

Paralleles Höchstleistungsrechnen

Distributed-Memory Programmiermodelle II

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 425
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

17. November 2009



Distributed-Memory Programmiermodelle II

Kommunikation über Nachrichtenaustausch

- MPI Standard
- Globale Kommunikation auf verschiedenen Topologien
 - ▶ Feld (1D / 2D / 3D)
 - ▶ Hypercube
- Lokaler Austausch



MPI: Einführung

Das *Message Passing Interface* (MPI) ist eine portable Bibliothek von Funktionen zum Nachrichtenaustausch zwischen Prozessen.

- MPI wurde 1993/94 von einem internationalen Gremium entworfen.
- Ist auf praktisch allen Plattformen verfügbar, inklusive der freien Implementierungen MPICH und LAM.
- Merkmale:
 - ▶ Bibliothek zum Binden mit C-, C++- und FORTRAN-Programmen (keine Spracherweiterung).
 - ▶ Große Auswahl an Punkt-zu-Punkt Kommunikationsfunktionen.
 - ▶ Globale Kommunikation.
 - ▶ Datenkonversion für heterogene Systeme.
 - ▶ Teilmengenbildung und Topologien.
- MPI besteht aus über 125 Funktionen, die auf über 200 Seiten im Standard beschrieben werden. Daher können wir nur eine kleine Auswahl der Funktionalität betrachten.
- MPI-1 hat keine Möglichkeiten zur dynamischen Prozesserzeugung, dies ist in MPI-2 möglich, ebenso Ein-/Ausgabe.



MPI: Hello World

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
    int my_rank, P;
    int dest, source;
    int tag=50;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&P);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

    if (my_rank!=0)
    {
        sprintf(message,"I am process %d\n",my_rank);
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,
                dest,tag,MPI_COMM_WORLD);
    }
    else
    {
        puts("I am process 0\n");
        for (source=1; source<P; source++)
        {
            MPI_Recv(message,100,MPI_CHAR,source,tag,
                    MPI_COMM_WORLD,&status);
            puts(message);
        }
    }
    MPI_Finalize();

    return 0;
}
```

- SPMD-Stil!

- Übersetzen und starten geht mit

mpicc -o hello hello.c

mpirun -machinefile machines -np 8 hello

- machines enthält Namen der zu benutzenden Rechner.



MPI: Blockierende Kommunikation I

- MPI unterstützt verschiedene Varianten blockierender und nicht-blockierender Kommunikation, Wächter für die **receive**-Funktion, sowie Datenkonversion bei Kommunikation zwischen Rechnern mit unterschiedlichen Datenformaten.

- Die grundlegenden blockierenden Kommunikationsfunktionen lauten:

```
int MPI_Send(void *message, int count, MPI_Datatype dt,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *message, int count, MPI_Datatype dt,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status);
```

- Eine Nachricht in MPI besteht aus den eigentlichen *Daten* und einer Hülle (*engl.* envelope).
- Die Daten sind immer ein Array von elementaren Datentypen. Dies erlaubt MPI eine Datenkonversion vorzunehmen.



MPI: Blockierende Kommunikation II

- Die Hülle besteht aus:
 - 1 Nummer des Senders,
 - 2 Nummer des Empfängers,
 - 3 Tag,
 - 4 und einem Communicator.
- Nummer des Senders und Empfängers wird als Rank bezeichnet.
- Tag ist auch eine Integer-Zahl und dient der Kennzeichnung verschiedener Nachrichten zwischen identischen Kommunikationspartnern.
- Ein Communicator ist gegeben durch eine Teilmenge der Prozesse und einen Kommunikationskontext. Nachrichten, die zu verschiedenen Kontexten gehören, beeinflussen einander nicht, bzw. Sender und Empfänger müssen den selben Communicator verwenden.
- Zunächst verwenden wir nur den Default Communicator `MPI_COMM_WORLD` (alle gestarteten Prozesse).



MPI: Blockierende Kommunikation III

- `MPI_Send` ist grundsätzlich blockierend, es gibt jedoch diverse Varianten:
 - ▶ *buffered send* (B): Falls der Empfänger noch keine korrespondierende **recv**-Funktion ausgeführt hat, wird die Nachricht auf Senderseite gepuffert. Ein „buffered send“ wird, genügend Pufferplatz vorausgesetzt, immer sofort beendet. Im Unterschied zur asynchronen Kommunikation kann der Sendepuffer `message` sofort wiederverwendet werden.
 - ▶ *synchronous send* (S): Ende des `synchronous send` zeigt an, dass der Empfänger eine **recv**-Funktion ausführt und begonnen hat, die Daten zu lesen.
 - ▶ *ready send* (R): Ein `ready send` darf nur ausgeführt werden, falls der Empfänger bereits das korrespondierende **recv** ausgeführt hat. Ansonsten führt der Aufruf zum Fehler.
- Die entsprechenden Aufrufe lauten `MPI_Bsend`, `MPI_Ssend` und `MPI_Rsend`.
- Der `MPI_Send`-Befehl hat entweder die Semantik von `MPI_Bsend` oder `MPI_Ssend`, je nach Implementierung. `MPI_Send` kann also, muss aber nicht blockieren. In jedem Fall kann der Sendepuffer `message` sofort nach Beenden wieder verwendet werden.



MPI: Blockierende Kommunikation IV

- Der Befehl `MPI_Recv` ist in jedem Fall blockierend.
- Das Argument `status` enthält Quelle, Tag, und Fehlerstatus der empfangenen Nachricht.
- Für die Argumente `src` und `tag` können die Werte `MPI_ANY_SOURCE` bzw. `MPI_ANY_TAG` eingesetzt werden. Somit beinhaltet `MPI_Recv` die Funktionalität von **`recv_any`**.
- Eine nicht-blockierende Wächterfunktion für das Empfangen von Nachrichten steht mittels

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

zur Verfügung.



MPI: Nichtblockierende und globale Kommunikation I

- Für nichtblockierende Kommunikation stehen die Funktionen

```
int MPI_Isend(void *buf, int count, MPI_Datatype dt,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *req);
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype dt,
              int src, int tag, MPI_Comm comm,
              MPI_Request *req);
```

zur Verfügung.

- Mittels der `MPI_Request`-Objekte ist es möglich, den Zustand des Kommunikationsauftrages zu ermitteln (entspricht unserer **msgid**).
- Dazu gibt es (unter anderem) die Funktion

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status
```

- Das `flag` wird auf `true` ($\neq 0$) gesetzt, falls die durch `req` bezeichnete Kommunikationsoperation abgeschlossen ist. In diesem Fall enthält `status` Angaben über Sender, Empfänger und Fehlerstatus.

Zu beachten ist dabei, dass das `MPI_Request`-Objekt ungültig wird, sobald `MPI_Test` mit `flag==true` zurückkehrt. Es darf dann nicht mehr verwendet werden.



MPI: Nichtblockierende und globale Kommunikation II

- Für die globale Kommunikation stehen zur Verfügung (u. a.):

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

blockiert alle Prozesse eines Communicators bis alle da sind.

- ```
int MPI_Bcast(void *buf, int count, MPI_Datatype dt,
 int root, MPI_Comm comm);
```

verteilt die Nachricht in Prozess `root` an alle anderen Prozesse des Communicator.

- Für das Einsammeln von Daten stehen verschiedene Operationen zur Verfügung. Wir beschreiben nur eine davon:

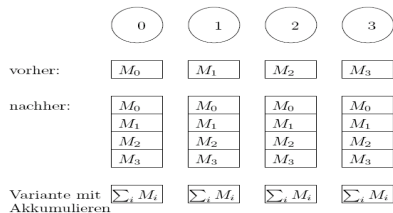
```
int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype
 MPI_Op op, int root, MPI_Comm comm);
```

kombiniert die Daten im Eingangspuffer `sbuf` aller Prozesse mittels der assoziativen Operation `op`. Das Endergebnis steht im Empfangspuffer `rbuf` des Prozesses `root` zur Verfügung. Beispiele für `op` sind `MPI_SUM`, `MPI_MAX`.

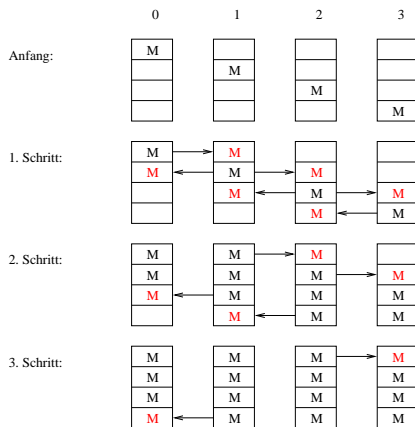


# Alle-an-alle: 1D Feld, Prinzip

Jeder will ein Datum an alle schicken (Variante: Akumulieren mit assoziativem Operator):



Ring lassen wir weg und gehen gleich zum 1D-Feld: Jeder schickt in beide Richtungen.



Wir verwenden synchrone Kommunikation. Entscheide Wer sendet/empfängt durch schwarz-weiß Färbung:



# Alle-an-alle: 1D Feld, Code

## Programm (Alle an alle auf 1D-Feld)

**parallel** *all-to-all-1D-feld*

```
{
 const int P;
 process Π [int p \in {0, ..., P - 1}]
 {
 void all_to_all_broadcast(msg m[P])
 {
 int i,
 from_left= p - 1, from_right= p + 1,
 to_left= p, to_right= p;
 for (i = 1; i < P; i++)
 {
 if ((p%2) == 1)
 {
 if (from_left \geq 0) recv(Π_{p-1} , m[from_left]);
 if (to_right \geq 0) send(Π_{p+1} , m[to_right]);
 if (from_right < P) recv(Π_{p+1} , m[from_right]);
 if (to_left < P) send(Π_{p-1} , m[to_left]);
 }
 else
 {
 if (to_right \geq 0) send(Π_{p+1} , m[to_right]);
 if (from_left \geq 0) recv(Π_{p-1} , m[from_left]);
 if (to_left < P) send(Π_{p-1} , m[to_left]);
 if (from_right < P) recv(Π_{p+1} , m[from_right]);
 }
 }
 ...
 }
 }
}
```

# Alle-an-alle: 1D Feld, Code

## Programm (Alle an alle auf 1D-Feld cont.)

**parallel** *all-to-all-1D-feld cont.*

```
{

 ...

 from_left--; to_right--;
 from_right++; to_left++;
}
}
...
m[p] = „Das ist von p!“;
all_to_all_broadcast(m);
...
}
```



# Alle-an-alle: 1D Feld, Laufzeit

- Für die Laufzeitanalyse betrachte  $P$  ungerade,  $P = 2k + 1$ :

$$\underbrace{\Pi_0, \dots, \Pi_{k-1}}_k, \Pi_k, \underbrace{\Pi_{k+1}, \dots, \Pi_{2k}}_k$$

|                 |          |          |             |
|-----------------|----------|----------|-------------|
| Prozess $\Pi_k$ | empfängt | $k$      | von links   |
|                 | schickt  | $k + 1$  | nach rechts |
|                 | empfängt | $k$      | von rechts  |
|                 | schickt  | $k + 1$  | nach links. |
| $\Sigma =$      |          | $4k + 2$ |             |
|                 |          | $= 2P$   |             |

- Danach hat  $\Pi_k$  alle Nachrichten. Nun muss die Nachricht von 0 noch zu  $2k$  und umgedreht. Dies dauert nochmal

$$\left( \underbrace{k}_{\text{Entfernung}} - 1 \right) \cdot \underbrace{2}_{\text{senden u. empfangen}} + \underbrace{1}_{\text{der Letzte empfängt nur}} = 2k - 1 = P - 2$$

also haben wir

$$T_{\text{all-to-all-array-1d}} = (t_s + t_h + t_w \cdot n)(3P - 2)$$

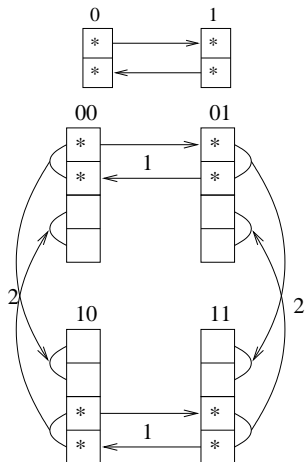


# Alle-an-alle: Hypercube

Der folgende Algorithmus für den Hypercube ist als *Dimensionsaustausch* bekannt und wird wieder rekursiv hergeleitet.

Beginne mit  $d = 1$ :

Bei vier Prozessen tauschen erst 00 und 01 bzw 10 und 11 ihre Daten aus, dann tauschen 00 und 10 bzw 01 und 11 jeweils zwei Informationen aus



# Alle-an-alle: Hypercube

```
void all_to_all_broadcast(msg m[P]) {
 int i, mask = 2d - 1, q;
 for (i = 0; i < d; i++) {
 q = p ⊕ 2i;
 if (p < q) { // wer zuerst?
 send(Πq, m[p&mask], ..., m[p&mask + 2i - 1]);
 rcv(Πq, m[q&mask], ..., m[q&mask + 2i - 1]);
 }
 else {
 rcv(Πq, m[q&mask], ..., m[q&mask + 2i - 1]);
 send(Πq, m[p&mask], ..., m[p&mask + 2i - 1]);
 }
 mask = mask ⊕ 2i;
 }
}
```

## • Laufzeitanalyse:

$$\begin{aligned} T_{all-to-all-bc-hc} &= \underbrace{2}_{\substack{\text{send u.} \\ \text{receive}}} \sum_{i=0}^{\text{ld } P-1} t_s + t_h + t_w \cdot n \cdot 2^i = \\ &= 2 \text{ld } P (t_s + t_h) + 2 t_w n (P - 1). \end{aligned}$$

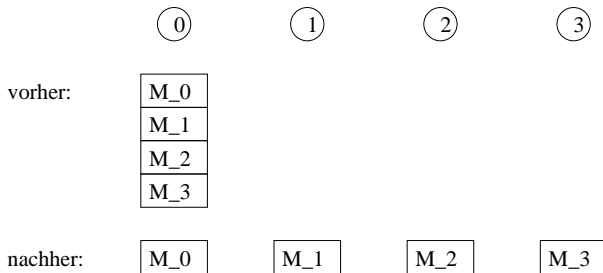
- Für große Nachrichten hat der HC keinen Vorteil: Jeder muss  $n$  Worte von jedem empfangen, egal wie die Topologie aussieht.



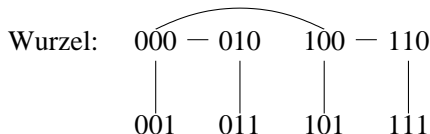


# Einer-an-alle indiv. Nachrichten: Hypercube, Prinzip

- Prozess 0 schickt an jeden eine Nachricht, aber an jeden eine andere!



- Beispiel ist die Ein/Ausgabe in *eine* Datei.
- Der Abwechslung halber betrachten wir hier mal die Ausgabe, d.h. alle-an-einen mit persönlichen Nachrichten.
- Wir nutzen die altbekannte Hypercubestruktur:



# Einer-an-alle mit indiv. Nachrichten: Hypercube, Code

Programm (*Einsammeln* persönlicher Nachrichten auf dem Hypercube)

**parallel** *all-to-one-personalized*

{

**const int**  $d, P = 2^d;$

**process**  $\Pi[\text{int } p \in \{0, \dots, P - 1\}]$ {

**void** *all\_to\_one\_pers(msg m)* {

**int** *mask, i, q, root;*

*// bestimme p's Wurzel: Wieviele Bits am Ende sind Null?*

*mask = 2<sup>d</sup> - 1;*

**for** ( $i = 0; i < d; i++$ )

{

*mask = mask  $\oplus$  2<sup>i</sup>;*

**if** ( $p \& \text{mask} \neq p$ ) **break;**

*// p = p<sub>d-1</sub> ... p<sub>i+1</sub> 1 0 ... 0*  
*zuletzt 0 gesetzt in i-1, ..., 0*  
*mask*

**if** ( $i < d$ ) *root = p  $\oplus$  2<sup>i</sup>;*

*// meine Wurzelrichtung*

*// eigene Daten*

**if** ( $p == 0$ ) *selber-verarbeiten(m);*

**else** *send(root, m);*

*// hochgeben*

...

}

# Einer-an-alle mit indiv. Nachrichten: Hypercube, Code

Programm (*Einsammeln* persönlicher Nachrichten auf dem Hypercube cont.)

**parallel** *all-to-one-personalized* cont.

```
{

 ...

 // arbeite Unterbäume ab:
 mask = 2d - 1;
 for (i = 0; i < d; i++) {
 mask = mask ⊕ 2i; q = p ⊕ 2i;
 if (p & mask == p)

 for (k = 0; k < 2i; k++) {
 recv(Πq, m);
 if (p == 0) verarbeite(m);
 else send(Πroot, m);
 }
 }
 }
}
```

$$\begin{aligned} // \quad p &= p_{d-1} \dots p_{i+1} \quad 0 \quad \underbrace{0 \dots 0}_{i-1, \dots, 0} \\ // \quad q &= p_{d-1} \dots p_{i+1} \quad 1 \quad \underbrace{0 \dots 0}_{i-1, \dots, 0} \end{aligned}$$

// ⇒ ich bin Wurzel eines HC der Dim.  $i + 1$ !

# Einer-an-alle m. indiv. Nachrichten: Laufzeit, Varianten

Für die *Laufzeit* hat man für grosse ( $n$ ) Nachrichten

$$T_{all-to-one-pers} \geq t_w n(P - 1)$$

wegen dem Pipelining.

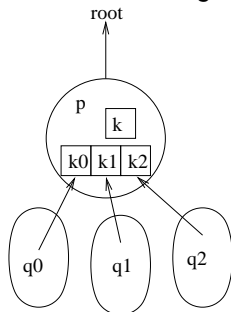
Einige Varianten sind denkbar:

- *Individuelle Länge der Nachricht*: Hier sendet man vor verschicken der eigentlichen Nachricht nur die Längeninformation (in der Praxis ist das notwendig  $\rightarrow$  MPI).
- *beliebig lange Nachricht* (aber nur endlicher Zwischenpuffer!): zerlege Nachrichten in Pakete fester Länge.
- *sortierte Ausgabe*: Es sei jeder Nachricht  $M_i$  (von Prozess  $i$ ) ein Sortierschlüssel  $k_i$  zugeordnet. Die Nachrichten sollen von Prozess 0 in aufsteigender Reihenfolge der Schlüssel verarbeitet werden, *ohne* dass alle Nachrichten zwischengepuffert werden.



# Einer-an-alle m. indiv. Nachrichten: Laufzeit, Varianten

- Bei *sortierter Ausgabe* folgt man der folgenden Idee:



$p$  habe drei „Untergebene“,  $q_0, q_1, q_2$ , die für ganze Unterbäume stehen.

Jeder  $q_i$  sendet seinen nächst kleinsten Schlüssel an  $p$ , der den kleinsten Schlüssel rausucht und ihn seinerseits, mit samt der inzwischen übertragenen Daten, weitergibt.