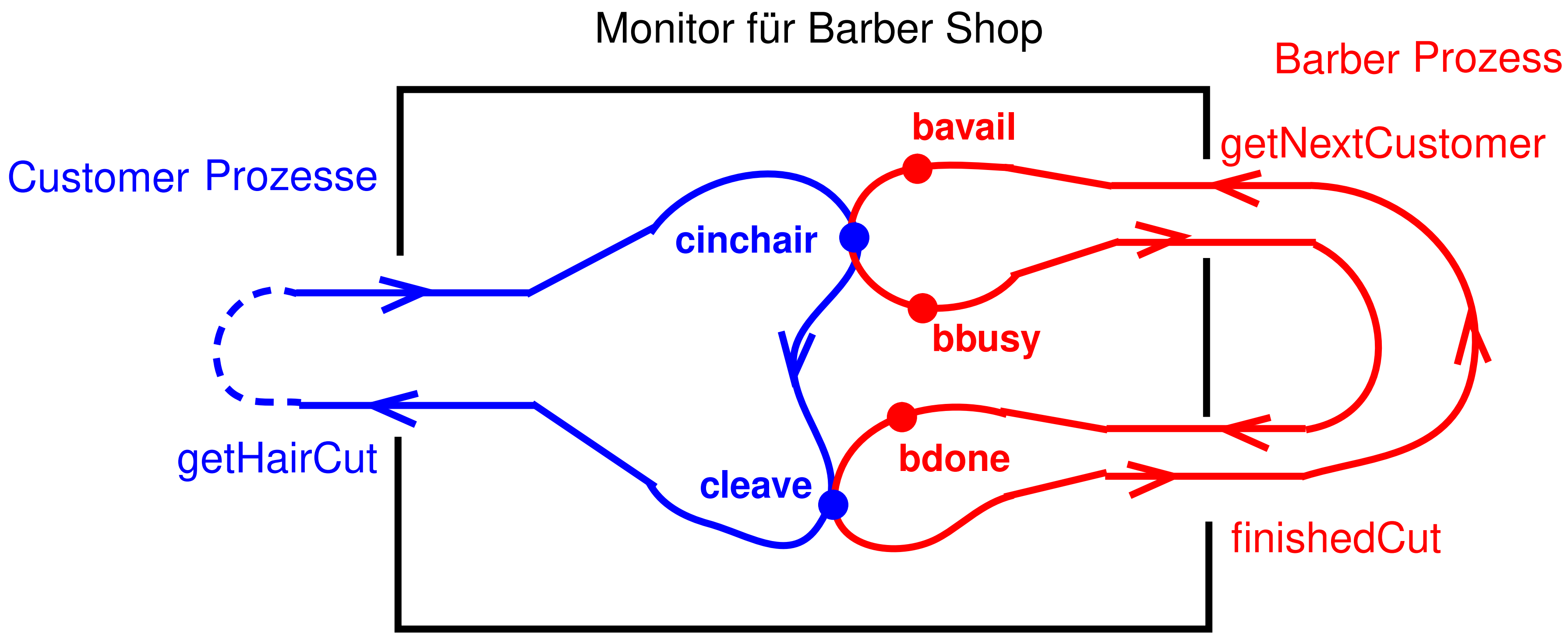
Ein Kunde, der den Friseurladen betritt, setzt sich in den Frisierstuhl, falls er frei ist, und wartet auf seinen Haarschnitt. Falls der Frisierstuhl besetzt ist, wartet er auf einem anderen Platz bis er dran ist.

2 Arten von Prozessen: Barber (1 Exemplar), Customer (viele Exemplare)

2 Begegnungen: Haarschnitt und verabschieden

Die Aufgabe ist auch ein Beispiel für das Client/Server-Schema.

Monitor-Entwurf zum Sleeping Barber



Invariante über Zählvariable:

C1: cinchair >= cleave and bavail >= bbusy >= bdone wird durch die Prozesse erfüllt.

C2: bavail >= cinchair >= bbusy führt zu 2 Wartebedingungen

C3: bdone >= cleave führt zu 1 Wartebedingung

Monitor-Invariante BARBER: C1 and C2 and C3

Grobentwurf mit Zählvariablen und Wartebedingungen

Monitor-Invariante

BARBER: cinchair >= cleave && bavail >= bbusy >= bdone &&
bavail >= cinchair >= bbusy &&
bdone >= cleave

	Wartebedingung:	**Zählvariable erhöhen:**	
getHairCut			Customer Prozesse:
	cinchair < bavail	cinchair++;	
	cleave < bdone	cleave++;	
getNextCustomer			Barber Prozess:
		bavail++;	
	busy < cinchair	bbusy++;	
finishedCut			
		bdone++;	
	bdone == cleave	Kunde verabschiedet	

Zählvariable substituieren

neue binäre Variable: barber = bavail - cinchair
chair = cinchair - bbusy
open = bdone - cleave

alte Invariante:
bavail >= cinchair >= bbusy &&
bdone >= cleave

substituierte Invariante:
barber >= 0 && chair >= 0 &&
open >= 0

Wartebedingung:	**Zustandswechsel:**	
getHairCut		Customer Prozesse:
barber > 0	barber--;	
	chair++;	wecken
open > 0	open--;	
getNextCustomer		Barber Prozess:
	barber++;	wecken
chair > 0	chair--;	
finishedCut		
	open++;	wecken
open == 0	Kunde verabschiedet	

Datenparallelität

Viele Prozesse bzw. Prozessoren führen zugleich die gleichen Operationen auf verschiedenen Daten aus; meist Datenelemente in regulären Datenstrukturen: Array, Folge Matrix, Liste.

Datenparallelität als **Architekturprinzip für Parallelrechner**:
Vectorrechner, z. B. Cray
SIMD-Rechner (Single Instruction Multiple Data), z. B. Connection Machine, MasPar

Datenparallelität als **Programmiermodell für Parallelrechner**:

- Berechnungen auf **Arrays in geschachtelten Schleifen**
- **Datenabhängigkeiten** der Berechnungen untersuchen, Schleifen transformieren, parallelisieren
- Iterative **Berechnungen in Runden**, Synchronisation durch **Barrieren**
- Systolische Berechnungen: 2 Phasen iterieren: rechnen - Daten weitergeben

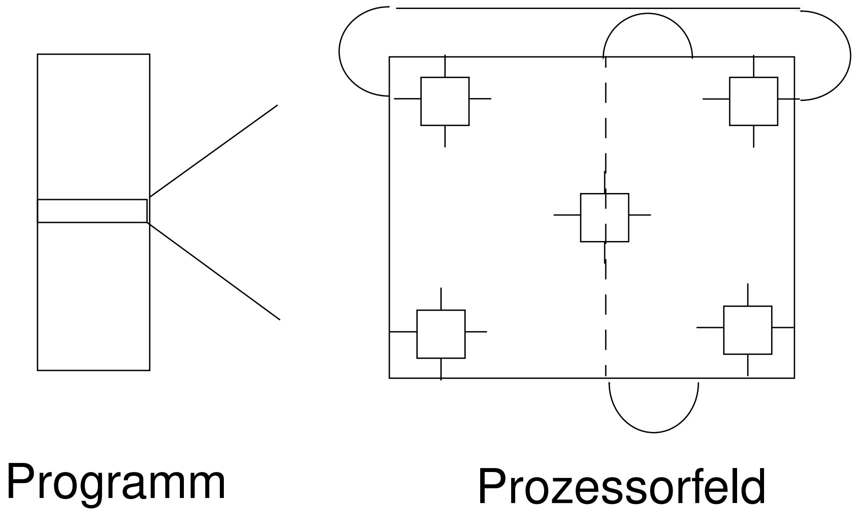
Anwendungen hauptsächlich im **technisch - wissenschaftlichen Rechnen**, z. B.

- Strömungsmechanik
- Bildverarbeitung
- Differentialgleichungen lösen
- Finite Element Methode in Entwurfssystemen

Datenparallelität als Architekturmodell

SIMD Rechner: Single Instruction Multiple Data

- sehr viele Prozessoren, massiv parallel
  z. B. 32 x 64 er Feld
- Prozessor-lokale Datenspeicher
- gleiche Instruktionen im **Gleichtakt**
- schnelle Kommunikation im **Gleichtakt**
- feste Topologie, meist Gitter
- Rechnertypen z. B. Connection Machine, MasPar



Datenparallelität als Programmiermodell

- reguläre Datenstrukturen (Arrays, Listen) auf ein Prozessfeld abbilden
- Prozesse führen gleiches Programm auf individuellen Daten im Gleichtakt aus
- Kommunikation mit Nachbarn in gleicher Richtung im Gleichtakt

einfaches Beispiel Matrixaddition:

C = A + B

sequentiell:

```
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
      c[i,j] = a [i,j] + b[i,j];
```

datenparallel:

```
A, B verteilen
c = a + b
C zusammenfügen
```

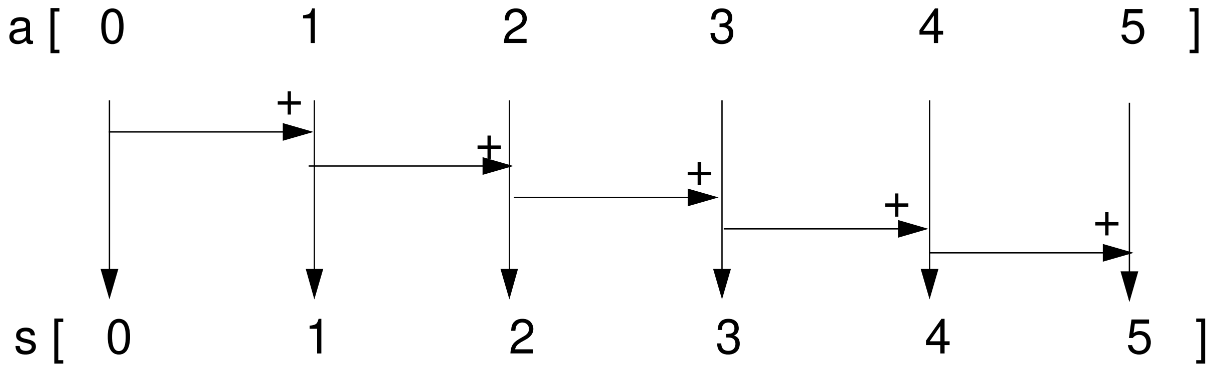
1 Schritt!

- diese Schleifen sind unmittelbar parallelisierbar, da keine **Datenabhängigkeiten**
- **Datenabbildung** ist trivial: Array-Elemente [i,j] auf Prozess [i,j]
- **Kommunikation** ist nicht nötig
- keine **algorithmische Idee** nötig

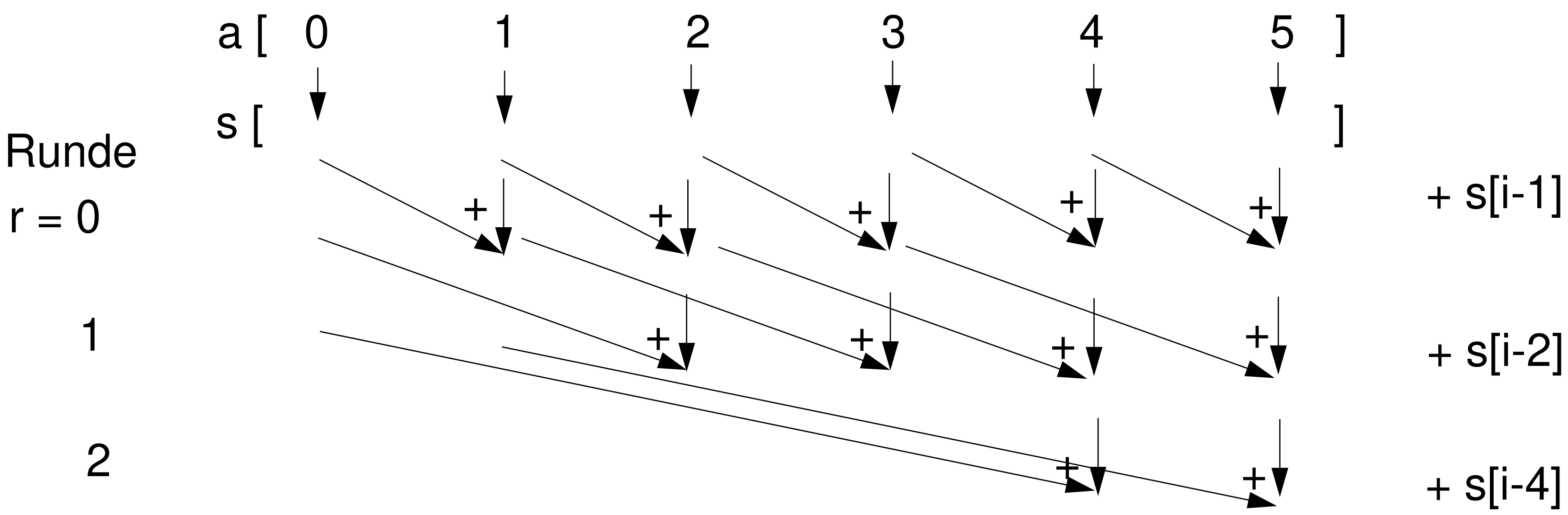
Beispiel Präfixsummen

gegeben: Folge a von Zahlwerten;
gesucht: Folge s Summen der Anfänge von a

$$s[i] = \sum_{j=0}^{i} a[j]$$



parallele algorithmische Idee:



Präfixsummen: angewandte Prinzipien

- Berechnungsschema **Reduktion**:
  alle Array-Elemente werden mit einer Operation zu einem Wert zusammengefasst
- iterative **Berechnung in Runden**:
  in jeder Berechnungsrunde führen alle Prozesse einen Berechnungsschritt aus
- **Barrieren**-Synchronisation:
  Prozesse dürfen erst dann in die nächste Runde gehen, wenn alle die vorige beendet haben
- **Abstandsverdopplung**:
  Datenaustausch in jeder Runde mit doppelt soweit entferntem Nachbarn

Barrieren

Mehrere Prozesse synchronisieren sich an einem gemeinsamen Synchronisationspunkt

Regel: Alle Prozesse müssen die Barriere (zum j-ten mal) erreicht haben,
bevor ein Prozess sie (zum j-ten mal) verlässt.

Anwendungen:

- iterative Verfahren, wo Iteration j Ergebnisse aus Iteration j-1 benutzt
- Trennung von Berechnungsphasen

Schema:

```
public void run ()
{  do {  computeNewValues (i);
         b.barrier();
      }
   while (!converged);
}
```

Implementierungstechniken für Barrieren:

- zentrale Lösungen: Monitor oder Koordinatorprozess
- Worker-Prozesse als Baum koordiniert
- Worker-Prozesse symmetrisch koordiniert (Butterfly Barrier, Dissemination Barrier)

Barrieren als Monitor implementiert

Monitor hält vorgegebene Anzahl von Prozessen an und gibt sie gemeinsam frei:

```
class BarrierMonitor
{  private int  processes          // number of processes to be synchronized
                arrived = 0;      // number of processes arrived at the barrier

   public BarrierMonitor (int procs)
   {  processes = procs; }

   synchronized public barrier ()
   {  arrived++;
      if (arrived < processes)
         try { wait(); } catch (InterruptedException e) {}
                                  // exception destroys barrier behaviour
      else
      {  arrived = 0;                            // reset arrival count
         notifyAll();                  // release the other processes
}  }  }
```

Verteilte Barriere als Baum

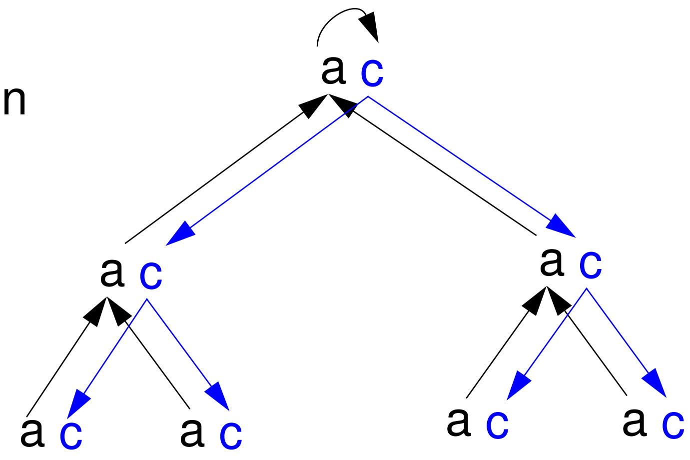
Barrierensynchronisation der Worker-Prozesse als **Binärbaum** organisiert.
Vermeidet den Engpass der zentralen Synchronisation.

2 Synchronisationsvariable (Flags) an jedem Knoten:

arrived: alle Prozesse im Teilbaum sind angekommen
wird nach oben propagiert

continue: alle Prozesse imTeilbaum können weiterlaufen
wird nach unten propagiert

Nachteil: Erfordert unterschiedlichen Code für
Wurzel, innere Knoten und Blätter



Flag zur Synchronisation zwischen genau 2 Prozessen (in Java als Monitor):

Ein Prozess wartet, dass das Flag gesetzt ist.
Der andere Prozess setzt das Flag.

Flag-Regeln: Wer auf ein Flag wartet setzt es auch zurück.
Wenn ein Flag gesetzt wird, darf es nicht schon gesetzt sein.

Symmetrische, verteilte Barriere (Dissemination)

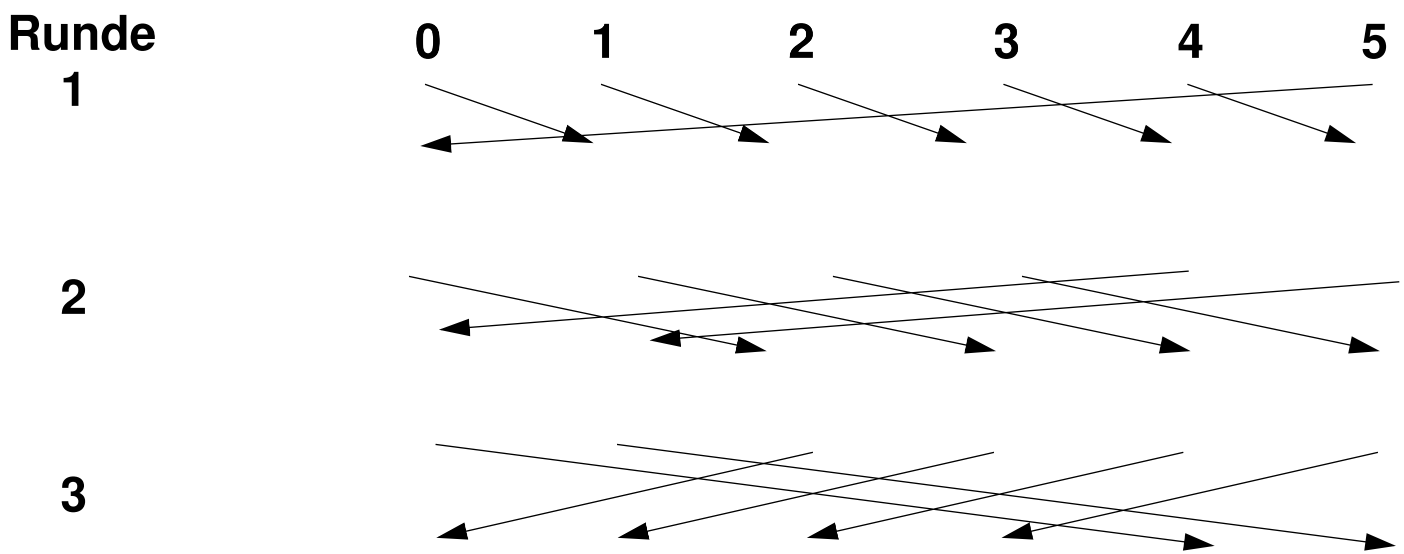
Prozesse synchronisieren sich paarweise in Runden mit Abstandsverdopplung

In r Runden werden $N <= 2^r$ Prozesse synchronisiert

In Runde s

zeigt Prozess i seine Ankunft an und

wartet auf die Ankunft von Prozess $(i + N - 2^{s-1})$ modulo N



Präfixsummen mit Barriere

```
class PrefixSum extends Thread
{  private int procNo;                              // Prozessnummer
   private BarrierMonitor bm;                    // Barrieren-Objekt

   public PrefixSum (int p, BarrierMonitor b)
   { procno = p; bm = b; }

   public void run ()
   {  int dist = 1;                                     // Abstand
                                              // globale Arrays a und s
      s[procNo] = a[procNo];              // Ziel-Array initialisieren
      bm.barrier();

      // Invariante SUM: s[procNo] == a[procNo-dist+1]+...+a[procNo]
      while (dist < N)
      {  if (procNo - dist >= 0)
         {  int addIt = s[procNo - dist]; // Wert vor Überschreiben
            bm.barrier();
            s[procNo] += addIt;
            bm.barrier();
         }
         dist = dist * 2;                          // Abstand verdoppeln
}  }  }
```

Präfixsummen synchron parallel

Notation in Modula-2* mit synchronen (und asynchronen) Schleifen für Parallelrechner

```
VAR a, s, t: ARRAY [0..N-1] OF INTEGER;
VAR dist: CARDINAL;
BEGIN
   ...
   FORALL i: [0..N-1] IN SYNC
      s[i] := a[i];
   END;

   dist := 1;

   WHILE dist < N
      FORALL i: [0..N-1] IN SYNC
         IF (i-dist) >= 0 THEN
            t[i] := s[i - dist];
            s[i] := s[i] + t[i];
         END
      END;
      dist := dist * 2;
   END
END
```

Parallele Schleife im Gleichtakt (zu `FORALL i: [0..N-1] IN SYNC` nach `...`)

Parallele Schleife im Gleichtakt (zu `FORALL i: [0..N-1] IN SYNC` in der `WHILE`-Schleife)

Barrieren automatisch für jede Anweisung

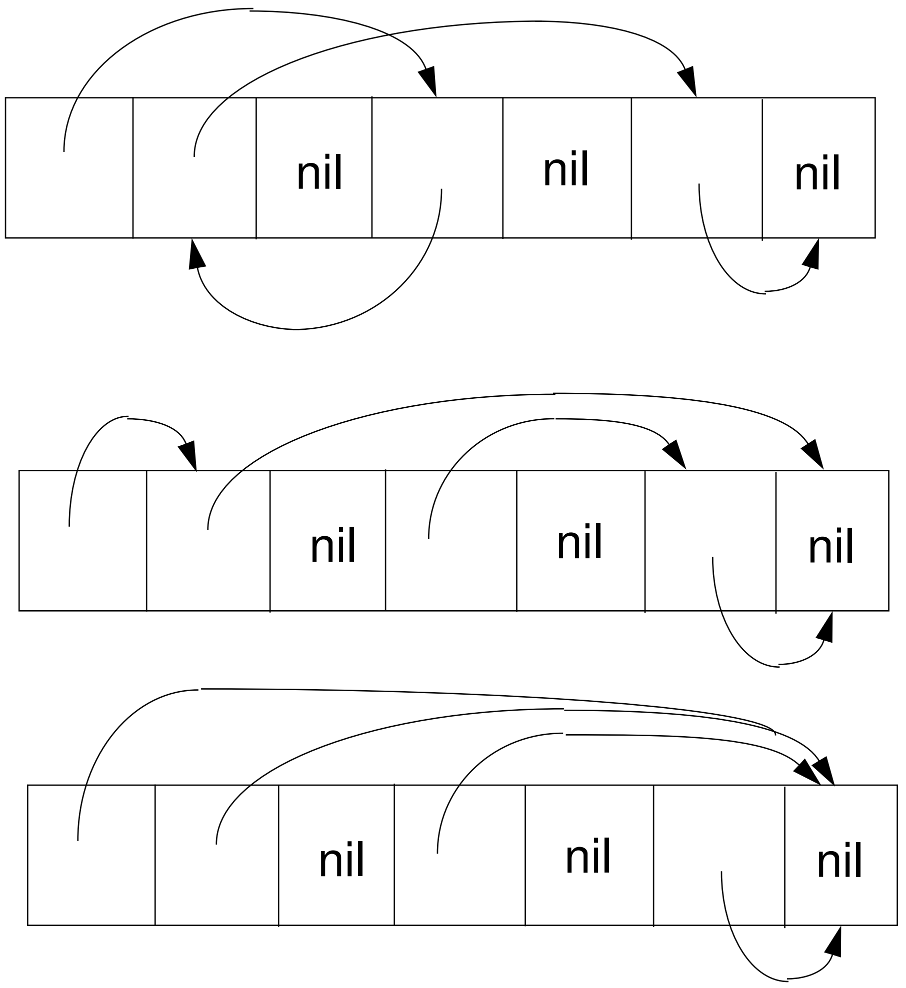
Datenparallel Listenenden finden

geg.: int-Array link speichert Listen; link[i] enthält den Index des Nachfolgers oder nil

ges.: int-Array last; last[i] enthält den Index des letzten Elementes der Liste link[i]

Methode: Worker Prozess i berechnet last[i] = last[last[i]] in log N **Runden**

```
int d = 1;
last[i] = link[i];
Barriere

while (d < N)
{  int newlast = nil;
   if ( last[i] != nil &&
        last[last[i]] != nil)
      newlast = last[last[i]];
   Barriere
   if (newlast != nil)
      last[i] = newlast;
   Barriere
   d = 2*d;
}
```



last[i] zeigt auf Listenende, wenn es höchstens d Elemente entfernt ist

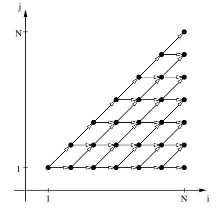
Schleifenparallelisierung

Entwicklungsschritte:

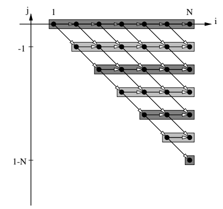
- **geschachtelte Schleifen**, die sequentiell **auf Arrays** operieren

```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
    FOR J := 1 .. I
        B[I,J] :=
            B[I-1,J]+B[I-1,J-1]
    END FOR
END FOR
```

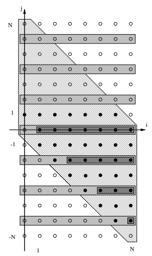
- **Datenabhängigkeiten** analysieren



- **Schleifen transformieren**, Datenabhängigkeiten beibehalten
- **innere Schleifen parallel** auf Prozessorfeld abbilden



- **Arrays** so auf Prozessorfeld **verteilen**, dass **viele Zugriffe lokal** sind



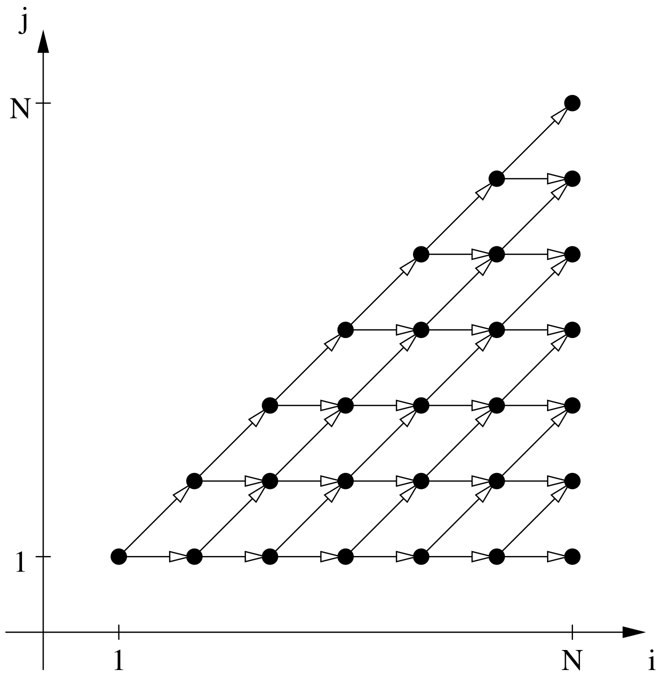
Iterationsraum von Schleifenschachteln

Iterationsraum n-fach geschachtelter Schleifen:

- **n-dimensionaler Raum von ganzzahligen Punkten** (Polytop)
- jeder Punkt (i_1, ..., i_n) steht für eine Ausführung des innersten Schleifenrumpfes
- Schleifengrenzen i. a. erst zur Laufzeit bekannt
- Iterationsraum ist nicht notwendig rechtwinklig begrenzt
- Iterationsraum wird sequentiell abgearbeitet

Beispiel:
Berechnung des Pascalschen Dreiecks

```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
    FOR J := 1 .. I
        B[I,J] :=
            B[I-1,J]+B[I-1,J-1]
    END FOR
END FOR
```



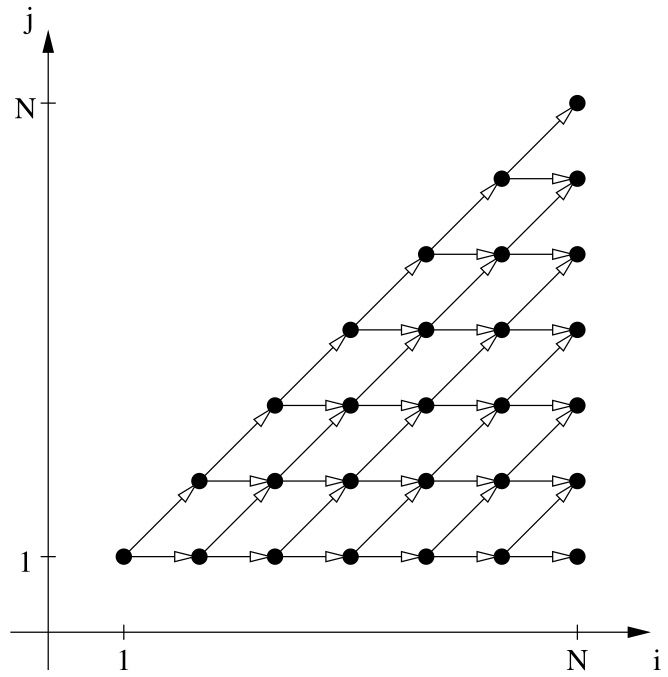
Datenabhängigkeiten

Datenabhängigkeit von Iterationspunkt i zu j:

- Iteration **i** berechnet einen Wert, der in Iteration **j** benutzt wird; (Daten-)Flussabhängigkeit
- relativer **Abhängigkeitsvektor d** = **j - i** = $(j_1 - i_1, ..., j_n - i_n)$
  gilt für alle Iterationspunkte außer am Rand
- Flussabhängigkeiten können **nicht entgegen der Ausführungsreihenfolge** gerichtet sein, nicht rückwärts in der Zeit

Beispiel:
Berechnung des Pascalschen Dreiecks

```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
    FOR J := 1 .. I
        B[I,J] :=
            B[I-1,J]+B[I-1,J-1]
    END FOR
END FOR
```



Schleifentransformation

Der **Iterationsraum** geschachtelter Schleifen wird auf **neue Koordinatenachsen** transformiert.

Ziele:

- **innere Schleife(n) parallel** ausführen
- **Lokalität** der Datenzugriffe verbessern; räumlich und zeitlich (Wiederverwendung im Cache)
- **systolisches** Rechen- und Kommunikationsschema erreichen

Datenabhängigkeiten nicht zeitlich rückwärts und nicht gleichzeitig

3 lineare **Grundtransformationen**, kombinierbar:

- **Reversal**: eine Dimension des Iterationsraumes spiegeln
- **Permutation**: Schleifen vertauschen
- **Skewing**: Iterationsraum schräg verschieben

nicht-lineare Transformationen, z. B.:

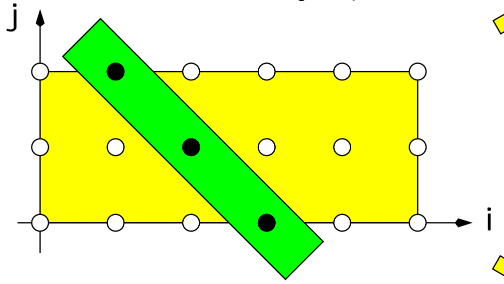
- **Scaling**: streckt den Iterationsraum in einer Dimension (Lücken)
- **Tiling**: zusätzliche Schleife bearbeitet Kacheln fester Größe

Transformationen von

Daten

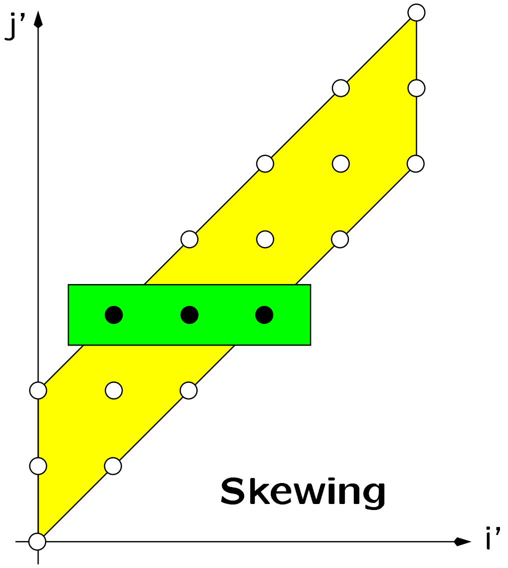
```
REAL B(1:n, 0:m)
```

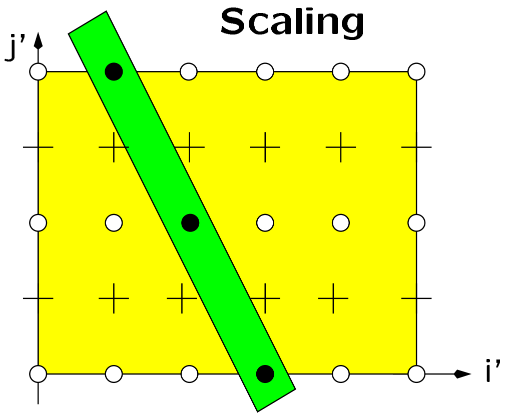
konvexes Polytop

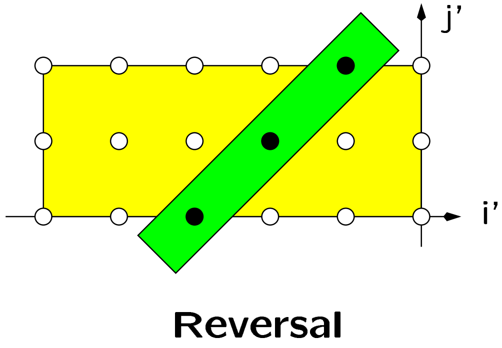


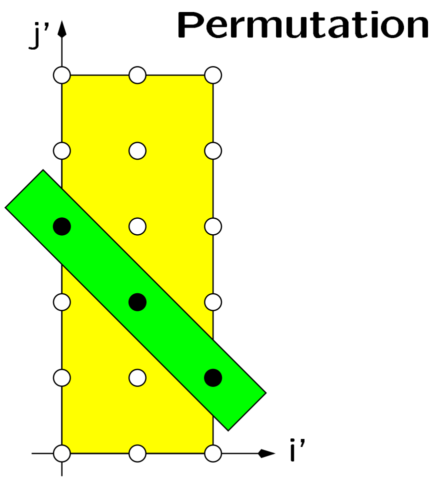

```
DO i = 0, m-1
   DO j = 0, k-1
      ...
   END
END
```

Schleifenschachteln









Transformationen definiert durch Matrizen

Transformationsmatrizen: systematisches Transformieren, Prüfen von Abhängigkeitsvektoren

Reversal
$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} -i \\ j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

Skewing
$$\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ f*i+j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

Permutation
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

Benutzung von Transformationsmatrizen

- mit Transformationsmatrix **T** neue Iterationspunkte durch alte ausdrücken: **T * i = i´**

z. B. $$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} -i \\ j \end{pmatrix} = \begin{pmatrix} i' \\ j' \end{pmatrix}$$

- mit Transformationsmatrix T alte Abhängigkeitsvektoren in neue umrechnen: **T * d = d´**

z. B. $$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

- mit inverser Transformationsmatrix $\mathbf{T^{-1}}$ i durch i‘ ausdrücken, für Indexausdrücke: $\mathbf{T^{-1} * i = i'}$

z. B. $$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} -i' \\ j' \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix}$$

- Hintereinanderausführung erst $\mathbf{T^1}$ dann $\mathbf{T^2}$: $\mathbf{T^2 * T^1 = T}$

z. B. $$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

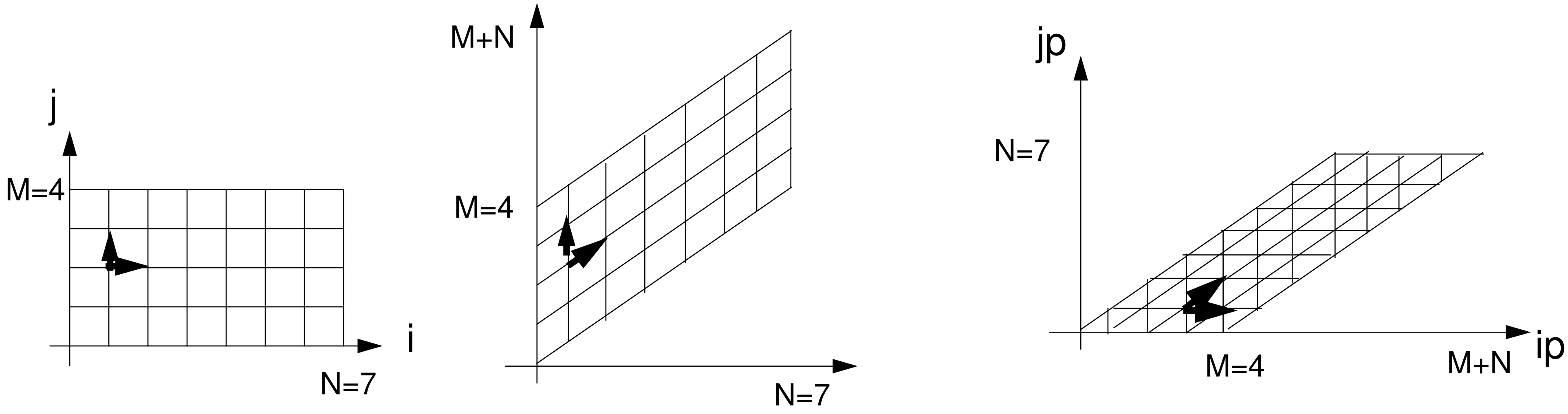
Beispiel: Schleife transformieren und parallelisieren

```
for i = 0 to N
   for j = 0 to M
      a[i, j] = (a[i, j-1] + a[i-1, j]) / 2;
```

Die obige Schleife soll parallelisiert werden.

1. Zeichnen Sie den Iterationsraum.
2. Bestimmen Sie die Abhängigkeitsvektoren und zeichnen sie Beispiele davon ein. Warum kann man die innere Schleife nicht parallel ausführen?
3. Wenden Sie eine Skewing-Transformation an und zeichnen den Iterationsraum.
4. Wenden Sie die Permutation-Transformation an und zeichnen Sie den Iterationsraum. Begründen Sie, dass man nun die innere Schleife parallelisieren kann.
5. Geben Sie die zusammengesetzte Transformationsmatrix an und transformieren Sie damit die Abhängigkeitsvektoren.
6. Geben Sie die Inverse der Transformationsmatrix an und transformieren Sie damit die Indexausdrücke.
7. Schreiben Sie die Schleifen vollständig und mit neuen Laufvariablen ip und jp und neuen Schleifengrenzen auf.

Lösung: Schleife transformieren und parallelisieren



$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}}_{\text{Inverse}}$$

2. Eine Abhängigkeit in Richtung der parallelen Dimension j ist nicht zulässig.

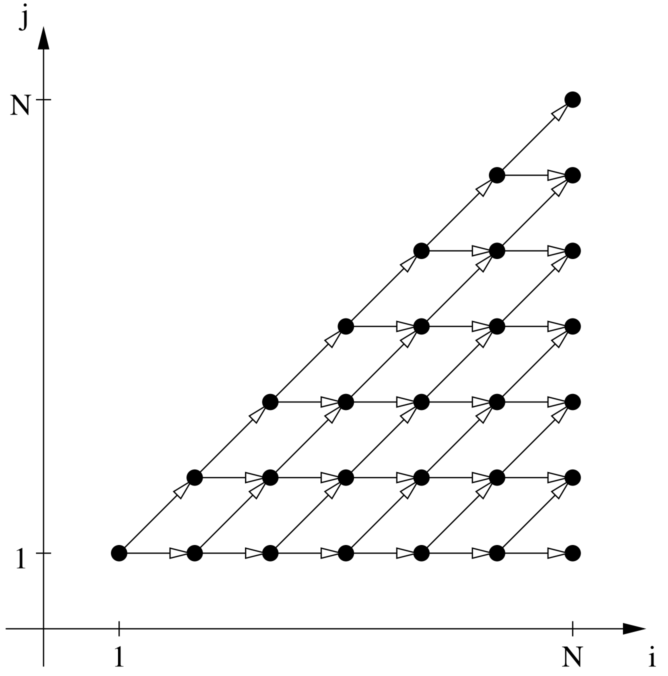
4. Beide Abhängigkeitsvektoren zeigen vorwärts in ip-Richtung.

7.
```
for ip = 0 to M+N
    for jp = max (0, ip-M) to min (ip, N)
        a[jp, ip-jp] = (a[jp, ip-jp-1] + a[jp-1, ip-jp]) / 2;
```

Transformation und Parallelisierung

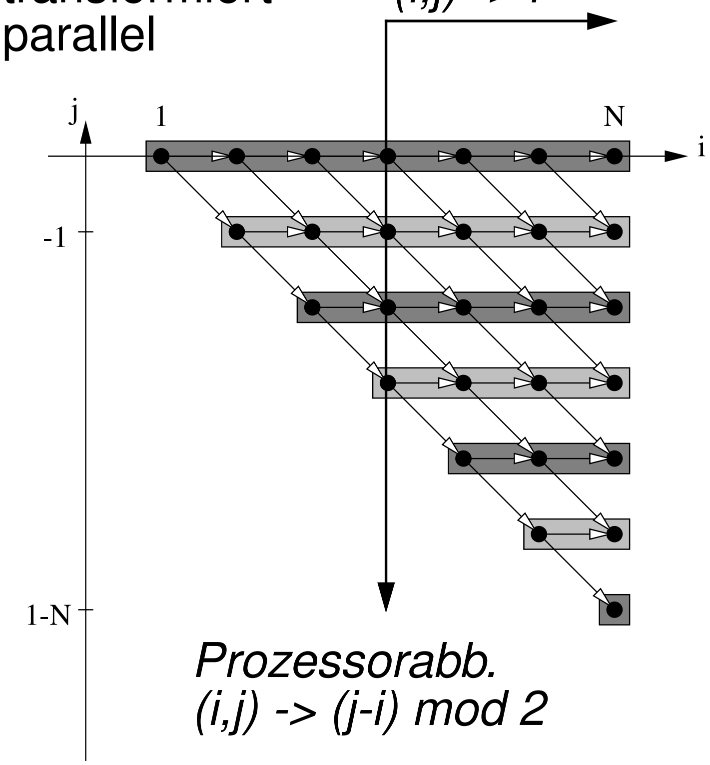
Iterationsraum

original




```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
   FOR J := 1 .. I
       B[I,J] :=
           B[I-1,J]+B[I-1,J-1]
   END FOR
END FOR
```

transformiert
parallel




```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
   FOR J := 1-I.. 0
       B[I,J+I] :=
           B[I-1,J+I]+B[I-1,J-1+I]
   END FOR
END FOR
```

Datenverteilung

Ziel: Array-Elemente so auf Prozessoren verteilen, dass möglichst viele der **Zugriffe lokal** sind.

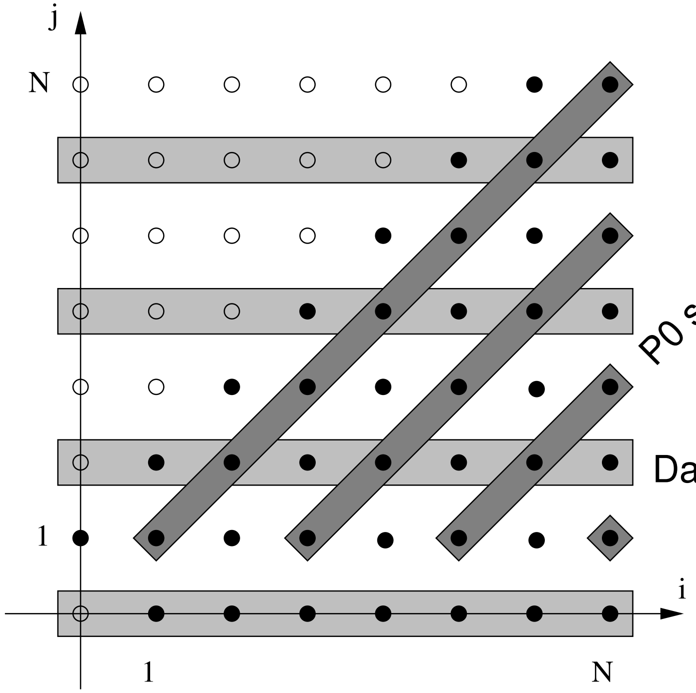
Indexraum eines Arrays: n-dimensionaler Raum von ganzzahligen Indexpunkten (Polytop)

- **gleiche Eigenschaften wie Iterationsraum**
- gleiches mathematisches Modell
- gleiche **Transformationen** anwendbar (Skewing, Reversal, Permutation, ...)
- **keine Restriktionen** durch Datenabhängigkeiten

Datenverteilung für parallele Schleifen

Indexraum

original



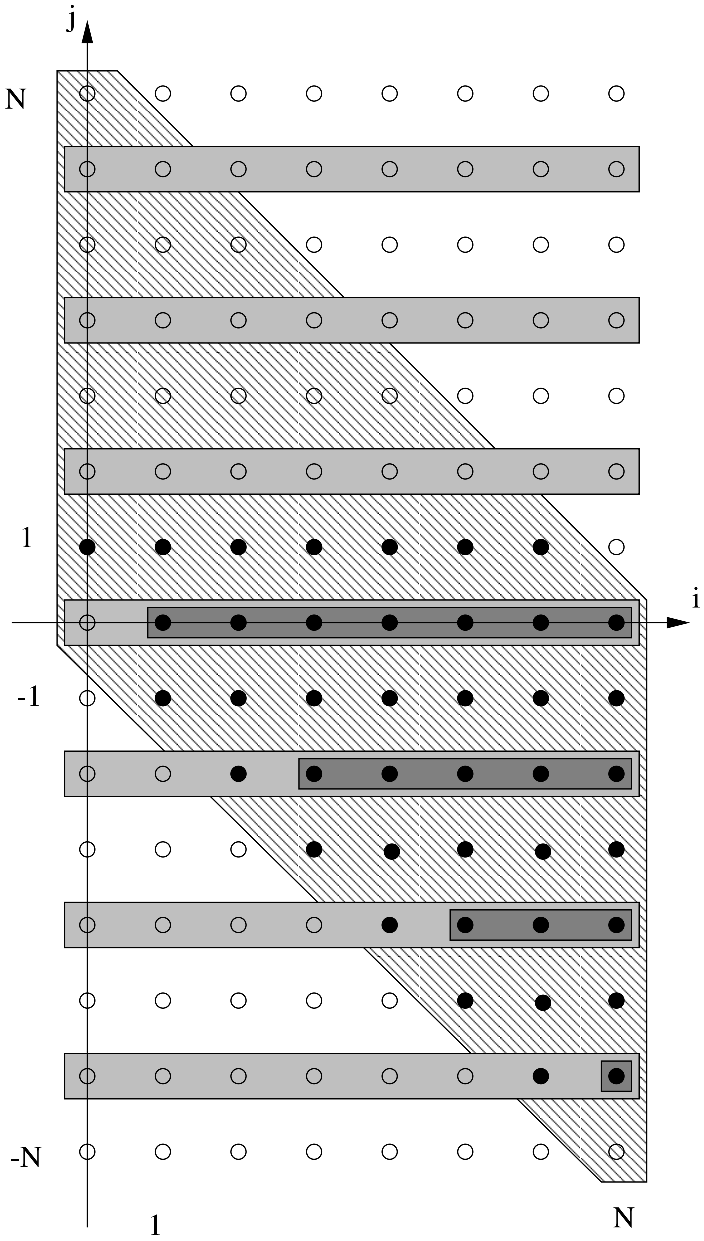

```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
   FOR J := 1-I.. 0
      B[I,J+I] :=
         B[I-1,J+I]+B[I-1,J-1+I]
   END FOR
END FOR
```

```
DECLARE B[0..N,-N..N]
...
   B[I,J] :=
      B[I-1,J-1]+B[I-1,J]
```

transformiert

Skewing
(i,j) -> (i,j-i)

100%lokal

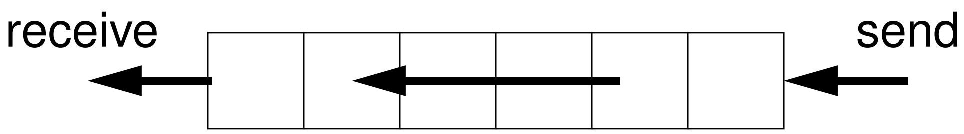


Asynchrone Botschaften

Prozesse senden und empfangen Botschaften über Kanäle.

Botschaft: Wert eines evtl. zusammengesetzten Datentyps oder Objekt einer Klasse.

Kanal: beliebig lange Schlange von Botschaften mit Operationen send und receive.



Operationen mit Kanälen:

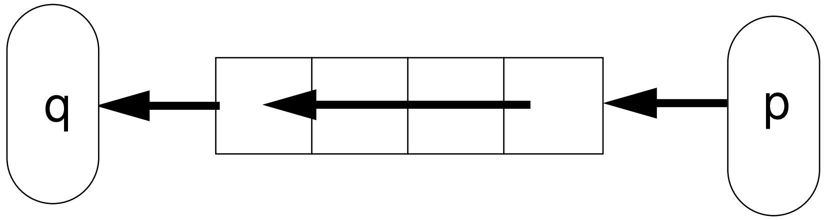
- **send (b)**: fügt die Botschaft b an die Schlange des Kanals an; **blockiert nicht** (im Gegensatz zu send bei synchronen Botschaften)
- **receive()**: liefert die älteste Botschaft und entfernt sie aus dem Kanal; **blockiert** den aufrufenden Prozess solange der Kanal leer ist.
- **empty()**: liefert true, falls der Kanal leer ist; das Ergebnis ist **nicht notwendig aktuell**.

Alle Operationen werden atomar ausgeführt.

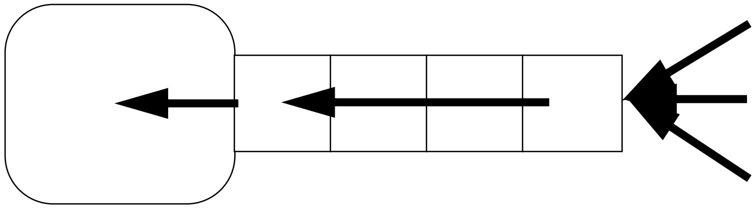
Kanal mit leeren Botschaften: dient nur zur Synchronisation; entspricht zählendem Semaphor

Prozesse und Kanäle

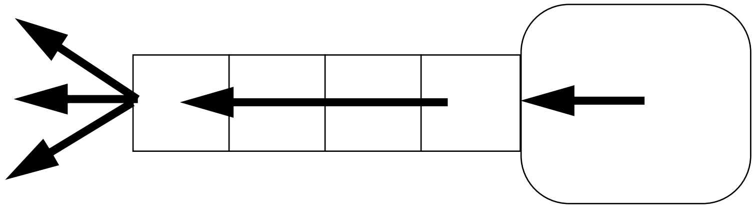
Link:
ein Sender mit **einem** Empfänger verbunden;
z. B. Prozesse bilden Ketten von Transformationen (Pipeline)



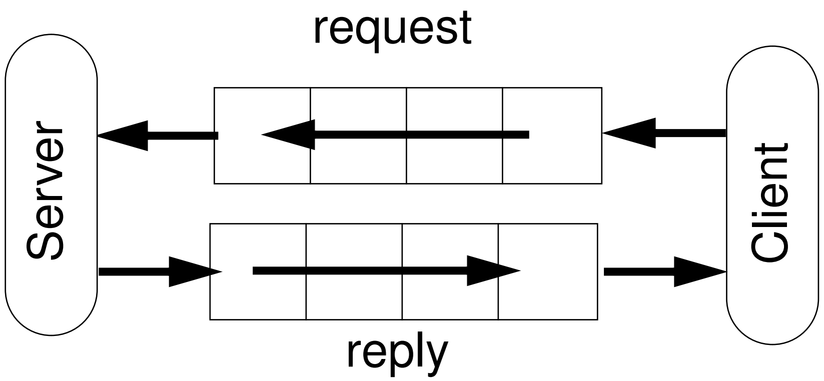
Input-Port eines Prozesses:
viele Sender ein Empfänger;
Kanal gehört zum empfangenden Prozess;
z. B. Server-Prozess empfängt Aufträge von vielen Client-Prozessen



Output-Port eines Prozesses:
ein Sender viele Empfänger;
Kanal gehört zum sendenden Prozess;
z. B. Prozess verteilt Aufträge an viele Bearbeite
(ungewöhnlich Struktur)



Paar von **Anfrage- und Antwort-Kanal**
request/reply;
enge Synchronisation,
z.B. von Client und Server



Kanäle in Java

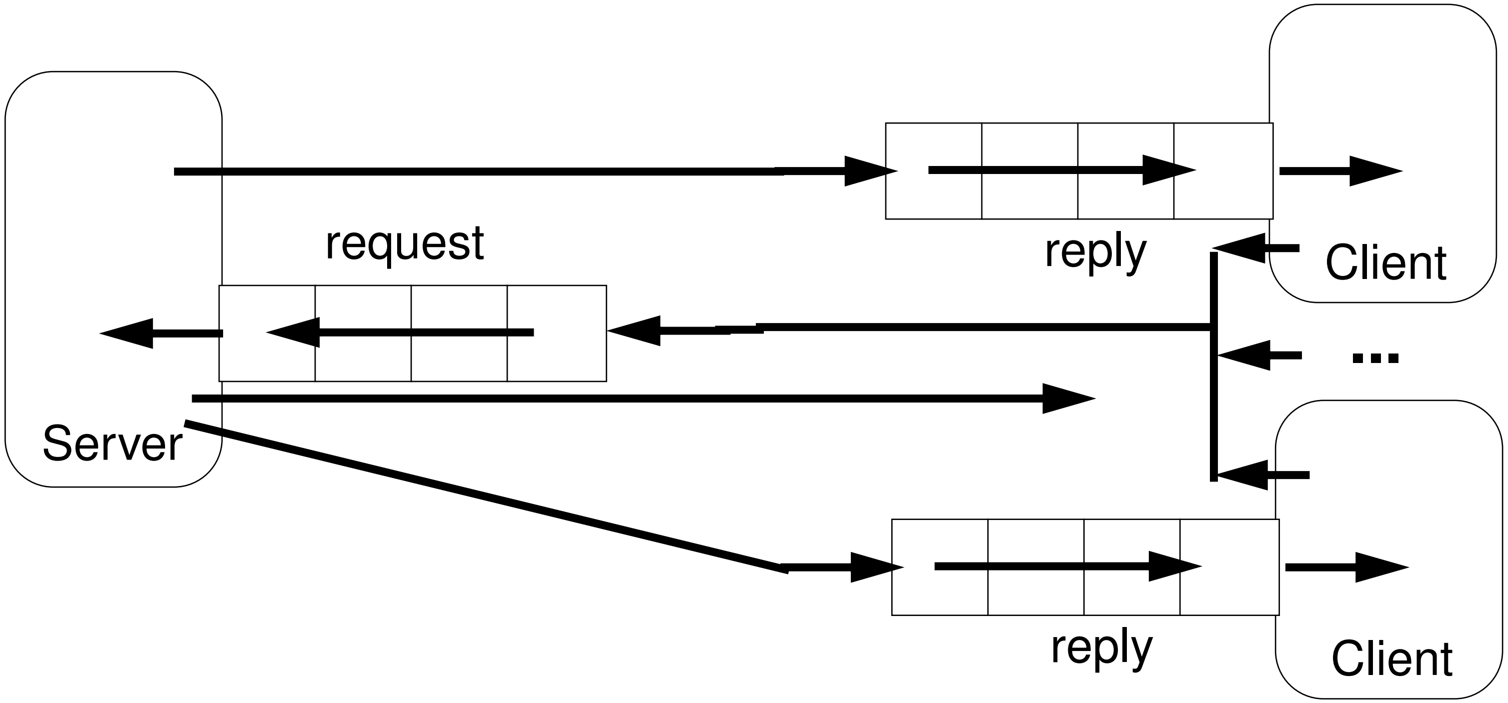
```
public class Channel
{  // Implementation of a Channel using a Queue of messages
   private Queue msgQueue;

   public Channel ()
      { msgQueue = new Queue (); }

   public synchronized void send (Object msg)
      { msgQueue.enqueue (msg); notify(); }// wake a receiving process

   public synchronized Object receive ()
      {  while (msgQueue.empty())
            try { wait(); } catch (InterruptedException e) {}
         Object result = msgQueue.front();// if this process is woken,
         msgQueue.dequeue();                // the Queue is not empty
         return result;
      }

   public synchronized boolean empty ()
      { return msgQueue.empty (); }
}
```

Botschaften in Java

```
import java.util.Vector;

public class BasicMessage
{             // Messages with an integral code and arbitrary arguments
   protected int kind;                    // encoding of the message kind
   protected Vector argList;// arbitrary number and types of arguments

   public BasicMessage () { argList = new Vector(); } // constructors
   public BasicMessage (int k) { kind = k; argList = new Vector (); }
   public BasicMessage (int k, Vector args) { kind=k; argList=args;}

   protected void setKind (int k) { kind = k; }    // kind of message
   protected int getKind () { return kind; }

   public void setArg (int index, Object arg) // set and get arguments
      { argList.insertElementAt (arg, index); }

   public Object getArg (int index)
      { return argList.elementAt (index); }

   public Vector getArgList ()                     // copy arguments
      { Vector listCopy = (Vector) argList.clone();
        return listCopy;
}     }
```

Client/Server mit asynchronen Botschaften

Ein oder mehrere gleichartige **Server-Prozesse bedienen Anfragen von Client-Prozessen**.



request-Kanal:
- Input-Port des Servers;
- auch bei mehreren Servern nur ein request-Kanal

reply-Kanal
- zu jedem Client (Input-Port);
- wird z. B. mit der request-Botschaft versandt

Server-Prozess

Der Server ist ein Prozess, er verarbeitet Anfragen in permanenter Schleife.

Unterschiedliche Anfragen (Operationen) durch verschiedene Arten von Botschaften.

```
public void run ()
{  while (running)
   {                                                   // Anfrage empfangen
      BasicMessage msg = (BasicMessage)servChan.receive();
      switch (msg.getKind())
      {  case REQUEST:  // Operation bearbeiten und Antwort senden:
            ...   ((Channel)msg.getArg(0)).send(...);
         case RELEASE:
            ...   ((Channel)msg.getArg(0)).send(...);
}  }  }
```

Antwortkanal ist in der Anfragebotschaft enthalten.

Falls **Anfragen nur unter bestimmten Bedingungen** erfüllt werden können,
werden sie in einer **Schlange** des Servers gespeichert.

Der Server-Prozess bearbeitet die Anfragen streng sequentiell,
deshalb ist gesichert, dass kritische Abschnitte nicht Verzahnt werden.

Server-Prozess als **„aktiver Monitor“**.

Server-Prozesse in Java

```
public class RessourceServer extends Thread
{  private Channel servChan; // this server receives messages from it
   private String id;

   public static final int REQUEST = 1;          // kinds of messages
   public static final int RELEASE = 2;

   private int avail = 5;            // number of available ressources
   private Queue pending;                              // pending requests

   public RessourceServer (Channel c, String name)     // constructor
      { servChan = c; id = name; pending = new Queue ();}

   public void stopIt () { running = false; } // external termination
   private boolean running = true;

   public void run ()                             // requests are processed
   {  while (running)
      {  BasicMessage msg = (BasicMessage)servChan.receive();
         switch (msg.getKind())
         {  case REQUEST: ...
            case RELEASE: ...
}  }  }  }
```

Client-Prozesse in Java

```
import java.util.Random;

public class RessourceClient extends Thread
{  private Channel clientReply;       // reply channel of this client
   private Channel serverChan;// the request channel of the server(s)

   private String id; private Random rand;

   public RessourceClient (Channel serv, String name)
      {  serverChan = serv; clientReply = new Channel ();
         id = name; rand = new Random (name.hashCode());
      }

   public void stopIt () { running = false; }
   private boolean running = true;

   public void run ()
   {  BasicMessage reqMsg = new BasicMessage(RessourceServer.REQUEST);
      reqMsg.setArg (0, clientReply); ...// reusable message instances

      while (running)
      {  serverChan.send (reqMsg); clientReply.receive (); ...
         serverChan.send (relMsg); clientReply.receive (); ...

}  }  }
```

Receive ohne Blockieren

Wenn mehrere Prozesse von einem Kanal empfangen, kann

```
if (!ch.empty()) msg = ch.receive();
```

trotz der Bedingung blockieren.
Wenn von mehreren Kanälen empfangen werden soll, ist das unerwünscht.

Die Kanaloperationen werden deshalb um eine Operation erweitert:

```
public class Channel
{  ...
   public synchronized Object receiveMsgOrNull ()
   {  if (msgQueue.empty()) return null;
      Object result = msgQueue.front();
      msgQueue.dequeue();
      return result;
}  }
```

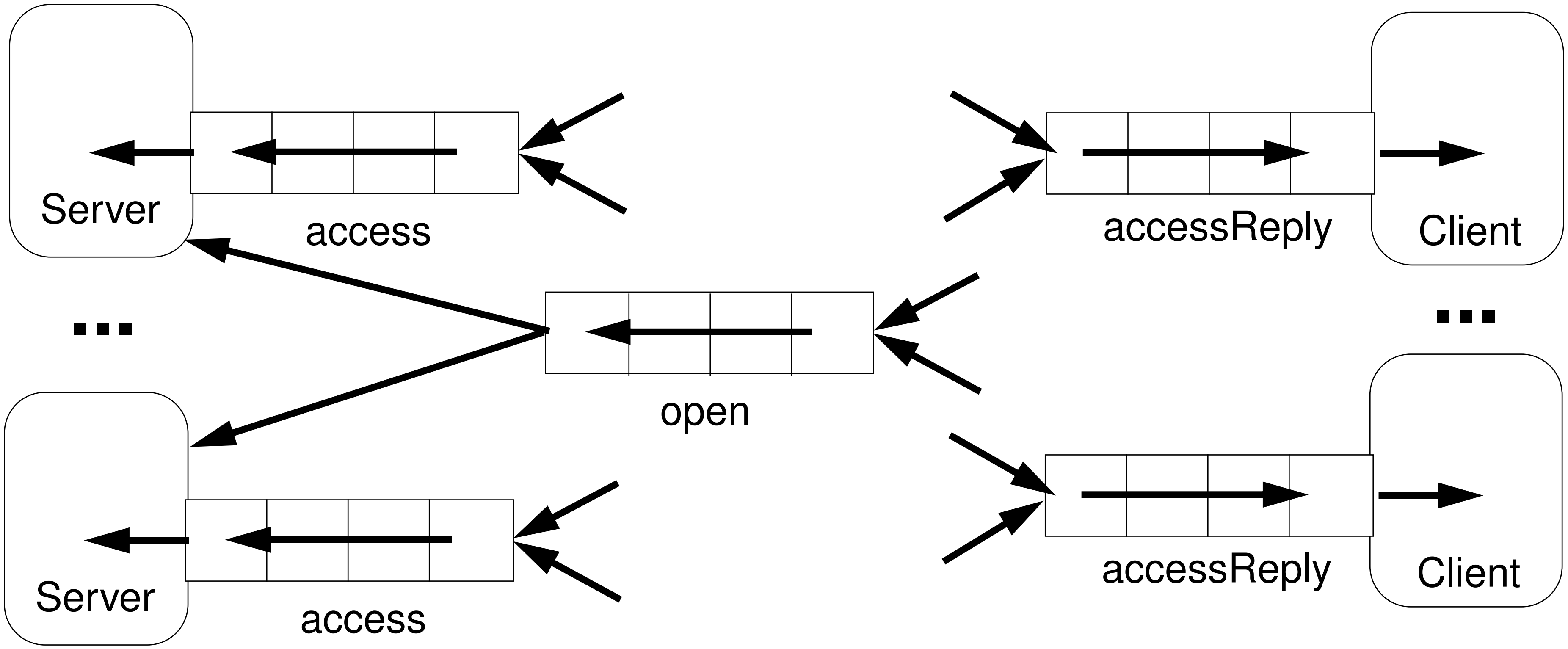
Abfrage mehrerer Kanäle:

```
while (msg == null)
{  if ((msg = ch1.receiveMsgOrNull()) == null)
   if ((msg = ch2.receiveMsgOrNull()) == null)
      Thread.sleep (500);
}
```

Konversationsfolgen zwischen Client und Server

Anwendungsmuster (z. B. Fileserver):

- mehrere gleichartige Server bedienen mehrere Clients
- Client schickt Eröffnungsanfrage auf allgemeinem Kanal (open)
- ein Server übernimmt den Auftrag und führt mit dem Client eine Konversation nach bestimmten Protokoll, z. B. ((read readReply)* | (write writeReply)) close closeReply
- Antwortkanäle werden mit den Botschaften versandt.



Broadcast im Netz

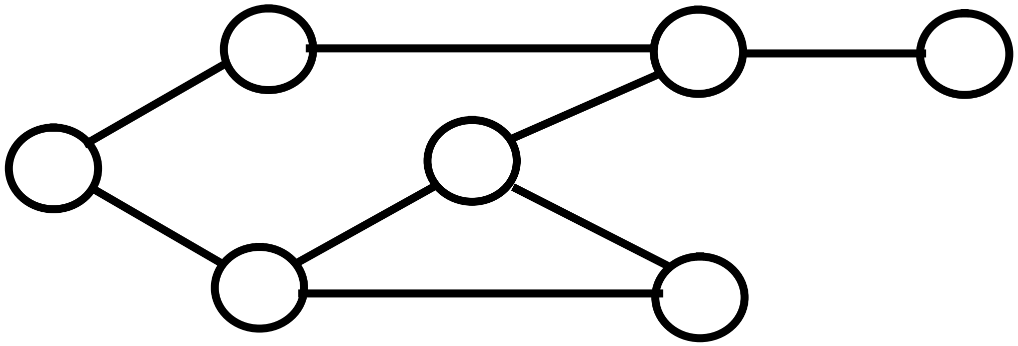
Netz: unregelmäßiger, zusammenhängender, bidirektionaler Graph;
Knoten: Prozess mit einem Input-Port;
Kante: Paar von Referenzen auf Input-Ports der beiden Prozesse an der Kante

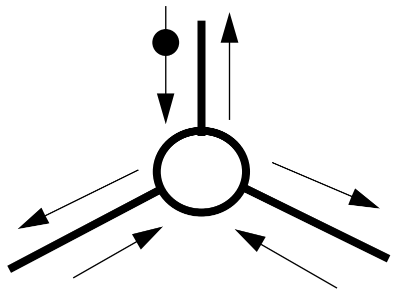
Knoten kennen nur ihre unmittelbaren Nachbarn, deren Identität und Input-Ports

Aufgabe: Von einem Initiator-Knoten aus eine Botschaft an alle Knoten im Netz verteilen.
Am Ende sollen alle Kanäle leer sein.

Methode:

- Knoten wartet auf Botschaft am Input-Port
- Nach erstem Empfang sendet er an **alle** n Nachbarn (auch an den Sender der Botschaft)
- Kanal des ersten Empfangs ist eine Kante des **Spannbaumes**
- Knoten empfängt n-1 redundante Botschaften von den nicht-Spannbaumkanten





insgesamt 2*|Kanten| Botschaften

Probe/Echo im Netz

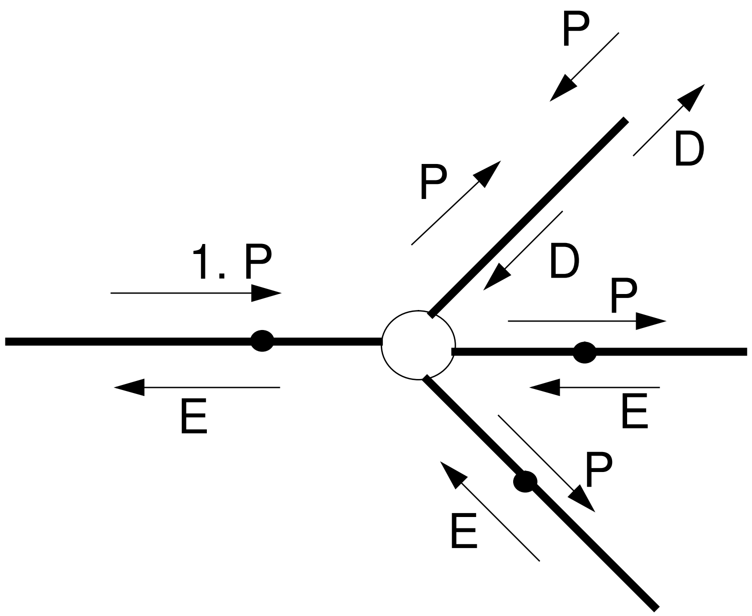
Ein Initiator ruft von allen Knoten im Netz Information ab (Probe);
auf dem Weg durch das Netz wird sie zusammengefasst (Echo),
z. B. Summe lokaler Werte, Topologie des Graphen, globalen Zustand berechnen

Methode:

- Anfragen (Probe) verteilen wie Broadcast
- dabei Spannbaumkanten bestimmen (jeweils erster Empfang)
- Antwort (Echo) auf Spannbaumkanten zurücksenden

Aktionen eines Knotens im Einzelnen:

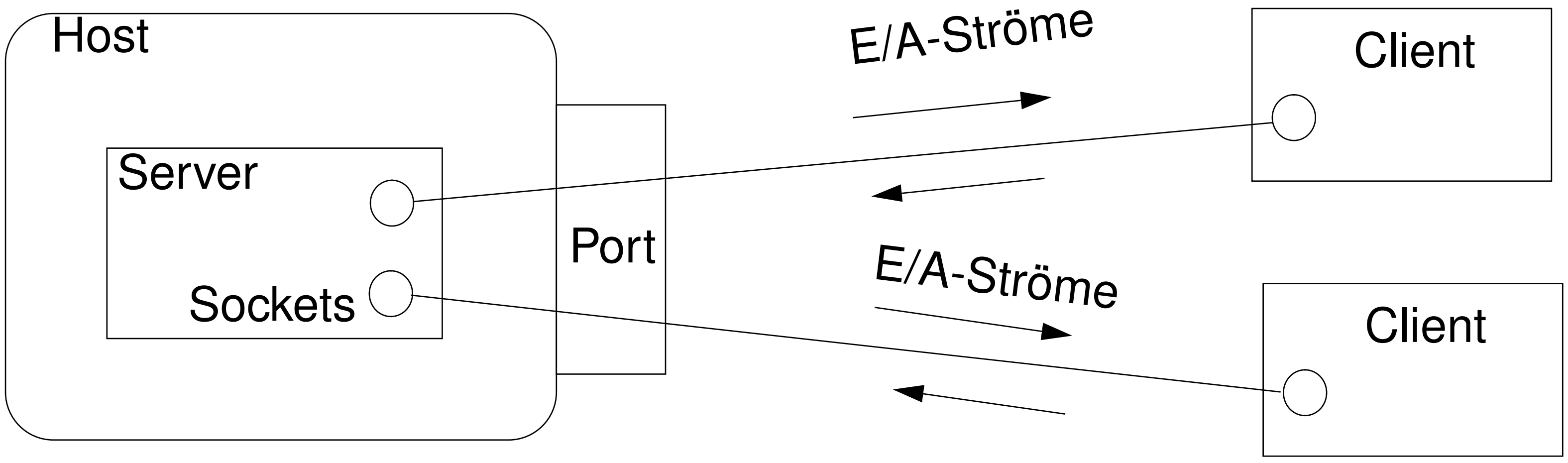
- Jeder Knoten hat einen Input-Port für Probes und Echos zusammen
- Nach erstem Empfang eines Probes von s: Probe an alle n Nachbarn außer s senden
- Alle weiteren eintreffenden Probes mit einer Dummy-Botschaft beantworten
- Insgesamt müssen n-1 Dummies und Echos eintreffen, dann Echo berechnen, an s senden



Rechnerverbindungen über Ports und Sockets

Port:

- Abstrakte Verbindungsstelle eines Rechners; numerisch codiert;
- Server-Prozesse können bestimmte Ports bedienen, z. B. 13: Datum und Uhrzeit;
- Client-Prozesse auf anderen Rechnern können über
- Rechnernamen und Port-Nummer Verbindung aufnehmen



Socket:

- Abstraktion der Netzwerk-Software zur Kommunikation über Ports.
- Sockets werden aus Rechneradresse und Portnummer erzeugt
- Mehrere Sockets auf einem Port zur Bedienung mehrerer Clients
- Auf einem Socket können E/A-Ströme eingerichtet werden.

Sockets und E/A-Ströme

Rechneradresse bestimmen:

```
InetAddress  addr1 = InetAddr.getByName ("java.sun.com"),
             addr2 = InetAddr.getByName ("206.26.48.100"),
             addr3 = InetAddr.getLocalHost();
```

Client-Seite, Socket mit Verbindung zum Server-Rechner erzeugen

```
Socket myServer = new Socket (addr, port);
```

E/A-Ströme auf Socket einrichten:

```
BufferedReader in =
   new BufferedReader
      (new InputStreamReader (myServer.getInputStream()));

PrintWriter out =
   new PrintWriter (myServer.getOutputStream(), true);
```

Server-Seite, speziellen Socket erzeugen, eintreffende Verbindungen annehmen:

```
ServerSocket listener = new ServerSocket (port);
...
Socket client = listener.accept(); ... client.close();
```

Worker-Paradigma

Eine **Menge von gleichartigen Worker-Prozessen**
arbeitet gemeinsam an der Lösung einer Aufgabe.
Jeder bearbeitet einige Teilaufgaben.

Geschwindigkeitsgewinn bei **verteilten Prozessen**.

Anwendung z. B. bei **Lösungsverfahren Branch & Bound**, Divide & Conquer, Backtracking für **kombinatorische Probleme**.

Manager-Prozess
verwaltet die zu lösenden Teilaufgaben und
sammelt die Teillösungen ein.

Worker-Prozess
bearbeitet nacheinander einige Teilaufgaben,
generiert neue Teilaufgaben und
berechnet Teillösungen.

Branch & Bound

Lösungsverfahren Branch & Bound für kombinatorische Probleme (z. B. Travelling Salesman)

Baumstrukturierter Lösungsraum wird nach einer besten Lösung durchsucht.

Allgemeines Schema:

- **Teillösung L erweitern** zu L_1, L_2, ... (z. B. Weg um eine Kante verlängern)
- Ist Teillösung **zulässig**? (z. B. ist der erreichte Knoten neu?)
- Ist L eine **vollständige** Lösung? (z. B. werden alle Knoten erreicht?)
- **MinKosten (L)** = C: jede aus L erweiterte Lösung hat mindestens die Kosten C
  (z. B. Summe der Kantengewichte)
- **Bound**: Kosten der bisher gefundenen besten Lösung

Datenstrukturen: nach MinKosten geordnete Schlange von Teillösungen; Bound-Variable

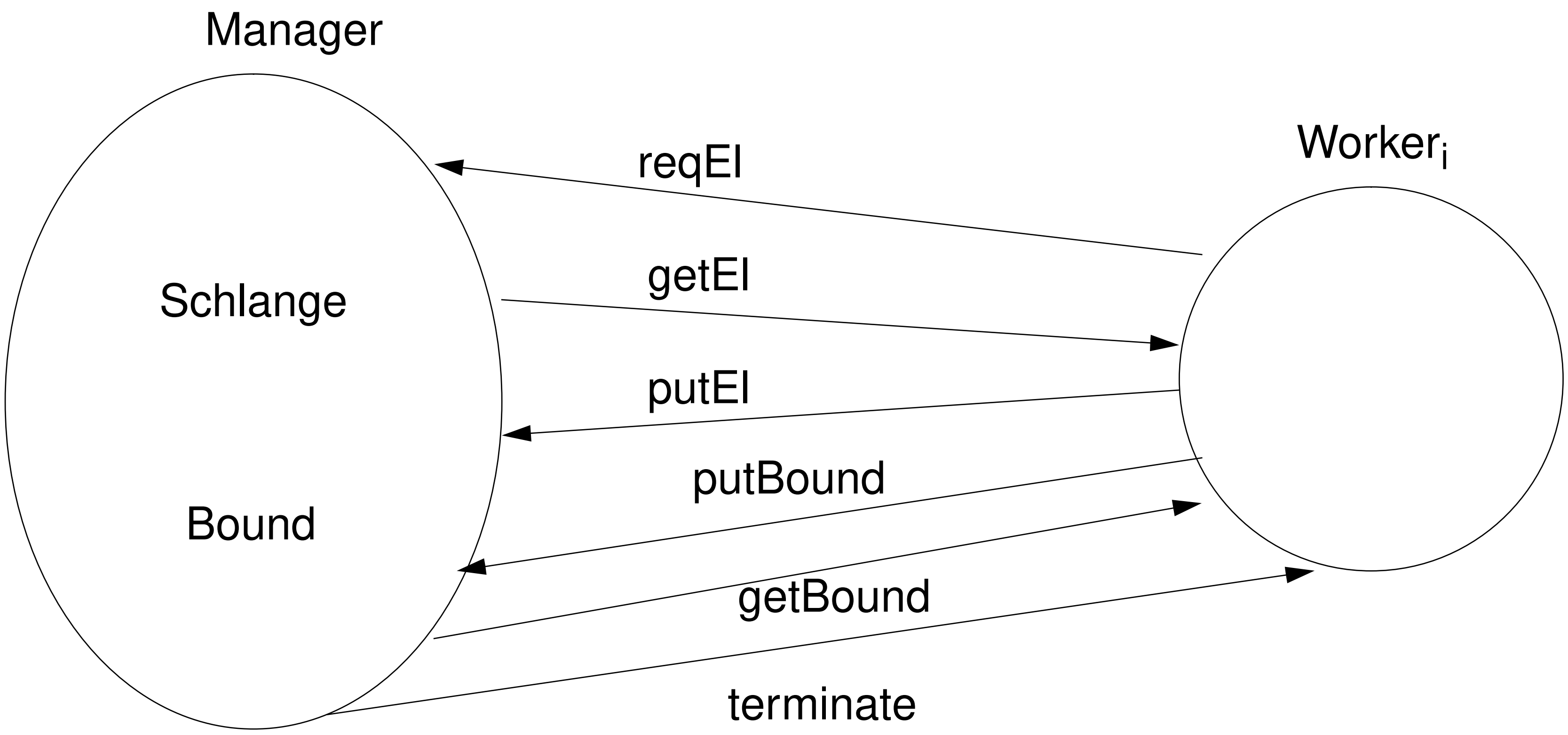
sequentieller Algorithmus:

```
iterieren bis Schlange leer:
    erstes Element erweitern
    neue Elemente prüfen
    ggf. neue Lösung und neuer Bound gefunden
    Schlange aktualisieren
```

Paralleles Branch & Bound (zentral)

Ein **zentraler Manager-Prozess** verwaltet die Schlange und die Bound-Variable.

Ein Worker-Prozess erweitert ein Element, prüft es, berechnet die Kosten, ggf. neuen Bound

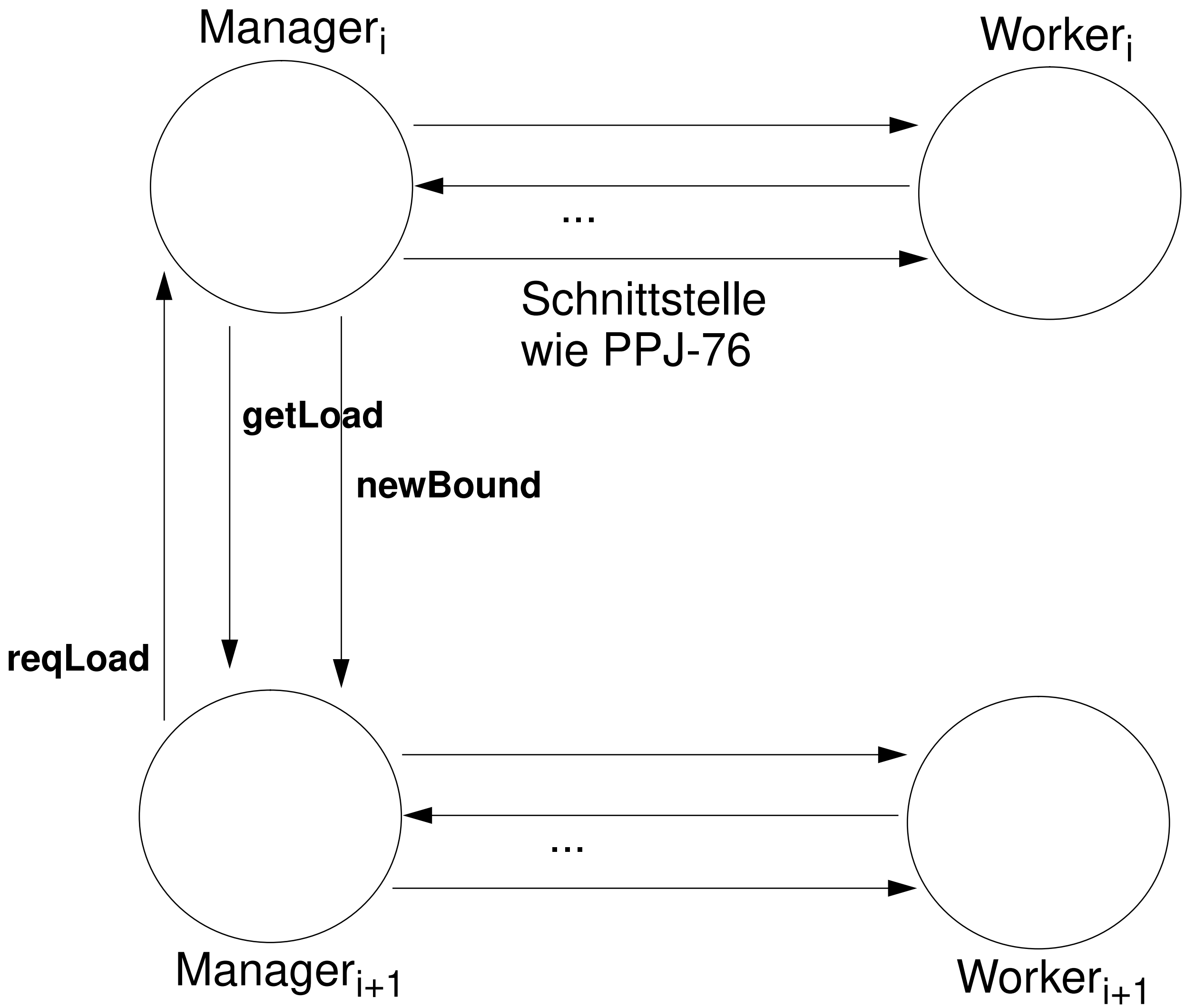


Protokoll: reqEl (getEl [getBound] (putEl | putBound)* reqEl)* terminate

Paralleles Branch & Bound (verteilt)

Verteilte Manager-Prozesse, einer für jeden Worker-Prozess.

Last balancieren mit Nachbarn, z. B. im Ring

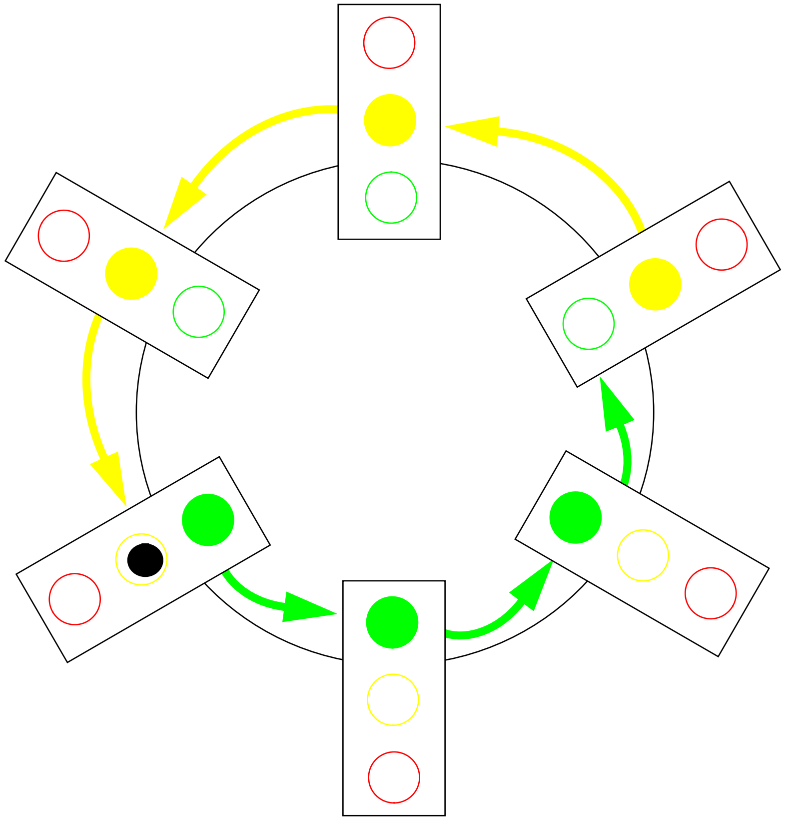


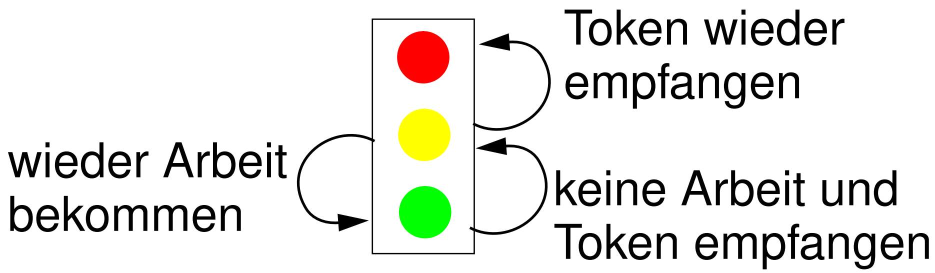
Terminierung?

Terminierung im Ring

Aufgabe: Einen globalen Zustand von Prozessen im unidirektionalen Ring feststellen und allen Prozessen mitteilen, z. B. „alle Prozesse sind inaktiv“.

Idee: Ein Token kreist im Ring und kennzeichnet Prozesse, die den Zustand (inaktiv) eingenommen haben.
In gleicher Richtung kann er wieder zurückgesetzt werden.
Wenn das Token seine eigene Spur erreicht, gilt der Zustand global.



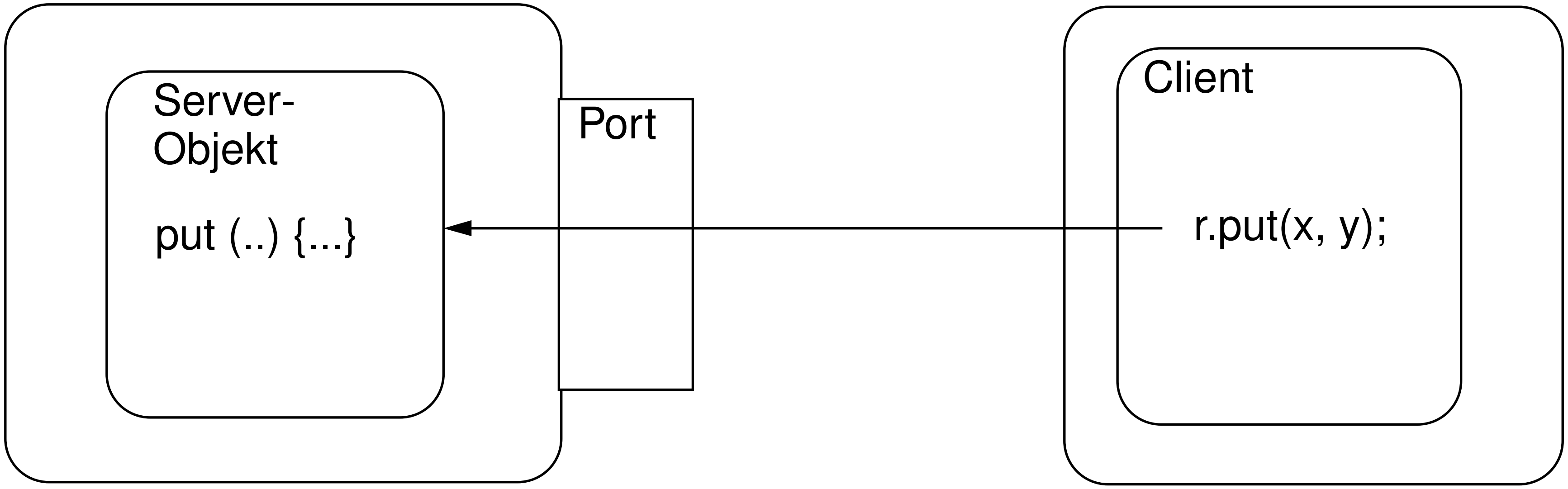


Methodenaufrufe für Objekte auf anderen Rechnern (RMI)

Remote Method Invocation (RMI): Aufruf von Methoden zu Objekten auf anderen Rechnern

In Java durch die Bibliotheken java.rmi verfügbar.

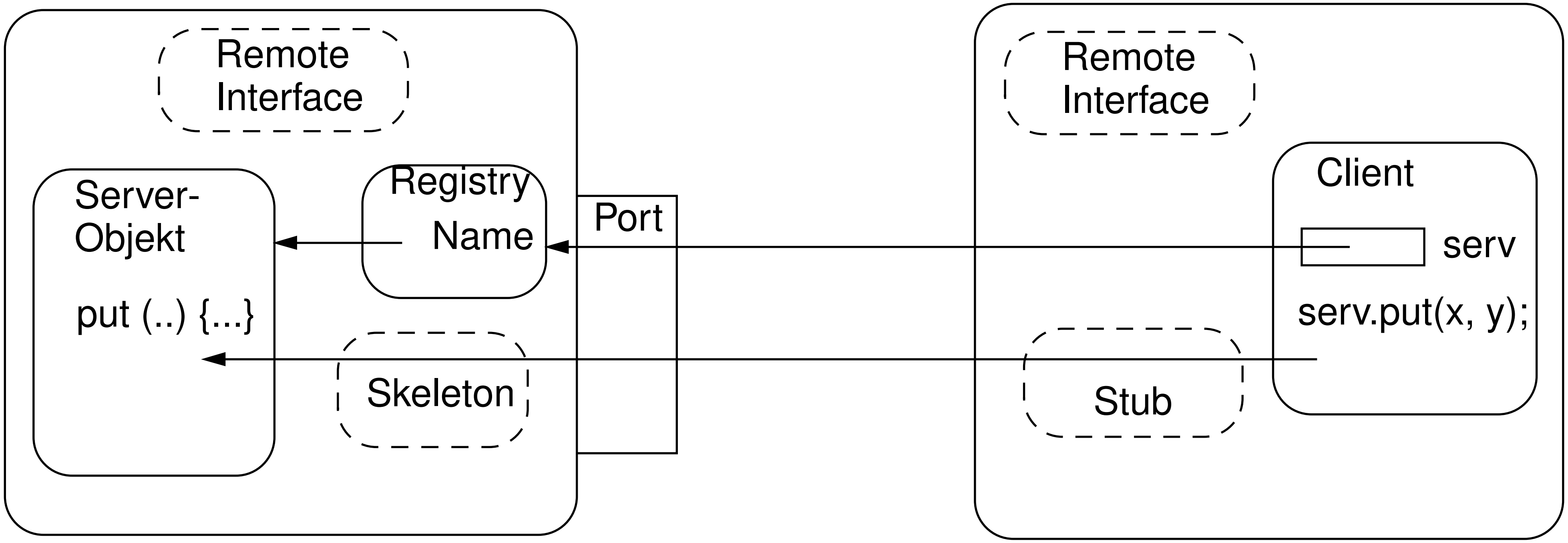
Vergleichbare Techniken: CORBA mit IDL, Microsoft DCOM mit COM



Aufgaben:

- Objekte über Rechnergrenzen hinweg **identifizieren** (Objektverwaltung, Naming Service)
- **Schnittstelle** für Fernzugriffe und ausführbare Repräsentanten der Objekte (Skeleton, Stub)
- **Methodenaufruf**, Parameter und Ergebnis übertragen (Object Serialization)

RMI in Java



Remote Interface: Spezielle Anforderungen an Schnittstellenmethoden

Registry: Prozess auf Systemebene für Rechnerund Port;
leistet Zuordnung von Namen zu Objektreferenzen

Server Skeleton: Repräsentant des Servers für Außenzugriffe auf Server-Objekte,
leistet E/A-Umsetzung Server-seitig,

Client Stub: Repräsentant des Servers, leistet E/A-Umsetzung Client-seitig

RMI Entwicklungsschritte

Am Beispiel: **`Hashtable`** als Server-Objekte verfügbar machen

1. Remote Interface definieren:

```
public interface RemoteMap extends java.rmi.Remote
{ public Object get (Object key) throws RemoteException; ...}
```

2. Adapterklasse zur Anpassung der Server-Klasse an Remote Interface entwickeln:

```
public class RemoteMapAdapter extends UnicastRemoteObject
       implements RemoteMap
{ public RemoteMapAdapter (Hashtable a) { adaptee = a; }
  public Object get (Object key) throws RemoteException
  { return adaptee.get (key); }
  ...
}
```

3. Server-Hauptprogramm erzeugt das Server-Objekt und trägt es in die Registry ein:

```
Hashtable adaptee = new Hashtable();
RemoteMapAdapter adapter = new RemoteMapAdapter (adaptee);
Naming.rebind (registeredObjectName, adapter);
```

4. Skeleton und Stub aus der adaptierten Server-Klasse erzeugen;
   Client-Stub zum Client kopieren:

```
rmic RemoteMapAdapter
```

RMI Entwicklungsschritte (Forts.)

5. Client identifiziert das Server-Objekt auf einem Zielrechner und ruft Methoden auf:
```
Registry remoteRegistry = LocateRegistry.getRegistry (hostName);
RemoteMap serv = (RemoteMap) remoteRegistry.lookup (remObjectName);
v = serv.get (key);
```
6. Auf Server-Rechner eine Registry starten:
```
rmiregistry [port] &
```
Default Port ist 1099
7. Auf Server-Rechner einige Server starten.
8. Auf Client-Rechnern ggf. mehrere Clients starten

Objekte als Parameter von RMI-Aufrufen

Parameter und Ergebnis von RMI-Aufrufen werden über E/A-Ströme übertragen.

Das ist problemlos für Werte von Grundtypen und String.

Bei allgemeinen Objekten wird der Datengehalt der Objektvariablen übertragen und daraus aus der Empfangsseite ein neues Objekt erzeugt.
Die Klasse solcher Objekte muss das Interface Serializable implementieren:

```
import java.io.Serializable;

class SIPair implements java.io.Serializable
{  private String s;
   private int i;

   public SIPair (String a, int b) { s = a; i = b; }
   public String toString () { return s + "-" + i; }
}
```

Synchrone Botschaften

Prozesse kommunizieren und synchronisieren sich direkt miteinander,
bzw. über Kanäle, die höchstens eine Botschaft aufnehmen.

Operationen:

- **send (b):** blockiert bis Partnerprozess bereit ist, die Botschaft zu empfangen
- **receive (v)**: blockiert bis Partnerprozess bereit ist, eine Botschaft zu senden.

Wenn Sender und Empfänger bereit sind, wir die Botschaft übertragen; Zuweisung v := b;

send-receive-Paar ist **Datenübertragung und Synchronisationspunkt** zugleich

Ursprung: Communicating Sequential Processes (CSP) [C.A.R. Hoare, CACM 21, 8, 178]

Notation in CSP und Occam:

```
p:   ...   q ! Ausdruck ...
```
sende den Wert des Ausdrucks an Prozess q

```
q:   ...   p ? Variable ...
```
empfange von Prozess p einen Wert in der Variablen

auch mit verschiedenen Ports an einem Prozess und mit zusammengesetzten Botschaften:

```
p:   ...   q ! Port1 (a1,..,an) ...

q:   ...   p ? Port1 (v1,..,vn) ...
```

Selektives Warten

Verzweigung abhängig von Bedingungen und **Kommunikationsbereitschaft (Guard)**

```
if Bedingung1; p ! x-> Anweisung1
[] Bedingung2; q ? y-> Anweisung2
[] Bedingung3; r ? z-> Anweisung3
fi
```

Kommunikationsanweisung im Guard liefert
wahr, falls Partnerprozess bereit ist, **falsch**, falls er terminiert ist, **offen** sonst
Sind einige Guards wahr, wird einer gewählt, die Kommunikation und die Anweisung ausgeführt
Sind alle Guards falsch, wird die Selektion Ausführen einer Anweisung beendet.
Sonst wird gewartet, bis einer der beiden Fälle eintritt.

Selektive Schleifen werden wiederholt bis alle Guards falsch liefern:

```
do Bedingung1; p ! x-> Anweisung1
[] Bedingung2; r ? z-> Anweisung2
fi
```

Auch indizerte Selektion:

```
if (i: 1..n) Bedingung; p[i] ? v -> Anweisungen fi
```

Beispiel: Beschränkter Puffer:

```
do cnt < N; Prod ? buf[rear]  -> cnt++; rear = rear % N + 1;
[] cnt > 0; Cons ! buf[front] -> cnt--; front = front % N + 1;
od
```

Präfix-Summen mit synchronen Botschaften

Synchrone Kommunikation leistet **Übertragung des Datums und Synchronisation.**
Nur notwendige Synchronisation (vergl. Barrieren, PPJ-47)

```
const N := 6; var a [0:N-1] : int;

process Worker (i := 0 to N-1)                  ein Prozess für jedes Element
   var d := 1, sum, new: int

   sum := a[i];

                    {Invariante SUM: sum = a[i-d+1] + ... + a[i]}
   do d < N-1 ->

      if (i+d) < N -> Worker(i+d) ! sum fi          alter Wert nach rechts

      if (i-d) >= 0-> Worker(i-d) ? new; sum := sum + new fi
                                                    neuer Wert von links
      d := 2*d                                      Abstandsverdopplung
   od                                               {SUM and d >= N-1}
end
```

Es gibt immer paarige Operationen **`Worker(i+d) ! sum`** und **`Worker(j-d) ? new`**
=> kein Deadlock

Client/Server-Schema mit synchronen Botschaften

Technik: für jede vom Server angebotene Operationsart Kommunikation über 2 Ports:

- oprReq für die Übertragung der Parameter
- oprRepl für die Übertragung der Antwort

Schema der Client-Prozesse:

```
process Client (I := 1 to N)
   ...
   Server ! oprReq (myArgs)
   Server ? oprRepl (myRes)
   ...
end
```

Schema des Server-Prozesses:

```
process Server ()
   ...
   do (c: 1..N) BedingungOpr1; Client[c] ? oprReq(oprArgs)
                     -> Anfrage bearbeiten ...
                        Client[c] ! oprRepl(oprResults)
   [] ebenso für weitere Operationen ...
   od
end
```

Verständnisfragen (1)

Grundbegriffe

1. Erklären Sie die Begriffe sequentielle parallele, verzahnte, nebenläufige Prozesse.
2. Beschreiben Sie die 2 Arten in Java Threads erzeugen (3 Schritte).
3. Geben Sie die wichtigsten Methoden der Klasse Thread an.
4. Erklären Sie die Begriffe Atomare Aktion, Art-most-once Eigenschaft
5. Wie wird Interferenz zwischen Prozessen definiert?

Monitore

6. Erklären Sie wie Monitore die 2 Synchronisationskonzepte realisieren.
7. Semantik von Bedingungsvariablen und Varianten dazu.
8. Aus welchen 3 Gründen können Prozesse am Monitor warten?
9. Beschränkter Puffer auch mit 1 Bedingungsvariablen?
10. Warum muss man bei signal-and-continue die Bedingung in Schleife prüfen?
11. Eigenschaften der Monitore in Java.
12. Unter welchen Umständen kann man notify statt notifyAll verwenden?
13. Wann gilt eine Monitor-Invariante?

Verständnisfragen (2)

14. Systematischer Monitor-Entwurf in 5 Schritten

15. Monitor-Invariante zum Leser/Schreiber-Schema

16. Strategien ausdrücken: Schreiber bevorzugen. Keinen bevorzugen.

17. Entwurfsmethode zur „Begegnung von Prozessen“

18. Wie werden Wartebedingungen und Weck-Operationen bei der Zählvariablenmethode eingefügt?

Barrieren

19. Abstandsverdopplung am Beispiel Präfixsummen

20. Barrieren-Regel

21. Baum-Barriere

22. Symmetrische, verteilte Barriere

Datenparallelität

23. Datenparallel Listenenden finden

24. Schleifen zu trapezförmigem Iterationsraum angeben

25. Abhängigkeitsvektoren, Zulässigkeit in sequentiellen Schleifen

Verständnisfragen (3)

26. SRP-Transformationen
27. Benutzung der Transformationsmatrizen
28. Mit welchen Transformationen kann man die innere Schleife parallelisieren, wenn die Abhängigkeitsvektoren (0,1) und (1,0) vorliegen?

Asynchrone Botschaften

29. Begriff des Kanals und seine Operationen
30. wichtige Kanal-Strukturen
31. Client/Server Kanal-Strukturen
32. Welches Problem tritt auf, wenn mehrere Prozesse jeweils von mehreren Kanälen empfangen?
33. Erklären Sie den Begriff Konversationsfolgen.
34. Welche Operationen führt ein Knoten bei Broadcast im Netz aus?
35. Welche Operationen führt ein Knoten bei Probe/Echo im Netz aus?
36. Wieviele Botschaften werden bei Probe/Echo versandt?

Verständnisfragen (4)

Botschaften in verteilten Systemen

37. Wie geht man in Java mit Ports und Sockets um?
38. Erläutern Sie das Worker-Paradigma.
39. Skizzieren Sie die Prozess-Schnittstellen für verteiltes Branch-and-Bound.
40. Erläutern Sie die Technik Terminierung im Ring.
41. Was sind die Aufgaben bei Methodenfernanrufen (RMI)?
42. Geben Sie die Entwicklungsschritte zur RMI-Implementierung an.

Synchrone Botschaften

43. Vergleichen Sie die Grundkonzepte synchroner und asynchroner Botschaften.
44. Erklären Sie das Konstrukt zu selektiven Warten bei synchronen Botschaften.
45. Warum sind Programme mit synchronen Botschaften weniger redundant als mit asynchronen Botschaften.
46. Skizzieren Sie einen Server zur Ressourcenvergabe nach dem Schema für synchrone Botschaften.