Parallel Computing I

Einführung in das
Hochleistungsrechnen

Message Passing Interface

14./21. Mai 2012
Paralleles Rechnen
SS 2012
Thorsten Grahs

Message Passing Interface

- Grundlagen
- Punkt-zu-Punkt-Kommunikation
- Kollektive Kommunikation
- Kommunikatoren

Message Passing Interface

I am not sure how I will Programm a Petaflop machine, but I am Sure I will need MPI somewhere

Horst D. Simon, Director Berkeley Super Computing Lab

- Rechner sind über ein Netzwerk verbunden
- Datentransport/Kommunikation über eine Netzwerk

Single Program Multiple Data (SPMD)

Im Allgemeinen wir in MPI nach diesem Prinzip verfahren

- Ein Programm welches auf allen Rechnern/Prozessoren ausgeführt wird
- Spezifikation spezieller Tasks für spezielle Rechner im Programm

```
if (my_rank != 0) ...
```

- Bearbeiten/Ausgeben verschiedener Daten

Historisches

Situation vor MPI

- Zahlreiche konkurrierende Variationen zum Thema Message Passing
  ⇒ Programme schwer portierbar
- Parallelverarbeitung stellte neue Technik/Wissenschaft dar
  ⇒ Untersuchungen notwendig über nützlichste Konzepte
- Es existierte kein gemeinsamer Standard
  ⇒ Hersteller betrachten eigene Lösung als Wettbewerbsvorteil

1994 Veröffentlichung des MPI-1 Standards und erste Implementation freie Implementierungen (MPICH, LAM-MPI, OpenMPI)

1997 Veröffentlichung MPI-2
(dynamisches Erzeugen von Prozessen, parallele I/O)

2008 Beginn der Vorbereitungen für MPI-3

Eigenschaften

MPI ist ...

- **groß** besitzt mittlerweile ca. 200 Funktionen
- **klein** ca. sechs Basisfunktionalitäten beschreiben die Grundkonzepte der Kommunikation

`MPI_Init`	Initialisierung vom MPI
`MPI_Comm_size`	Ermittlung der Prozessoranzahl
`MPI_Comm_rank`	Ermittlung der eigenen Prozessnummer
`MPI_Send`	Senden einer Nachricht
`MPI_Recv`	Empfangen einer Nachricht
`MPI_Finalize`	Beenden von MPI

Vorteile des Message Passing Modells

Universalität

Im Prinzip auf allen Rechnern und Architekturen einzusetzen. Sowohl auf parallelen Supercomputer wie auch auf Workstation und PC-Clustern einsetzbar, d.h. leicht portierbar.

Ausdrucksstärke

Vollständiges Modell zur Definition paralleler Algorithmen. Datenlokalität erlaubt direkte Kontrolle der Algorithmen, was Übersetzerbasierten Modellen fehlt.

Leistungsfähigkeit

Leistungsfähigkeit] Ideal für distributed Memory-Maschinen. Diese besitzen eine großen Vorteil: Sie verfügen über mehr Speicher und mehr Cache. Daraus resultiert besserer Speedup für große Modelle. Außerdem: Programmierer hat volle Kontrolle über Daten, d.h er kann sie ggf. direkt einzelnen Prozessen zuordnen.

Semantik von MPI – blockierend/nicht blockierend

blockierend

ist eine MPI-Anweisung, falls die Kontrolle zum aufrufenden Prozess erst zurückkehrt,wenn alle Ressourcen, die für den Aufruf genutzt werden, wieder für andere Operationen zur Verfügung stehen.

Die Steuerung kehrt erst zurück, **nachdem** die Nachricht empfangen wurde.

nicht blockierend

ist eine MPI-Anweisung, falls die Kontrolle zum aufrufenden Prozess zurückkehrt, **bevor** die durch sie ausgelösten Operationen und Ressourcen beendet sind bzw. wieder benutzt werden dürfen

Semantik von MPI – synchrone/asynchrone

synchrone Kommunikation

Die Übertragung einer Nachricht findet nur statt, wenn Sender und Empfänger **gleichzeitig** an der Kommunikation teilnehmen.

asynchrone Kommunikation

Übertragung findet statt, ohne sicher zu sein, dass der Empfänger bereit ist, die Nachricht zu empfangen. Der Sender übermittelt die Nachricht **einseitig**

Konventionen und erstes Syntax

Konventionen/Schreibweise

- `MPI_Xroutine`: Alle Funktionen beginnen MPI_ 1. Buchstabe groß.
- `PMPI_X`: Vorbehalten für Profiling Tools.
- `MPI_CONST`: Schreibkonvention für Konstante aus mpi.h.

```
#include <mpi.h>
main(int argc, char **argv) {
/* Initialise MPI */
MPI_Init (&argc, &argv);
/* There is no main program */
/* Terminate MPI */
MPI_Finalize ();
exit(1); }
```

MPI – Init & Finalize

```
int MPI_Init(int * argc, char * argv);
```

- Erste Routine zur Initialisierung von MPI.
- Mehrfacher Aufruf führt zu Fehlermeldungen

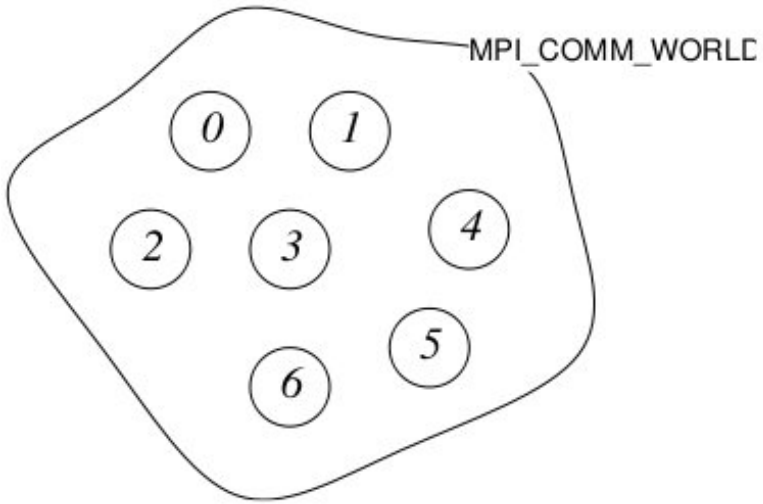
```
MPI_Finalize();
```

- Routine bereinigt/beseitigt alle MPI-Datenstrukturen
- Der Aufruf bereinigt **nicht** ausstehende Kommunikation.
  Dies ist Aufgabe des Programmierers
- Nach Aufruf der Routine können keine weiteren MPI-Routinen ausgeführt werden, nicht einmal `MPI_Init`

MPI – Kommunikatoren

MPI_COMM_WORLD

- Der Aufruf von `MPI_Init` definiert den Default-Kommunikator `MPI_COMM_WORLD`
- Jeder aktive Prozessor wird Mitglied diese Objektes.
- Diese werden nummeriert, das heißt ihnen wird ein Rang (rank) zugewiesen.
- Jeder MPI-Kommunikationsaufruf benötigt ein Kommunikator-Argument
- MPI-Prozesse können nur miteinander kommunizieren, wenn sie einen gemeinsamen Kommunikator besitzen



MPI_Comm_rank & MPI_Comm_size

```
MPI_Comm_rank (comm, & rank)
```

- Routine gibt in `rank` den Rang des aufrufenden Prozessors in der zum Kommunikator `comm` gehörende Prozessorgruppe zurück
- *Wer bin ich?*

```
MPI_Comm_size (comm, & size)
```

- Routine gibt in `size` die Gesamtanzahl der zum Kommunikator `comm` gehörenden Prozessoren zurück
- *... und wenn ja – Wie viele?*

Ein erstes Beispiel

Einfaches Beispiel für zwei Tasks – first.c

```
#include "mpi.h"
main(int argc,char **argv)
{
char text[20];
int myrank, size, sender=0, adressat=1, tag=99;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if(size > 1) {
printf("Beispiel für 2 Tasks\n");
MPI_Finalize();
exit(1);
}
```

Kompilieren und Ausführen

Kompilieren des Programms

```
mpicc ...
```

- MPI-Standartversion gibt es für Fortran/Fortran90, C, C++.
- andere über eigene Libraries verfügbar (z.B. mpi4py)

Ausführen des Programms (plattformabhängig)

```
mpiexec -np # Prozesse/Prozessoren <Programmname>
```

Vom MPI-Forum empfohlener Standard. Alternativ

```
mpirun -np # Prozesse/Prozessoren <Programmname>
```

Kompilieren und Ausführen des Beispiels

Kompilieren

```
mpicc -o first first.c
```

- Ausführen von 2 Prozessen:

```
> mpirun -np 2 first
Beispiel für 2 Tasks
Beispiel für 2 Tasks
```

- Ausführen von 4 Prozessen

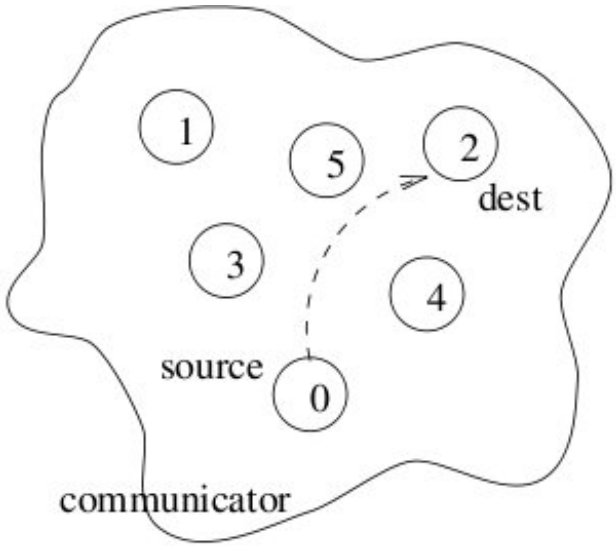
```
> mpirun -np 4 first
Beispiel für 2 Tasks
Beispiel für 2 Tasks
Beispiel für 2 Tasks
Beispiel für 2 Tasks
```

Punkt-zu-Punkt-Kommunikation

Einzeltransferoperation

Einfachste Form des Datenaustausches zwischen genau zwei Prozessoren, dem *Sender* und dem *Empfänger*. Dies unterscheidet es von kollektiver Kommunikation in MPI, bei der eine Gruppe von Prozessoren gleichzeitig kommunizieren.

- **senden**
  source-Prozess führt MPI call aus, mit *dest* als Zielprozessor
- **empfangen**
  dest-Prozessor führt ebenfalls einen MPI call aus, um die Nachricht zu empfangen



Kommunikation-Modi

Einzeltransferoperation

Paarweise Kommunikation: Je zwei Tasks kommunizieren miteinander. Eine Task sendet Daten (charakterisiert durch Adresse, Menge, Typ), die andere Task empfängt Daten (charakterisiert durch Adresse, verf. Platz, Typ).

- Keine Kontrolle auf Konformität
- Typen dürfen verschieden sein
- Konvertierung bei unterschiedlichen Zahlendarstellungen ?
- Fehler, falls verfügbarer Platz nicht ausreicht.

MPI_Send/Recv

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

- Sender `source` sendet Nachricht `buf` and Empfänger `dest`

```
int MPI_Recv(void *buf, int maxbuf, MPI_Datatype type, int source,
int tag, MPI_Comm comm, MPI_Status &status)
```

- Empfänger `dest` empfängt Nachricht `buf` von Sender `source`
- Paarweise Kommunikation setzt voraus, dass Sender und Empfänger kompatible Argumente verwenden
- Dies wird nicht überprüft, d.h. der Programmierer muss dies selber sicher stellen

MPI_Send/Recv – Argumente

Dabei bezeichnen

- `buf` Sende- bzw. Empfangspuffer, in dem die zu sendenden/empfangenden Elemente fortlaufend enthalten sind.
- `count/maxbuf` Anzahl bzw. Obergrenze der zu sendenden/Empfangenden Elemente
- `type` MPI-Datentype der Pufferelemente
- `dest/source` Nummer des Ziel-/Sendeprozessors
- `tag` Markierung, zur Unterscheidung verschiedener Nachrichten desselben Senders
- `comm` Kommunikator, d.h. Gruppe der Prozessoren, welche sich Nachrichten senden können.
- `status` Datenstruktur, welche Informationen über die empfangende Nachricht enthält

Status

```
MPI_Status *status
```

- Die Variable vom Typ `MPI_Status` ist eine Struktur
- Sie enthält Informationen über die gesendete Nachricht für den Empfänger
- Die Komponenten sind

```
typedef struct {
int MPI_SOURCE;
int MPI_TAG;
int MPI_ERROR;
} MPI_Status
```

spezifiziert Sender der empfangenen Nachricht

gibt die Markierung der empfangenen Nachricht an

enthält den Fehlercode

MPI_Get_Count

```
MPI_Get_count(&status, recv_type, &count)
```

liefert in count die Anzahl der empfangenen Elemente zurück
Typisch Anwendung ist das sondieren einer eingehenden Nachricht mit

```
MPI_Probe(source, tag, comm, & status)
```

- Funktion kehrt zum Aufrufer zurück, sobald Nachricht vom Absender source mit dem Etikett tag zum Empfangen vorliegt.
- Die Nachricht selber wird nicht empfangen; es wird die Statusvariable gesetzt
- Empfänger kann mit `MPI_Get_count(&status,...)` die Größe der wartenden Nachricht ermitteln, ausreichend Speicherplatz reservieren und die Nachricht empfangen.

Erweitern des Beispiels um senden/empfangen

```
#include "mpi.h"
main(int argc,char **argv) {
char text[20];
int myrank, size, sender=0, adressat=1, tag=99;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if(myrank == 0) {
  strcpy(text,"Hallo zusammen");
MPI_Send(text,strlen(text),MPI_CHAR,adressat,tag,MPI_COMM_WORLD);
} else {
MPI_Recv(text,20,MPI_CHAR,sender,tag,MPI_COMM_WORLD,&status);
  printf("Task %d empfing:%s:\n",myrank,text); }
MPI_Finalize();
exit(1);
}
```

Ausführen des erweiterten Beispiels

Zwei Prozesse

```
>mpirun -np 2 first2
Task 1 empfing:Hallo zusammen:
>
```

Prozess terminiert, alles wie es sein soll

Drei Prozesse

```
>mpirun -np 3 first2
Task 1 empfing:Hallo zusammen:
```

Der dritte Prozess wartet auf das Senden der angekündigten Nachricht...

Joker/Wild cards

- Ev. sind bestimmte Informationen über den sendenden Prozess nicht bekannt, bzw. unwichtig.
- Andererseits werden alle anderen Nachrichten ignoriert, solange der Empfänger auf eine bestimmte Nachricht wartet
- Ev. kommt es auf die Reihenfolge der eintreffenden Nachrichten nicht an...

  $\Rightarrow$ Wildcards für `source` und `tag`

`MPI_ANY_SOURCE`

stellt eine *Wildcard* als Senderangabe beim Empfangen dar

`MPI_ANY_TAG`

dient zum Empfangen von Nachrichten mit beliebigen `tag`

MPI_Send/Recv – Datentypen

MPI Datentyp	C Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long 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	

Protokollarten

Short protocol

Kurze Mitteilung (bis zu 16 Byte), die ohne weitere Vorkehrung durch das Netzwerk und seine Komponenten jederzeit sicher zugestellt werden kann.

Eager protocol

mittlere Mitteilung, die üblicherweise im Netzwerk zwischengespeichert wird, bis sie zugestellt werden kann. Speicherfähigkeit im Netzwerk wird durch eine short message überprüft.

Rendevouz protocol

Lange Mitteilung, die die Speicherkapazität im Netzwerk übersteigt und zur Zustellung verlangt, dass der Empfänger die Mitteilung verarbeiten kann.

Protokollarten II

Warum verschiedenen Protokollarten?

- Gezielter Einsatz verschiedener Protokollarten in MPI-Funktionen (Geschwindigkeit)
- Benutzer gesteuert
- Größte Performance

Einsatz

Vier verschiedene Kommunikationsformen (Senden)

Kommunikationsfunktionen – Senden

MPI_Send

- Standardfunktion
- Implementationsabhängig
- Üblicherweise als MPI_Ssend implementiert

MPI_Ssend

- Synchrones Senden
- Verwendet alle Protokolle in Abhängigkeit von Paketgröße
- Implizite Synchronisation, short message vor eager kann entfallen

Kommunikationsfunktionen – Senden

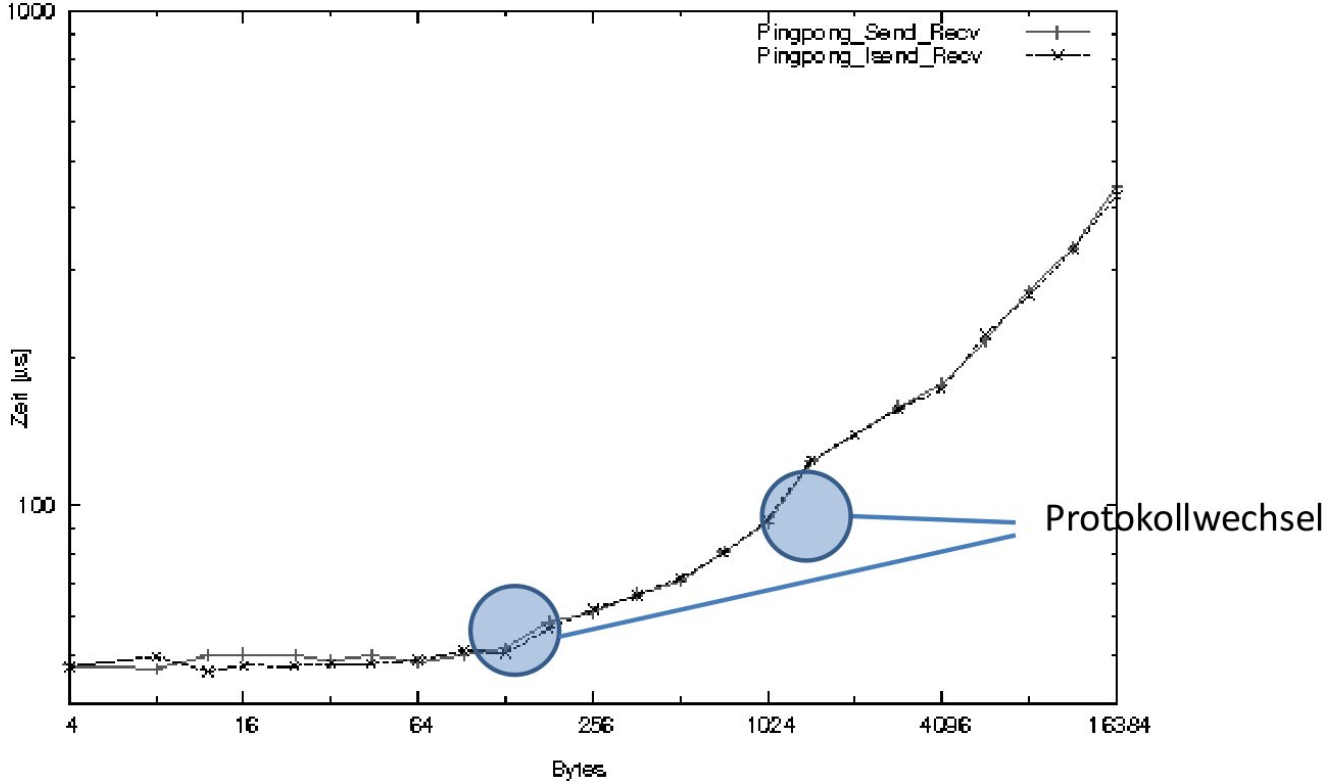
MPI_Rsend

- Senden nach Bestätigung durch Adressaten (Ready Send)
- Implizite Verwendung des Rendevouz Protokolls
- Bevorzugt bei großen Mitteilungen

MPI_Bsend

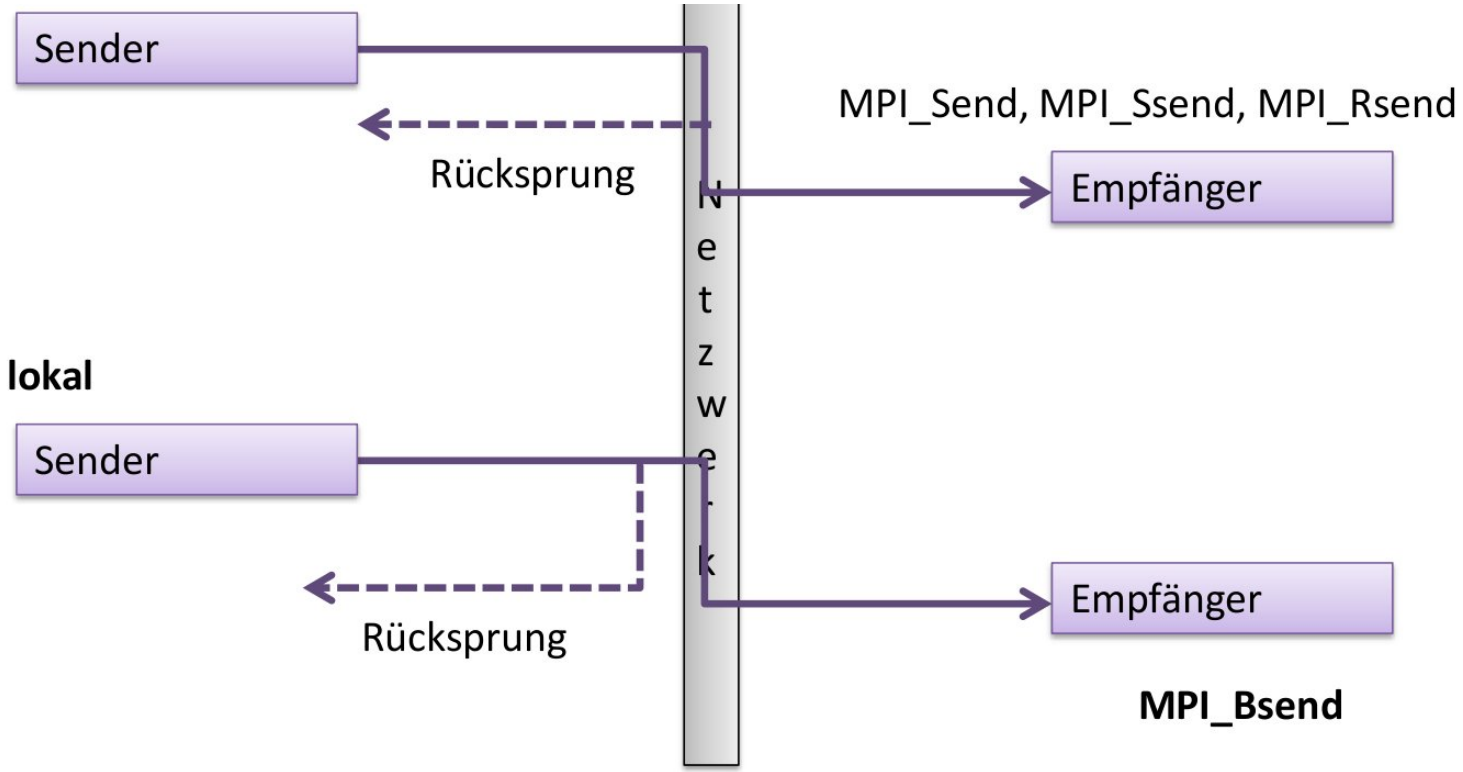
- Buffered Send
- Kopie der Mitteilung und Verschicken der Kopie
- Ungeeignet für große Mitteilungen und sehr kleine
- Lokale Operation

Vergleich MPI_Send/MPI_Isend



- Protokollwechsel bei 128 Byte und zwischen 1024 und 4096 Byte
- Geschwindigkeit von MPI_Send und MPI_Isend gleich
- $\Rightarrow$ nicht blockierendes Senden lohnt sich erst, wenn während des Sendens Berechnungen durchgeführt werden

lokale/nicht lokale Kommunikation



lokal: Rücksprung unabhängig von Kontakt mit Zielfunktion
nicht lokal: Rücksprung erst, wenn Nachricht empfangen wurde.
(Rücksprung hängt vom Zustand des Empfänger ab (also ggf. von anderen Prozessen)

Gepuffertes Senden

```
int MPI_Bsend(void *buf, int count, MPI_Datatype type, int dest,
int tag, MPI_Comm comm)
```

- Nachricht wird aus dem Sendepuffer ausgelesen und lokal in einem Puffer zwischengespeichert
- MPI_Bibliothek liefert Nachricht versetzt an Empfänger aus
  - \+ Senden/Empfangen entkoppelt
    Sender muss nicht auf Empfangsbereitschaft warten
  - \+ Bsend i.a. schneller
    Sender muss nur Umkopieren in lokalen Puffer abwarten
  - \+ Zeitgewinn für lokale Arbeit (Berechnung)
  - – Erhöhter Speicherverbrauch, besonders bei vielen langen Nachrichten
  - – Mehr Kommunikation bei nicht-blockierenden Senden notwendig.

```
int MPI_Buffer_attach(void *buffer, int size)
```

Puffer zum Zwischenspeichern der Nachricht muss vom Sender bereitgestellt werden. Dazu dient Funktion `MPI_Buffer_attach`.

MPI_attach/MPI_detach

```
int MPI_Buffer_attach(void *buffer, int size)
```

Puffer zum Zwischenspeichern der Nachricht muss vom Sender bereitgestellt werden. Dazu dient Funktion `MPI_Buffer_attach`.

- Der Speicher `buffer` muss natürlich vorher allokiert werden.
- Fehlermeldung von MPI_Bsend wenn Puffer nicht ausreichend für zu übertragende Nachricht.

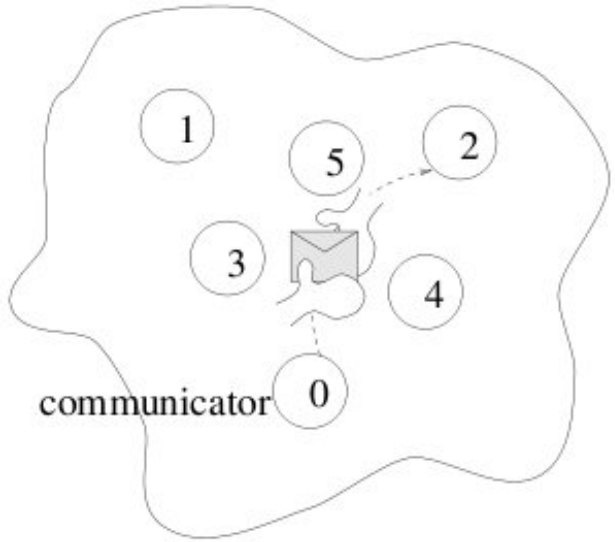
```
int MPI_Buffer_detach(void *buffer, int size)
```

Der MPI-Implementation wird so mitgeteilt, dass der Pufferspeicher nicht mehr benötigt wird.

Synchrones Senden

```
int MPI_Ssend(void *buf, int count, MPI_Datatype type, int dest,
int tag, MPI_Comm comm)
```

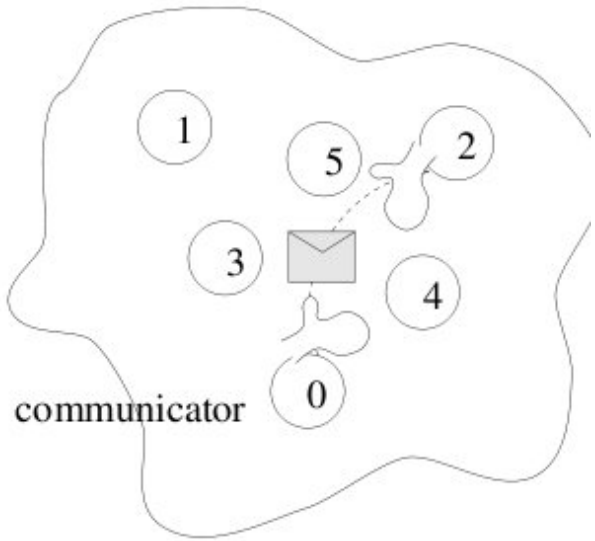
- Synchrones Senden ist eine blockierende Funktion
- Handshake zwischen Sender und Empfänger
- Bei Rückkehr kann der Sendepuffer sofort überschrieben werden



Empfangsbereites Senden

```
int MPI_Rsend(void *buf, int count, MPI_Datatype type, int dest,
int tag, MPI_Comm comm)
```

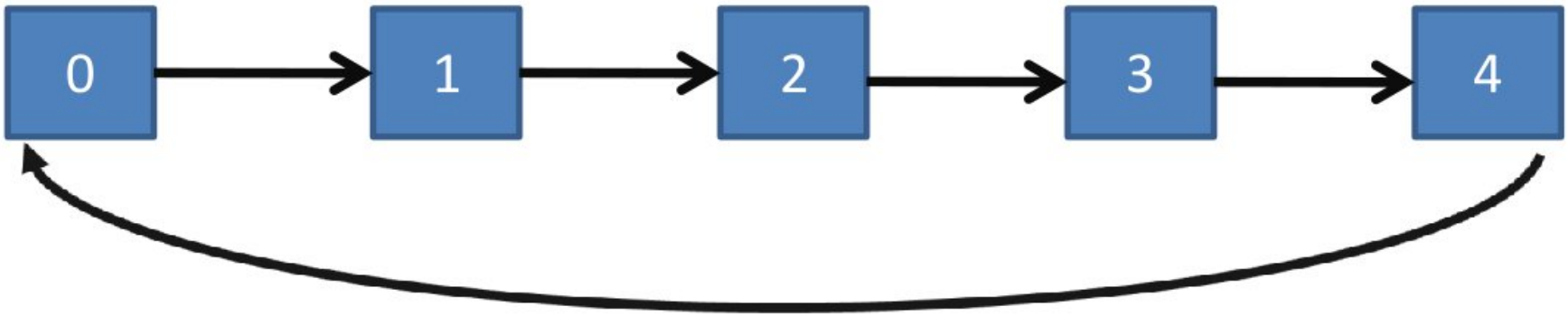
- Empfangsbereites Senden ist eine blockierende Funktion
- Fehlermeldung, falls zum Zeitpunkt des Aufrufs kein adäquater Empfangsaufruf vorliegt.
- Bei Rückkehr kann der Sendepuffer sofort überschrieben werden



Beispiel: Ringkommunikation

Prozessor P_i empfängt von P_{i-1} und sendet an P_{i+1}

```
src = (myrank-1+np)%np      linker Nachbar
dest= (myrank+1)%np         rechter Nachbar
```



Senden und Empfangen

```
for (i=0; i<np; ++i) {
  MPI_Send(sendbuf, bufsize, MPI_INT, dest, tag, comm);
  MPI_Recv(recvbuf, bufsize, MPI_INT, src , tag, comm,
                &status);}
```

Deadlock

Tasks warten gegenseitig aufeinander

```
if (myrank == src){
  MPI_Send(sendbuf, bufsize, MPI_INT, dest, tag, comm);
  MPI_Recv(recvbuf, bufsize, MPI_INT, dest, tag, comm,&status)
}
elseif (myrank == dest){
  MPI_Send(sendbuf, bufsize, MPI_INT, src , tag, comm);
  MPI_Recv(recvbuf, bufsize, MPI_INT, src , tag, comm,&status)
}
```

Sendebefehle blockieren sich gegenseitig, da die Funktionsaufrufe jeweils nicht terminieren können und so kein Empfangen erfolgt.

$\Rightarrow$ zyklisches Warten

Auflösung

Unsicheres Senden

Falls MPI_Send die Nachrichten intern puffert, funktioniert der Ringtausch.
Das ist aber implementationsabhängig oder von der Größe des Buffers.

Ersetze MPI_Send durch MPI_Ssend.

Entsteht kein Deadlock, ist das Senden sicher.

Ansonten

```
if (myrank == src){
  MPI_Send(sendbuf, bufsize, MPI_INT, dest, tag, comm);
  MPI_Recv(recvbuf, bufsize, MPI_INT, dest, tag, comm,&status)
}
elseif (myrank == dest){
  MPI_Recv(sendbuf, bufsize, MPI_INT, src , tag, comm);
  MPI_Send(recvbuf, bufsize, MPI_INT, src , tag, comm,&status)
}
```

Auflösung Ringtausch

- Gerade Prozessoren senden, ungerade Prozessoren empfangen
- Ungerade Prozessoren senden, gerade Prozessoren empfangen

```
if (myrank%2 == 1) {
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,  MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,  MPI_COMM_WO
}
else {
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,  MPI_COMM_WO
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,  MPI_COMM_WORLD);
}
```

Auflösung Deadlock mit Sendrecv

Simultane sende/empfangs-Operation

```
MPI_Sendrecv(sbuf, scount, stype, dest, stag, rbuf, rcount,
                          rtype, source, rtag, comm, &status)
```

- Funktion ist gleichzeitig sende und empfangsbereit.
- Sendenachricht im `sendbuf`
- Zu empfangende Nachricht im `recbuf`

Auflösung Deadlock zwischen zwei Prozessoren

```
Process 0       Process 1
Sendrecv(1)     Sendrecv(0)
```

Auflösung Deadlock ISend/IRecv bzw. Bsend

Nicht blockierende ISend/IRecv

```
Process 0       Process 1
ISend(1)        ISend(0)
IRecv(1)        IRecv(0)
Waitall         Waitall
```

- ISend/IRecv blockiert nicht, aber...
- Kommunikation benden, befor Speicherpuffer genutzt werden dürfen

Gepuffertes Bsend

```
Process 0       Process 1
Bsend(1)        Bsend(0)
Recv(1)         Recv(0)
```

- Nutzt vom User bereitgestellten Extrapuffer zum zwischenspeichern der Nachricht, unabhängig von Systemresourcen.

Kollektive Kommunikation

Kommunikation zwischen mehreren Tasks

- Kollektive Kommunikation,
  d.h. nicht unbedingt alle (=global)
- Auswahl des Kollektivs ggf. durch Definition eines entsprechenden Kommunikators
- Gegenwärtig nur blockierende Funktionen
- Alle Teilnehmer werden mit identischem Aufruf erreicht.
- Kein Tag

Kollektive Kommunikation

Funktionen

- Barriereoperation
  `MPI_Barrier()`
  Alle Tasks warten aufeinander
- Broadcastroperation
  `MPI_Bcast()`
  Ein Task sendet an alle
- Akkumulationsoperation
  `MPI_Reduce()`
  Ein Task verknüpft/operiert auf verteilten Daten
- Gatheroperationen
  `MPI_Gather()`
  Ein Task sammelt Daten ein
- Scatteroperationen
  `MPI_Scatter()` Ein Task verteilt/streut Daten (z.B. Vektor)

Kollektive Kommunikation

Multi-Task-Funktionen

- Multi-Broadcastoperation
  `MPI_Allgather()`
  Alle teilnehmenden Tasks stellen den anderen teilnehmenden Tasks Daten zur Verfügung

- Multi-Akkumulationsoperation
  `MPI_Allreduce()`
  Alle teilnehmenden Tasks erhalten Ergebnis der Operation

- Totaler Austausch
  `MPI_Alltoall()`
  Jeder beteiligt Task sendet und empfängt an bzw. von allen

Barriereoperation

`MPI_Barrier(comm)`:

Alle Tasks im comm warten gegenseitig auf das Erreichen einer Barriere.
Wird meistens zur Ablaufsynchronisation von Tasks eingesetzt.

Tasks warten aufeinander

`MPI_Isend` ist nicht beendet. Auf Daten darf nicht zugegriffen werden.

Barriereoperation

`MPI_Barrier(comm)`

Alle Tasks im comm warten gegenseitig auf das Erreichen einer Barriere.
Wird meistens zur Ablaufsynchronisation von Tasks eingesetzt.

```
if(myrank==0) {
 MPI_Isend(...);
 MPI_Barrier(comm);
} else {
 MPI_Barrier(comm);
}
```

Task 0

Task 1

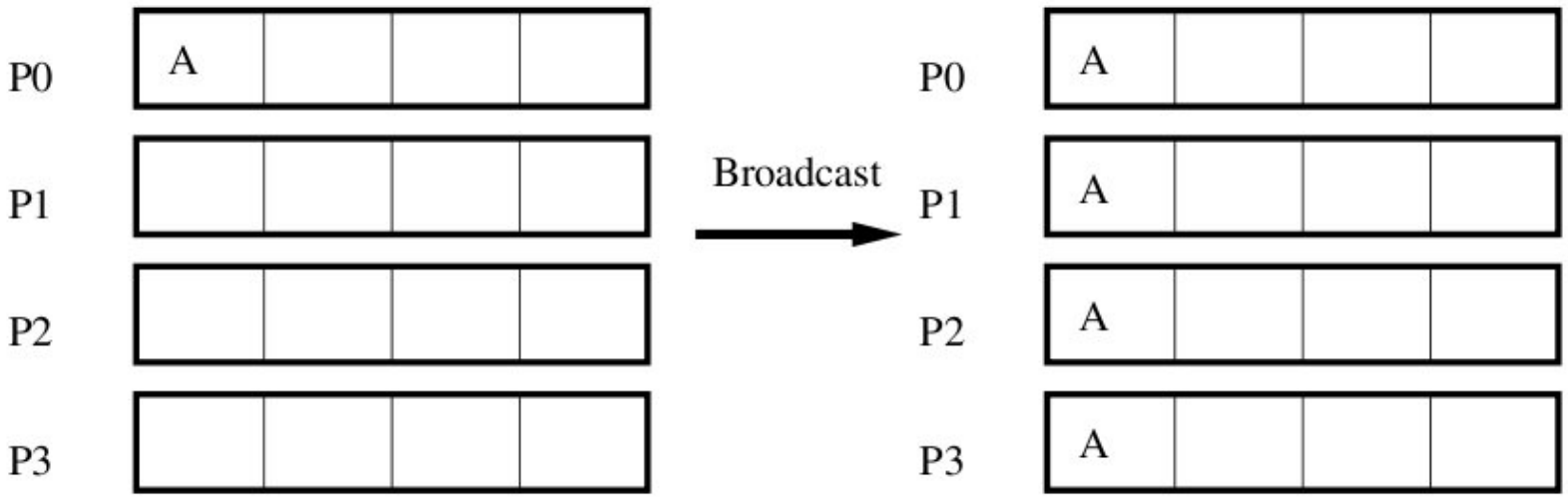
Tasks warten aufeinander

`MPI_Isend` ist nicht beendet. Auf Daten darf nicht zugegriffen werden.

Broadcastoperation

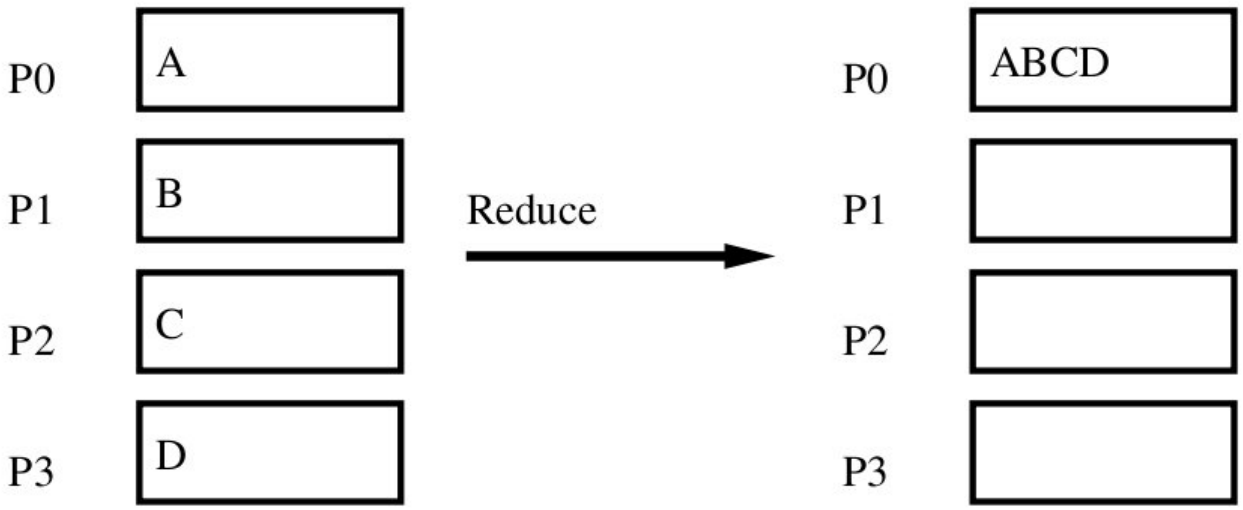
`MPI_Bcast(buffer,count,datatype,root,communicator)`

Alle Tasks im communicator benutzen den gleichen Funktionsaufruf. Daten von der Task mit rank root werden an alle Tasks im communicator verteilt. Der Aufruf ist blocking, aber nicht mit einer Synchronisation verbunden



Akkumulationsoperation

```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,master,comm)
```



- Aufrufender Prozess ist `master`
- Verknüpfungsoperation `op` (z.B. Summation)
- In `sendbuf` legen beteiligte Prozesse ihre lokalen Daten ab
- In `recvbf` sammelt `master` das Ergebnis ein

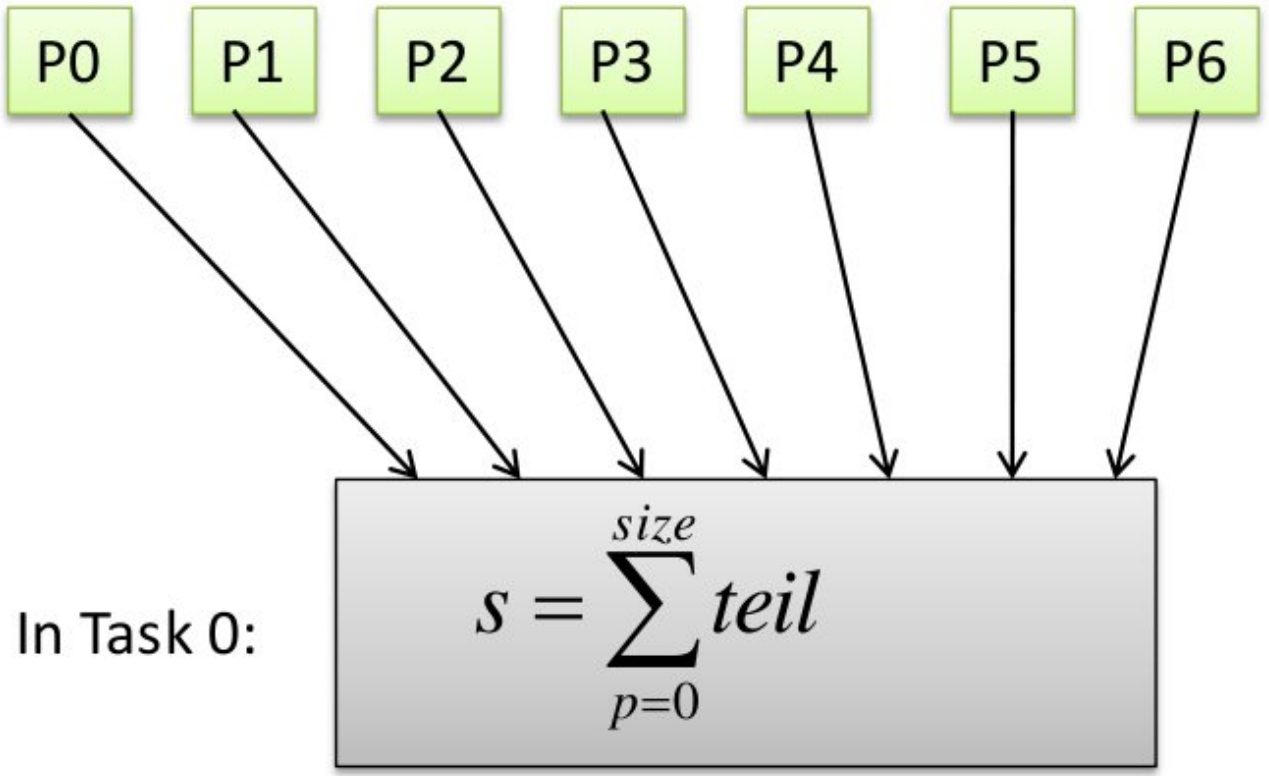
Reduceoperationen

Vordefinierte Operationen

- `MPI_MAX` Maximum
- `MPI_MAXLOC` Maximum und dessen Index
- `MPI_MIN` Minimum
- `MPI_SUM` Summation
- `MPI_PROD` Produktbildung
- `MPI_LXOR` logisches exclusives Oder
- `MPI_BXOR` Bitweises exclusives Oder
- ...

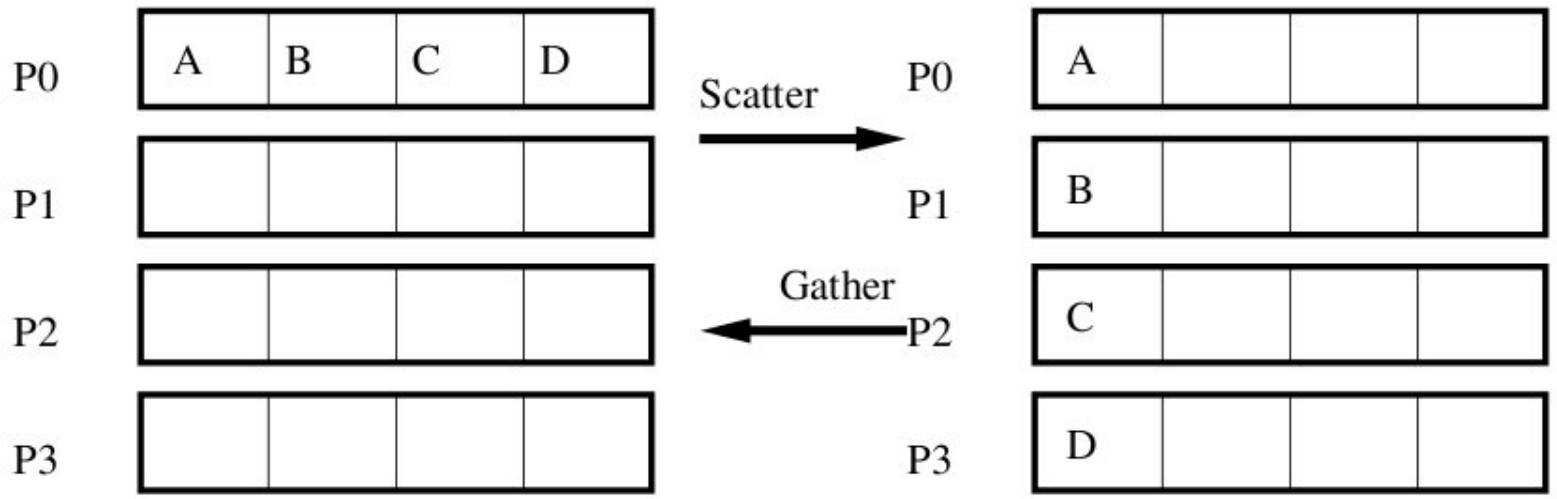
Beispiel: Reduce – Summation

```
MPI_Reduce(teil,s,1,MPI_DOUBLE,MPI_SUM,0,comm)
```



Gatheroperation

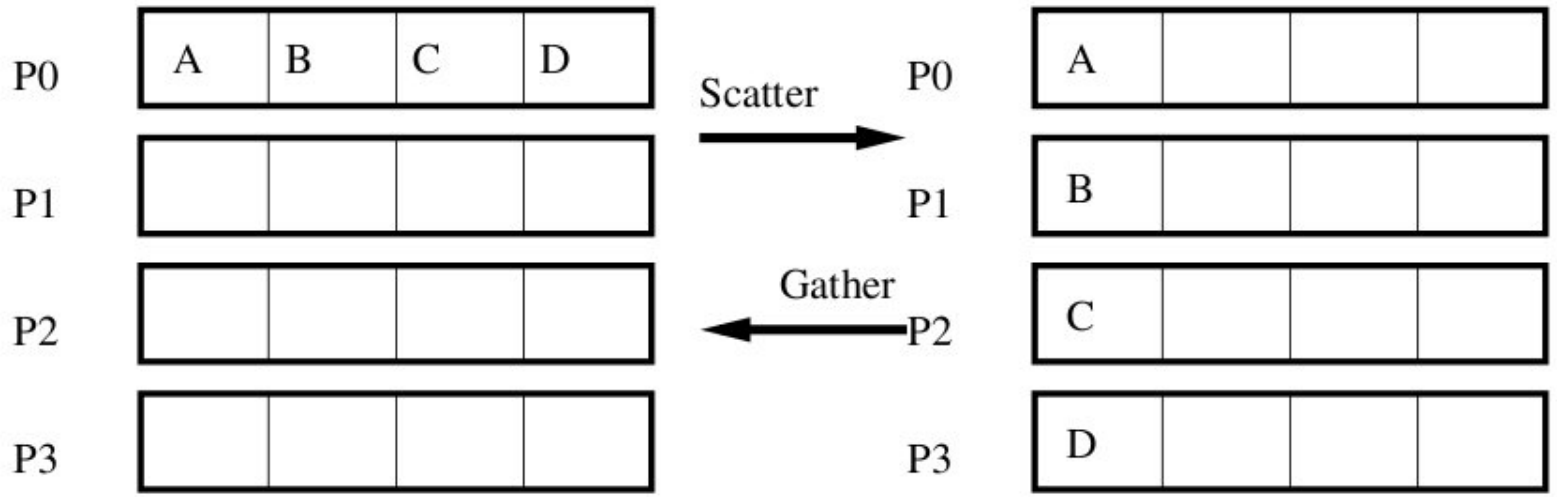
```
MPI_Gather(sbuf,scount,stype,rbuf,rcount,rtype,master,comm)
```



- `sbuf` lokaler Sendepuffer
- `rbuf` Empgangsbuffer von `master`
- jeder Prozessor sendet `rcount` Elemente vom Typ `rtype` an `master`
- Reihenfolge der Daten im `rbuf` in Reihenfolge der Nummerierung im Kommunikator `comm`

Scatteroperation

```
MPI_Scatter(sbuf,scount,stype,rbuf,rcount,rtype,master,comm)
```



- `master` verteilt /verstreut Daten aus `sendbuf`
- Jeder Prozessor empfängt Teilbuffer aus `sbuf` in lokalen Empfangspuffer`rbuf`
- `master` sendet auch an sich selber.
- Reihenfolge der empfangenen Daten im `rbuf` in Reihenfolge der Nummerierung im Kommunikator `comm`

Beispiel: Scatter

Drei Prozessoren in `comm` beteiligt

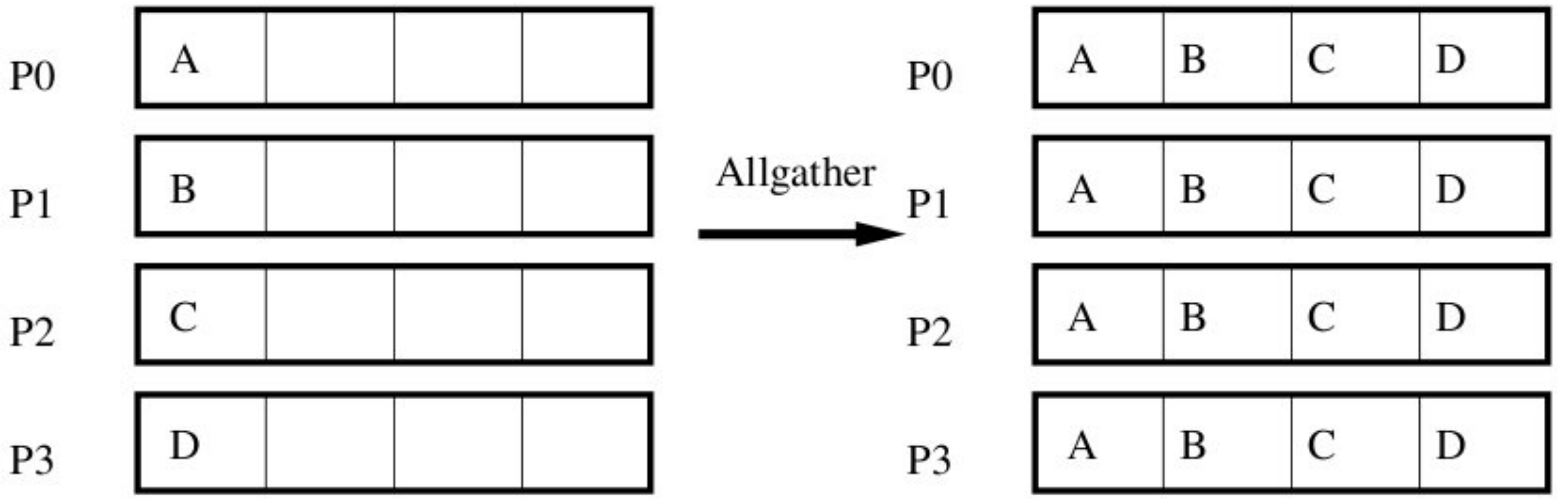
```
int sbuf[6]={3, 14, 15, 92, 65, 35};
int rbuf[2];
...
MPI_Scatter(sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 0, comm);
```

führt zur folgenden Verteilung:

Prozess	`rbuf`
0	`{ 3, 14}`
1	`{15, 92}`
2	`{65, 35}`

Multibroadcastoperation

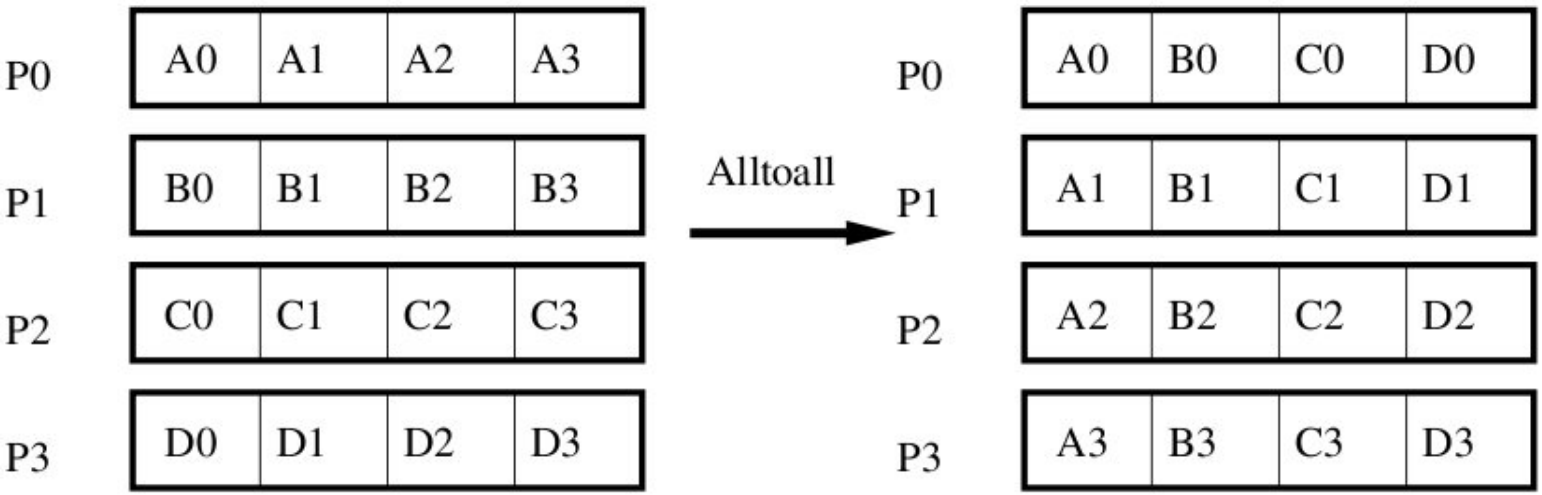
```
MPI_Allgather(sbuf,scount,stype,rbuf,rcount,rtype,comm)
```



- Daten aus lokalem `sbuf` werden an alle in `rbuf` versendet
- Angabe von Mastrer redundant, da alle die Gleichen Daten erhalten
- `MPI_Allgather` entspricht einem `MPI_Gather` gefolgt von einem `MPI_Bcast`

Totaler Austausch

```
MPI_Alltoall(sbuf,scount,stype,rbuf,rcount,rtype,comm)
```



- Matrixsichtweise

  Vor `MPI_Alltoall` besitzt Prozessor *k* die k-te Zeile der Matrix

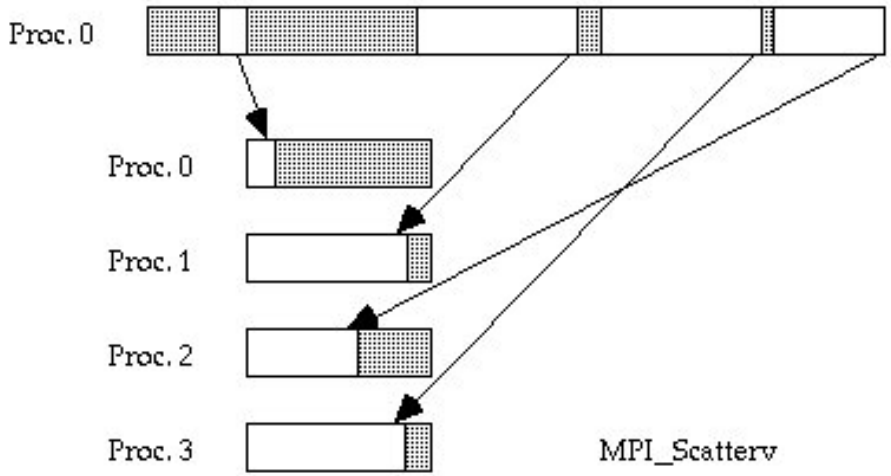
  Nach `MPI_Alltoall` besitzt Prozessor *k* die k-te Spalte der Matrix
- `MPI_Alltoall` entspricht einem `MPI_Gather` gefolgt von einem `MPI_Scatter`

Variable Austauschoperationen

Variable Scatter und Gather- Varianten

`MPI_Scatter` und `MPI_Gather` besitzen eine variable Version.
Variabel sind dabei

- Anzahl der Datenelemente, welche an die einzelnen Prozessoren verteilt wird
- deren Position im Sendebuffer `sbuf`



Variable Austauschoperationen

```
MPI_Scatterv(sbuf,scount,displs,styp,rbuf,rcount,rtyp,ma,comm)
```

- `scount[i]` enthält die Anzahl der an Prozess *i* zu versendende Datenelemente
- `displs[i]` legt den Beginn des Datenblocks für Prozess *i* relativ zu `sbuf` fest.

```
MPI_Gatherv(sbuf,scount,styp,rbuf,rcount,displs,rtyp,ma,comm)
```

- Es existieren auch die Funktionen

  `MPI_Allgatherv`

  `MPI_Allscatterv`

  `MPI_Alltoallv`

Beispiel – MPI_Scatterv

```
 /*Initialisierung */
if(myrank==root) init(sbuf,N);
/* Aufteilung der Arbeit und der Daten */
MPI_Comm_size(comm,&size);
Nopt=N/size;
Rest=N-Nopt*size;
displs[0]=0;
for(i=0;i<N;i++) {
scount[i]=Nopt;
if(i>0) displs[i]=displs[i-1]+scount[i-1]*sizeof(double);
if(Rest>0) { scount[i]++; Rest--;}
}
/* Verteilung der Daten */
MPI_Scatterv(sbuf,scount,displs,MPI_DOUBLE,rbuf,
                    scount[myrank],MPI_DOUBLE,root,comm);
```

Beispiel: Reduce

Multiplikation Matrix mit Vector

Beispiel: Reduce

Aufteilung zeilenweise, Ergebnisvektor verteilt

Beispiel: Reduce

Building block – Multiplikation Matrix*Vektor (BLAS: dgemv)

N: Dim/Anzahl Zeilen A, M: Dim/Anzahl Spalten A

```
void local_mv(N,M,y,A,lda,x) {
double x[N],A[N*M],y[M],s;
/*partial sum { local operation*/
for(i=0;i<M;i++) {
s=0;
for(j=0;j<N;j++)
s+=A[i*lda+j]*x[j];
y[i]=s;
}
}
```

Zeiten

arithmetisch	2*N*M*Ta
Speicherzugriff	
X	M*Tm(N,1)
Y	Tm(M,1)
A	M*Tm(N,1)

Beispiel: Reduce

Ausgangsbasis: Alle Daten auf Task 0, Endergebnis auf Task 0

Vektor y:

MPI_Gather bzw. MPI_Gatherv

Aufwand (p-1 mal Empfangen):
(p-1)*Tk(M)

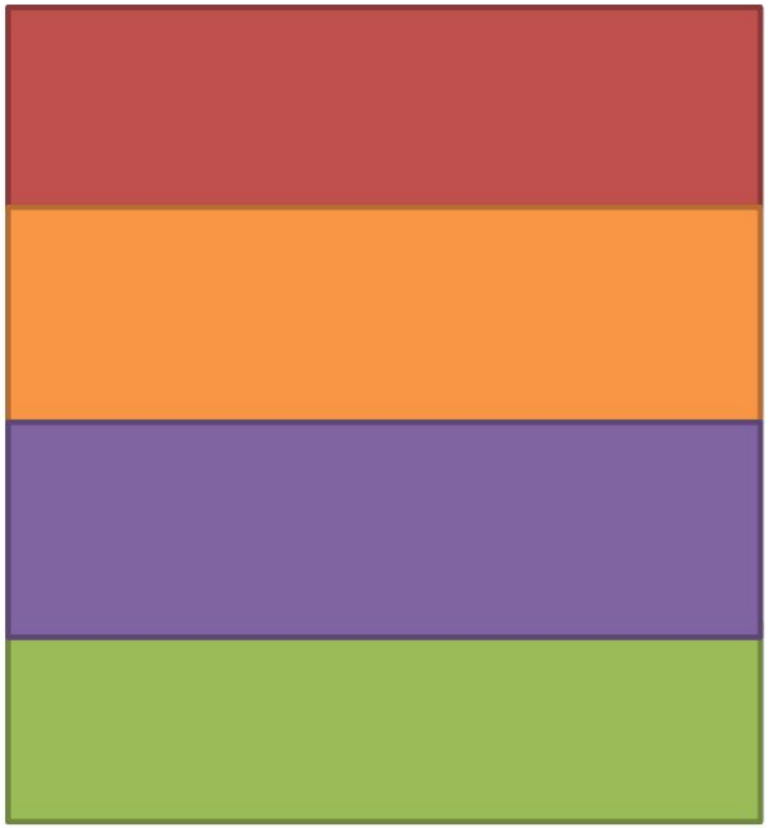
Beispiel: Reduce

Operationen

Verteile Vektor x an alle Prozessoren: MPI_Bcast (p-1)*Tk(N)
Verteilen Zeilen Matrix A: MPI_Scatter/MPI_Scatterv (p-1)*Tk(M*N)

Vektor x

Matrix A



Beispiel: Matrix-Vektor-Multiplikation

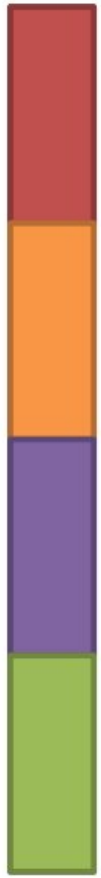
Aufteilung spaltenweise, Ergebnisvektor durch Reduktion

Beispiel: Matrix-Vektor-Multiplikation

Aufteilung Vektor x

Vektor x

`MPI_Scatter` (p-1)*Tk(M)

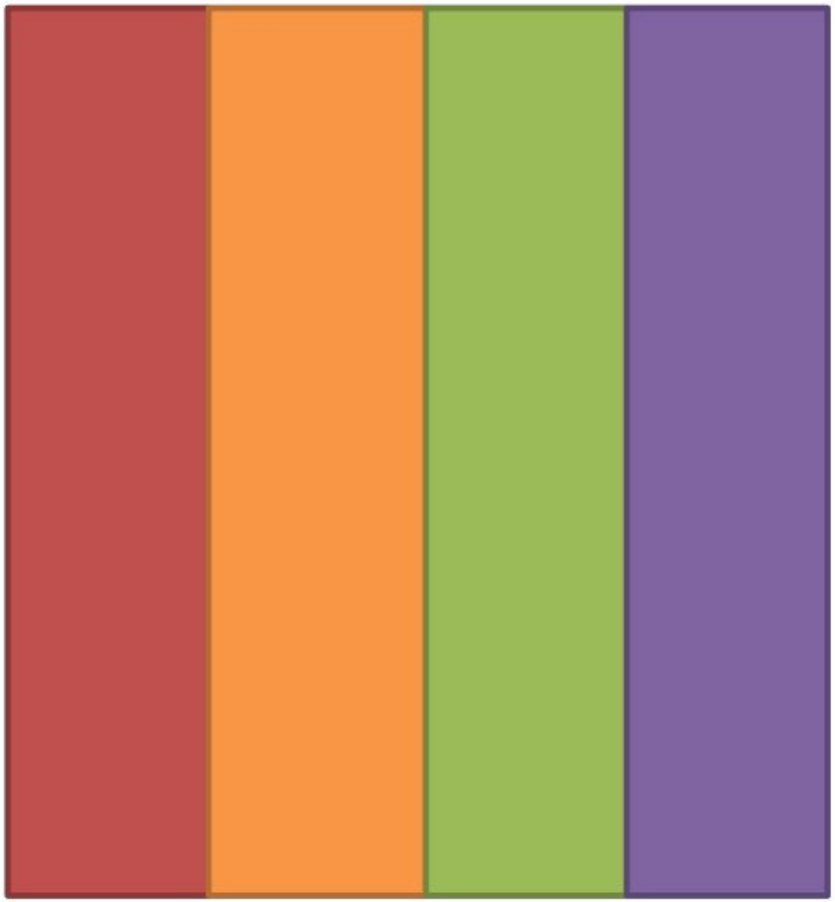


Beispiel: Matrix-Vektor-Multiplikation

Aufteilung Matrix A

Matrix A

?

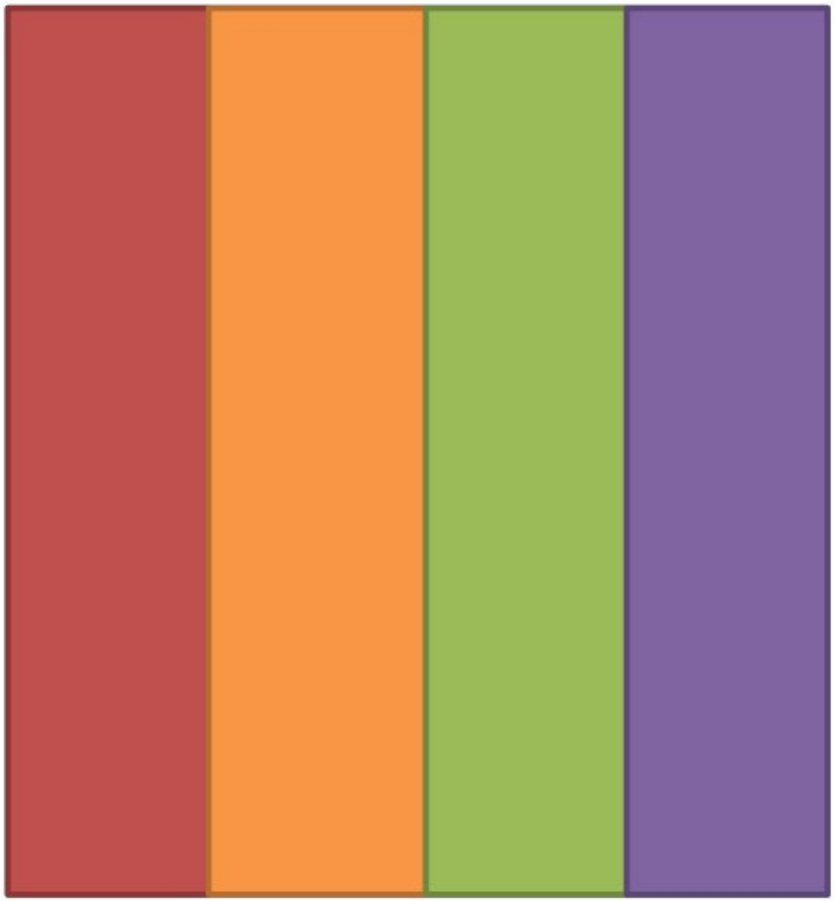


Beispiel: Matrix-Vektor-Multiplikation

Aufteilung Matrix A

Matrix A
Packen der Blöcke in Puffer:
Speicher:
N*Tm(M,1)+M*Tm(N,1)

Senden:
(p-1)Tk(M*N)



Beispiel: Matrix-Vektor-Multiplikation

Vektor y – `MPI_Reduce`

Reduktion von y: log(p)(Tk(N)+NTa+2Tm(N,1))

- Arithmetik: 2*N*M*Ta
- Speicher: N*Tm(M,1) + Tm(N,1) + N*Tm(M,1)

Algorithmus unwesentlich schneller

Parallelisierung ist nur sinnvoll, wenn entsprechende Datenaufteilung schon vor dem Programmteil vorliegt.

Kommunikatoren

Motivation

- Communicator: Unterscheidung verschiedener Kontexte
- Konfliktfreie Organisation von Gruppen
- Einbindung von Software von dritter Seite
  z.B. Unterscheidung
  – Libary-Funktionen
  – Anwendung

Vordefinierte Kommunikatoren

- `MPI_COMM_WORLD`
- `MPI_COMM_SELF`
- `MPI_COMM_NULL`

Kommunikatoren duplizieren

```
MPI_Comm_dup(MPI_COMM comm, MPI_COMM &newcomm);
```

- Erzeugt eine Kopie `newcomm` von `comm`
- Identische Prozessorgruppe
- Erlaubt z.B. eindeutige Abgrenzung/Charakterisierung von Nachrichten

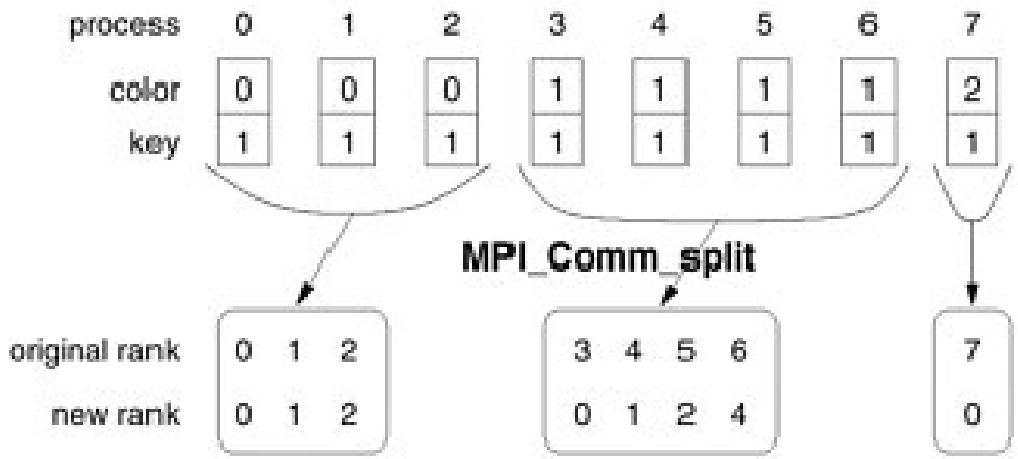
Beispiel

```
MPI_COMM myworld;
...
MPI_Comm_dup(MPI_COMM_WORLD, &myworld)
```

Kommunikatoren teilen

```
MPI_Comm_split(MPI_COMM comm, int color, int key, MPI_COMM
&newcomm);
```

- Unterteilt Kommunikator `comm` in mehrere Kommunikatoren mit disjunkten Prozessorgruppen
- `MPI_Comm_split` muss von allen Prozessen in `comm` aufgerufen werden
- Prozesse mit gleichen Wert `color` bilden gemeinsamen neuen Kommunikator



Kommunikatoren teilen

```
MPI_COMM comm1, comm2;
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);
i=rank%3;
j=size-rank;
if(i==0) MPI_Comm_split(comm,MPI_UNDEFINED,0,&newcomm);
else if(i==1) MPI_Comm_split(comm,i,j,&comm1);
else
MPI_Comm_split(comm,i,j,&comm2)
```

`MPI_UNDEFINED` Null-Handle `MPI_COMM_NULL`

Kommunikatoren teilen

MPI_COMM_WORLD

Rang	P0	P1	P2	P3	P4	P5	P6	P7	P8
color	$\perp$	1	2	$\perp$	1	2	$\perp$	1	2
key	8	7	6	5	4	3	2	1	0

MPI_COMM_WORLD

comm1

P1	P4	P7
2	1	0

comm2

P2	P5	P8
2	1	0

P0	P3	P6
0	1	2

Kommunikatoren auflösen

```
MPI_COMM_free(MPI_COMM *comm);
```

- Löschen des Kommunikators comm
- Die von comm belegten Ressource werden von MPI freigegeben
- Kommunikator hat nach dem Aufruf den Wert des Null-Handles MPI_COMM_NULL Funktion muss von allen Prozessen aus comm aufgerufen werden

Prozessorgruppen

```
MPI_COMM_group(MPI_COMM comm, MPI_Group *grp)
```

Zugriff auf die Prozessgruppe eines Kommunikators

- `MPI_COMM_create`
  Erzeugen eines Kommunikators aus einer Gruppe
- `MPI_Group_incl`
  Inkludieren von Prozessen in eine Gruppe
- `MPI_Group_excl`
  Exkludieren von Prozessen in eine Gruppe
- `MPI_Group_range_incl`
  Bilden einer Gruppe aus einfachen Mustern
- `MPI_Group_range_incl`
  Exkludieren von Prozessoren aus einfachen Mustern

Beispiel: Gruppenbildung

Gruppe

```
grp=(a,b,c,d,e,f,g),
n=3,
rank=[6,0,2]
```

- `MPI_Group_incl(grp, 3, &rank, &newgrp)` liefert
  `newgrp=(f,a,c)`
- `MPI_Group_excl(grp, 3, &rank, &newgrp)` liefert
  `newgrp=(b,d,e,g)`

Beispiel: Gruppenbildung II

Gruppe

```
grp=(a,b,c,d,e,f,g,h,i,j),
n=3,
ranges=[[6,7,1],[1,6,2],[0,9,4]]
```

Ranges Tripelbildung [Anfang, Ende, Abstand]

- `MPI_Group_range_incl(grp, 3, ranges, &newgrp)` liefert

  `newgrp=(g,h,b,d,f,a,e,i)`

- `MPI_Group_excl(grp, 3, ranges, &newgrp)` liefert

  `newgrp=(c,j)`

Operationen auf Kommunikatorgruppen

Weiter Funktionen

Darüber hinaus existieren weitere Funktionen zur Gruppierung:
z.B.

- Zusammenfassen von Gruppen `MPI_Group_union`
- Schnittmenge von Gruppen `MPI_Group_intersection`
- Differenz von Gruppen `MPI_Group_difference`
- Vergleich von Gruppen `MPI_Group_compare`
- Auflösen von Gruppen `MPI_Group_free`
- Größe einer Gruppe `MPI_Group_size`
- Rang einer Gruppe `MPI_Group_rank`
- ...

Intra & Interkommunikatoren

Intrakommunikator

- Kommunikator, der eine zusammenhängende Gruppe beschreibt.
- Alle bisherigen Kommunikatoren

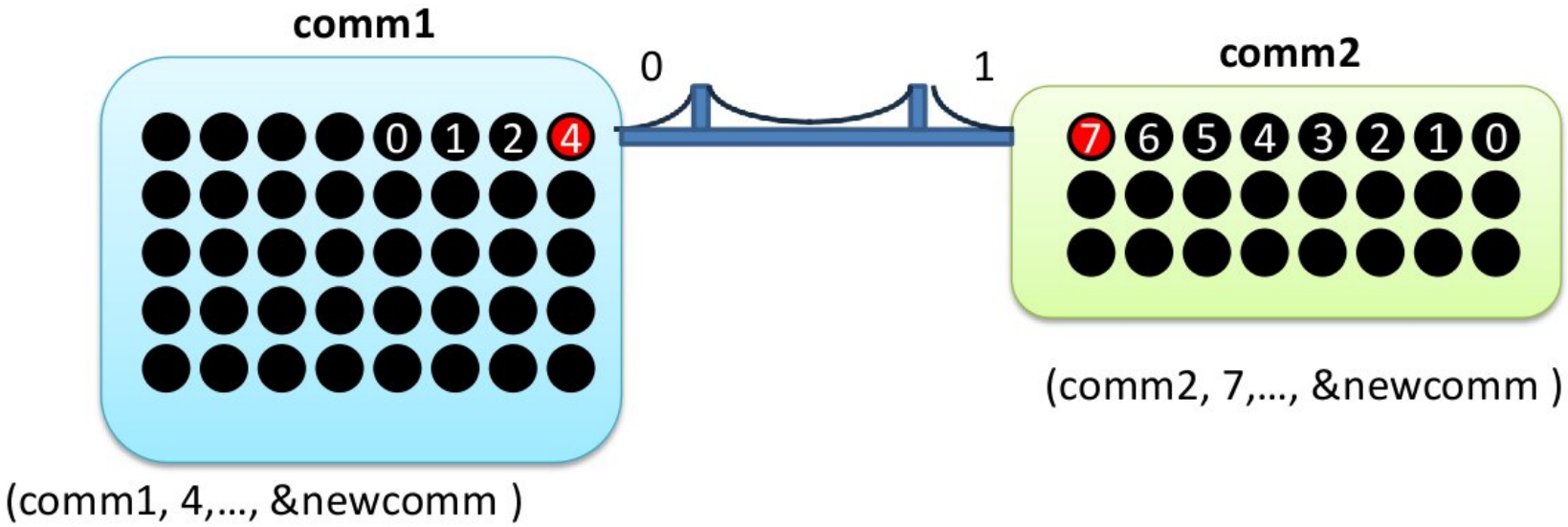
Interkommunikator

- Kommunikator, der einen Kontext zwischen Gruppen herstellt.
- Kommunikation zwischen MPI-Tasks und Benutzer-Tasks
- Kommunikation zwischen heterogenen Maschinen

Kommunikation zwischen Gruppen

Interkommunikator

```
MPI_Intercomm_create (local_comm, local_bridge,bridge_comm,
remote_bridge, tag, &newcomm )
```



(comm1, 0,MPI_COMM_WORLD,1,tag,&newcomm)

(comm2, 1,MPI_COMM_WORLD,0,tag,&newcomm)

Beispiel I

```
int main(int argc, char **argv)
{
    MPI_Comm   myComm;       /* intra-communicator of local sub
    MPI_Comm   myFirstComm;  /* inter-communicator */
    MPI_Comm   mySecondComm; /* second inter-communicator (grou
    int membershipKey;
    int rank;

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

    /* User code must generate membershipKey in the range [0, 1
    membershipKey = rank % 3;

    /* Build intra-communicator for local sub-group */
    MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm
```

Beispiel II

```
/* Build inter-communicators.  Tags are hard-coded. */
if (membershipKey == 0)
{                   /*Group 0 communicates with group 1. */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                        1, &myFirstComm);  }
else if (membershipKey == 1)
{         /* Group 1 communicates with groups 0 and 2. */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                        1, &myFirstComm);
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                        12, &mySecondComm);
}
else if (membershipKey == 2)
{                 /* Group 2 communicates with group 1. */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                        12, &myFirstComm);
```

Beispiel III

```
    /* Do work ... */

    switch(membershipKey)  /* free communicators appropriatel
    {
    case 1:
       MPI_Comm_free(&mySecondComm);
    case 0:
    case 2:
       MPI_Comm_free(&myFirstComm);
       break;
    }

    MPI_Finalize();
}
```

Motivation Interkommunikation

Interkommunikator – wozu?

- Meta-Computing
- Cloud-Computing
- Geringere Bandbreite zwischen den Komponenten
  z.B. Cluster - PC
- Brückenkopf kontrolliert Kommunikation mit remote-Rechner