Karlsruhe Institute of Technology

Paralleles Programmieren mit OpenMP und MPI

Einführung in MPI

Vorlesung “Parallelrechner und Parallelprogrammierung”, SoSe 2016

Hartmut Häfner, Steinbuch Centre for Computing (SCC)

STEINBUCH CENTRE FOR COMPUTING - SCC

Danksagungen

- Dieser Kurs basiert teilweise auf den Folien von Herrn Rabenseifner vom HLRS in Stuttgart. Vielen Dank an ihn dafür, dass er seine Folien uns zur Verfügung gestellt hat.

- Damit basiert der Kurs auch teilweise auf dem MPI-Kurs des “EPCC Training and Education Centre” vom “Edinburgh Parallel Computing Centre” an der Universität Edinburgh.

Informationen zu MPI

MPI-1: A Message-Passing Interface Standard (June,1995)
https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html

MPI-2: A Message-Passing Interface Standard (July, 1997)
https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

MPI-3: A Message-Passing Interface Standard (September, 2012)
https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Marc Snir und William Gropp und andere:
MPI: The Complete Reference. (2-volume set). The MIT Press, 1998.
(MPI-1.2 und MPI-2 Standard in lesbarer Form)

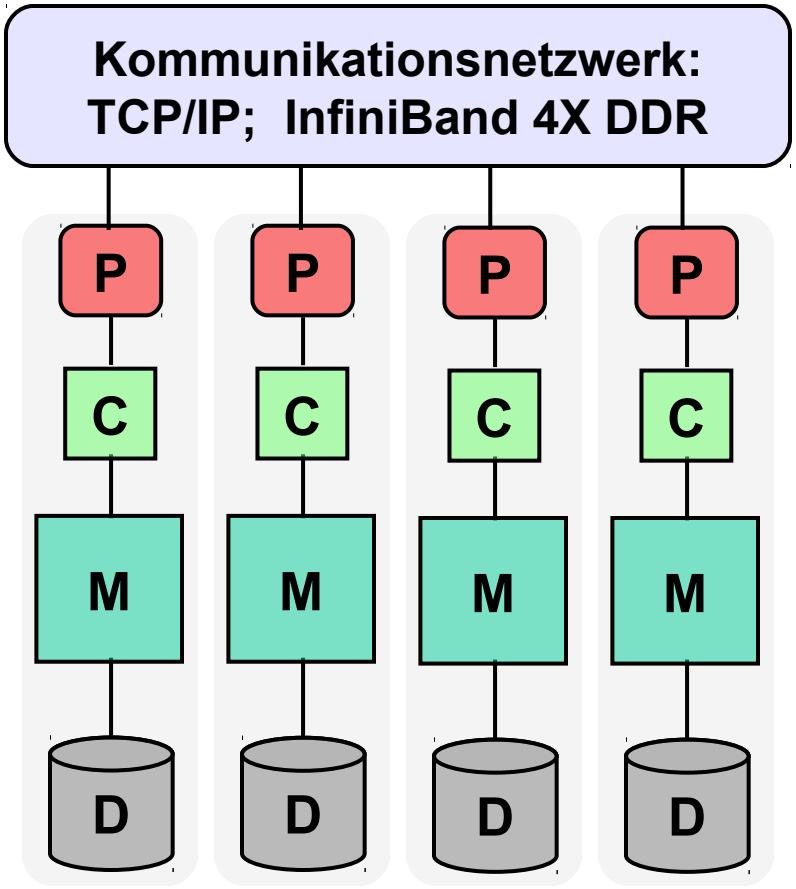
William Gropp, Ewing Lusk und Rajeev Thakur:
Using MPI, Third Edition: Portable Parallel Programming With the Message-Passing Interface, MIT Press, Nov. 2014, und
Using Advanced MPI: Advanced Features of the Message-Passing Interface.
MIT Press, Nov. 2014.

Peter S. Pacheco: Parallel Programming with MPI. Morgen Kaufmann Publishers, 1997
(Sehr gute Einführung, kann als begleitender Text für Vorlesungen zu MPI benutzt werden).

MPI-Tutorial vom Livermore Computing Center:
https://computing.llnl.gov/tutorials/mpi/

http://www.mpi-forum.org

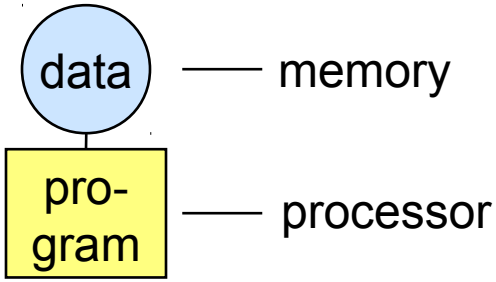
„Distributed Memory“ System



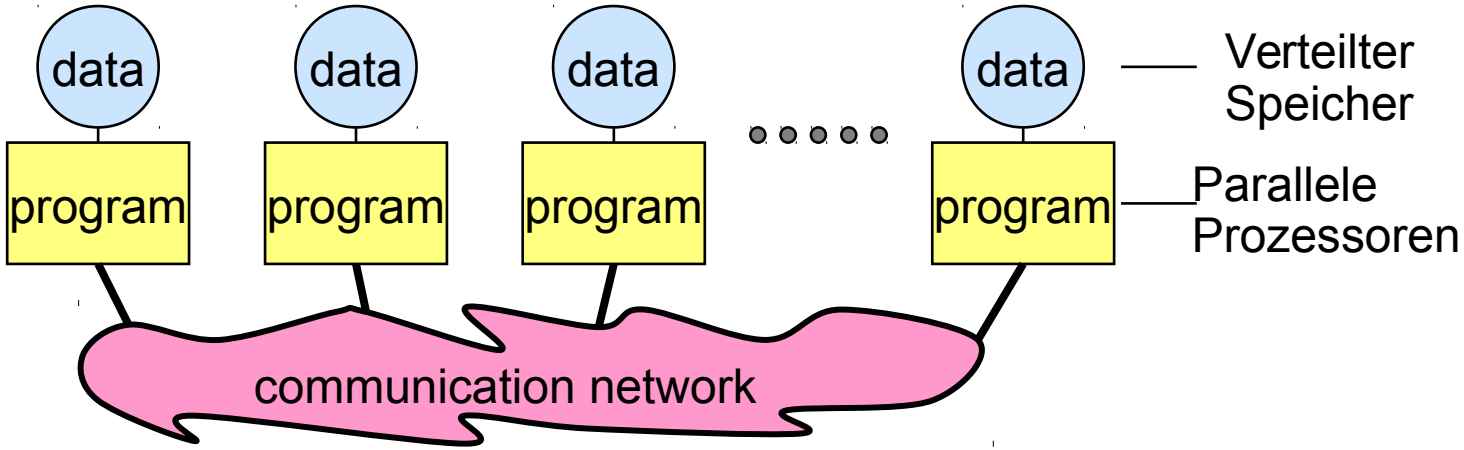
- Jeder Knoten agiert als eigenständiges Computersystem
- Eine Kopie des Betriebsystems pro Knoten
- Jeder Prozessor hat nur Zugriff auf seinen eigenen lokalen Speicher
- Parallelisierung über “Message Passing Interface“
- Beispiele: HPC-Systeme am KIT, vernetzte Workstations

“Message Passing” Paradigma

Sequentielles Paradigma

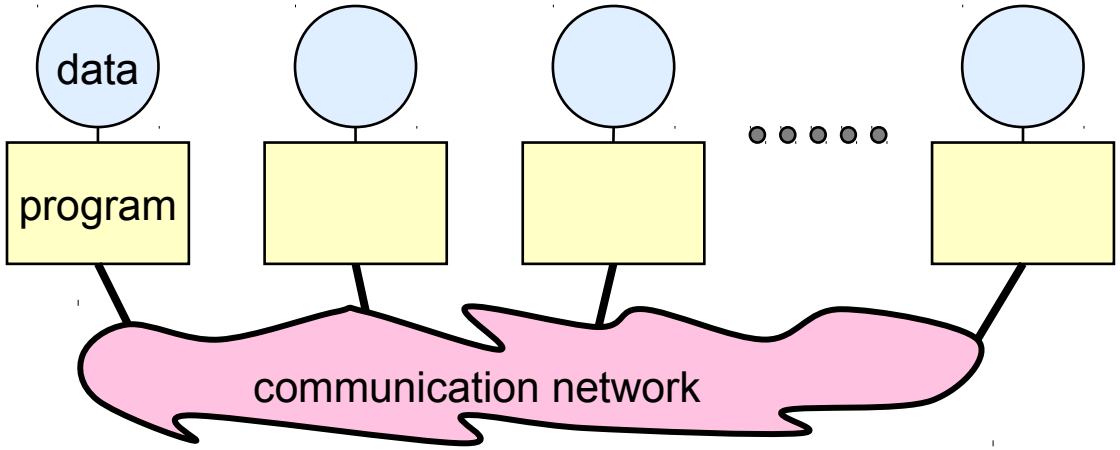


“Message-Passing” Paradigma



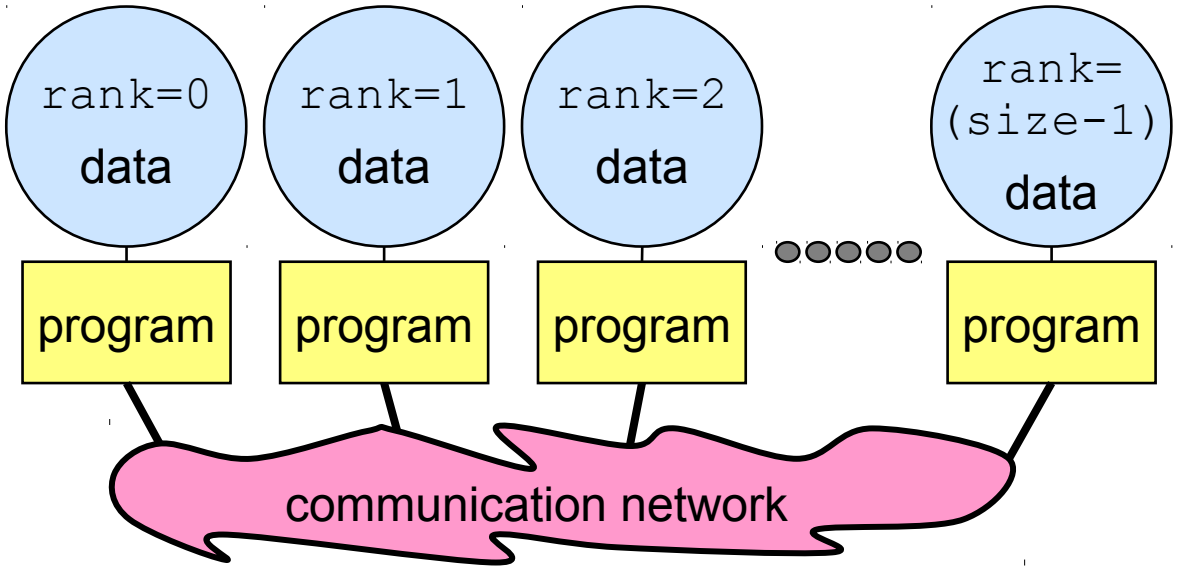
“Message Passing” Paradigma (2)

- Auf jedem Prozessor in einem “message passing” Programm läuft (genau) ein *Programm bzw. Prozess:*
  - geschrieben in einer konventionellen Programmiersprache wie z.B. C oder Fortran
  - typischerweise das gleiche “Executable” auf jedem Prozessor (SPMD)
  - Variable jedes Programms bzw. Prozesses haben
    - den gleichen Namen
    - aber unterschiedliche Lokationen (“distributed memory”) und unterschiedliche Daten
    - → alle Variablen sind privat
  - kommunizieren über Send & Receive Routinen (“M*essage Passing Interface”*)



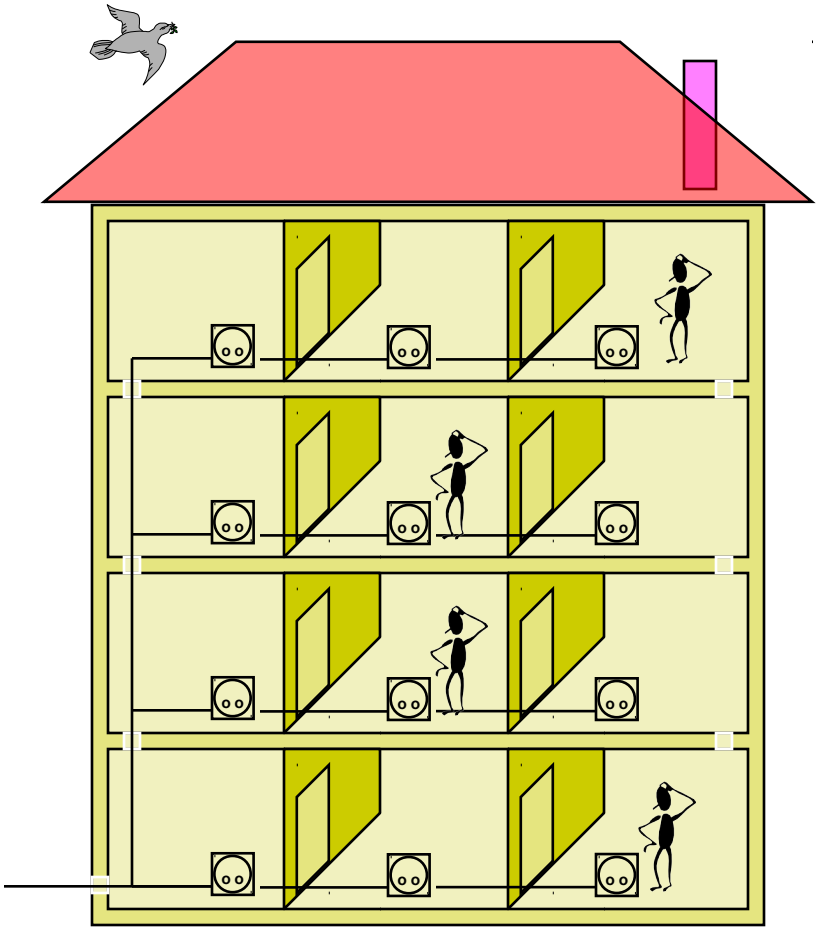
Daten- und Arbeitsverteilung

- Der Wert der Variablen `rank` wird von einer MPI-Bibliotheksroutine festgelegt
- Alle (`size`) Prozesse werden durch ein MPI Initialisierungs-programm (`mpirun` oder `mpiexec`) gestartet
- Alle Verteilungsentscheidungen basieren auf `rank`, d.h. welche Prozesse auf welchen Daten arbeiten



Analogie: Parallele Elektrische Installation

- **MPI Prozess**
  = Arbeit eines Elektrikers auf einem Stockwerk
- **Daten**
  = elektrische Installation
- **MPI Kommunikation**
  = reale Kommunikation, die garantiert, dass die Drähte an derselben Stelle durch die Stock-werke laufen



Was ist SPMD?

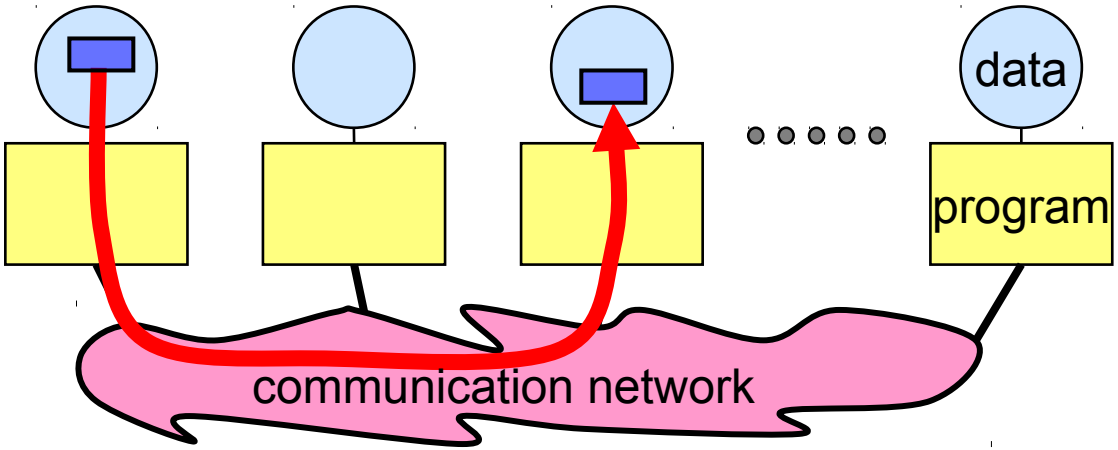
- Single Program, Multiple Data
- Dasselbe Programm läuft auf jedem Prozessor, allerdings auf unterschiedlichen Datensätzen
- MPI erlaubt auch MPMD, d.h. Multiple Program, …
- (Einige MPI-Implementierungen sind beschränkt auf SPMD)
- MPMD kann durch SPMD emuliert werden

Emulation von MPMD durch SPMD

```
main(int argc, char **argv)
{
  if (myrank < .... )
  {
          ocean( /* arguments */ );
  } else {
          weather( /* arguments */ );
  }
}
```

```
program simulated_MPMD
if (myrank < ... ) then
  call ocean( some arguments )
else
  call weather( some arguments )
endif
end program simulated_MPMD
```

“Messages“ - Botschaften



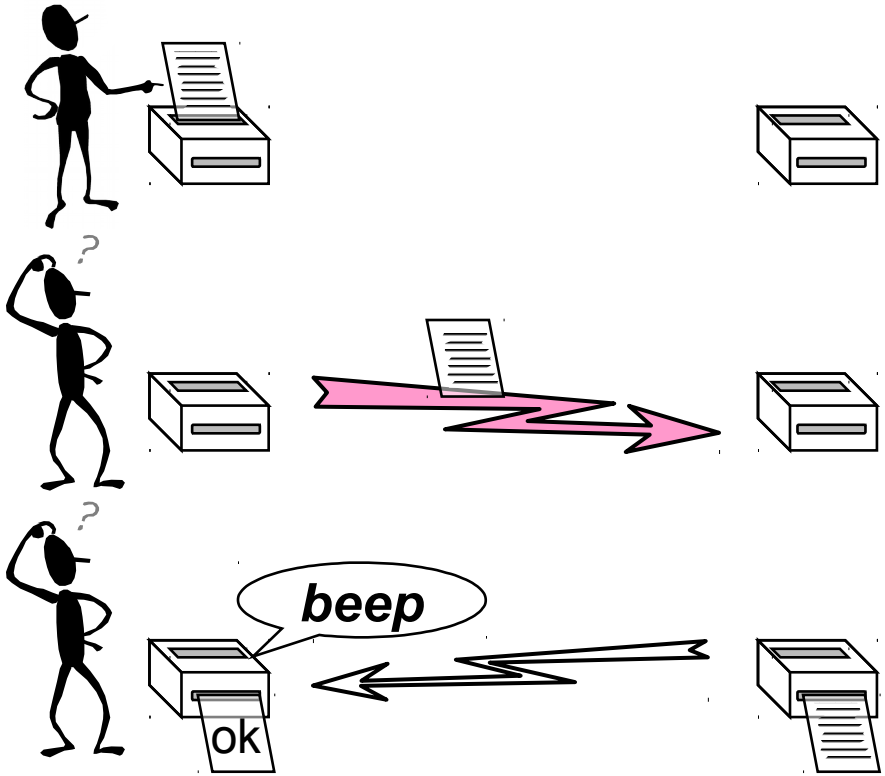
- Botschaften sind Datenpakete, die von einem Prozess zu einem anderen übermittelt werden
- Notwendige Informationen für die Botschaften sind:
  - Sendender Prozess – Empfangender Process d.h. deren “ranks“
  - Lokation der Quelle – Ziellokation
  - Datentyp der Quelle – Datentyp des Ziels
  - Datengröße der Quelle – Datengröße des Ziels

Punkt-zu-Punkt (P2P) Kommunikation

- Einfachste Form des “message passing”
- Ein Prozess sendet eine Botschaft an einen anderen Prozess
- Verschiedene Typen von P2P Kommunikationsarten
  - Synchrones Senden
  - Asynchrones (gepuffertes) Senden

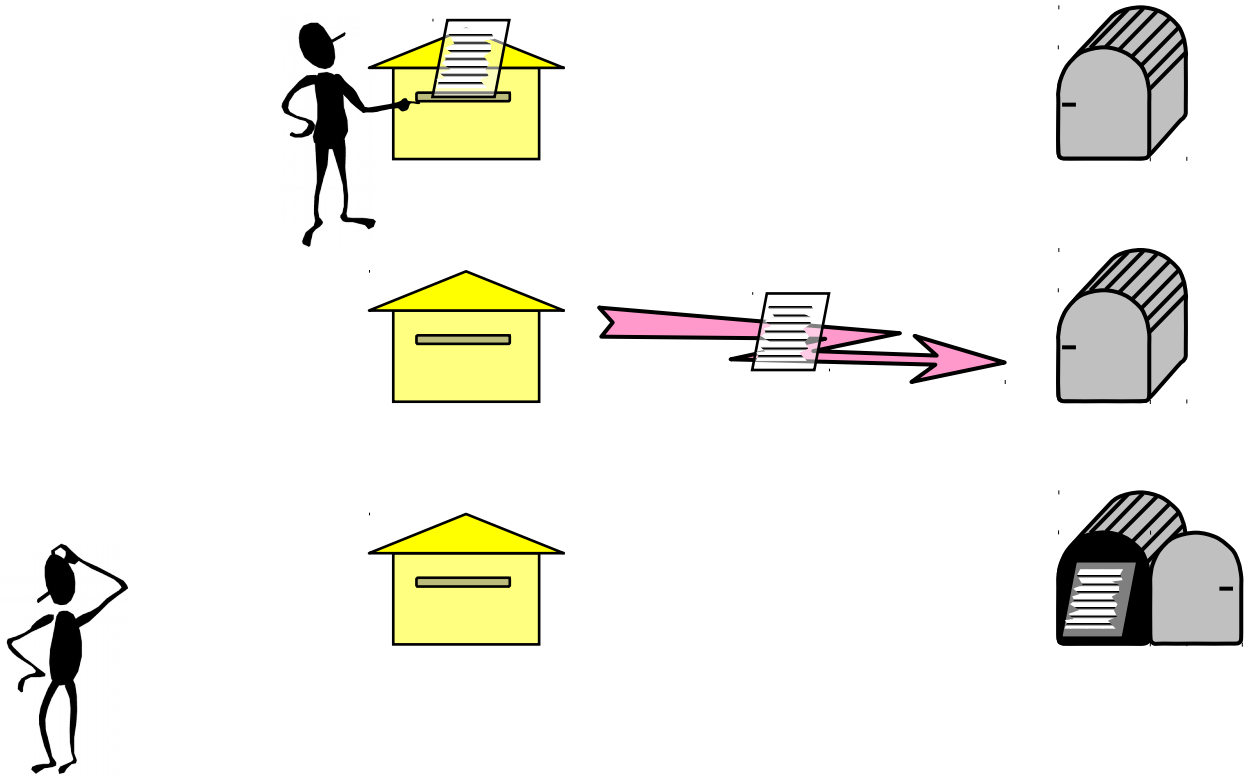
Synchrones Senden

- Sender erhält Information, dass die Nachricht empfangen wurde
- Analogie zur Bestätigung eines Faxgeräts.



Asynchrones (gepuffertes) Senden

- Man weiss nur, wann die Botschaft abgeschickt wurde



Blockierende Operationen

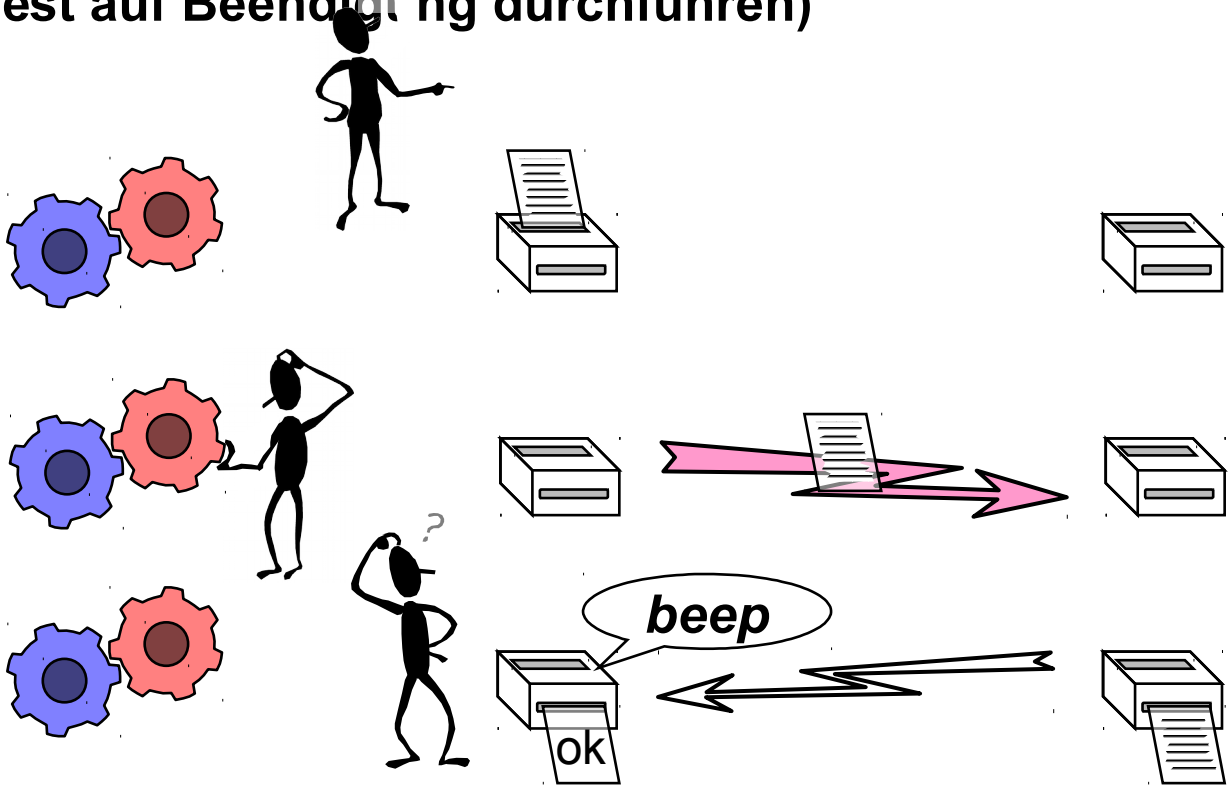
- Operationen sind lokale Aktivitäten wie Senden oder Empfangen einer Botschaft
- Blockierendes Sende- oder Empfangs-Unterprogramm wird erst verlassen, wenn die zugehörige Operation beendet ist.
- *Bei synchronem Senden*: Senderoutine wird erst verlassen, wenn die Botschaft beim Empfänger angekommen ist

  Bei a*synchrones Senden*: Senderoutine wird verlassen, sobald die Daten vollständig verschickt wurden

  Beim *Empfangen*: Empfangsroutine wird erst verlassen, wenn die Daten vollständig im Anwendungsspeicher stehen

Nicht-blockierende Operationen

- Nicht-blockierende Operation: stößt Kommunikation an, kehrt dann sofort zurück (verlässt Routine) und erlaubt somit diesem Prozess andere Arbeit durchzuführen
- Zu gegebener Zeit muss der Prozess auf die Beendigung der nicht-blockierenden Operation warten (oder eventuell nur einen Test auf Beendigung durchführen)



Nicht-blockierende Operationen (2)

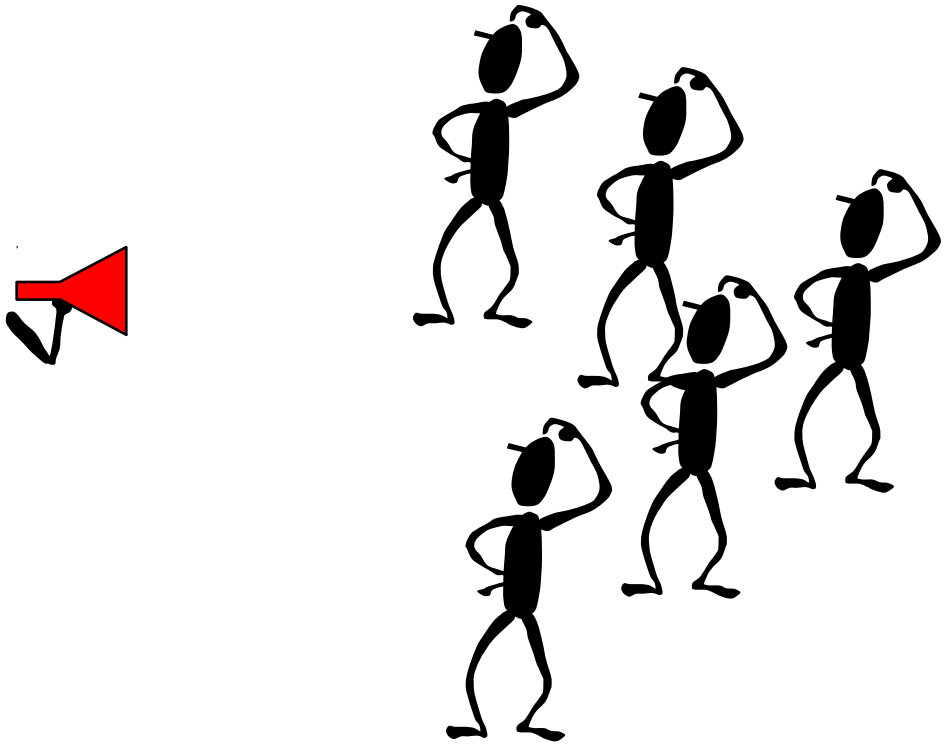
- Zu allen nicht-blockierenden Operationen sollte eine zugehörige WAIT- oder TEST-Operation ausgeführt werden (Einige MPI-Systemressourcen können nur freigegeben werden, wenn die nicht-blockierende Operation vervollständigt ist!?)
- Eine nicht-blockierende Operation, auf die sofort danach eine zugehörige WAIT-Routine folgt, ist äquivalent zu einer blockierenden Operation
- Nicht-blockierende Operationen sind nicht dasselbe wie sequentielle Unterprogramm-Aufrufe
  - Die Operation kann andauern, während die Anwendung nachfolgende Anweisungen ausführt!

Kollektive Kommunikationen

- Viele Prozesse sind gleichzeitig beteiligt
- Üblicherweise vom MPI-Hersteller optimierte Implementierungen wie z.B. “tree based” Algorithmen
- Kann aus P2P-Routinen implementiert werden

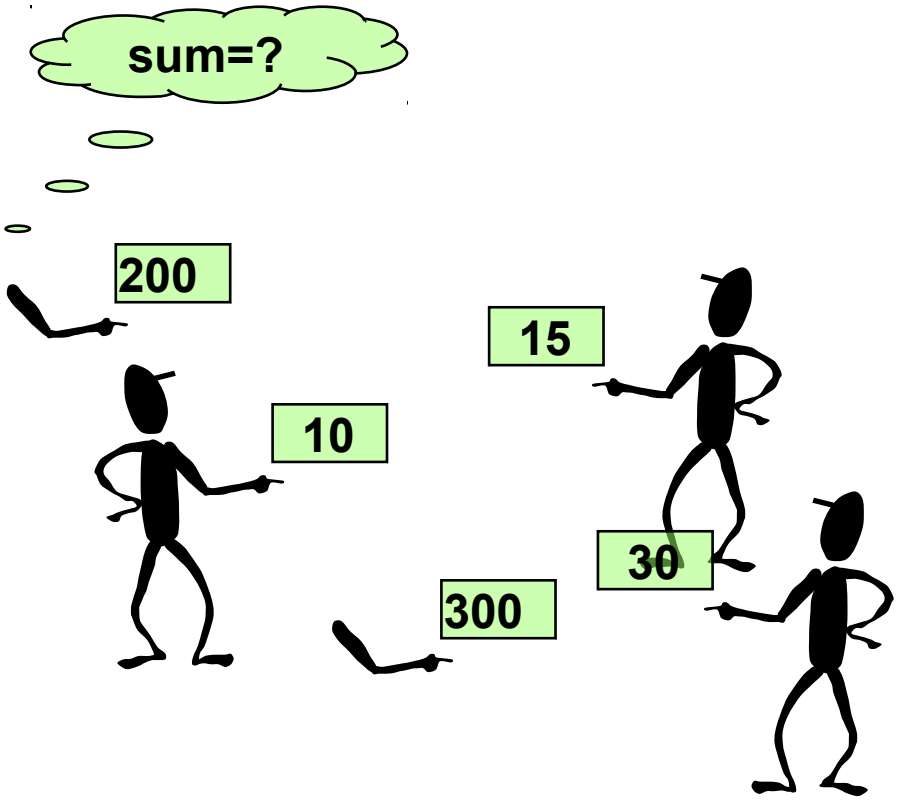
Broadcast

- Eine “einer-an-alle” Kommunikation



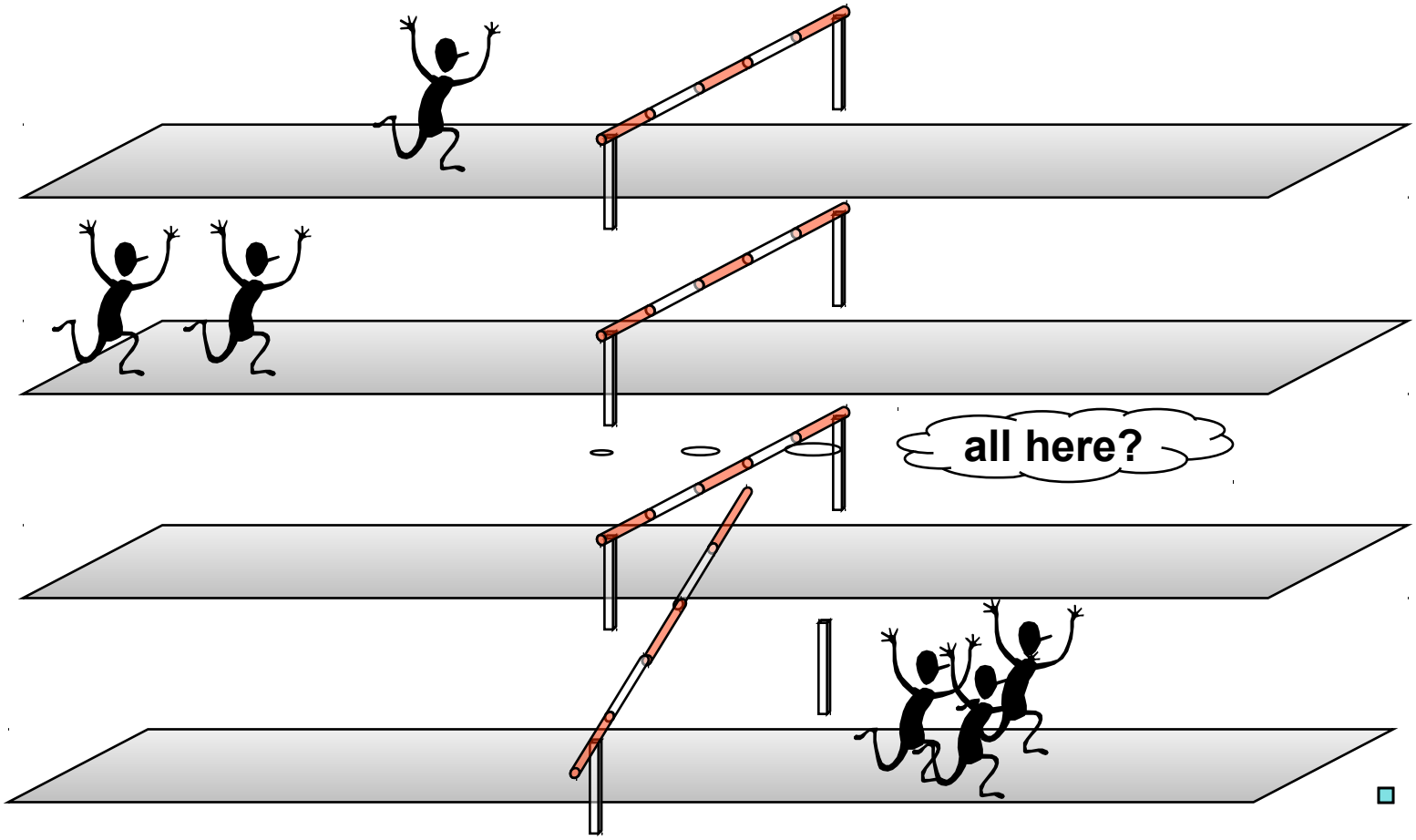
Reduktions-Operationen

- Kombiere Daten von (allen) Prozessen um ein einzelnes Ergebnis zu erzeugen



Barrieren

- Synchronisiere (alle) Prozesse



MPI-1

- MPI-1 Forum
  - Erster “Message Passing Interface” Standard
  - 60 Leute von 40 verschiedenen Organisationen beteiligt
  - MPI 1.0 — Juni, 1994
  - MPI 1.1 — Juni, 1995

- Nur SPMD-Programmierstil
- Keine Multithreading-Unterstützung
- Keine „active messages“
- Keine virtuellen Kanäle
- Keine Unterstützung von paralleler Ein-/Ausgabe, Debugging, Programmerstellung
- Auf der anderen Seite: schon heute (zu) viele Funktionen, MPI-Implementierungen etwa 10 mal größer als PVM-Implementierungen

MPI-2

- MPI-2 Forum
  - MPI-2: Erweiterungen zum “Message Passing Interface” Dokument (Juli, 1997)
  - MPI 1.2 — hauptsächlich Klarstellungen
  - MPI 2.0 — Erweiterungen zu MPI 1.2
  - MPI 2.1 — Zusammenführung der MPI-Linien (Juni, 2008)
  - MPI 2.2 — Kleine Verbesserungen und Erweiterungen (4.9.2009)
- Möglichkeit der dynamischen Prozesserzeugung und des dynamischen Prozeßmanagements ähnlich wie in PVM
- Einführung einseitiger Kommunikationsfunktionen im Hinblick auf “distributed shared memory” Systeme
- Erweiterungen der kollektiven Funktionen
- Ein-/Ausgabe-Funktionen
- zusätzliche Sprachanbindungen für C++ und FORTRAN 90/95 (in MPI-1 nur für C und FORTRAN 77)

MPI-3

- MPI Forum hat am 21.9.2012 MPI-3 veröffentlicht.

- Neue Funktionalitäten:
  - nicht blockierende Kollektive
  - verbessertes einseitiges Kommunikationsmodell (RMA, Remote Memory Access)
  - neues Interface für Fortran 2008
  - topographiebezogene Kommunikation
  - nicht blockierende parallele Ein- und Ausgabe

Initialisieren und Beenden von MPI

- **C:**

```
int MPI_Init( int *argc, char ***argv)
 …
int MPI_finalize()
```

```
#include <mpi.h>
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ....
    MPI_Finalize();
```

- **Fortran:**

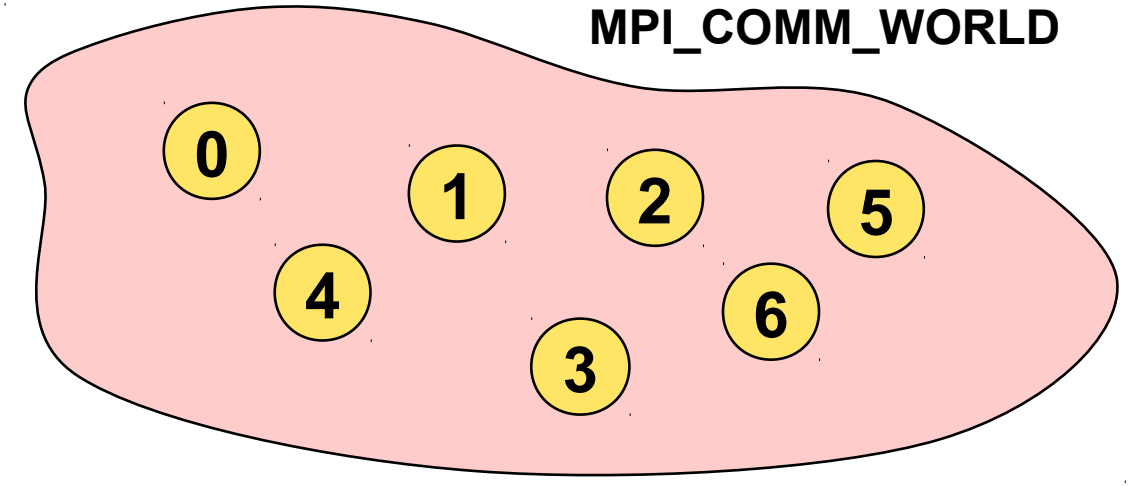
```
MPI_INIT( IERROR )
INTEGER IERROR
 …
MPI_FINALIZE(IERROR)
```

```
program xxxxx
implicit none
include ´mpif.h´
integer ierror
call MPI_Init(ierror)
....
call MPI_Finalize(ierror)
```

- **MPI_INIT muss die erste MPI-Routine sein; nach MPI_Finalize sind weitere MPI-Kommandos und auch eine Reinitialisierung mit MPI_INIT verboten!**

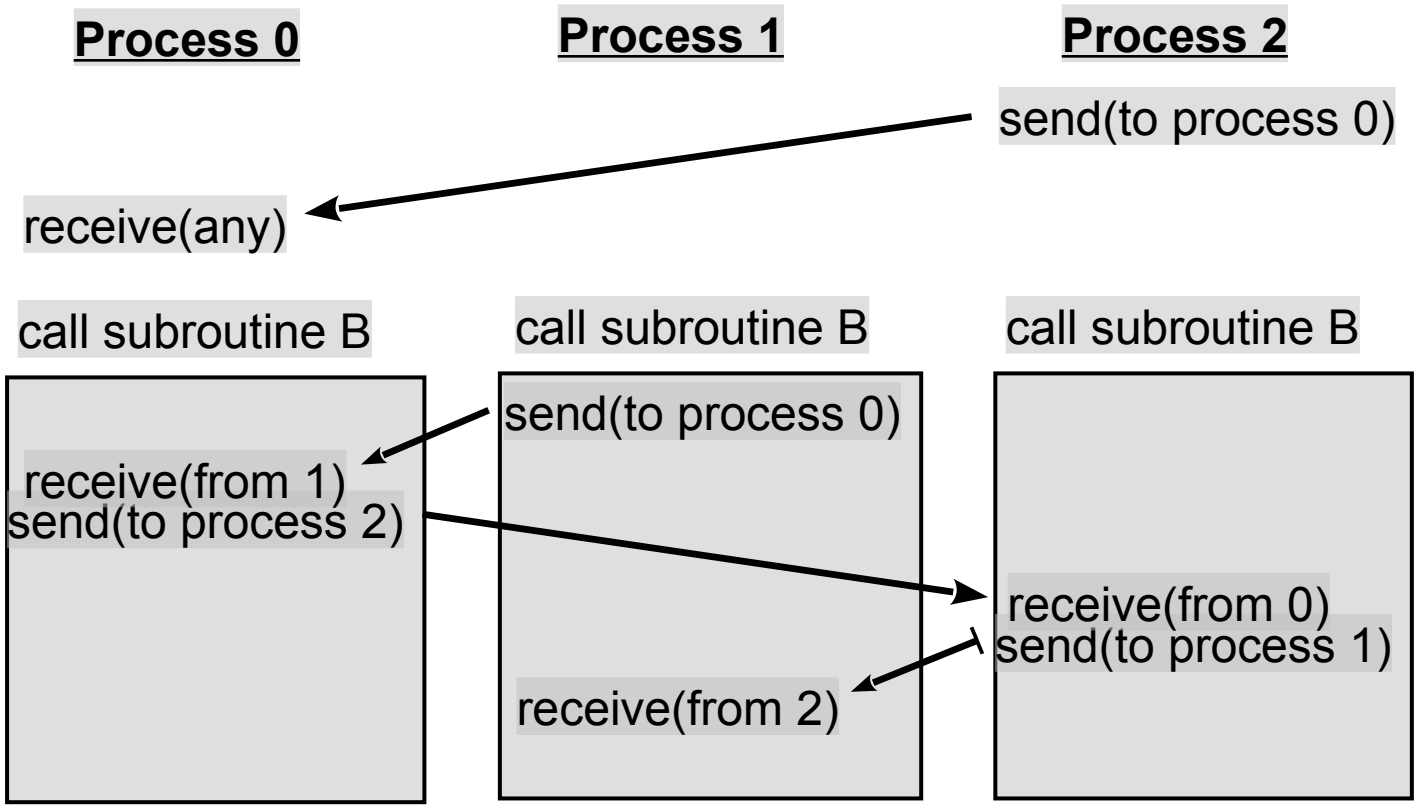
Kommunikator MPI_COMM_WORLD

- Alle Prozesse eines MPI-Programms “hängen” an dem Kommunikator MPI_COMM_WORLD
- MPI_COMM_WORLD ist ein vordefiniertes “handle” in mpi.h and mpif.h
- Jeder Prozess hat seinen eigenen Rang in einem Kommunikator mit den Zahlen 0..(size-1)

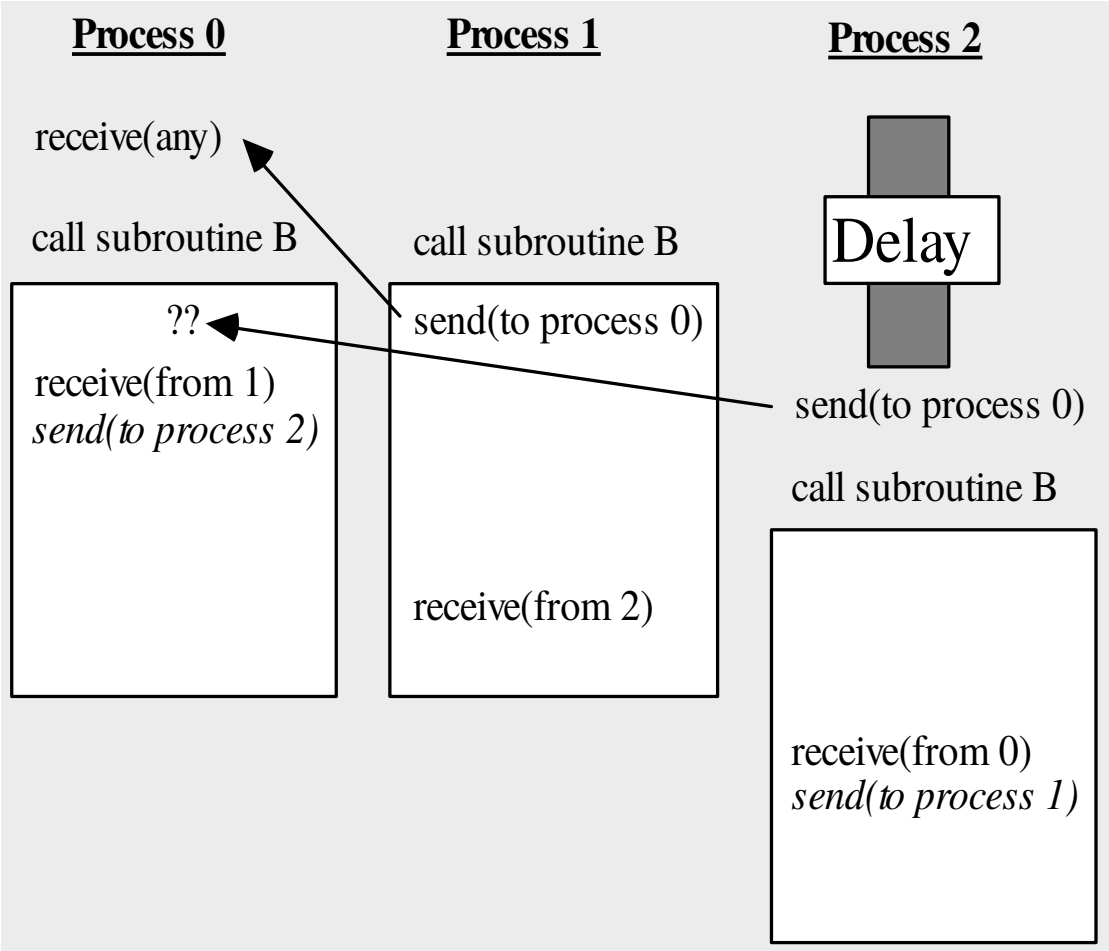


Warum den Kontext beachten?

wegen der Verwendung von Unterprogrammbibliotheken!



MPI_COMM_WORLD: Was kann passieren!



Der Kommunikator dient dazu, eine Nachricht auf ihren Kontext (hier die Subroutine) einzuschränken!

Kommunikator

- Kommunikator `MPI_COMM_WORLD`, der alle Prozesse vereinigt, von MPI vordefiniert
- Aus einer Prozessgruppe lassen sich Untergruppen bilden (z.B. mit `MPI_Group_excl`; dann kann mit `MPI_Comm_create` ein neuer Kommunikator erzeugt werden
- MPI unterstützt Topologien wie z.B. Gitter (cartesian grid), allgemeine Graphenstruktur oder auch keine Struktur durch Prozessgruppen
- Aufbau von Kommunikatoren mit angehängter Topologieinformation
  - `MPI_Dims_create` und `MPI_Cart_create` für kartesische Geometrien
  - `MPI_Graph_create` für Graphen
- Neben dem Intra-Kommunikator, der innerhalb einer Prozessgruppe und Topologie definiert ist und insbesondere für kollektive Operationen verwendet werden kann, gibt es noch den Inter-Kommunikator
  Ein Inter-Kommunikator betrifft zwei verschiedene Prozessgruppen und wird für die Punkt-zu-Punkt-Kommunikation zwischen Prozessgruppen eingesetzt.

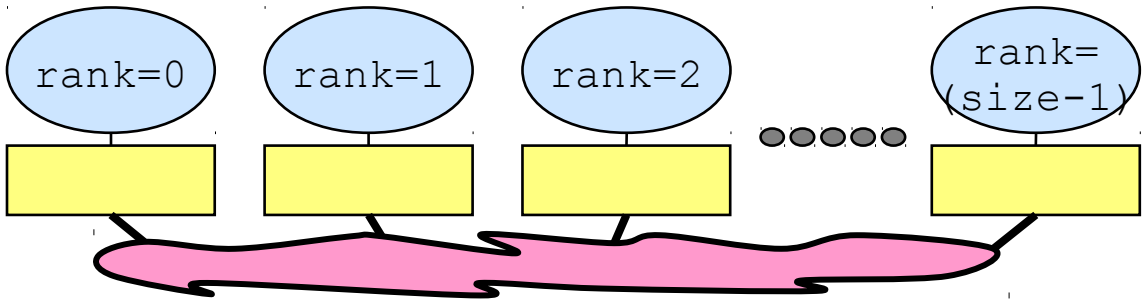
2 notwendige MPI-Kommandos

- Der Rang identifiziert unterschiedliche Prozesse mit Werten zwischen `0` und `size-1`; der Wert von `size` sollte vom MPI-System ermittelt werden!
- Der Rang ist die Basis für parallelen Code und die Verteilung der Daten
- C:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Fortran:

```
INTEGER comm, rank, size, ierror
MPI_COMM_RANK( comm, rank, ierror)
MPI_COMM_SIZE( comm, size, ierror)
```

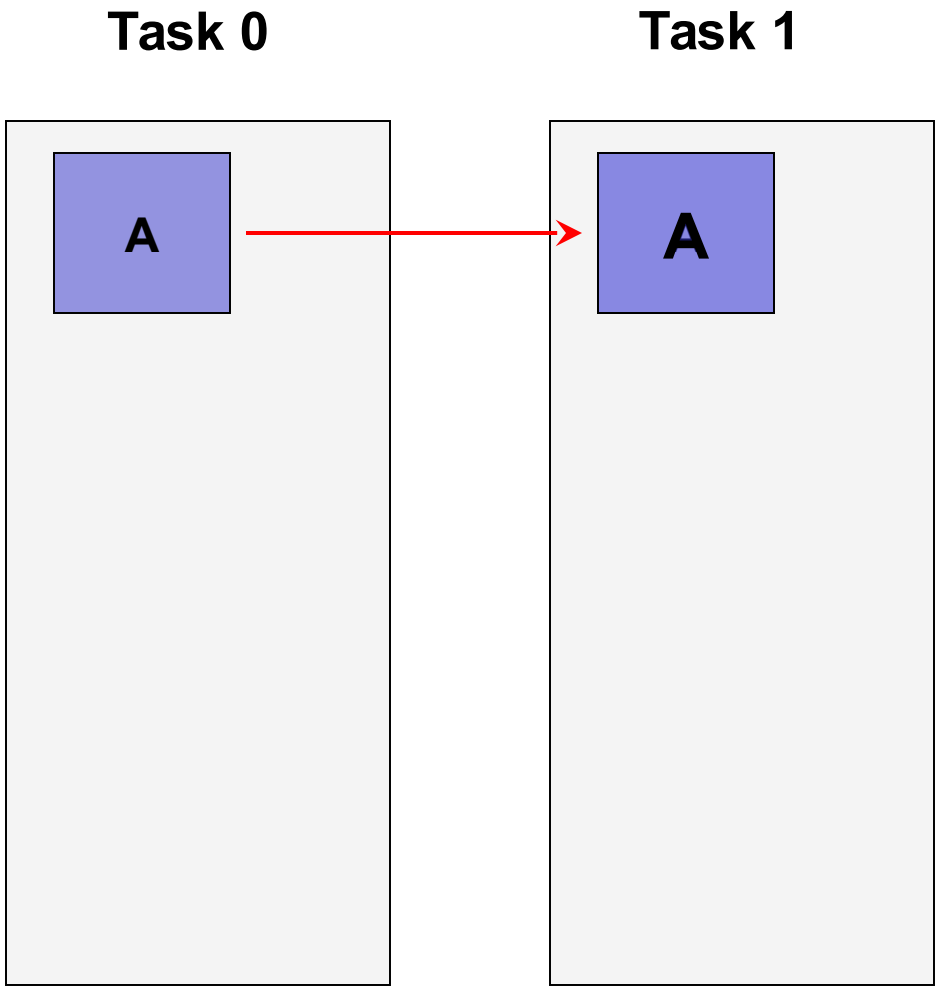


MPI Ausführungsmodell

```
PROGRAM main

REAL A(n,n)
INTEGER ierr
…
CALL MPI_Init(ierr)
CALL MPI_Comm_Size(...)
CALL MPI_Comm_Rank(...)
…
IF (rank == 0) THEN
 CALL MPI_Send(A, …)
ELSE
 CALL MPI_Recv(A, …)
ENDIF
…
CALL MPI_Finalize (...)

END PROGRAM main
```



MPI Basis Datentypen für C

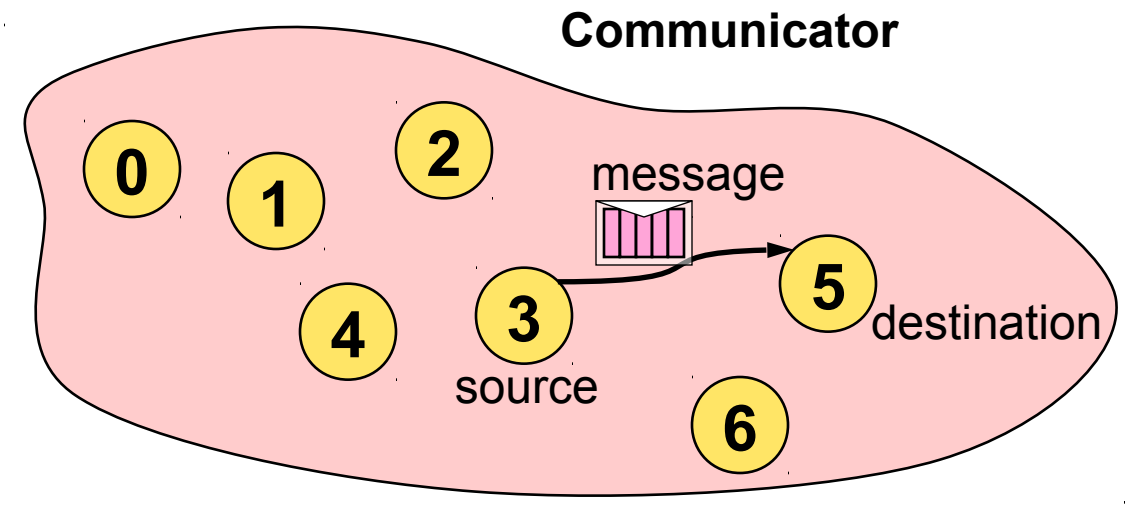
MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long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

MPI Basis Datentypen für Fortran

MPI Datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_INTEGER1	INTEGER (1 Byte)
MPI_INTEGER2	INTEGER (2 Byte)
MPI_INTEGER4	INTEGER (4 Byte)
MPI_REAL	REAL
MPI_REAL2	REAL (2 Byte)
MPI_REAL4	REAL (4 Byte)
MPI_REAL8	REAL (8 Byte)
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE PRECISION COMPLEX
MPI_ LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

P2P Kommunikation

- Kommunikation zwischen 2 Prozessen
- Prozess sendet Botschaft an anderen Prozess
- Kommunikation findet innerhalb eines Kommunikators statt, standardmässig MPI_COMM_WORLD
- Prozesse werden identifiziert durch ihre “ranks” in dem Kommunikator



Versenden von Botschaften

- C:
```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```
- Fortran:
```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```
- `buf` ist das erste Element der Botschaft; es werden `count` Werte vom Typ `datatype` übertragen
- `dest` ist der Rang des Zielprozesses innerhalb des Kommunikators `comm`
- `tag` ist eine zusätzliche nicht-negative Information vom Typ integer
- `tag` kann verwendet werden, um unterschiedliche Botschaften zu unterscheiden

Empfangen von Botschaften

- C:
```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Status *status)
```
- Fortran:
```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
                STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```
- `buf/count/datatype` beschreiben den Empfangspuffer
- Empfang der Botschaft, die von dem Prozess mit `rank` innerhalb des Kommunikators `comm` geschickt wurde
- Umschlagsinformation befindet sich in `status`
- Ausgabeargumente sind blau gedruckt
- Allein Botschafte mit passendem `tag` werden empfangen

Anforderungen an P2P Kommunikationen

Karlsruhe Institute of Technology

Für eine erfolgreiche Kommunikation muss

- **der Sender einen gültigen Rang spezifizieren**
- **der Empfänger einen gültigen Rang spezifizieren (Wildcard erlaubt!)**
- **der Kommunikator der gleiche sein**
- **eine Übereinstimmung der Tags vorliegen (Wildcard erlaubt!)**
- **der Datentyp der Botschaft übereinstimmen**
- **der Empfangspuffer groß genug sein**

Gebrauch von Wildcards

- Auf Empfangsseite können Wildcards benutzt werden
- Für den Empfang von beliebiger Quelle
  `source = MPI_ANY_SOURCE`
- Für den Empfang bei beliebigem tag
  `tag = MPI_ANY_TAG`
- Aktuelle Parameter `source` and `tag` befinden sich im Feld `status` auf Empfängerseite

Kommunikationsarten

Kommunikationsart	Definition	Bemerkung
Synchronous send `MPI_SSEND`	Ausführung wird nur beendet, wenn das zugehörige Empfangsroutine gestartet wurde	
Buffered send `MPI_BSEND`	Ausführung wird immer beendet	Benötigt Puffer, der von der Applikation mit `MPI_BUFFER_ATTACH` angelegt wurde
Standard send `MPI_SEND`	Entweder „synchronous“ oder „buffered“	Benutzt einen internen Puffer
Ready send `MPI_RSEND`	Wird nur gestartet, wenn zugehörige Empfangsroutine bereits gestartet wurde	Hoch gefährlich
Receive `MPI_RECV`	Wird beendet, sobald eine Botschaft abgespeichert wurde	Gleiche Routine für alle Kommunikationsarten

Regeln für Kommunikationsarten

- Standard send (`MPI_SEND`)
  - Minimale Transferzeit
  - Kann bei „synchronous mode“ blockieren
  - → Risiken bei Kommunikationsart „synchronous send“
- Synchronous send (`MPI_SSEND`)
  - Risiko von „deadlock“
  - Risiko der Serialisierung
  - Warte-Risiko → „idle“ Zeit
  - Hohe Latenzzeit / Beste Bandbreite
- Buffered send (`MPI_BSEND`)
  - Geringe Latenzzeit / Schlechte Bandbreite
- Ready send (`MPI_RSEND`)
  - Sollte nicht benutzt werden, außer man hat die 200% Garantie, dass `MPI_Recv` „sicher“ bereits aufgerufen wurde und dies auch für zukünftige Codeversionen gilt

(Nicht-)Blockierende Kommunikation

- Senden und Empfangen kann blockierend oder nicht-blockierend sein
- Ein blockierendes Senden kann zusammen mit einem nicht-blockierenden Empfagen benutzt werden und umgekehrt
- Nicht-blockierendes Senden kann in jeder Kommunikations-art benutzt werden
  - standard – `MPI_ISEND`
  - synchronous – `MPI_ISSEND`
  - buffered – `MPI_IBSEND`
  - ready – `MPI_IRSEND`

Test- und Informationsfunktionen

- Blockierender Empfangstest
  `MPI_Probe(source, tag, comm, status)`
- Nicht blockierender Empfangstest
  `MPI_Iprobe(source, tag, comm, flag, status)`
- Test, ob eine Sende- oder eine Empfangsoperation fertig ist
  `MPI_Test(request, flag, status)`
- Warten, bis eine Sende- oder eine Empfangsoperation fertig ist
  `MPI_Wait(request, status)`
- Länge der empfangenen Nachricht
  `MPI_Get_count(status, datatype, count)`
- Dabei gilt für C

```
void *buf
int count, source, dest, tag, *flag
MPI_Datatype datatype
MPI_Comm comm
MPI_Status *status
MPI_Request *request
```

Abgeleitete strukturierte Daten

- Ähnlich wie in PVM besteht die Möglichkeit, eine Struktur Element für Element in einen Puffer zu packen, zu versenden und dann wieder auszupacken (aufwendig, langsam und eine potentielle Fehlerquelle)

- Alternative: einfach alle Bytes einer Struktur am Stück zu versenden und zu empfangen; bei diesem Verfahren ist aber in heterogenen Workstation-Clustern keine Kontrolle über die internen Datenformate möglich

- Um trotzdem ein komfortables Arbeiten mit strukturierten Datentypen zu ermöglichen, kann man den vorhandenen Satz an MPI-Datentypen in einem C-konformen Umfang erweitern und danach komplette Datenstrukturen direkt mit dem Routinenpaar `MPI_Send` und `MPI_Recv` austauschen.

Abgeleitete strukturierte Daten (2)

- **Funktionen zum Packen und Entpacken sind**

`MPI_Pack(inbuf, incount, datatype, outbuf, outsize, pos, comm)`
packt Daten in einen Puffer

`MPI_Unpack(inbuf, incount, datatype, outbuf, outsize, pos, comm)`
holt Daten aus einem Puffer

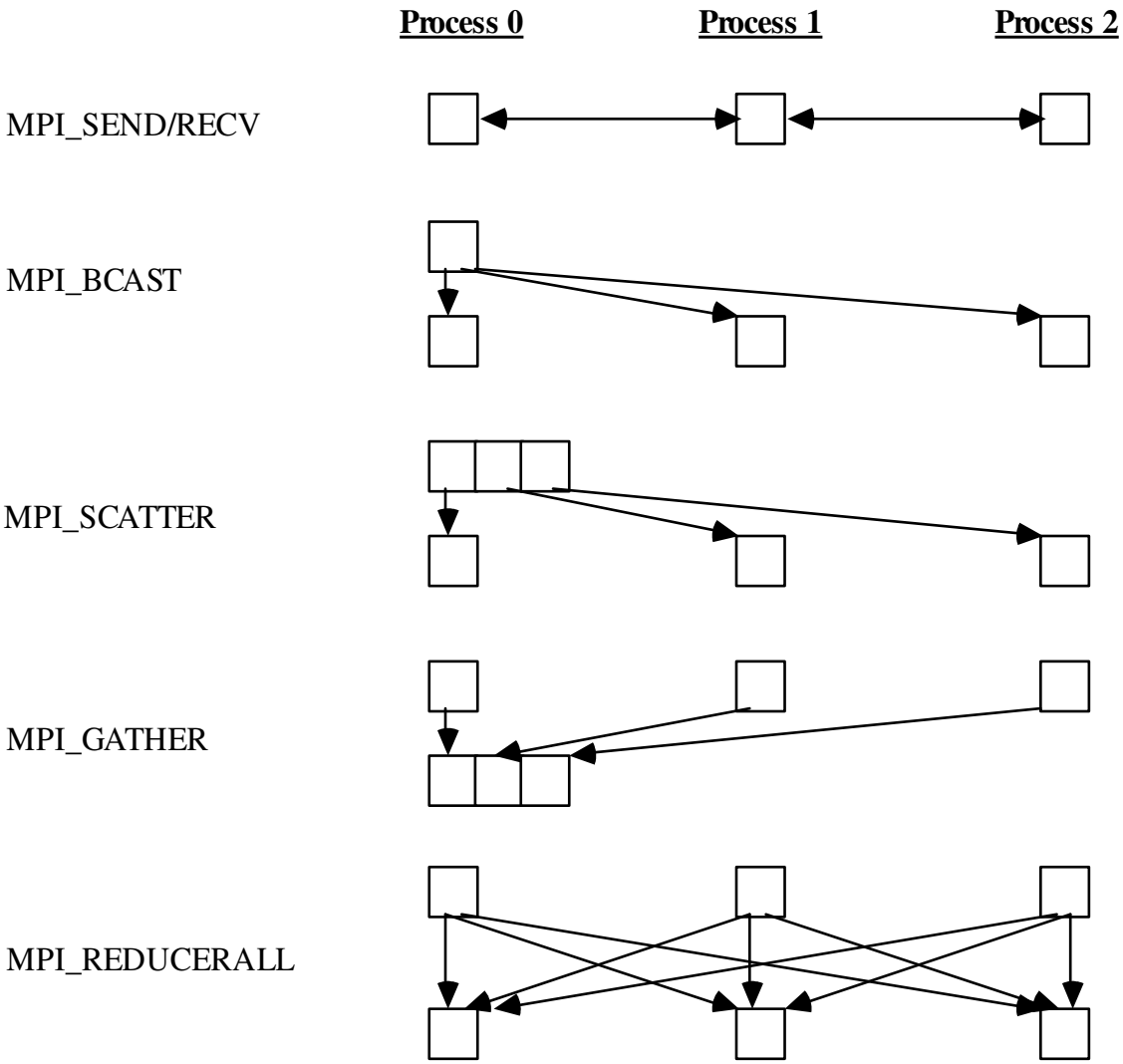
- **Abgeleitete Datentypen entstehen durch**

`MPI_Type_contiguous(count, oldtype, newtype)`
bildet einen neuen Typ `newtype` **aus** `count * oldtype`**.**

`MPI_Type_vector(count, blocklen, stride, oldtype, newtype)`
konstruiert ein MPI-Array mit Elementabstand `stride`**.**

`MPI_Type_struct(count, alens, adispls, atypes, newtype)`
konstruiert einen MPI-Verbund.

Kollektive Operationen



Kollektive Operationen (2)

Datentypen

```
MPI_Comm comm
MPI_Datatype datatype, intype, outtype
MPI_Op op
MPI_Uop *function()
int count, root, incnt, outcnt, commute
void *buffer, *inbuf, *outbuf
```

Funktionen

Warte auf die anderen Prozesse

```
MPI_Barrier(comm)
```

Broadcast: `buffer` an alle Prozesse schicken

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

Sammle Daten von allen Prozessen

```
MPI_Gather(outbuf, outcnt, outtype, inbuf, incnt, intype, root,
comm)
```

Kollektive Operationen (3)

Karlsruhe Institute of Technology

Daten von allen Prozessen an alle

```
MPI_Allgather(outbuf, outcnt, outtype, inbuf, incnt, intype, comm)
```

Verteile Daten auf alle Prozessoren

```
MPI_Scatter(outbuf,outcnt,outtype,inbuf,incnt,intype,root,comm)
```

Operationen auf verteilten Daten

```
MPI_Reduce(outbuf, inbuf, count, datatype, op, root, comm)
```

Ergebnis der Operation an alle

```
MPI_Allreduce(outbuf, inbuf, count, datatype, op, comm)
```

Neue Operation für „Reduce“

```
MPI_Op_create(function, commute, op)
```

Operation wieder entfernen

```
MPI_Op_free(op)
```

MPI Provider

- Computer-Hersteller (Intel, IBM Platform, …)
- MPICH2 – „public domain“ MPI-Bibliothek von Argonne
  - für alle UNIX Platformen, für Linux und Windows
  - MVAPICH/MVAPICH2 für MPI über InfiniBand und iWARP
- OpenMPI www.open-mpi.org
  - Verschmelzung von FT-, LA- und LAM/MPI