6. Inter-Prozess-Kommunikation (4)

6. IPC
6.5 Message Passing Interface

Diese Folien basieren auf zwei Foliensätzen:

- William Gropp, Ewing Lusk: „An Introduction to MPI“, http://www-unix.mcs.anl.gov/mpi/tutorial/mpiintro/MPIIntro.PPT
- F. Chris MacPhee, Wesley Huang: „Parallel Programming Techniques“, http://acrl.cs.unb.ca/php/training/mpi/macphee-Intro_to_MPI_Parallel_Programming_Techniques.ppt

/home/esser/Daten/Dozent/Folien/bs-esser-20.odp

Was ist MPI?

- Spezifikation für Message-Passing-Bibliotheken
  - komplexes Message-Passing-Protokoll
  - keine Sprach- oder Compiler-Spezifikation
  - keine spezielle Implementation / Produkt
- für Parallelrechner, Cluster, heterogene Netzwerke
- Nutzung fortschrittlicher Parallel-Hardware für
  - Endanwender
  - Bibliotheksprogrammierer
  - Tool-Entwickler

6.5 Message Passing Interface (MPI)

Message Passing Interface

- 1994: MPI 1.0 Standard
- 1995: MPI 1.1
- 1997: MPI 2.0
  - enthält MPI 1.1 und zusätzliche Features:
    - MPI-IO (parallele I/O-Operationen)
    - einseitige Kommunikation
    - dynamische Prozesse

MPI-Features

- Verschiedene Arten von Point-to-Point Message Passing
  - blockierend (*e.g.* MPI_SEND)
  - nicht-blockierend (*e.g.* MPI_ISEND)
  - synchron (*e.g.* MPI_SSEND)
  - gepuffert (*e.g.* MPI_BSEND)
- „kollektive“ Kommunikation (alle Prozesse führen einen MPI-Befehl gemeinsam aus), z. B. MPI_REDUCE
- Synchronisation, z. B. MPI_BARRIER
- benutzerdefinierte Datentypen
- unterstützt Topologien (z. B. 3x4-Grid), dadurch entstehen „Nachbarschafts“-Konzepte“

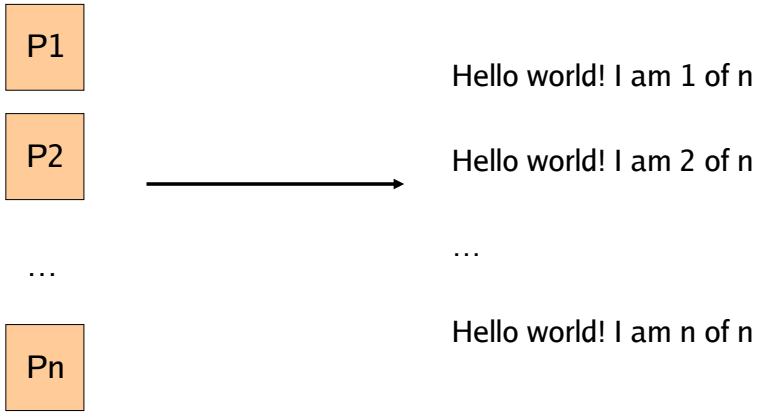
Minimales MPI-Programm

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

int main( int argc, char *argv[] ) {

    /* MPI-System initialisieren */
    MPI_Init( &argc, &argv );

    printf( "Hello, world!\n" );

    /* MPI-System beenden */
    MPI_Finalize();

    return 0;
}
```


Wie viele Prozesse / Identität

- Zwei wichtige Fragen in Parallelprogrammen:
  - Wie viele Prozesse nehmen an der Berechnung teil?
  - Welcher davon bin ich?
- MPI bietet Funktionen, die beide Fragen beantworten:
  - **`MPI_Comm_size`** gibt die Anzahl der beteiligten Prozesse zurück
  - **`MPI_Comm_rank`** gibt den *rank* zurück, eine Zahl zwischen 0 and *size*-1, die den aufrufenden Prozess identifiziert

Einfaches MPI-Beispiel (1)

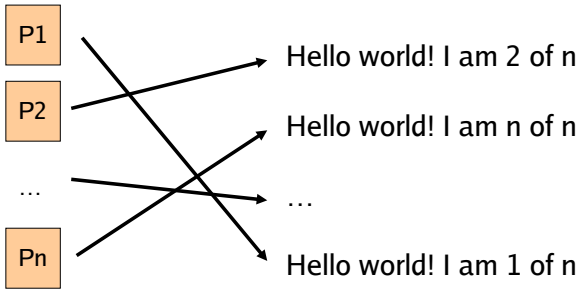


Einfaches MPI-Beispiel (2)

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

int main(int argc, char * argv[]) {
   int taskid, ntasks;

   /* MPI-System initialisieren */
   MPI_Init(&argc, &argv);

   /* Wer bin ich und wie viele sind wir? */
   MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

   printf("Hello world!  I am %d of %d\n", taskid, ntasks);

   /* MPI-Nutzung beenden */
   MPI_Finalize();
   return(0);
}
```

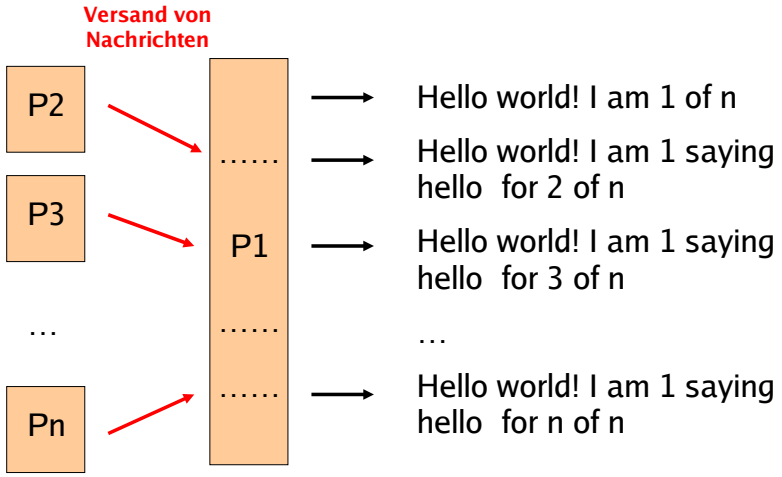

Einfaches MPI-Beispiel (3)

- Realistischeres Ergebnis:



- Prozessor-IDs legen keine Reihenfolge fest – das ist Aufgabe des Programmierers

Koordiniertes MPI-Beispiel (1)



Koordiniertes MPI-Beispiel (2)

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char * argv[]) {
   int taskid, ntasks, slaveid, srank, tag, err;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
   if (taskid == 0) {    /* Master */
      printf("Hello world!  I am %d of %d\n", taskid, ntasks);
      for (slaveid = 1; slaveid < ntasks; slaveid++) {
         err = MPI_Recv(srank, 1, MPI_INTEGER, slaveid, tag,
                        MPI_COMM_WORLD, &status);
         printf("Hello World! I am %d saying hello for %d of %d\n",
                taskid, srank, ntasks);
      }
   }
   else {                /* Slave */
      err = MPI_Send(srank, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD);
   }
}
```

Send-Funktion MPI_Send

```
int MPI_Send ( void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm )
```

Eingabe-Parameter:

buf - send buffer (choice)
count - number of elements in send buffer (nonnegative integer)
datatype - datatype of each send buffer element (handle)
dest - rank (id) of destination (integer)
tag - message tag (integer)
comm - communicator (handle)

Receive-Funktion MPI_Recv

```
int MPI_Recv ( void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Status *status )
```

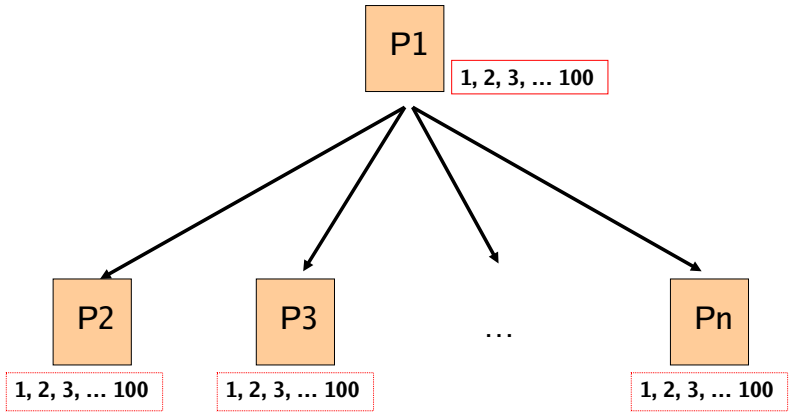
Ausgabe-Parameter:

buf - address of receive buffer (choice)
status - status object (Status)

Eingabe-Parameter:

count - maximum number of elements in receive buffer (integer)
datatype - datatype of each receive buffer element (handle)
source - rank (id) of source (integer)
tag - message tag (integer)
comm - communicator (handle)

Broadcasting



Broadcast-Funktion MPI_Bcast

```
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype,
                int root, MPI_Comm comm )
```

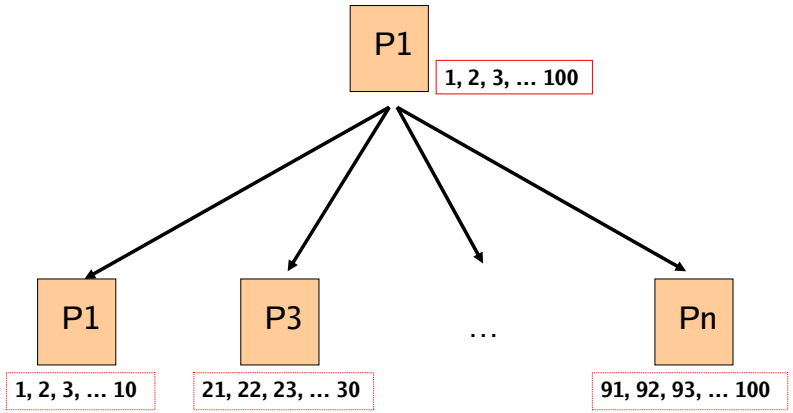
Parameter

buffer - address of buffer (choice)
count - number of entries in buffer (integer)
datatype - data type of buffer (handle)
root - rank of broadcast root (integer)
comm - communicator (handle)

Broadcast-Beispiel

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])    {
   MPI_Comm comm;
   int myarray[100];
   int root = 0;
   /* … */
   /* Broadcast 100 integers to all from myarray to all processes */
   MPI_Bcast(myarray, 100, MPI_INTEGER, root, comm);
   /* … */
}
```

Datendistribution



Distributionsfunktion MPI_Scatter

```
int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
```

Eingabe-Parameter

- **sendbuf** - send buffer (choice, significant only at root)
- **sendcount** - number of elements sent to each process (integer, significant only at root)
- **sendtype** - data type of send buffer elements (handle, significant only at root)
- **recvcount** - number of elements in receive buffer (integer)
- **recvtype** - data type of receive buffer elements (handle)
- **root** - rank of sending process (integer)
- **comm** - communicator (handle)

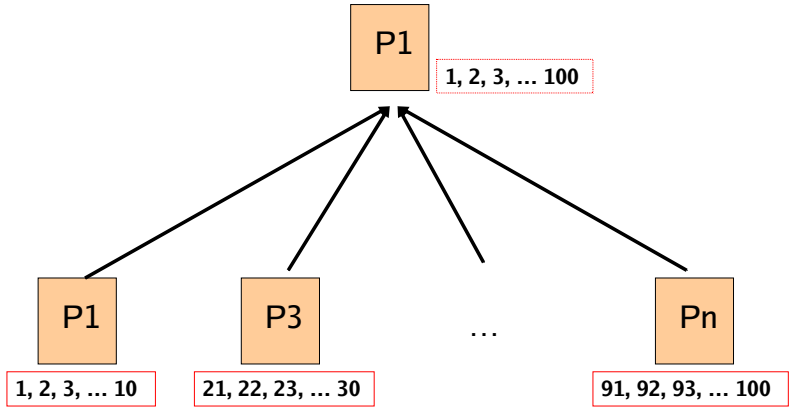
Ausgabe-Parameter

- **recvbuf** - receive buffer (choice)

Datendistribution, Beispiel

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])    {
   MPI_Comm comm;
   int sendarray[100], *recvarray;
   int root = 0;
   /* … */
   /* Scatter 100 integers evenly across all processes */
   MPI_Scatter(sendarray, 100, MPI_INTEGER, recvarray, root, comm);
   /* … */
}
```


Daten sammeln (1)



Daten sammeln (2)

```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm )
```

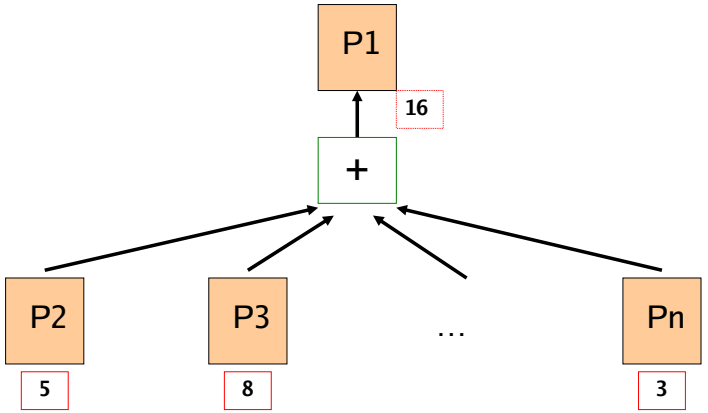
Eingabeparameter

- **sendbuf** - send buffer (choice)
- **sendcount** - number of elements in send buffer (integer)
- **sendtype** - data type of send buffer elements (handle)
- **recvcount** - number of elements for any single receive (integer, significant only at root)
- **recvtype** - data type of recv buffer elements (handle, significant only at root)
- **root** - rank of receiving process (integer)
- **comm** - communicator (handle)

Ausgabeparameter

- **recvbuf** - receive buffer (choice, significant only at root)

Daten reduzieren (1)



Daten reduzieren (2)

```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op, int root,
                 MPI_Comm comm )
```

Eingabe-Parameter

- **sendbuf** - send buffer (choice)
- **count** - number of elements in send buffer (integer)
- **datatype** - data type of elements of send buffer (handle)
- **op** - reduce operation (handle)
- **root** - rank of root process (integer)
- **comm** - communicator (handle)

Ausgabe-Parameter

- **recvbuf** - receive buffer (choice, significant only at root)

Übersicht

Broadcast:
`MPI_BCast ()`

Scatter:
`MPI_Scatter ()`

Gather:
`MPI_Gather ()`

All gather:
`MPI_Allgather ()`

All to all:
`MPI_Alltoall`

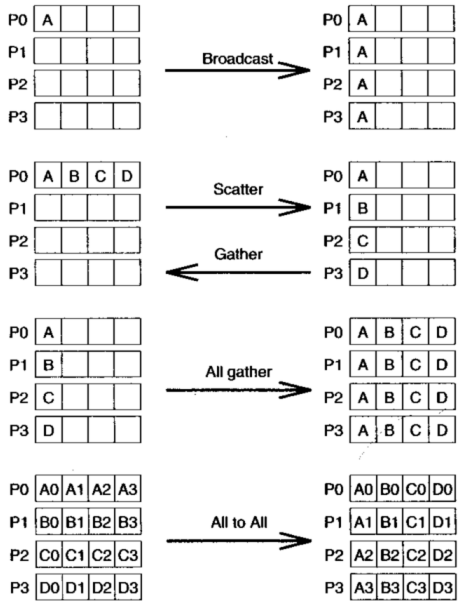


Bild: Gropp/Lusk/Skjellum: „UsingMPI", 1994

Fehlerbehandlung

- Standardmäßig führt ein Fehler zum Abbruch aller Prozesse
- Alternativ geben die Routinen einen Fehlercode zurück
  - In C++: Exceptions (MPI-2)
- Programmierer kann auch eigene Error-Handler schreiben und einsetzen
- Für Bibliotheken ist evtl. eine andere Fehlerbehandlung als für Anwendungen sinnvoll

Ein paar Grundkonzepte

- Prozesse kann man in **Gruppen** einteilen
- Jede Nachricht wird in einem **Kontext** verschickt und muss im gleichen Kontext empfangen werden
- Gruppe+ Kontext = **Kommunikator**.
- Prozess-Identifikation über seinen **Rang** in Gruppe / Kommunikator.
- Es gibt einen Standardkommunikator, dessen Gruppe alle Ausgangsprozesse enthält: **`MPI_COMM_WORLD`**.

MPI-Datentypen (1)

- Daten in einer Nachricht werden durch ein Tripel (Adresse, Zähler, Datentyp) definiert:
- MPI-**Datentypen** werden rekursiv definiert durch:
  - vordefinierte, die einem Datentyp der Programmiersprache entsprechen, z. B. MPI_INT, MPI_DOUBLE_PRECISION
  - zusammenhängedes Array von MPI-Datentypen
  - strided block von MPI-Datentypen
  - indexed array von Blöcken von MPI-Datentypen
  - beliebige Struktur aus MPI-Datentypen
- Es gibt MPI-Funktionen, mit denen man eigene Datentypen zusammensetzen kann, z. B. ein Array von (int, float)-Paaren

MPI-Datentypen (2)

MPI-Datentyp	C-Datentyp
`MPI_BYTE`	
`MPI_CHAR`	`signed char`
`MPI_DOUBLE`	`double`
`MPI_FLOAT`	`float`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_LONG_DOUBLE`	`long double`
`MPI_PACKED`	
`MPI_SHORT`	`short`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_UNSIGNED_SHORT`	`unsigned short`

MPI-Tags

- Nachrichten werden mit einem begleitenden, benutzerdefinierten **Tag** verschickt, das dem Empfänger helfen soll, die Nachricht zu identifizieren
- Der Empfänger kann durch Angabe eines Tags nur solche Nachrichten entgegennehmen, die das passende Tag haben (alternativ: **`MPI_ANY_TAG`**)
- Einige andere Message-Passing Systeme nennen Tags “message types”. MPI nennt sie Tags, damit man sie nicht mit Datentypen verwechselt

weitere Informationen abrufen

`status` ist eine Datenstruktur, die im Programm als Variable deklariert wird

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;

MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )

recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```


Synchronisation (1)

- In vielen Algorithmen ist es nötig, alle beteiligten Prozesse in einen bestimmten Zustand zu überführen, bevor die parallele Berechnung fortgesetzt wird
- Das löst man i.d.R. mit einem Synchronisationspunkt, der sicherstellt, dass alle Prozesse (oder eine festgelegte Teilmenge) einen bestimmten Programmpunkt erreicht haben, bevor es weiter geht
- In einigen Fällen erzielt schon das Blockieren von Message-Funktionen (implizit) die gewünschte Wirkung, aber manchmal muss dies explizit festgelegt werden
- MPI: **MPI_Barrier ()**
  (vgl. Barrieren im Kapitel über Synchronisation)

Synchronisation (2)

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

main(int argc, char** argv) {
  int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

  if (my_PE_num==0) {   /* Master */
    for (index=1; index<4; index++) {
      MPI_Send( &numbertosend, 1, MPI_INT, index, 10,MPI_COMM_WORLD);
    }
  }
  else {                /* Slave */
    MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
    result = numbertoreceive * my_PE_num;
  }
```

Synchronisation (3)

```
  for (index=1; index<4; index++) {
    /* Barriers sorgen für Ausgabe in der richtigen Reihenfolge */
    MPI_Barrier(MPI_COMM_WORLD);
    if (index==my_PE_num)
      printf("PE %d's result is %d.\n", my_PE_num, result);
  }

  if (my_PE_num==0) {
    for (index=1; index<4; index++) {
      MPI_Recv( &numbertoreceive, 1, MPI_INT, index, 10, MPI_COMM_WORLD,
                &status );
      result += numbertoreceive;
    }
    printf("Total is %d.\n", result);
  }
  else {
    MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD );
  }
  MPI_Finalize();
}
```


Beispiel-Parallelisierung (1)

Berechnung von π über Integration:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \approx \frac{1}{N}\sum_{k=1}^{N} \frac{4}{1+[(k-1/2)/N]^2}$$

$$\pi = \int_0^1 f(x)\,dx \approx \frac{1}{N}\sum_{k=1}^{N} f\left(\frac{k-1/2}{N}\right), \quad f(x) = \frac{4}{1+x^2}$$

Nicht-paralleler Code

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define f(x)
((float)(4.0/(1.0+x*x)))
#define pi ((float)(4.0*atan(1.0)))
```

Code in diesen Beispielen basiert auf http://www.pdc.kth.se/training/Tutor/MPI/Templates/pi/

```
main() {
  float err, sum, w;
  int i, N;

  while (1) {
    printf ("Anzahl Approximationen (0 = Ende): ");
    scanf("%d",&N);
    if (N==0) exit(0);
    w = 1.0/(float)N;
    sum = 0.0;
    for (i = 1; i <= N; i++)
      sum = sum + f(((float)i-0.5)*w);
    sum = sum * w;
    err = sum - pi;
    printf("sum, err = %10.10f, %1.10f\n", sum, err);
  }
}
```


Beispiel-Parallelisierung (2)

Parallele MPI-Variante

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

#define f(x) ((float)(4.0/(1.0+x*x)))
#define pi ((float)(4.0*atan(1.0)))

MPI_Status status;

void solicit (int *pN, int *pnprocs, int mynum) {
  /* Instanz 0 fragt N ab und verteilt den Wert */
  int source = 0;
  if (mynum == 0) {
    printf ("Anzahl Approximationen (0 = Ende): ");
    scanf ("%d", pN);
  }
  MPI_Bcast(pN, 1, MPI_INT, source, MPI_COMM_WORLD);
}
```


Beispiel-Parallelisierung (3)

```
main(int argc, char **argv) {
  float err, sum, w, x;
  int   i, N, info, mynum, nprocs, source, dest = 0,
        type = 2, nbytes = 0, EUI_SUCCEED = 0;

  MPI_Init(&argc, &argv);                  /* Initialisierung              */
  MPI_Comm_rank(MPI_COMM_WORLD, &mynum);   /* Wer bin ich?                 */
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  solicit (&N, &nprocs, mynum);            /* 1. Schritt: N bestimmen      */

  if (N <= 0) {                            /* 2. Schritt: Programm beenden? */
    printf("node %d left\n", mynum); exit(0);
  }

  /* 3. Schritt: "nprocs" teilnehmende Instanzen. Jede erledigt
   * Anteil 1/nprocs der Berechnung. */
  while (N > 0) {
    w = 1.0/(float)N;
    sum = 0.0;
    for (i = mynum+1; i <= N; i+=nprocs)  /* Schrittweite nprocs statt 1  */
      sum = sum + f(((float)i-0.5)*w);
    sum = sum * w;
    err = sum - pi;
```

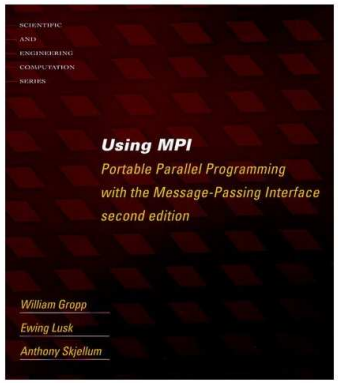
Beispiel-Parallelisierung (4)

```
    /* 4. Schritt: partielle Ergebnisse sammeln; Master gibt sie aus */
    if (mynum==0) {  /* Ich bin der Master */
      printf ("host calculated x = %7.5f\n", sum);
      for (i=1; i<nprocs; i++) {
        source = i;
        info = MPI_Recv(&x, 1, MPI_FLOAT, source, type, MPI_COMM_WORLD, &status);
        printf ("host got x = %7.5f\n", x);
        sum=sum+x;
      }
      err = sum - pi;
      printf ("sum, err = %7.5f, %10e\n", sum, err);
      fflush(stdout);
    }
    /* Andere Instanzen senden nur ihre Ergebnisse */
    else {            /* Ich bin ein Slave */
      info = MPI_Send(&sum, 1, MPI_FLOAT, dest, type, MPI_COMM_WORLD);
      printf ("inst %d sent partial sum %7.2f to inst 0\n", mynum, sum);
      fflush(stdout);
    }
    /* noch mal ausführen... */
    solicit (&N, &nprocs, mynum);
  }

  MPI_Finalize();
}
```


Literatur

Gutes Buch zu MPI:

William Gropp, Anthony Skjellum, Ewing Lusk: „Using MPI: Portable Parallel Programming with the Message Passing Interface“



Vorschau

nächste Vorlesung: 15.01.2007

Deadlocks

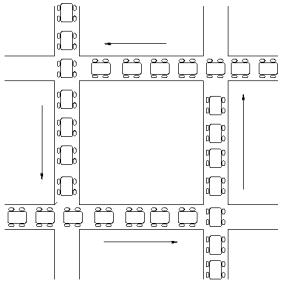


Bild: http://www.cs.rpi.edu/academics/courses/fall04/os/c10/gridlock.gif