

Distributed-Memory Programmiermodelle I

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 11/12



Distributed-Memory Programmiermodelle I

Kommunikation über Nachrichtenaustausch

- Synchroner Nachrichtenaustausch
- Asynchroner Nachrichtenaustausch
- Globale Kommunikation bei
 - ▶ Store-and-Forward oder
 - ▶ Cut-Through Routing
- Globale Kommunikation auf verschiedenen Topologien
 - ▶ Ring
 - ▶ Feld (2D / 3D)
 - ▶ Hypercube

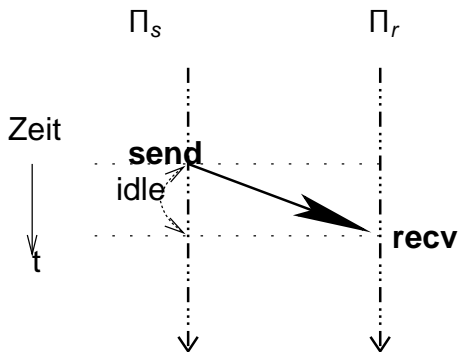


Synchroner Nachrichtenaustausch I

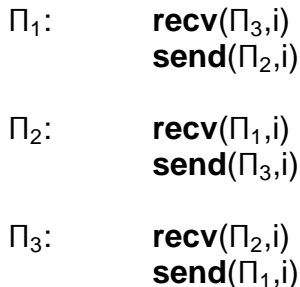
- Für den Nachrichtenaustausch benötigen wir mindestens zwei Funktionen:
 - ▶ **send**: Überträgt einen Speicherbereich aus dem Adressraum des Quellprozesses in das Netzwerk mit Angabe des Empfängers.
 - ▶ **recv**: Empfängt einen Speicherbereich aus dem Netzwerk und schreibt ihn in den Adressraum des Zielprozesses.
- Wir unterscheiden:
 - ▶ Zeitpunkt zu dem eine Kommunikationsfunktion beendet ist.
 - ▶ Zeitpunkt zu dem die Kommunikation wirklich stattgefunden hat.
- Bei synchroner Kommunikation sind diese Zeitpunkte identisch, d.h.
 - ▶ **send** blockiert bis der Empfänger die Nachricht angenommen hat.
 - ▶ **recv** blockiert bis die Nachricht angekommen ist.
- Syntax in unserer Programmiersprache:
 - ▶ **send**(*dest* – *process*, *expr*₁, ..., *expr*_{*n*})
 - ▶ **recv**(*src* – *process*, *var*₁, ..., *var*_{*n*})



Synchroner Nachrichtenaustausch II



(a)



(b)

(a) Synchronisation zweier Prozesse durch ein **send/recv** Paar

(b) Beispiel für eine Verklemmung



Synchroner Nachrichtenaustausch III

Es gibt eine Reihe von Implementierungsmöglichkeiten.

- Senderinitiiert, *three-way handshake*:
 - ▶ Quelle Q schickt *ready-to-send* an Ziel Z.
 - ▶ Ziel schickt *ready-to-recv* wenn **recv** ausgeführt wurde.
 - ▶ Quelle überträgt Nachricht (variable Länge, single copy).
- Empfängerinitiiert, *two-phase protocol*:
 - ▶ Z schickt *ready-to-recv* an Q wenn **recv** ausgeführt wurde.
 - ▶ Q überträgt Nachricht (variable Länge, single copy).
- Gepuffertes Senden
 - ▶ Q überträgt Nachricht sofort, Z muss eventuell zwischenspeichern.
 - ▶ Hier stellt sich das Problem des endlichen Speicherplatzes!



Synchroner Nachrichtenaustausch IV

- Synchrones **send/recv** ist nicht ausreichend um alle Kommunikationsaufgaben zu lösen!
- Beispiel: Im Erzeuger-Verbraucher-Problem wird der Puffer als eigenständiger Prozess realisiert. In diesem Fall kann der Prozess nicht wissen mit welchem Erzeuger oder Verbraucher er als nächstes kommunizieren wird. Folglich kann ein blockierendes **send** zur Verklemmung führen.
- Lösung: Bereitstellung zusätzlicher *Wächterfunktionen*, die überprüfen ob ein **send** oder **recv** zur Blockade führen würde:

- ▶ **int sprobe**(*dest* – *process*)
- ▶ **int rprobe**(*src* – *process*).

sprobe liefert 1 falls der Empfängerprozess bereit ist zu empfangen, d.h. ein **send** wird nicht blockieren:

- ▶ **if (sprobe(Π_d)) send(Π_d, \dots);**

Analog für **rprobe**.

- Wächterfunktionen blockieren nie!



Synchroner Nachrichtenaustausch V

- Man braucht nur eine der beiden Funktionen.
 - ▶ **rprobe** lässt sich leicht in das senderinitiierte Protokoll integrieren.
 - ▶ **sprobe** lässt sich leicht in das empfängerinitiierte Protokoll integrieren.
- Ein Befehl mit ähnlicher Wirkung wie **rprobe** ist:
 - ▶ **recv_any**(*who*, *var*₁, . . . , *var*_{*n*}).

Er erlaubt das empfangen von einem *beliebigen* Prozess, dessen ID in der Variablen *who* abgelegt wird.

- **recv_any** wird am einfachsten mit dem senderinitiierten Protokoll implementiert.



Asynchroner Nachrichtenaustausch I

- Befehle zum asynchronen Nachrichtenaustausch:
 - ▶ **asend**(*dest* – *process*, *expr*₁, ..., *expr*_{*n*})
 - ▶ **arecv**(*src* – *process*, *var*₁, ..., *var*_{*n*})
 - Hier zeigt das Beenden der Kommunikationsfunktion *nicht* an, dass die Kommunikation tatsächlich stattgefunden hat. Dies muss mit extra Funktionen erfragt werden.
 - Man stellt sich vor, es wird ein *Auftrag* an das System gegeben die entsprechende Kommunikation durchzuführen, sobald sie möglich ist. Der Rechenprozess kann währenddessen andere Dinge tun (*communication hiding*).
 - Syntax:
 - ▶ **msgid asend**(*dest* – *process*, *var*₁, ..., *var*_{*n*})
 - ▶ **msgid arecv**(*src* – *process*, *var*₁, ..., *var*_{*n*})
- blockieren nie! **msgid** ist eine Quittung für den Kommunikationsauftrag.



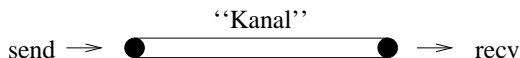
Asynchroner Nachrichtenaustausch II

- Vorsicht: Die Variablen var_1, \dots, var_n dürfen nach Absetzen des Kommunikationsbefehls nicht mehr modifiziert werden!
- Dies bedeutet, dass das Programm den Speicherplatz für die Kommunikationsvariablen selbst verwalten muss. Alternative wäre das gepufferte Senden, was aber mit Unwägbarkeiten und doppeltem Kopieraufwand verbunden ist.
- Schließlich muss man testen ob die Kommunikation stattgefunden hat (d.h. der Auftrag ist bearbeitet):
 - ▶ **int** success(**msgid** *m*)
- Danach dürfen die Kommunikationsvariablen modifiziert werden, die Quittung ist ungültig geworden.



Synchroner/Asynchroner Nachrichtenaustausch

- Synchroner und Asynchroner Operationen dürfen gemischt werden. Ist im MPI Standard so implementiert.
- Bisherige Operationen waren *verbindungslos*.
- Alternative sind kanalorientierte Kommunikationsoperationen (oder virtuelle Kanäle):



- ▶ Vor dem erstmaligen senden/empfangen an/von einem Prozess muss mittels **connect** eine Verbindung aufgebaut werden.
 - ▶ **send/recv** erhalten einen Kanal statt einen Prozess als Adresse.
 - ▶ Mehrere Prozesse können auf einen Kanal senden aber nur einer empfangen.
 - ★ **send**(*channel*, *expr*₁, ..., *expr*_{*n*})
 - ★ **recv**(*channel*, *var*₁, ..., *var*_{*n*}).
- Wir werden keine kanalorientierten Funktionen verwenden.



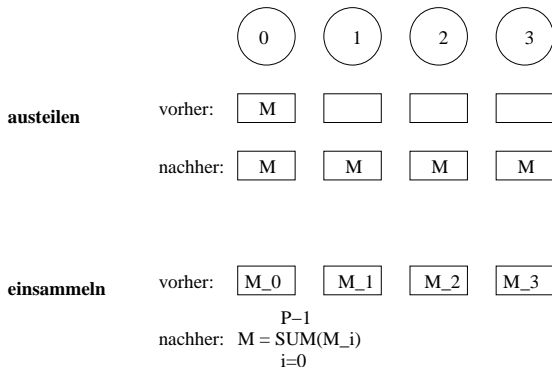
Globale Kommunikation

- Ein Prozess will ein *identisches* Datum an alle anderen Prozesse
- *one-to-all broadcast*
- duale Operation ist das Zusammenfassen von individuellen Resultaten auf einem Prozeß, z.B. Summenbildung (alle assoziativen Operatoren sind möglich)
- Wir betrachten Austeilen auf verschiedenen Topologien und berechnen Zeitbedarf für store & forward und cut-through routing
- Algorithmen für das Einsammeln ergeben sich durch Umdrehen der Reihenfolge und Richtung der Kommunikationen
- Folgende Fälle werden einzeln betrachtet:
 - ▶ Einer-an-alle
 - ▶ Alle-an-alle
 - ▶ Einer-an-alle mit individuellen Nachrichten
 - ▶ Alle-an-alle mit individuellen Nachrichten



Einer-an-alle: Ring

Ein Prozess will ein *identisches* Datum an alle anderen Prozesse versenden:



Hier: Kommunikation im Ring mit store & forward:



Einer-an-alle: Ring

Programm (Einer-an-alle auf dem Ring)

parallel *one-to-all-ring*

```
{  
  const int P;  
  process  $\Pi$ [int  $p \in \{0, \dots, P - 1\}$ ]{  
    void one_to_all_broadcast(msg *mptr) {  
      // Nachrichten empfangen  
      if ( $p > 0 \wedge p \leq P/2$ )  
        recv( $\Pi_{p-1}$ , *mptr);  
      if ( $p > P/2$ )  
        recv( $\Pi_{(p+1)\%P}$ , *mptr);  
      // Nachrichten an Nachfolger weitergeben  
      if ( $p \leq P/2 - 1$ )  
        send( $\Pi_{p+1}$ , *mptr);  
      if ( $p > P/2 + 1 \vee p == 0$ )  
        send( $\Pi_{(p+P-1)\%P}$ , *mptr);  
    }  
    ...;  
    m = ...;  
    one_to_all_broadcast(&m);  
  }  
}
```

Der Zeitbedarf für die Operation beträgt (nearest-neighbor Kommunikation!):

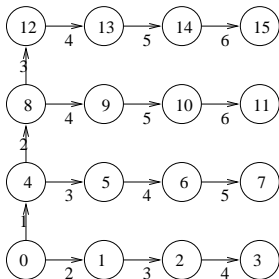
$$T_{\text{one-to-all-ring}} = (t_s + t_h + t_w \cdot n) \left\lceil \frac{P}{2} \right\rceil,$$

wobei $n = |*mptr|$ immer die Länge der Nachricht bezeichne.



Einer-an-alle: Feld

Nun setzen wir eine 2D-Feldstruktur zur Kommunikation voraus. Die Nachrichten laufen folgende Wege:



Beachte den zweidimensionalen Prozessindex:



Einer-an-alle: Feld

Programm (Einer an alle auf dem Feld)

```
parallel one-to-all-array
{
  int P, Q; // Feldgröße in x- und y-Richtung
  process  $\Pi$ [int[2] (p, q)  $\in$  {0, ..., P - 1}  $\times$  {0, ..., Q - 1}] {
    void one_to_all_broadcast(msg *mptr) {
      if (p == 0) // erste Spalte
      {
        if (q > 0) recv( $\Pi$ (p, q-1), *mptr);
        if (q < Q - 1) send( $\Pi$ (p, q+1), *mptr);
      }
      else
      if (p < P - 1)
      {
        recv( $\Pi$ (p-1, q), *mptr);
        send( $\Pi$ (p+1, q), *mptr);
      }
    }

    msg m=...;
    one_to_all_broadcast(&m);
  }
}
```

Die Ausführungszeit für $P = 0$ beträgt für ein 2D-Feld

$$T_{\text{one-to-all-array2D}} = 2(t_s + t_h + t_w \cdot n)(\sqrt{P} - 1)$$

und für ein 3D-Feld

$$T_{\text{one-to-all-array3D}} = 3(t_s + t_h + t_w \cdot n)(P^{1/3} - 1).$$

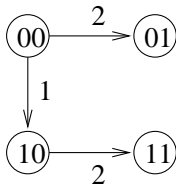


Einer-an-alle: Hypercube

- Wir gehen rekursiv vor. Auf einem Hypercube der Dimension $d = 1$ ist das Problem trivial zu lösen:



- Auf einem Hypercube der Dimension $d = 2$ schickt 0 erst an 2 und das Problem ist auf 2 Hypercubes der Dimension 1 reduziert:



- Allgemein schicken im Schritt $k = 0, \dots, d - 1$ die Prozesse

$$\begin{array}{l}
 \underbrace{p_{d-1} \dots p_{d-k}}_{k \text{ Dimens.}} \quad 0 \quad \underbrace{0 \dots 0}_{d-k-1 \text{ Dimens.}} \\
 \text{je eine Nachricht an} \quad \underbrace{p_{d-1} \dots p_{d-k}}_{k \text{ Dimens.}} \quad 1 \quad \underbrace{0 \dots 0}_{d-k-1 \text{ Dimens.}}
 \end{array}$$



Einer-an-alle: Hypercube

Programm (Einer an alle auf dem Hypercube)

parallel *one-to-all-hypercube*

```
{
  int d, P = 2d;
  process  $\Pi$ [int p  $\in$  {0, ..., P - 1}]{
    void one_to_all_broadcast(msg *mptr) {
      int i, mask = 2d - 1;
      for (i = d - 1; i  $\geq$  0; i --){
        mask = mask  $\oplus$  2i;
        if (p & mask == 0) {
          if (p & 2i == 0) //die letzten i Bits sind 0
            send( $\Pi_{p \oplus 2^i}$ , *mptr);
          else
            rcv( $\Pi_{p \oplus 2^i}$ , *mptr);
        }
      }
    }
  }
  msg m = „bla“; one_to_all_broadcast(&m);
}
```

Der Zeitbedarf ist

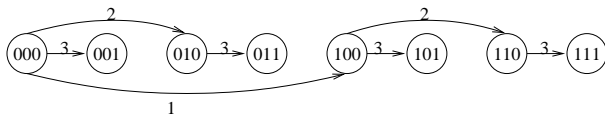
$$T_{\text{one-to-all-HC}} = (t_s + t_h + t_w \cdot n) \text{ld } P$$

Beliebige Quelle $src \in \{0, \dots, P - 1\}$: Ersetze jedes p durch $(p \oplus src)$.



Einer-an-alle: Ring und Feld mit cut-through routing

- Bildet man den Hypercubealgorithmus auf einen Ring ab, erhalten wir folgende Kommunikationsstruktur:



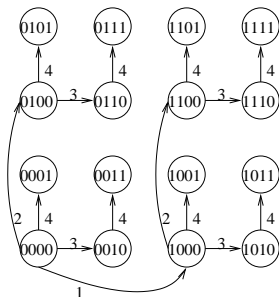
- Es werden keine Leitungen doppelt verwendet, somit erhält man bei cut-through routing:

$$\begin{aligned} T_{\text{one-to-all-ring-ct}} &= \sum_{i=0}^{\text{ld } P - 1} (t_s + t_w \cdot n + t_h \cdot 2^i) \\ &= (t_s + t_w \cdot n) \text{ld } P + t_h(P - 1) \end{aligned}$$



Einer-an-alle: Ring und Feld mit cut-through routing

Bei Verwendung einer Feldstruktur erhält man folgende Kommunikationsstruktur:



Wieder gibt es keine Leitungskonflikte und wir erhalten:

$$T_{\text{one-to-all-field-ct}} = \underbrace{2}_{\substack{\text{jede} \\ \text{Entfernung 2} \\ \text{mal}}} \sum_{i=0}^{\frac{\log P}{2} - 1} (t_s + t_w \cdot n + t_h \cdot 2^i)$$



Einer-an-alle: Ring und Feld mit cut-through routing

$$\begin{aligned} T_{\text{one-to-all-field-ct}} &= (t_s + t_w \cdot n) 2^{\frac{\text{ld } P}{2}} + t_h \cdot 2 \underbrace{\sum_{i=0}^{\frac{\text{ld } P}{2} - 1} 2^i}_{= 2^{\frac{\text{ld } P}{2}} - 1} \\ &= \text{ld } P (t_s + t_w \cdot n) + t_h \cdot 2(\sqrt{P} - 1) \end{aligned}$$

Insbesondere beim Feld ist der Term mit t_h vernachlässigbar und wir erhalten die Hypercubepformance auch auf weniger mächtigen Topologien! Selbst für $P = 1024 = 32 \times 32$ fällt t_h nicht ins Gewicht, somit werden dank cut-through routing keine physischen Hypercube-Strukturen mehr benötigt.

