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.

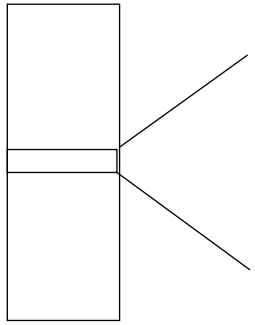
- Strömungsmechanik
- Bildverarbeitung
- Differentialgleichungen lösen
- Finite Element Methode in Entwurfssystemen

© 2000 bei Prof. Dr. Uwe Kastens

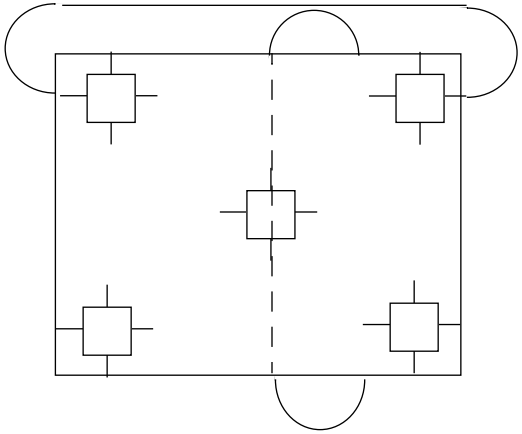
Datenparallelität als Architekturmodell

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



Programm



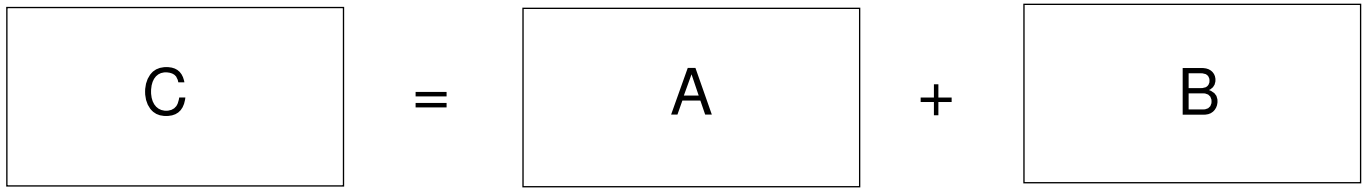
Prozessorfeld

© 2000 bei Prof. Dr. Uwe Kastens

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:



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!

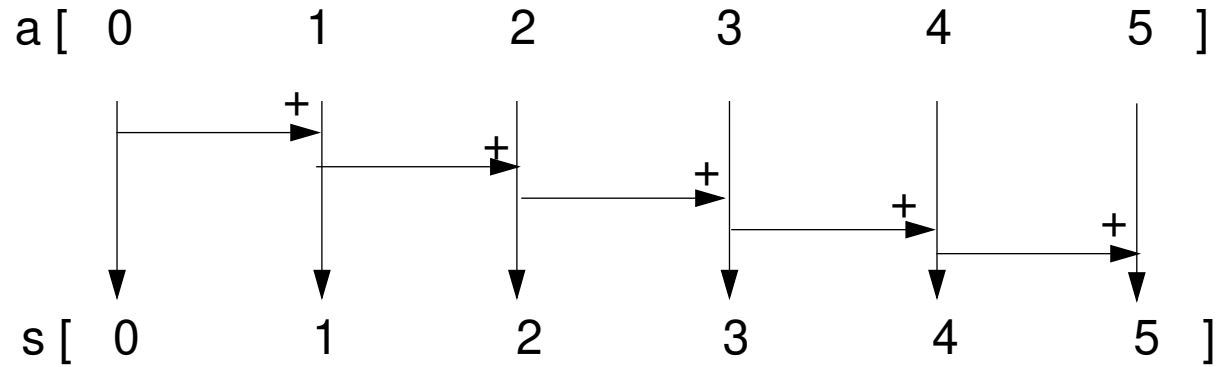
- diese Schleifen sind unmittelbar parallelisierbar, da keine **Datenabhängigkeiten**
- **Datenabbildung** ist trivial: Array-Elemente [i,j] auf Prozess [i,j]
- **Kommunikation** ist nicht nötig
- keine **algorithmische Idee** nötig

© 2000 bei Prof. Dr. Uwe Kastens

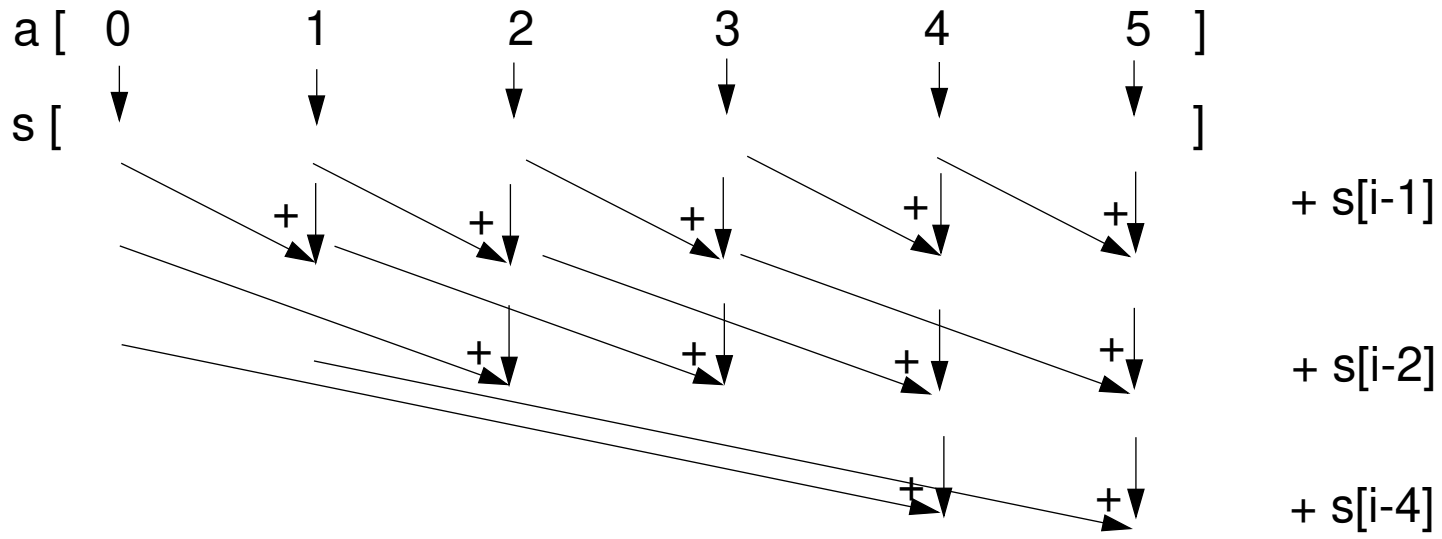
Beispiel Präfixsummen

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

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



parallele algorithmische Idee:



© 2000 bei Prof. Dr. Uwe Kastens

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

© 2000 bei Prof. Dr. Uwe Kastens

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)

© 2000 bei Prof. Dr. Uwe Kastens

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
}  }  }
```

© 2000 bei Prof. Dr. Uwe Kastens

Verteilte Barriere als Baum

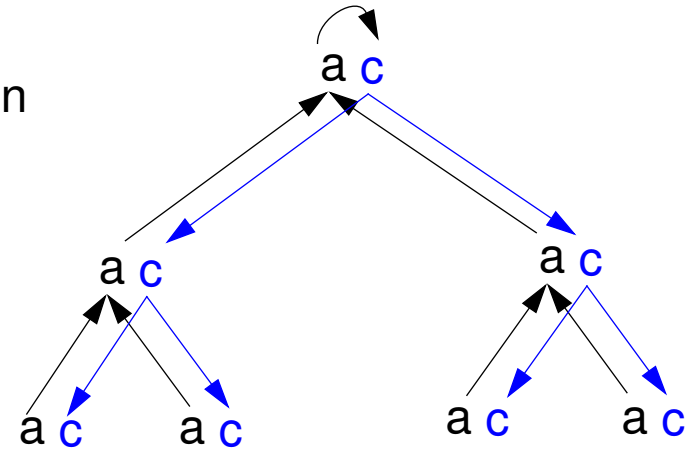
Barrierensynchronisation der Worker-Prozesse als **Binärbaum** organisiert.
Vermeidet den Engpass der zentralen Synchronisation.

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.

© 2000 bei Prof. Dr. Uwe Kastens

Symmetrische, verteilte Barriere (Dissemination)

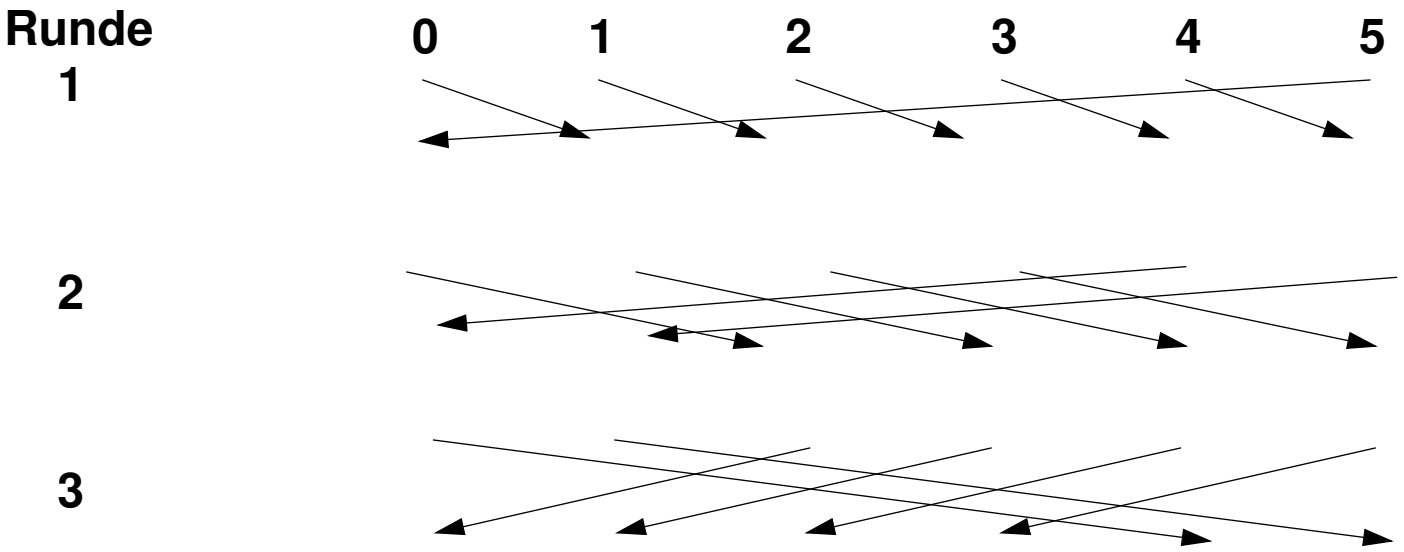
Prozesse synchronisieren sich paarweise in Runden mit Abstandsverdopplung

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



© 2000 bei Prof. Dr. Uwe Kastens

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
}  }  }
```

© 2000 bei Prof. Dr. Uwe Kastens

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

Parallele Schleife im Gleichtakt

Barrieren automatisch für jede Anweisung

© 2000 bei Prof. Dr. Uwe Kastens