

Einführung in die Parallelprogrammierung

MPI Teil 1: Einführung in MPI

Annika Hagemeier

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich



Inhalt

- Was ist MPI?
- MPI Historie
- Programmiermodell
- Kommunikationsmodell
- Laufzeitumgebung
- Syntax
- Initialisieren und Beenden von MPI
- Kommunikatoren
 - Handles
 - Zugriff auf Kommunikator-Informationen
- Kommunikationsarten
 - Blockierende und nicht-blockierende Kommunikation

Literatur

- *MPI: A Message-Passing Interface Standard (Version 3.1)*
Message Passing Interface Forum (2015)
<http://www mpi-forum.org/docs/>
- *Writing Message Passing Parallel Programs with MPI*
N. MacDonald, E. Minty, J. Malard, T. Harding, S. Brown, M. Antonioletti
- *Parallel Programming with MPI*
Peter Pacheco (Morgan Kaufmann)
<http://www.cs.usfca.edu/mpi>
- *MPI: The Complete Reference*
Snir, Otto, Huss-Lederman, Walker, Dongarra
<http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>
- *MPI-Funktionsübersicht:*
<http://www.cc.gatech.edu/projects/ihpcl/mpichdoc/www3/>

Was ist MPI? (1)

- MPI = **M**essage **P**assing **I**nterface
- Industriestandard für Message-Passing-Systeme
 - Hersteller von Parallelrechnern und Forscher an Universitäten, Laboren und in der Industrie sind an der Entwicklung beteiligt
 - <http://www mpi-forum.org>
- die Implementierung des Standards ist eine Bibliothek von Unterfunktionen
 - kann mit Fortran, C und C++ benutzt werden
 - enthält mehr als 300 Funktionen
 - 6 Funktionen reichen oft aus!
 - frei verfügbare MPI-Implementierungen:
 - MPICH
 - MPICH2
 - Open MPI

Was ist MPI? (2)

Vorteile von MPI:

- standardisiert
- portabel
- weit verbreitet (unterstützt von allen namhaften Herstellern)

MPI ist ein Standard, der den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen beschreibt.

Er legt dabei eine Sammlung von Operationen und ihre Semantik, also eine Programmierschnittstelle fest, aber kein konkretes Protokoll und keine Implementierung.

MPI Historie (1)

- **Version 1.0** (1994):
 - Unterstützung von Fortran77 und C
 - Anzahl neuer Funktionen: 129
- **Version 1.1** (1995):
 - Berichtigungen und Klarstellungen, kleine Änderungen
- **Version 1.2** (1997):
 - Weitere Berichtigungen und Klarstellungen
 - Anzahl neuer Funktionen: 1
- **Version 2.0** (1997):
 - Wesentliche Erweiterungen:
 - Einseitige Kommunikation
 - MPI-IO
 - dynamische Generierung von Tasks
 - Unterstützung von Fortran 90 und C++
 - Anzahl neuer Funktionen: 193

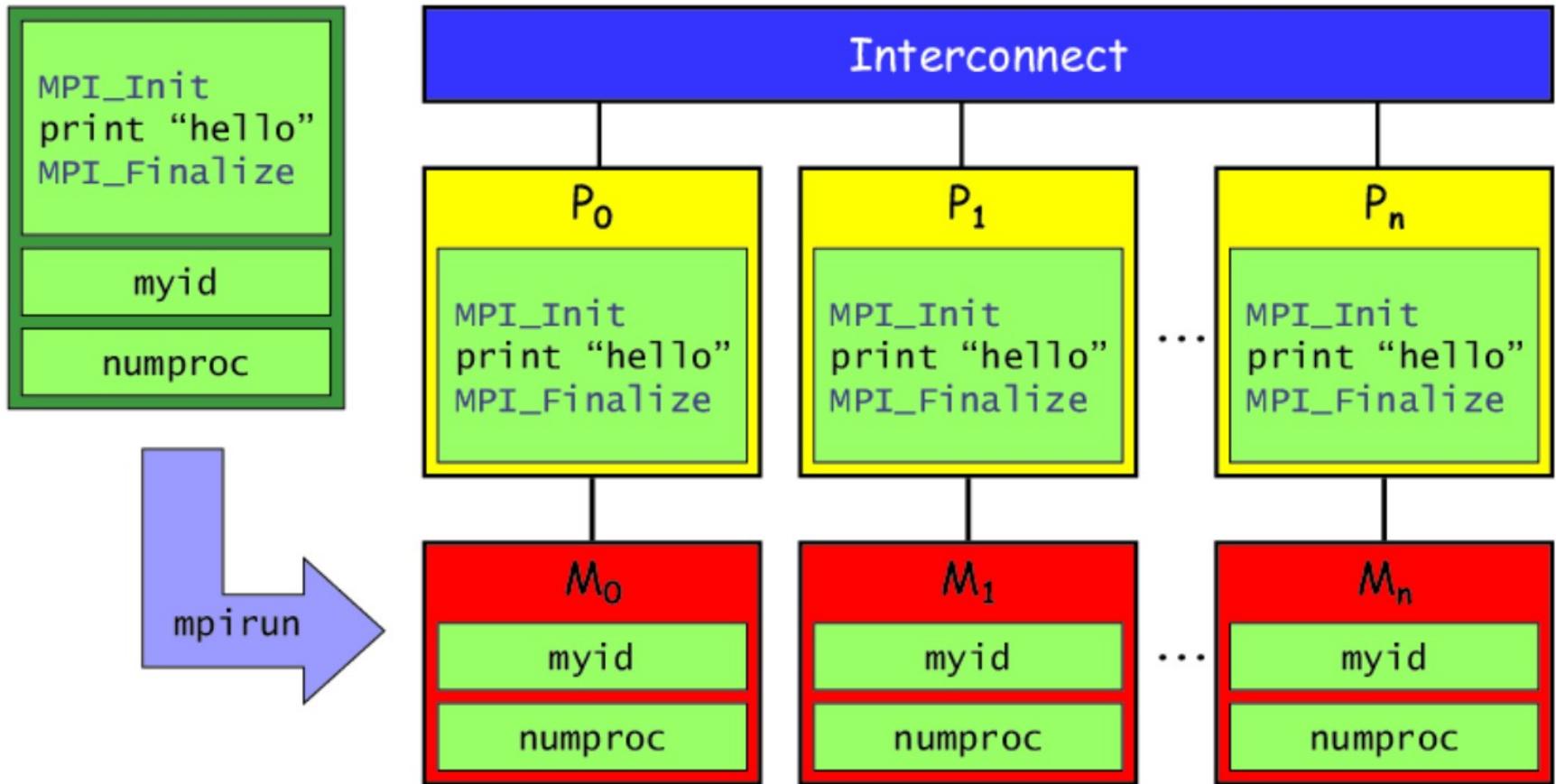
MPI Historie (2)

- **Version 2.1** (2008):
 - Zusammenfassung von MPI 1.2 und 2.0 in ein Dokument
- **Version 2.2** (2009)
- **Version 3.0** (September 2012):
 - Wesentliche Erweiterungen:
 - Nicht-blockierende kollektive Kommunikation
 - Keine C++ Bindings mehr
 - MPI Tool Information Interface
 - Neue Funktionen zur einseitigen Kommunikation
- **Version 3.1** (seit Juni 2015):
 - Berichtigungen und Klarstellungen, kleine Änderungen

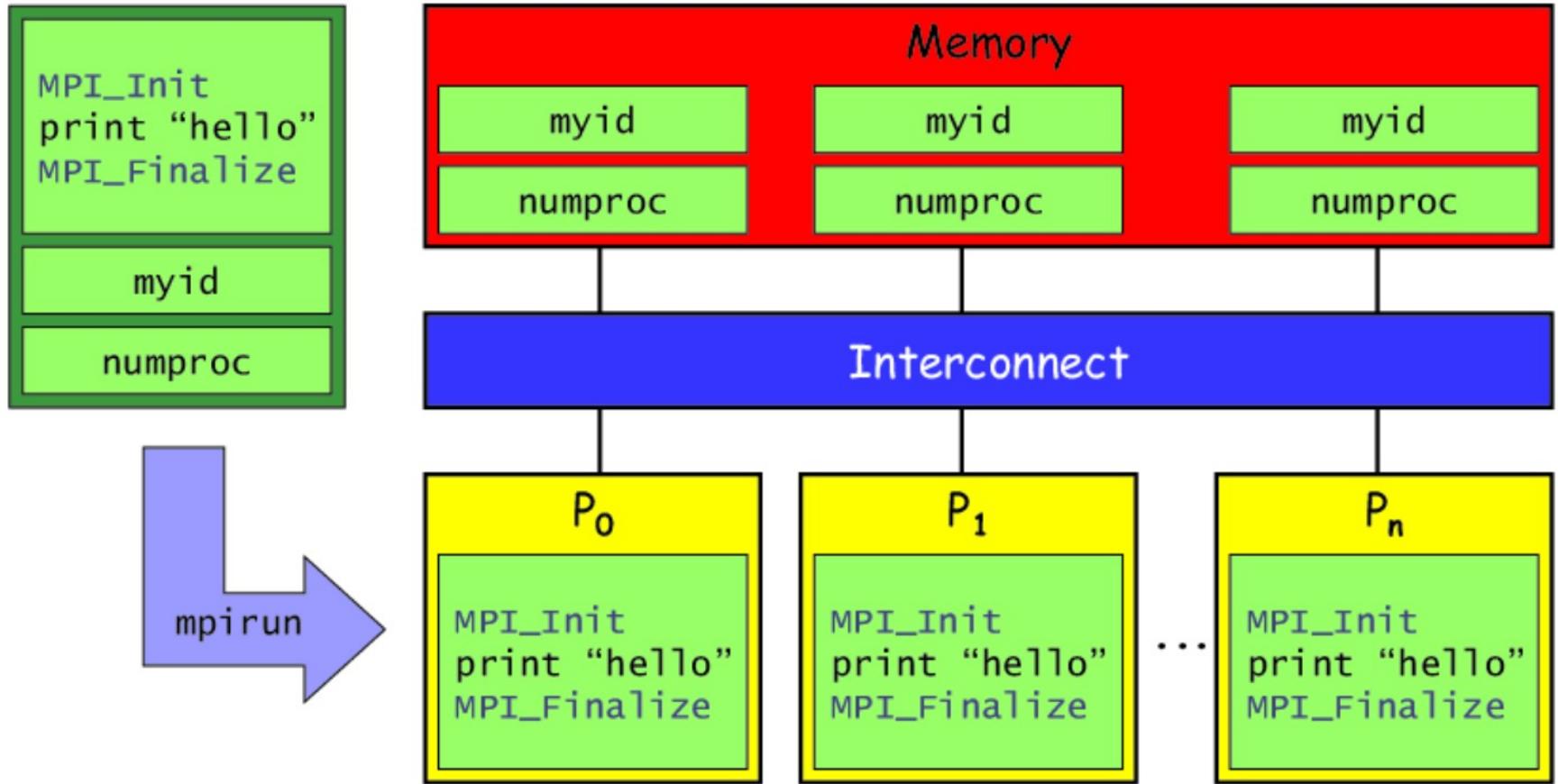
Programmiermodell

- MPI basiert auf dem Message-Passing-Programmiermodell
- jeder Prozess hat seinen eigenen **privaten Speicher**
 - der Speicher kann physikalisch verteilt oder gemeinsam sein
 - ein Prozess kann nicht auf die Daten eines anderen Prozesses zugreifen
- **jeder** Prozess führt **ein** Programm aus (i.a. SPMD)
- die Variablen eines Programms
 - haben den gleichen Namen
 - können sich aber an verschiedenen Stellen im Speicher befinden und unterschiedliche Werte haben
- Datenaustausch unter Prozessen geschieht explizit durch Verschicken / Empfangen von Nachrichten
- Synchronisation durch Nachrichtenaustausch
- auch für Rechner mit gemeinsamem Speicher geeignet

MPI auf Systemen mit verteiltem Speicher



MPI auf Systemen mit gemeinsamem Speicher



Nachrichten

- Nachrichten sind Datenpakete, die zwischen den Prozessen hin- und hergeschickt werden
- Das Message-Passing-System muss folgende Informationen verwalten:
 - ✓ Sender
 - ✓ Sendepuffer
 - ✓ Größe des Sendepuffers
 - ✓ Datentyp des Senders
 - ✓ Empfänger
 - ✓ Empfangspuffer
 - ✓ Größe des Empfangspuffers
 - ✓ Datentyp des Empfängers

Kommunikationsmodell

- virtuell **vollständige Vernetzung**:
 - jeder kann **direkt** mit jedem kommunizieren
- **Ränge** werden als Adressen verwendet
- Kommunikation ist **zuverlässig**
 - MPI geht davon aus, dass eine gesendete Nachricht immer korrekt empfangen wird (Vorsicht bei Programm- und Ressourcen-Fehlern)
 - MPI unterstützt keine Mechanismen zur Behandlung von Fehlern im Kommunikationssystem
 - wenn möglich gibt MPI einen Fehlercode zurück, der die erfolgreiche Beendigung einer Operation bestätigt
 - standardmäßig führt jedoch jeder Fehler zu einem sofortigen Abbruch des Programms
- Nachrichten müssen **explizit** empfangen werden
- zu **jeder Sendeoperation** gehört **eine Empfangsoperation!**

MPI Laufzeitumgebung

- MPI verwendet **vollständige Prozesse**:
 - ein MPI-Programm besteht aus unabhängigen Prozessen
 - jeder Prozess führt seinen eigenen Code im MIMD-Stil aus
 - die Programm-Codes der Prozesse müssen nicht identisch sein
 - die Prozesse kommunizieren über MPI-Sende- und Empfangsroutinen miteinander
- die Prozesse müssen mittels eines MPI Tools gestartet werden
 - JURECA: **srun**
- MPI muss vor der ersten Verwendung initialisiert werden
- **Achtung**: vor Initialisierung ist der Status des Programms undefiniert!
- nach Beenden ebenfalls

Header Files

C

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

Fortran

```
include 'mpif.h'      (Fortran77)  
use mpi                (Fortran90)  
use mpi_f08            (Fortran08)
```

Beinhaltet die Definition von

- Konstanten
- Funktionen
- Subroutinen

Funktionsformat

C

```
error = MPI_Function( parameter, ... );
```

Fortran

```
call MPI_Function( parameter, ..., ierror);
```

- MPI garantiert zuverlässige Kommunikation
- alle Fehler, die im Zusammenhang mit Kommunikation auftreten führen zu einem Abbruch des Programms

!

Die Prefixe **MPI_** und **PMPI_** sind reserviert für MPI-Konstanten und MPI-Funktionen, d. h. Variablen und Funktionen im Programm dürfen **nicht** mit **MPI_** oder **PMPI_** beginnen.

!

Der Parameter **ierror** ist optional in Fortran08 aber zwingend notwendig in Fortran77/90.

Initialisieren und Beenden von MPI

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

Fortran
`MPI_Init(ierror)`
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

- muss als erste MPI-Funktion aufgerufen werden
- erst danach können MPI-Funktionen aufgerufen werden
- einzige Ausnahme: `MPI_Initialized()`

C `int MPI_Finalize(void);`

Fortran
`MPI_Finalize(ierror)`
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

- muss als letzte MPI-Funktion aufgerufen werden
- danach können keine MPI-Funktionen mehr aufgerufen werden
- einzige Ausnahme: `MPI_Finalized()`

`MPI_Init` und `MPI_Finalize` dürfen nur jeweils einmal aufgerufen werden!

Inout:

`argc, argv:` Parameter aus main

MPI: erstes Beispiel

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

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

    MPI_Init(&argc, &argv);          /*initialize MPI*/
    printf(„Process Started \n“); /*each process */
    MPI_Finalize();                /*clean up */

    return 0;
}
```

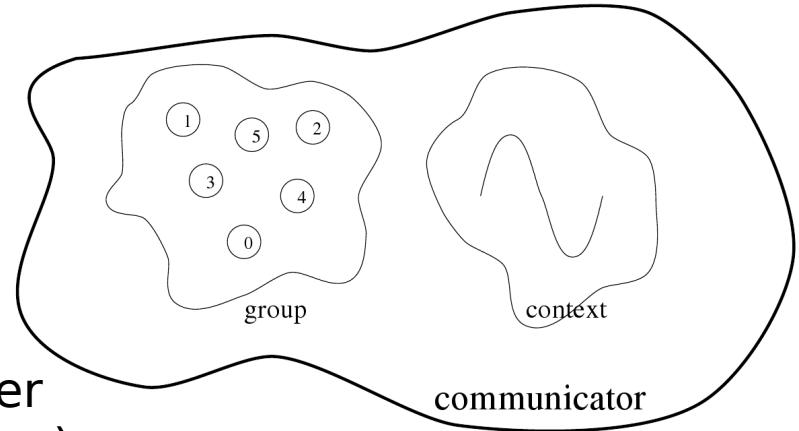
Adressierung in MPI

- zum Austausch von Nachrichten benötigt man Adressen, an die die Nachrichten gesendet werden
- die Kommunikation innerhalb einer Gruppe von Prozessen wird über einen **Kommunikator** abgewickelt
- jedem Prozess wird innerhalb eines Kommunikators ein eindeutiger Name (**Rang**) zugewiesen
- der Rang ist eine Zahl zwischen 0...N-1, wobei N die Anzahl der Prozesse im Kommunikator ist
- die Ränge der MPI-Prozesse werden als Adressen verwendet
- **Intra-Kommunikation:** Kommunikation zwischen Prozessen die der selben Gruppe angehören
- **Inter-Kommunikation:** Kommunikation zwischen Prozessen die sich in unterschiedlichen Gruppen befinden

betrachten hier nur **Intra**-Kommunikation, d.h. die kommunizierenden Prozesse müssen sich im **selben** (Intra-)Kommunikator befinden

Kommunikatoren

- jegliche Art von Kommunikation in MPI ist **nur** über einen Kommunikator möglich
- ein Kommunikator wird definiert durch:
 - **eine Gruppe von Prozessen:** Prozesse, die an der Kommunikation teilnehmen
 - **einen Kontext:** Umgebung, in der kommuniziert wird (Radio Frequenz)
- jeder Prozess hat einen eindeutigen Rang innerhalb des Kommunikators
- ein Prozess kann in verschiedenen Kommunikatoren verschiedene Ränge haben
- vordefinierte Kommunikatoren:
 - `MPI_COMM_WORLD` → enthält alle MPI-Prozesse
 - `MPI_COMM_SELF` → enthält nur den lokalen Prozess
 - `MPI_COMM_NULL` → Wert für ungültigen Kommunikator



Handles

- MPI verwaltet intern eigene Datenstrukturen, zur Speicherung von:
 - gepufferten Nachrichten
 - Objekten wie Gruppen, Kommunikatoren, Datentypen, Topologien, usw.
- diese Datenstrukturen können nur über Handles referenziert werden
- Manipulation ist nur über MPI-Funktionen möglich
- die Handles können sowohl als Parameter von MPI-Funktionen als auch in Zuweisungen und Vergleichen verwendet werden
- Typ eines Handles hängt von der Programmiersprache ab:
 - INTEGER in Fortran
 - meist typedefs oder Pointer in C
- **Es sollten immer die Handles benutzt werden, um auf MPI-Interne Objekte zuzugreifen!**

Zugriff auf Kommunikator-Informationen

C `int MPI_Comm_size(MPI_Comm comm, int* size);`

Fortran `MPI_Comm_size(comm, size, ierror)`

```
TYPE(MPI_COMM), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die Anzahl der Prozesse, die sich im Kommunikator befinden

C `int MPI_Comm_rank(MPI_Comm comm, int* rank);`

Fortran `MPI_Comm_rank(comm, rank, ierror)`

```
TYPE(MPI_COMM), INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: rank
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- bestimmt den Rang des rufenden Prozesses im Kommunikator

Input:

`comm:` betrachteter Kommunikator

Output:

`size:` Anzahl der Prozesse im Kommunikator `comm`

`rank:` Rang des Prozesses im Kommunikator `comm`

Kommunikationsarten

- **Punkt-zu-Punkt:**

EIN Prozess sendet an **EINEN** Prozess

- Blockierend/Nicht-blockierend

- **Kollektive Kommunikation:**

ALLE Prozesse innerhalb des Kommunikators sind an der Kommunikation beteiligt

- Barrieren: Synchronisation von Prozessen
- Broadcast: Ein Prozess sendet an viele
- Reduktionsoperationen: Kombination von Daten
- All-to-All: JEDER Prozess sendet an JEDEN
- Seit MPI 3.0 auch nicht-blockierend!

- **Einseitige Kommunikation** (ab MPI2):

EIN Prozess greift **ohne Mithilfe** des Kommunikationspartners auf dessen Daten zu

Übung

Übung 2:

Einführung in MPI



Einführung in die Parallelprogrammierung

MPI Teil 2: Blockierende Punkt-zu-Punkt Kommunikation

Annika Hagemeier

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich



Inhalt

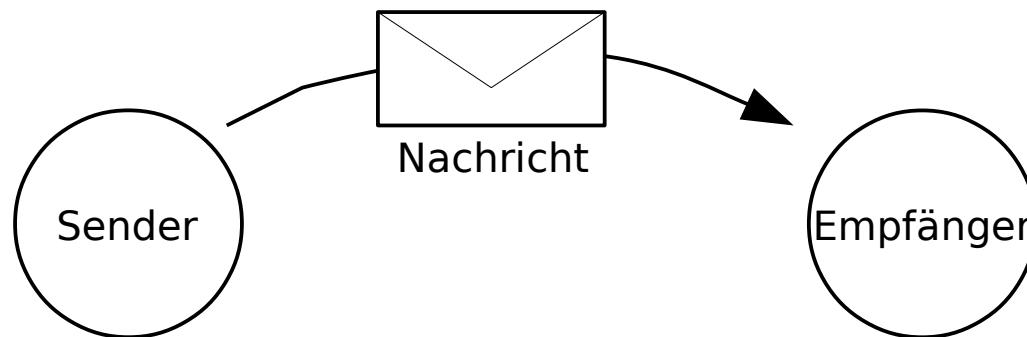
- Grundlagen der Punkt-zu-Punkt Kommunikation
- Senden und Empfangen von Nachrichten
- Daten einer Nachricht (MPI-Datentypen)
- Umschlag einer Nachricht (Status-Objekt)
- Kommunikationsmodi:
 - Standard Send (`MPI_Send`)
 - Synchrone Senden (`MPI_Ssend`)
 - Gepuffertes Senden (`MPI_Bsend`)
 - Ready Send (`MPI_Rsend`)
- Zeitmessung
- Visualisierung von Messergebnissen

Blockierend vs. nicht-blockierend

- alle Kommunikationsmodi der Punkt-zu-Punkt-Kommunikation existieren sowohl in blockierender als auch in nicht-blockierender Form
- bezieht sich auf die Fertigstellung einer Operation
- **Blockierend:**
 - Rückkehr aus Sende- oder Empfangsroutine erst wenn die Operation beendet ist (**lokal !**)
- **Nicht-Blockierend:**
 - sofortige Rückkehr aus Sende- oder Empfangsroutine und Ausführung der nächsten Statements (**lokal !**)
 - wird hauptsächlich zur Verbesserung der Performance durch Überlappung von Berechnung und Kommunikation verwendet
- Fertigstellung einer Operation ist immer **lokal** anzusehen:
 - Fertigstellung einer Sendeoperation: der Sendepuffer kann gefahrlos (ohne Informationsverlust) wiederverwendet werden

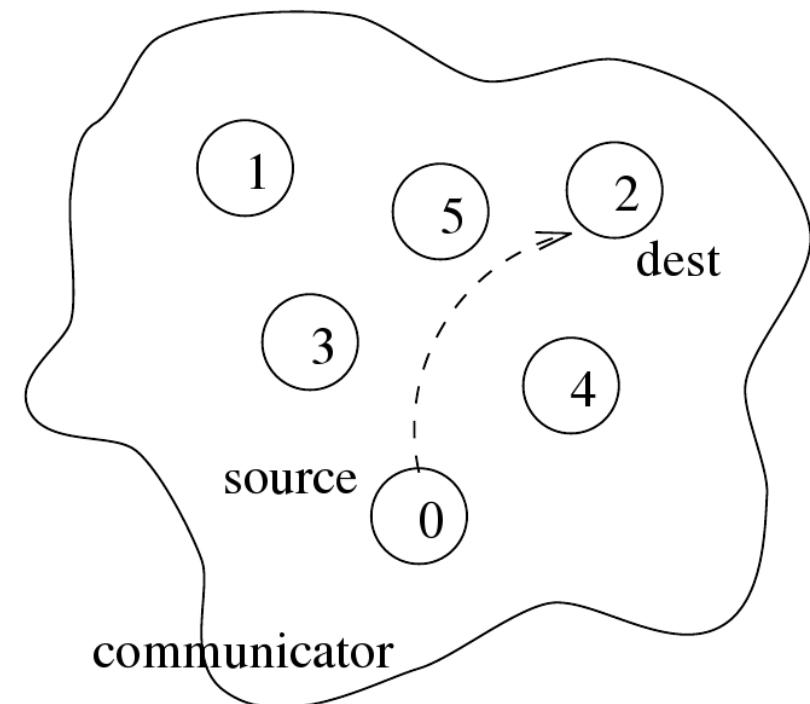
Punkt-zu-Punkt Kommunikation (1)

- Kommunikation zwischen genau **zwei** Prozessen
 - jeder Prozess kann dabei auch eine Nachricht an sich selbst schicken
 - **ein** Prozess sendet eine Nachricht an **einen** anderen Prozess
 - zu jeder Nachricht, die durch Punkt-zu-Punkt Kommunikation versendet wird, gehört:
 - eine Sendeoperation
 - die dazu passende Empfangsoperation



Punkt-zu-Punkt Kommunikation (2)

- Sende- und Empfangsprozess werden durch ihren Rang im Kommunikator (z.B. MPI_COMM_WORLD) spezifiziert
- die kommunizierenden Prozesse müssen sich im selben Kommunikator befinden
- beide Prozessoren nehmen **aktiv** an der Kommunikation teil:
 - der **Sendeprozess** sendet eine Nachricht an einen **Empfängerprozess** durch Aufruf einer MPI-Senderoutine
 - der **Empfängerprozess** muss die Nachricht **explizit** empfangen durch Aufruf einer MPI-Empfangsroutine



Nachrichten senden

C

```
int MPI_Send( const void* buf, int count, MPI_Datatype  
              datatype, int dest, int stag, MPI_Comm comm);
```

Fortran

```
MPI_Send( buf, count, datatype, dest, stag, comm, ierror )  
  
TYPE(*), DIMENSION(..), INTENT(IN) :: buf  
INTEGER, INTENT(IN) :: count, dest, stag  
TYPE(MPI_Datatype), INTENT(IN) :: datatype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- sendet eine Nachricht mit der Startadresse `buf` mit `count` Elementen vom Typ `datatype` zum Prozess mit dem Rang `dest` im Kommunikator `comm`

Input:

<code>buf:</code>	Adresse des Sendepuffers
<code>count:</code>	Anzahl der zu sendenden Elemente
<code>datatype:</code>	Typ der zu versendenden Daten
<code>dest:</code>	Rang des Empfängers in <code>comm</code>
<code>stag:</code>	Etikett zur Unterscheidung der Nachrichten
<code>comm:</code>	Kommunikator in dem kommuniziert wird

Nachrichten empfangen

C

```
int MPI_Recv( void* buf, int count, MPI_Datatype datatype,
              int src, int rtag, MPI_Comm comm, MPI_Status* status);
```

Fortran

```
MPI_Recv( buf, count, datatype, src, rtag, comm, status, ierror )
          TYPE(*), DIMENSION(..), INTENT(IN) :: buf
          INTEGER, INTENT(IN) :: count, src, rtag
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
          TYPE(MPI_Comm), INTENT(IN) :: comm
          TYPE(MPI_Status) :: status
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- sendet eine Nachricht mit der Startadresse `buf` mit `count` Elementen vom Typ `datatype` zum Prozess mit dem Rang `dest` im Kommunikator `comm`

Input:

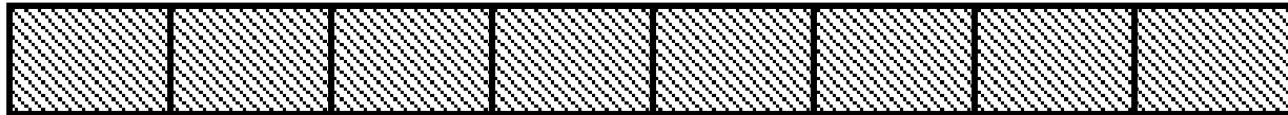
`count`: Anzahl der zu empfangenden Elemente (muss größer oder gleich der Länge der Nachricht sein)
`datatype`: Typ der zu empfangenden Daten
`src`: Rang des Senders in `comm`
`rtag`: Etikett zur Unterscheidung der Nachrichten
`comm`: Kommunikator in dem kommuniziert wird

Output:

`buf`: Adresse des Empfangpuffers
`status`: Information über die empfangene Nachricht (oder `MPI_STATUS_IGNORE`)

Daten einer Nachricht

- eine Nachricht enthält eine Anzahl von Elementen eines MPI-Datentyps



- eine Nachricht wird durch 3 Komponenten spezifiziert:
 - Position im Speicher (`buf`)
 - Anzahl der Elemente (`count`)
 - MPI-Datentyp (`datatype`)
- der Sendepuffer besteht aus `count` hintereinanderliegenden (im Speicher) Elementen vom Typ `datatype` mit der Startadresse `buf`
- die Länge einer Nachricht wird in Elementen und nicht in Bytes angegeben (portabler)
- MPI-Datentypen werden über ein Handle adressiert und enthalten die Beschreibung des Speicher-Layouts

Basistypen in MPI (Auszug)

C/C++

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

Basistypen in MPI

Fortran

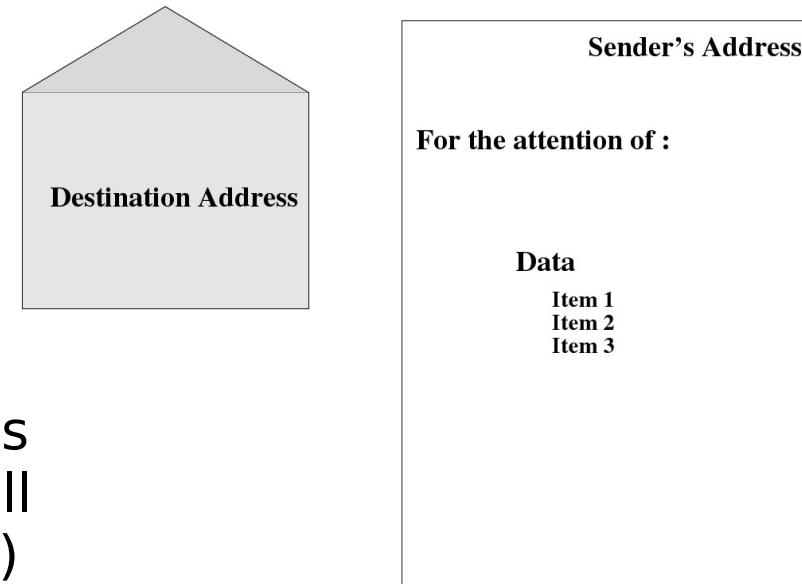
MPI Datentyp	Fortran Datentyp
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Behandlung von Datentypen

- der Nachrichtentransfer besteht aus 3 Phasen:
 1. Daten aus Sendepuffer kopieren und Nachricht erstellen
 2. Nachricht zum Empfänger transportieren
 3. Daten aus Nachricht lesen und in Empfangspuffer auspacken
- dabei müssen die Datentypen zueinander passen
 - Daten im Sendepuffer zum MPI_DATATYPE in Send
 - MPI_DATATYPE in Send zu MPI_DATATYPE in Recv
 - MPI_DATATYPE in Recv zu Daten im Empfangspuffer
- MPI kann das nicht prüfen!
-> **Verantwortung des Programmierers**

Umschlag einer Nachricht (Status-Objekt)

- zusätzlich zu den Daten tragen Nachrichten Informationen mit sich, die dazu verwendet werden Nachrichten zu unterscheiden
- diese Informationen werden Nachrichtenumschlag (engl. communication envelop) genannt
- zum Nachrichtenumschlag gehören:
 - Sender
 - Empfänger
 - Tag
 - Kommunikator
- der Empfänger kann die Informationen des Nachrichtenumschlags über das Status-Objekt abfragen (sinnvoll bei Verwendung von Wildcards)



Wildcards

- der **Empfänger** kann für `source` und `tag` Wildcards verwenden
 - **MPI_ANY_SOURCE**: kann verwendet werden, um eine Nachricht von einem beliebigen Sender zu empfangen
 - **MPI_ANY_TAG**: kann verwendet werden, um eine Nachricht mit einem beliebigen Tag zu empfangen
- der tatsächliche Sender und das tatsächliche Tag werden über das Status-Objekt zurück gegeben

Das Status-Objekt

- enthält die Informationen des Nachrichtenumschlags
- das Status-Objekt ist eine Struktur (struct) mit den folgenden Elementen:
 - **MPI_SOURCE** (Rang des Sendeprozesses)
 - **MPI_TAG** (Tag der Nachricht)
 - **MPI_ERROR** (Fehlerstatus der Sendeoperation)

```
int source, tag, error;  
MPI_Status status;  
  
...  
MPI_Recv(...,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);  
...  
source = status.MPI_SOURCE;  
tag    = status.MPI_TAG;  
error  = status.MPI_ERROR;
```

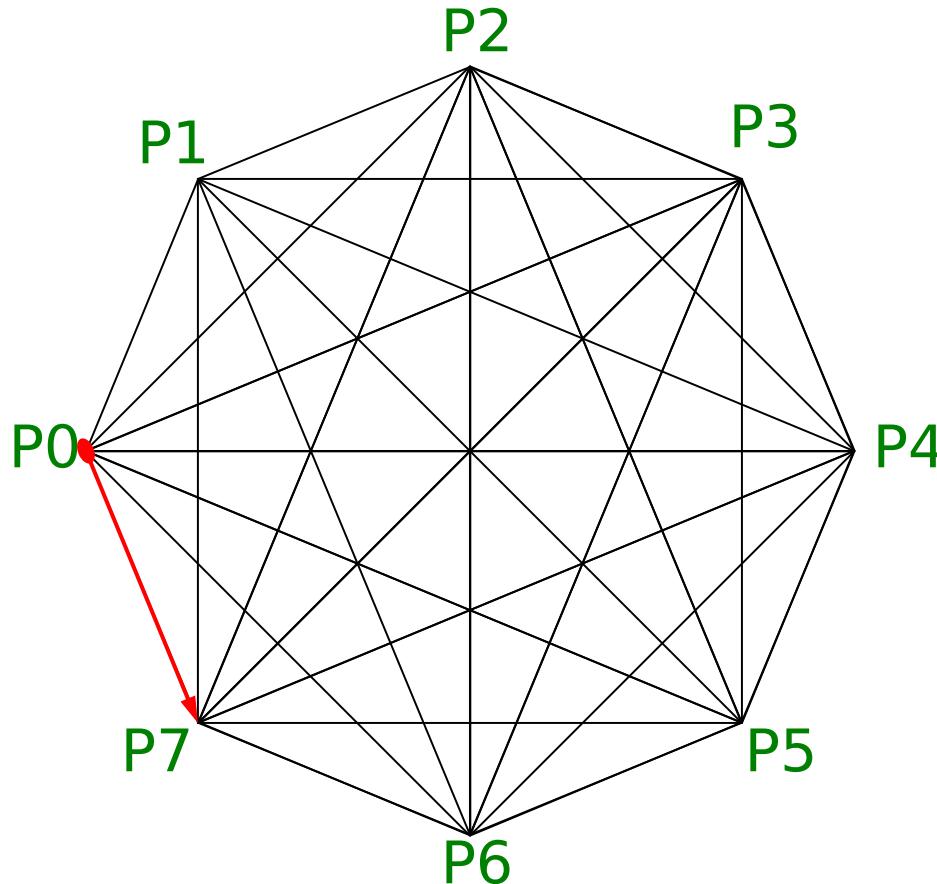
Das Status-Objekt

- enthält die Informationen des Nachrichtenumschlags
- das Status-Objekt ist ein Integerfeld der Größe MPI_STATUS_SIZE, wobei die folgenden Konstanten die Indizes zu den entsprechenden Daten sind:
 - **MPI_SOURCE** (Rang des Sendeprozesses)
 - **MPI_TAG** (Tag der Nachricht)
 - **MPI_ERROR** (Fehlerstatus der Sendeoperation)

```
integer source, tag, stat_error;
integer, dimension(MPI_STATUS_SIZE) :: status
...
call MPI_RECV(...., MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
              status, error)
...
source      = status(MPI_SOURCE)
tag         = status(MPI_TAG)
stat_error = status(MPI_ERROR)
```

Beispiel: Send/Recv

- Sende eine ganze Zahl von Prozess 0 zu Prozess $\text{num_procs}-1$



Send/Recv Programm

```
int msg;                      /* Inhalt */
int tag = 42;                  /* beliebig */
MPI_Status status;             /* Für MPI_Recv */

...                           /* Init, Comm_Size, Comm_Rank */

if (my_rank == 0) {
    msg = 4711;
    MPI_Send(&msg, 1, MPI_INT, num_procs-1, tag, MPI_COMM_WORLD);
}

if (my_rank == numprocs-1) {
    MPI_Recv(&msg, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    printf("Process %d received number %d\n", my_rank, msg);
}

...
```

Nachrichten prüfen

C

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

Fortran

```
MPI_Probe( source, tag, comm, status, ierror )  
  
INTEGER, INTENT(IN) :: source, tag  
TYPE(MPI_Comm), INTENT(IN) :: comm  
TYPE(MPI_Status) :: status  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- ermöglicht es, den Status einer anstehenden Nachricht abzufragen ohne diese explizit zu empfangen
- über die Status-Variable kann dann z. B. die Länge der anstehenden Nachricht ermittelt werden

Input:

source: Rang des Senders (oder `MPI_ANY_SOURCE`)
tag: Tag der Nachricht (oder `MPI_ANY_TAG`)
comm: Kommunikator in dem kommuniziert wird

Output:

status: Information über die empfangene Nachricht

Länge einer Nachricht abfragen

C

```
int MPI_Get_count( const MPI_Status* status, MPI_Datatype  
datatype, int* count);
```

Fortran

```
MPI_Get_count( status, datatype, count, ierror )  
  TYPE(MPI_Status), INTENT(IN) :: status  
  TYPE(MPI_Datatype), INTENT(IN) :: datatype  
  INTEGER, INTENT(OUT) :: count  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- ermittelt die Länge der tatsächlich empfangenen Nachricht
- sinnvoll, da das Argument `count` in `MPI_Recv` lediglich die maximale Länge einer Nachricht angibt
- `MPI_Get_count` ist auch hilfreich, um die Länge einer Nachricht vor dem eigentlichen Empfangen der Nachricht zu ermitteln (zusammen mit `MPI_Probe`)

Input:

`status`: Status-Objekt der Empfangsoperation
`datatype`: Datentyp der Nachricht

Output:

`count`: Anzahl der Elemente der Nachricht

Beispiel: Länge einer unbekannten Nachricht

```
int          myid, msglen, *msg;
MPI_Status   status;
MPI_Comm     comm = MPI_COMM_WORLD;
...
/* Init, Comm_Size, Comm_Rank */

if (myid == 0) {
    ...
    MPI_Send(msg, 10, MPI_INT, 1, 4711, comm);
}

else if (myid == 1) {
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
    MPI_Get_count(&status, MPI_INT, &msglen);

    msg = malloc(msglen * sizeof(int));

    MPI_Recv(msg, msglen, MPI_INT, status.MPI_SOURCE,
             status.MPI_TAG, comm, &status);
}
```

Punkt-zu-Punkt Kommunikation: Voraussetzungen

- kommunizierende Prozesse müssen sich im selben Kommunikator befinden
- der Sender muss einen gültigen Empfänger spezifizieren
- der Empfänger muss einen gültigen Sender spezifizieren
 - MPI_ANY_SOURCE ist ebenfalls gültig
- die Tags müssen übereinstimmen
 - MPI_ANY_TAG ist ebenfalls erlaubt
- die Datentypen der Nachrichten müssen übereinstimmen
- der Empfangspuffer muss groß genug sein, um die gesamte Nachricht zu speichern
 - falls das nicht der Fall ist, ist das Verhalten undefined
 - kann auch größer sein als die empfangenen Daten
- Eine Nachricht wird durch Sender, Empfänger, Tag und Kommunikator eindeutig zugeordnet. Dabei ist der Sender implizit gegeben.

Kommunikationsmodi (blockierend)

- **Sendemodi:**

- Synchrones Senden ⇒ MPI_Ssend
- Gepuffertes Senden ⇒ MPI_Bsend
- Standard Send ⇒ MPI_Send
- Ready Send ⇒ MPI_Rsend

- **Empfangsmodi:**

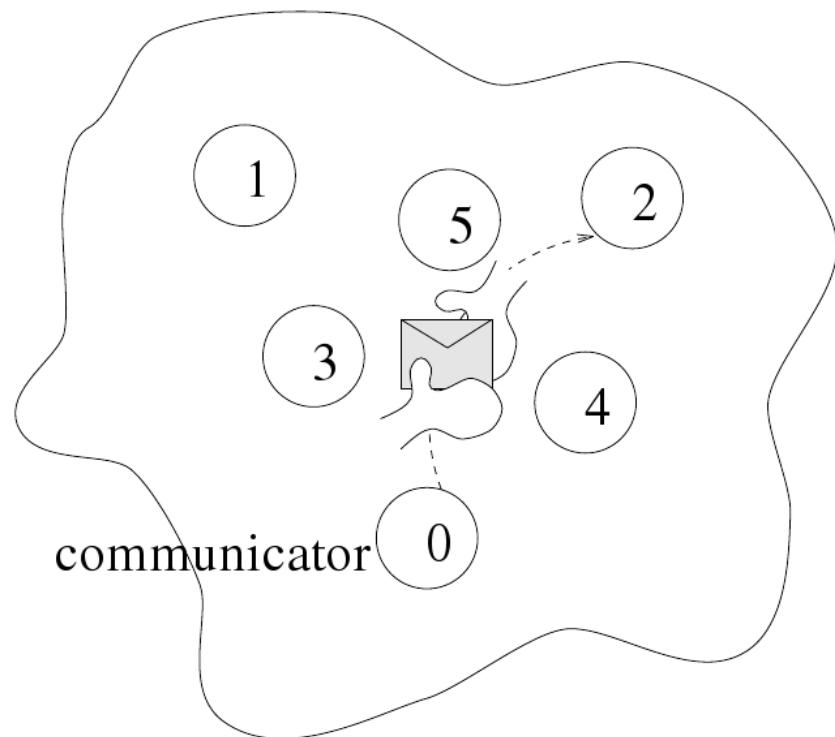
- Receive ⇒ MPI_Recv
- Probe ⇒ MPI_Probe

Standard Send (MPI_Send)

- ist grundsätzlich blockierend:
 - MPI_Send ist erst fertig, wenn die Nachricht gesendet wurde ⇒ kann heißen, dass die Nachricht bereits beim Empfänger angekommen ist, muss aber nicht !
- das Verhalten von MPI_Send hängt von der verwendeten Implementierung ab (nicht im Standard definiert):
 - MPI_Send kann sich wie ein synchrones oder bepuffertes Senden verhalten
 - kleinere Datenmengen: MPI_Send verhält sich wie MPI_Bsend
 - größer Datenmengen: MPI_Send verhält sich wie MPI_Ssend
- **Gefahr der Verklemmung !**

Synchrones Senden (MPI_Ssend)

- sollte verwendet werden, wenn der Sendeprozess wissen muss, dass die Nachricht beim Empfänger angekommen ist
- der Empfängerprozess schickt eine Bestätigung an den Sender, dass die Nachricht empfangen wurde
- erst wenn der Sendeprozess diese Bestätigung erhalten hat, fährt er mit seiner Berechnung fort
- der Sendeprozess ist solange idle, bis der Empfängerprozess bei der zu seiner Sendeoperation gehörigen Empfangsoperation angelangt ist



Synchron vs. asynchron

- Zusammenhang zwischen Sender und Empfänger
- **Synchron:**
 - Senden startet erst, wenn der Empfänger mit dem synchronen Empfangen starten kann
 - Sende-Operation ist erst beendet, wenn der Sender vom Empfänger eine Empfangsbestätigung erhalten hat (Daten sind vom Empfangspuffer in den Speicher des Empfängers kopiert worden)
- **Asynchron:**
 - Sendeoperation kann beendet sein, obwohl der Empfänger die Nachricht noch nicht empfangen hat
 - es ist nur sicher, dass die Nachricht gesendet wurde

Gepuffertes Senden (`MPI_Bsend`)

- Aufruf von `MPI_Bsend` kehrt sofort zurück, sobald die Nachricht in einen Puffer zur späteren Übertragung kopiert wurde
- **Vorteil:**
 - Sender und Empfänger werden nicht synchronisiert schneller
 - bei Überlastung des Netzwerkes tritt ein Fehler auf
- **Nachteil:**
 - der Benutzer kann nicht von vordefinierten Puffern ausgehen
 - der Benutzer muss explizit Pufferspeicher anlegen

Pufferverwaltung (1)

C

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

Fortran

```
MPI_Buffer_attach( buffer, size, ierror )
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer  
INTEGER, INTENT(IN) :: size  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- registriert einen Sendepuffer für die gepufferte Kommunikation
- der Puffer muss eindeutig sein und darf nicht angefasst werden
- der Puffer muss für alle gleichzeitigen Bsends ausreichen

Input:

buffer: Anfangsadresse des anzulegenden Puffers
size: Größe (in Bytes) des anzulegenden Puffers

Pufferverwaltung (2)

C `int MPI_Buffer_detach(void* buffer_addr, int* size);`

Fortran `MPI_Buffer_detach(buffer_addr, size, ierror)`

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
TYPE(C_PTR), INTENT(OUT) :: buffer_addr
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- entfernt den Puffer
- der Puffer kann danach wieder verwendet werden oder ein anderer kann registriert werden

Output:

`buffer_addr`: Anfangsadresse des entfernten Puffers

`buf_size`: Größe (in Bytes) des entfernten Puffers

Beispiel: Pufferverwaltung

```
#define BUFSIZE 10000

int main(int argc, char* argv[]) {
    int bsize;
    char *bufptr;
    ...
    MPI_Buffer_attach(malloc(BUFSIZE), BUFSIZE);
    ...
    MPI_Buffer_detach(&bufptr, &bsize);
}
```

Anmerkung:

- Die Verwendung der generischen Zeiger in `MPI_Buffer_attach` und `MPI_Buffer_detach` ist unterschiedlich!
- `MPI_Buffer_detach` erwartet einen Zeiger auf einen Zeiger

Grund:

- Typumwandlungen in generische Zeiger (`void *`) geschehen automatisch
- Typumwandlungen in Zeiger vom Typ `void **` müssen explizit durchgeführt werden

Ready Send (`MPI_Rsend`)

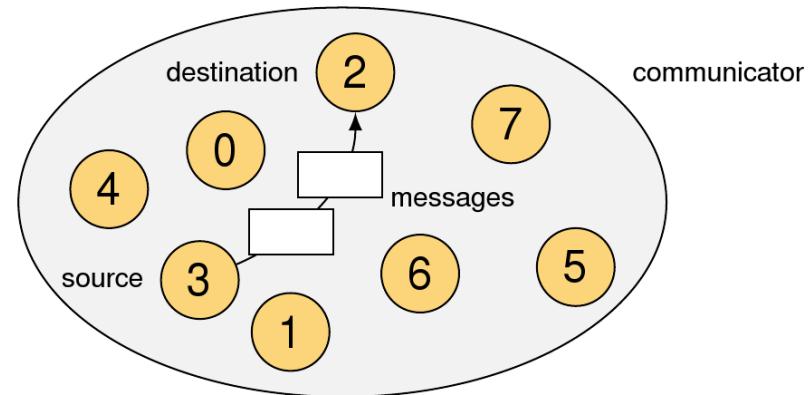
- kehrt wie `MPI_Bsend` sofort zurück
- ist ein passendes Recv-Kommando gepostet, wird die Kommunikation erfolgreich durchgeführt
- **ABER:** gibt es kein passendes Recv-Kommando, ist der Ausgang undefiniert
- **Grund:** der Sendeprozess “wirft” die Nachricht einfach raus und hofft, dass der Empfängerprozess darauf wartet, die Nachricht zu “fangen”
- `MPI_Rsend` ist sehr schwer zu debuggen
- **grundsätzlich sollte die Verwendung von `MPI_Rsend` vermieden werden !**
- macht nur Sinn in Performance-kritischen Anwendungen

Fertigstellung einer Operation

Modus	Fertigstellung	Bemerkung
Synchrones Senden	Fertigstellung erst wenn das Empfangen abgeschlossen ist	
Gepuffertes Senden	wird immer beendet (außer bei Auftreten eines Fehlers) unabhängig davon, ob das Empfangen begonnen hat	der Benutzer muss einen Puffer mit MPI_Buffer_attach zur Verfügung stellen
Standard Send	ist entweder gepuffert oder synchron	ein interner Puffer wird benutzt
Ready Send	wird immer beendet (außer bei Auftreten eines Fehlers) unabhängig davon, ob das Empfangen begonnen hat	wird nur ausgeführt, wenn das passende Empfangskommando bereits gepostet wurde
Empfangen	Fertigstellung wenn eine Nachricht empfangen wurde	gleiche Routine für alle Kommunikationsmodi

Bemerkungen (1)

- **Reihenfolge der Nachrichten bleibt erhalten:**
 - Nachrichten vom selben Sender, die an den selben Empfänger geschickt werden, werden in der Reihenfolge empfangen, in der sie gesendet wurden
 - Nachrichten mit dem selben Umschlag überholen sich nicht
- **Programm-Fortschritt:**
 - ein passendes Send/Recv-Paar kann nicht dauerhaft unerledigt bleiben
 - MPI garantiert, dass wenn es ein passendes Send/Recv-Paar gibt, dieses auch ausgeführt wird
 - MPI garantiert ein Weiterkommen im Programmablauf



Bemerkungen (2)

- **Fairness:**

- MPI garantiert keine Fairness in der Kommunikation
- sendet ein Prozess wiederholt Nachrichten und wollen mehrere Prozesse diese Nachrichten empfangen, ist nicht garantiert, dass auch alle Prozesse zum Zuge kommen
- es ist die Pflicht des Programmierers das “verhungern” von Prozessen zu vermeiden

- **Limitierung der Ressourcen:**

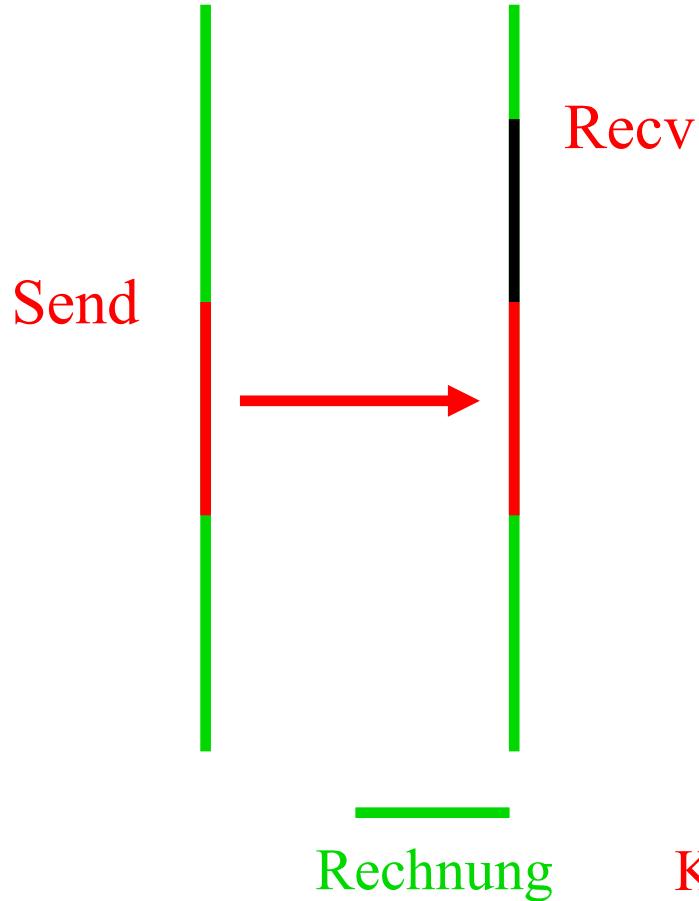
- jede unerledigte Kommunikationsoperation verbraucht Systemressourcen (z.B. Pufferspeicher)
- sind nicht genug Ressourcen vorhanden um eine MPI-Operation auszuführen, können Fehler auftreten

Nachteile

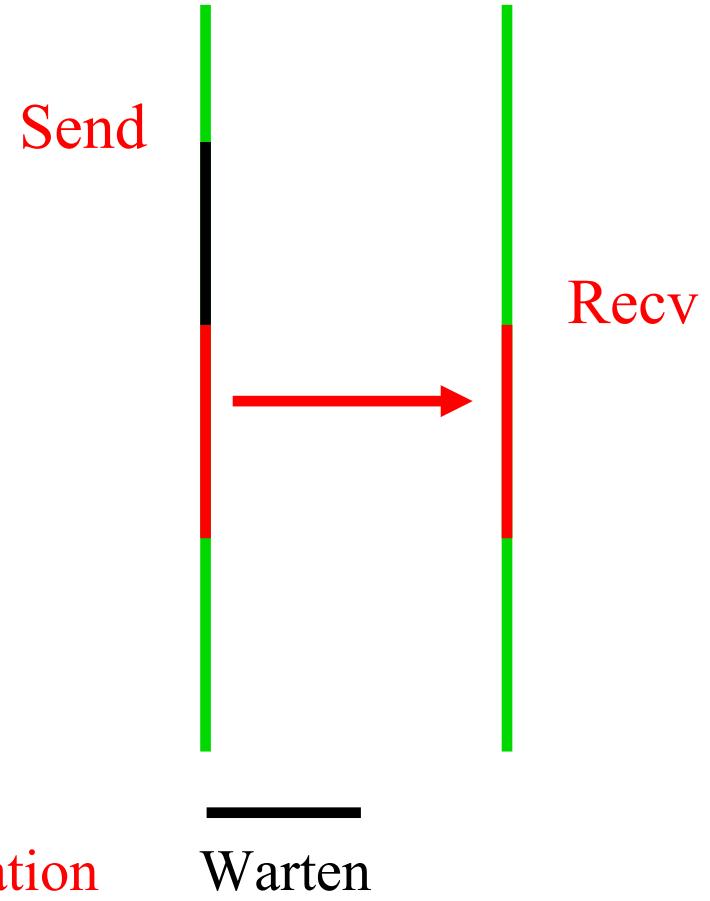
- Deadlocks mit Standard Send:
 - wird für das Standard Send der synchrone Modus gewählt, kann es zu Deadlocks kommen
 - es sollten niemals alle Prozesse zur gleichen Zeit mit **blockierenden** Aufrufen senden oder empfangen
- Performance-Verluste beim synchronen Senden:
 - **Late Receiver:**
Der Sender wartet, dass der Empfänger die Empfangsoperation startet.
 - **Late Sender:**
Der Empfänger blockiert seinen Aufruf, bis der Sender zu senden beginnt.

Mögliche Abläufe

Late Sender



Late Receiver



Zeitmessung: MPI_Wtime (1)

- **MPI_Wtime():**
 - gibt an, wieviel Zeit (in Sekunden) seit einem (willkürlich festgelegten) Zeitpunkt vergangen ist
 - Die Zeiten sind lokal im rufenden Prozess.
 - Der "Zeitpunkt in der Vergangenheit" bleibt, solange der Prozess lebt, unverändert.
- **Zeitmessung bei parallelen Programmen:**
 - Die Zeit, die das parallele Programm benötigt, ist das Maximum der Zeiten, die jeder einzelne Prozess benötigt.
 - Für die Zeitmessung eines MPI-parallelen Programms muss man daher das Maximum der auf allen Prozessoren gemessenen Zeit bestimmen.
 - Damit nicht unterschiedliche Vorlaufzeiten zu einer Verfälschung des Ergebnisses führen, muss außerdem vor der ersten Zeitmessung ein Barrier (`MPI_Barrier()`) aufgerufen werden

Zeitmessung: MPI_Wtime (2)

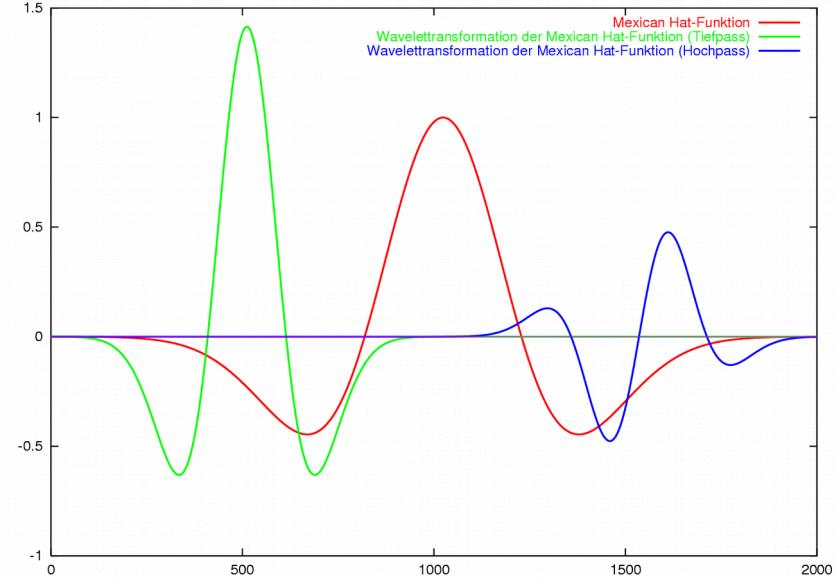
```
#include <mpi.h>

...
double st, ut, time;
...
MPI_Barrier(MPI_COMM_WORLD);
st = MPI_Wtime();
...
work ...
ut = MPI_Wtime() - st;
MPI_Reduce(&ut, &time, 1, MPI_DOUBLE, MPI_MAX, 0,
           MPI_COMM_WORLD);
if (my_rank == 0)
    printf("time used: %14.8f seconds\n", time);
```

Visualisierung von Messergebnissen (1)

- Gnuplot: Generierung von Liniendiagrammen
- Eingabe z.B.: Tabelle in ASCII-Datei
- Aufruf (interaktiv): gnuplot
- Aufruf (batch): gnuplot <cmdfile>
- Kommandos:
 - plot <file> <opt>
 - help
 - set <opt> <val>
 - ...

```
64 5.59634645e-05 2.89807795e-05 1.50456908e-05  
128 2.05009943e-04 1.15995877e-04 2.70577730e-05  
256 8.05962714e-04 4.03967744e-04 6.30193390e-05  
512 3.16096324e-03 1.58998877e-03 1.10046938e-04  
1024 1.25179816e-02 6.27897843e-03 2.26058415e-04  
...
```



Weitere Informationen zu Gnuplot:

- <http://www.gnuplot.info/>

Visualisierung von Messergebnissen (2)

- Batch-Datei mit Kommandos: cmdfile

```
# (PostScript output)
#set output "kompl.eps"
#set terminal postscript eps color solid 12
set xlabel "n"
set ylabel "Time (s)"
set xrange [2:1024*1024]
#set yrange [0:16]
set title "Zeitmessung"
#set logscale x
#set logscale y
plot      (x**4)  with lines lw 1, \
            "messung.dat" u 1:2 ti "algol" with linesp lw 1
pause -1 "Hit return to continue"
```

Übung

Übung 3:

Blockierende Punkt-zu-Punkt Kommunikation



Einführung in die Parallelprogrammierung

MPI Teil 3: Nicht-blockierende Punkt-zu-Punkt Kommunikation

Annika Hagemeier

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich



Inhalt

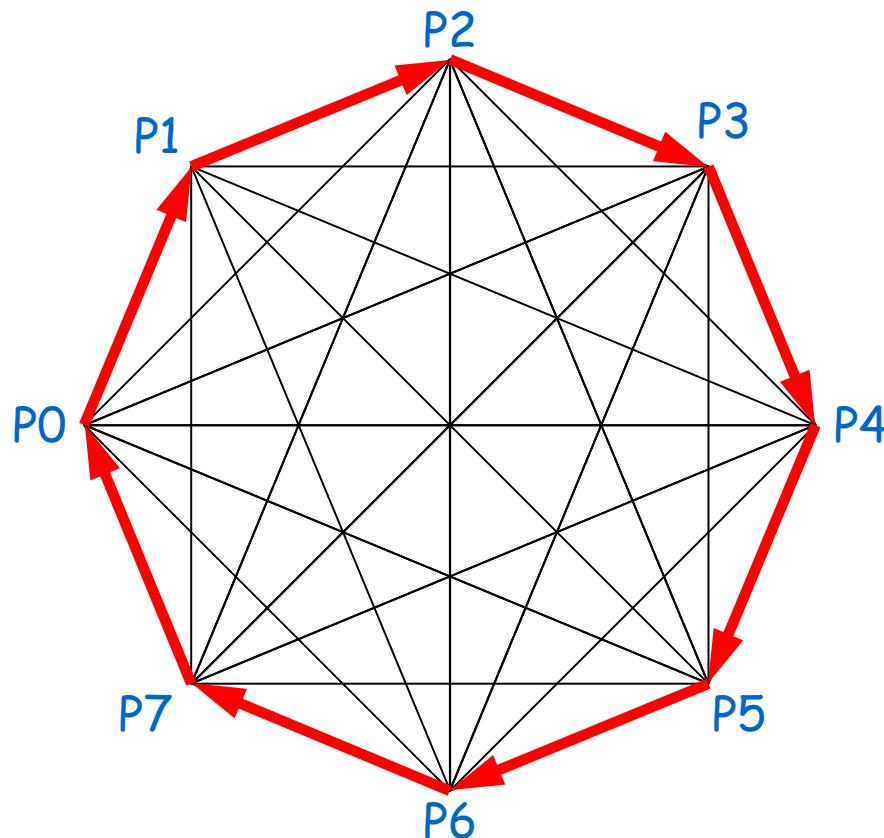
- Beispiel: Senden im Ring
- Phasen der nicht-blockierenden Kommunikation
- Das Request-Objekt
- Nicht-blockierende Kommunikation:
 - Nicht-blockierendes Senden
 - Nicht-blockierendes Empfangen
- Abschluss der Kommunikation:
 - MPI_Wait
 - MPI_Test
 - Mehrere Kommunikationsoperationen abfragen
- Kommunikation abbrechen

Blockierend vs. nicht-blockierend

- die bisher behandelten Kommunikationsarten waren alle **blockierend**
 - blockierend deshalb, da erst aus dem Funktionsaufruf zurückgekehrt wird, wenn die Kommunikation abgeschlossen ist
 - die Kommunikation gilt als beendet, wenn der jeweilige Puffer gefahrlos (ohne Informationsverlust) wieder verwendet werden kann
- im Folgenden werden wir **nicht-blockierende** Kommunikation kennen lernen
 - nicht-blockierend deshalb, da **direkt** aus dem Funktionsaufruf zurückgekehrt wird, bevor die Kommunikation abgeschlossen ist
 - der Abschluss der Kommunikation muss mit separaten MPI-Funktionen erfragt werden, um die jeweiligen Puffer gefahrlos wiederverwenden zu können

Beispiel: Senden im Ring

- jeder Prozess sendet seinen Rang an den rechten Nachbarn



Beispiel: Senden im Ring (blockierend)

```
int my_rank, size;          /* eigener Rang, Anzahl Prozesse */
int msg_s, msg_r;           /* Nachricht */
int left, right;            /* Ränge der Nachbarn */
MPI_Status status;          /* für MPI_Recv */

...                         /* init, size, rank */

right = (my_rank+1) % size;
left  = (my_rank-1 + size) % size;

MPI_Send(&msg_s, 1, MPI_INT, right, tag, MPI_COMM_WORLD);
MPI_Recv(&msg_r, 1, MPI_INT, left, tag, MPI_COMM_WORLD, &status);

MPI_Finalize();
```

Hinweise zum Beispiel

- alle Prozesse rufen zuerst das blockierende `MPI_Send` und danach erst `MPI_Recv` auf → **Gefahr der Verklemmung!**
 - keine Verklemmung bei gepuffertem Senden, also bei kleiner Datenmenge (implementierungsabhängig)
 - Verklemmung bei synchronem Senden, also bei großer Datenmenge (implementierungsabhängig)
- mögliche Lösung:
 - Prozess 0 ändert die Aufrufreihenfolge indem er zuerst empfängt und dann sendet
 - Nachteil: Serialisierung der Kommunikation
- bessere Lösung:
 - **nicht-blockierende Kommunikation**

Phasen der nicht-blockierenden Kommunikation

Nicht-blockierende Kommunikation lässt sich in 3 Phasen einteilen:

1. Kommunikation anstoßen:

- Funktionsnamen enthalten ein "I" für „immediate“
- nicht-blockierende Funktionen kehren zurück, bevor die Kommunikation abgeschlossen ist
- nicht-blockierende Funktionen haben die selben Argumente wie ihre blockierenden Gegenstücke bis auf das zusätzliche Request-Objekt

2. mit anderer Arbeit fortfahren:

- kann Berechnung oder weitere Kommunikation sein
- Ziel: Überlappung von Berechnung und Kommunikation

3. Kommunikation beenden:

- will man auf die verwendeten Puffer zugreifen, muss man warten, bis die Kommunikation beendet ist

Das Request-Objekt

- wird bei nicht-blockierender Kommunikation benötigt um abzufragen, ob die Kommunikation beendet ist, oder nicht
- das Handle für das Request-Objekt muss in lokalen Variablen abgespeichert werden:
 - C: `MPI_Request`
 - Fortran: `INTEGER`
- nicht-blockierende Funktionen geben einen Wert für das Request-Objekt zurück
- ist die Kommunikation bereits beendet, ist der Wert des Request-Objektes `MPI_REQUEST_NULL`
- dieser Wert dient als Argument für die Funktionen `MPI_Wait` und `MPI_Test`, mit denen abgefragt werden kann, ob die Kommunikation bereits beendet ist

Kommunikationsmodi (nicht-blockierend)

- **Sendemodi:**

- Synchrones Senden ⇒ MPI_Isend
- Gepuffertes Senden ⇒ MPI_Ibsend
- Standard Send ⇒ MPI_Isend
- Ready Send ⇒ MPI_Irsend

- **Empfangsmodi:**

- Receive ⇒ MPI_Irecv
- Probe ⇒ MPI_Iprobe

Nicht-blockierendes Senden

C

```
int MPI_Isend( const void* buf, int count, MPI_Datatype datatype,
                int dest, int stag, MPI_Comm comm, MPI_Request* request);
```

Fortran

```
MPI_Isend( buf, count, datatype, dest, stag, comm, request, ierror )
           TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
           INTEGER, INTENT(IN) :: count, dest, stag
           TYPE(MPI_Datatype), INTENT(IN) :: datatype
           TYPE(MPI_Comm), INTENT(IN) :: comm
           TYPE(MPI_Request), INTENT(OUT) :: request
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- nicht blockierende Senderoutinen haben die selben Argumente wie ihre blockierenden Verwandten, bis auf das `request`-Argument
- auf den Sendepuffer darf erst zugegriffen werden, nachdem das Senden erfolgreich auf Fertigstellung getestet wurde

Input:

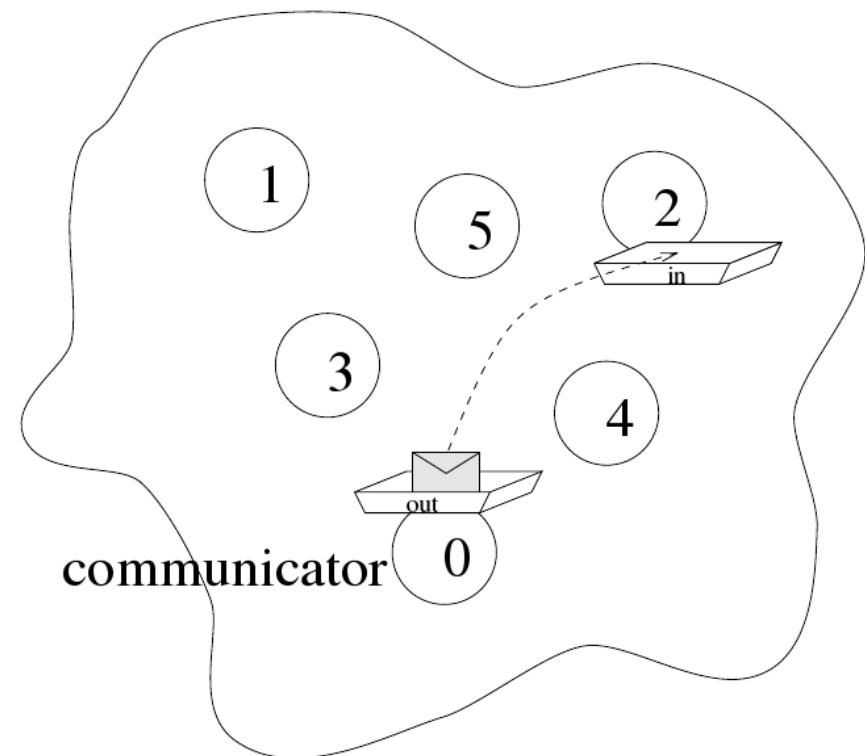
`buf`: Anfangsadresse des Sendepuffers
`count`: Anzahl der Elemente des Sendepuffers (nichtnegativ)
`datatype`: Typ der Elemente des Sendepuffers
`dest`: Rang des Empfängerprozesses in `comm`
`stag`: Tag zur Unterscheidung verschiedener Nachrichten
`comm`: Kommunikator in dem kommuniziert wird

Output:

`request`: Request-Objekt (zur Beendigung der Sendeoperation)

Nicht-blockierendes Senden: Prinzip

1. der Sender initiiert den Sendevorgang
2. die Funktion kehrt sofort aus dem Aufruf zurück (Kommunikation findet im Hintergrund statt)
3. der Sender kann direkt mit Berechnungen oder weiterer Kommunikation fortfahren, die den Sende-puffer nicht verändern
4. bevor der Sender den Sendepuffer aktualisieren kann, muss er überprüfen, ob der Sendevorgang beendet ist



Nicht-blockierendes Empfangen

C

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int src, int rtag, MPI_Comm comm, MPI_Request* request);
```

```
MPI_Irecv( buf, count, datatype, src , rtag, comm, request, ierror )  
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf  
  INTEGER, INTENT(IN) :: count, src, rtag  
  TYPE(MPI_Datatype), INTENT(IN) :: datatype  
  TYPE(MPI_Comm), INTENT(IN) :: comm  
  TYPE(MPI_Request), INTENT(OUT) :: request  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- nicht blockierende Empfangsroutinen haben die selben Argumente wie ihre blockierenden Verwandten, bis auf das `request`-Argument
- auf den Empfangspuffer darf erst zugegriffen werden, nachdem das Empfangen erfolgreich auf Fertigstellung getestet wurde

Input:

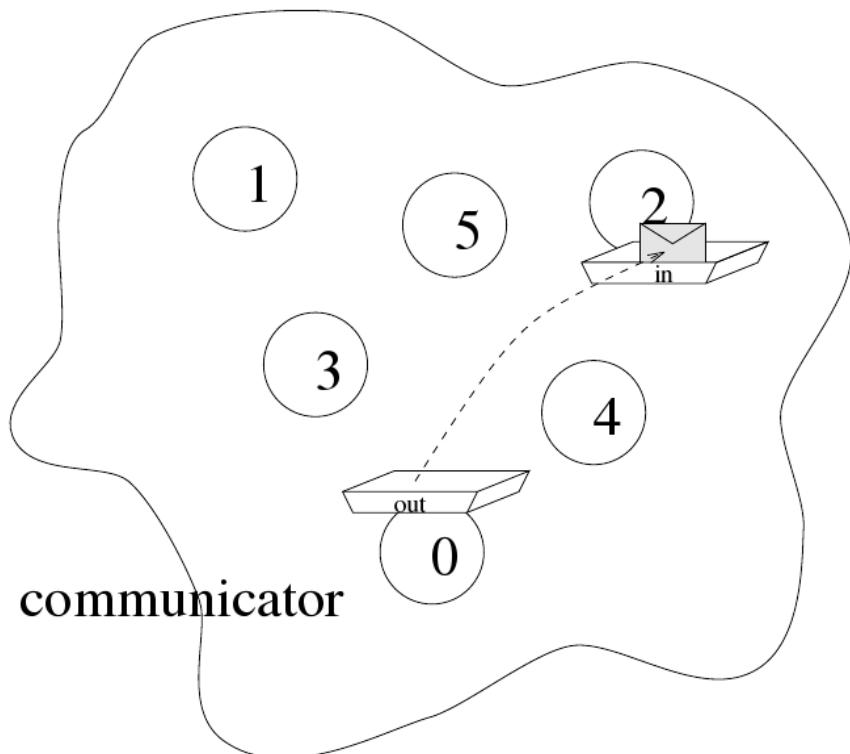
count: Anzahl der Elemente im Empfangspuffer
datatype: Datentyp der Elemente des Empfangspuffers
src: Rang des Senders in `comm`
Rtag: Tag zur Unterscheidung verschiedener Nachrichten
comm: Kommunikator in dem kommuniziert wird

Output:

buf: Anfangsadresse des Empfangpuffers
request: Request-Objekt (zur Beendigung der Sendeoperation)

Nicht-blockierendes Empfangen: Prinzip

1. der Empfänger setzt ein Empfangskommando ab
2. die Funktion kehrt sofort aus dem Aufruf zurück
(Kommunikation findet im Hintergrund statt)
3. solange der Empfänger keine Daten aus dem Empfangspuffer benötigt kann er mit anderen Operationen fortfahren
4. bevor der Empfänger auf den Empfangspuffer zugreifen kann, muss er überprüfen, ob die Nachricht komplett empfangen wurde



Beispiel: Senden im Ring (nicht-blockierend)

```
int my_rank, msg_s, msg_r; /* eigener Rank und Nachricht*/
int left, right;           /* Ränge der Nachbarn */
MPI_Status status;          /* für MPI_Wait */
MPI_Request request;      /* für Isend */

...
/* Init, Comm_Size, Comm_Rank */

right = (my_rank+1) % size;
left  = (my_rank-1 + size) % size;

MPI_Isend(&msg_s, 1, MPI_INT, right, tag, MPI_COMM_WORLD, &request);
MPI_Recv(&msg_r, 1, MPI_INT, left, tag, MPI_COMM_WORLD, &status);
MPI_Wait(&request, &status);

MPI_Finalize();
```

Abschluss der Kommunikation

- jede nicht blockierende Operation **muss** korrekt beendet werden bevor man auf die übertragenen Daten zugreift oder den Kommunikationspuffer wiederverwendet
- **2 Möglichkeiten:**
 - **WAIT** → **blockierend**: blockiert, bis die Kommunikation beendet ist
 - nützlich, bei Zugriff auf übertragene Daten oder Wiederverwendung der Kommunikationspuffer
 - **TEST** → **nicht blockierend**: gibt abhängig vom Kommunikationsstatus TRUE oder FALSE zurück
 - ist nicht blockierend
 - nützlich, wenn man sich nur über den Status der Kommunikation informieren will, und die übertragenen Daten oder die Kommunikationspuffer nicht benötigt

Warten auf Fertigstellung

C

```
int MPI_Wait( MPI_Request* request, MPI_Status* status);
```

Fortran

```
MPI_Wait( request, status, ierror )
  TYPE(MPI_Request), INTENT(INOUT) :: request
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- blockiert den rufenden Prozess solange, bis die entsprechende Sende- oder Empfangsoperation beendet ist

Inout:

request: ordnet den Wait-Aufruf einer vorher gestarteten Operation zu

Output:

status: Status-Objekt

Testen auf Fertigstellung

C

```
int MPI_Test( MPI_Request* request, int* flag,  
              MPI_Status* status);
```

Fortran

```
MPI_Test( request, flag, status, ierror )  
  TYPE(MPI_Request), INTENT(INOUT) :: request  
  LOGICAL, INTENT(OUT) :: flag  
  TYPE(MPI_Status) :: status  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- testet, ob die entsprechende Sende- oder Empfangsoperation beendet ist

Inout:

request: ordnet den Test-Aufruf einer vorher gestarteten Operation zu

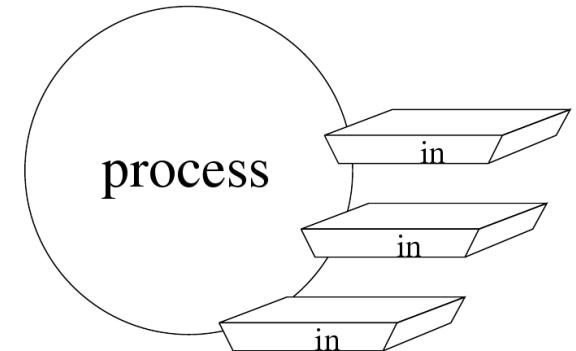
Output:

flag: TRUE, wenn die Operation beendet ist; FALSE, wenn nicht

status: Status-Objekt

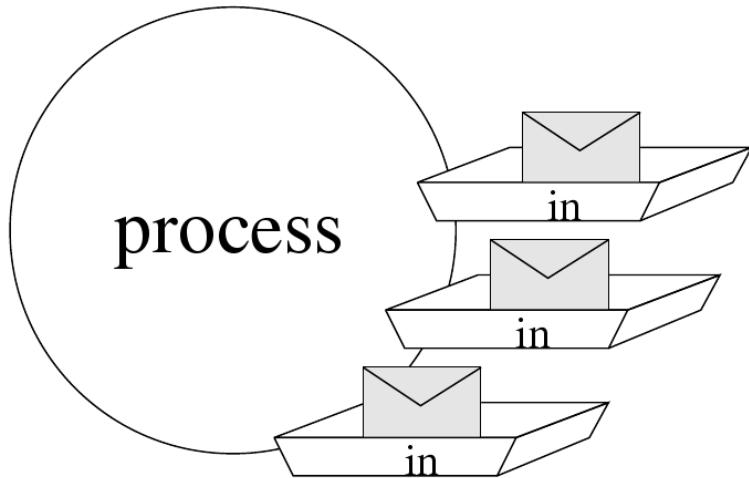
Mehrere Kommunikationsoperationen abfragen

- bei Verwendung nicht-blockierender Kommunikation ist es nicht unüblich, dass mehrere Kommunikationsprozesse zur gleichen Zeit initiiert werden
- aus diesem Grund stellt MPI 3 Arten von Funktionen zur Überprüfung mehrerer Kommunikationsprozesse jeweils in WAIT- und in TEST-Form zur Verfügung

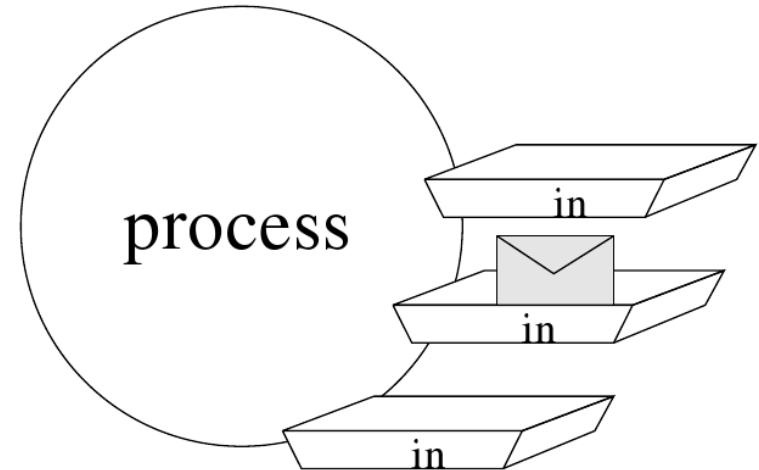


Test	WAIT-Typ	TEST-Typ
ist eine Operation von vielen beendet	<code>MPI_Waitany</code>	<code>MPI_Testany</code>
ist mindestens eine Operation unter vielen beendet	<code>MPI_Waitsome</code>	<code>MPI_Testsome</code>
sind alle Operationen beendet	<code>MPI_Waitall</code>	<code>MPI_Testall</code>

Mehrere Kommunikationsoperationen abfragen



MPI_Waitall
MPI_Testall



MPI_Waitany
MPI_Testany

Beispiel: Senden im Ring (nicht-blockierend)

```
int my_rank, msg_s, msg_r; /* eigener Rank und Nachricht*/
int left, right;           /* Ränge der Nachbarn */
MPI_Status status;          /* für MPI_Wait */
MPI_Request request;        /* für Isend */

...
/* Init, Comm_Size, Comm_Rank */

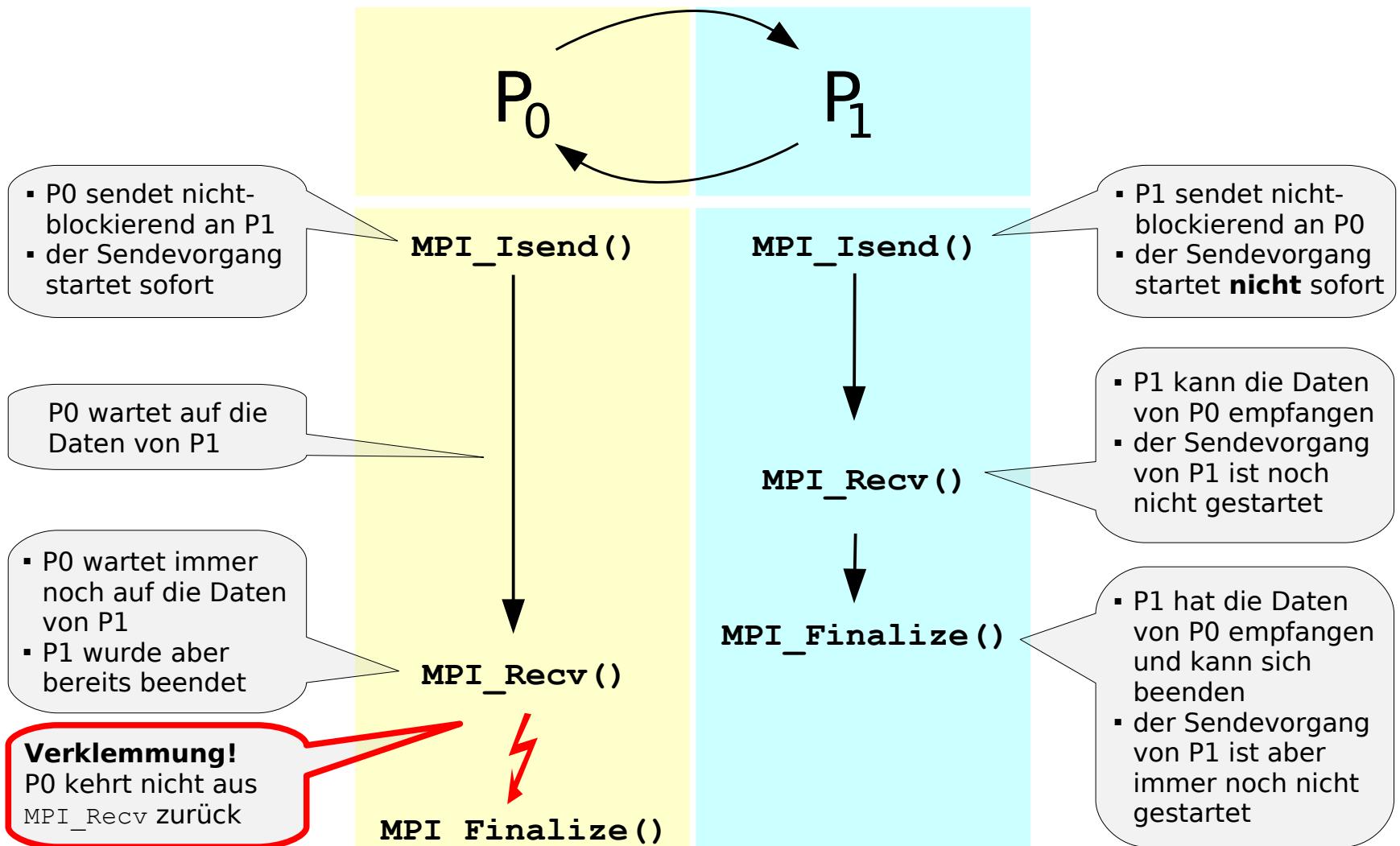
right = (my_rank+1) % size;
left  = (my_rank-1 + size) % size;

MPI_Isend(&msg_s, 1, MPI_INT, right, tag, MPI_COMM_WORLD, &request);
MPI_Recv(&msg_r, 1, MPI_INT, left, tag, MPI_COMM_WORLD, &status);
MPI_Wait(&request, &status);

MPI_Finalize();
```

Achtung: der Aufruf von **MPI_Wait** ist hier zwingend erforderlich, ansonsten kann es zu einer Verklemmung kommen!

Beispiel: Senden im Ring (Erläuterung)



Blockierend vs. nicht-blockierend

- ein blockierendes Senden kann mit einem nicht-blockierenden Empfangen kombiniert werden und umgekehrt
- nicht-blockierende Sendeoperationen können mit den gleichen Kommunikationsmodi verwendet werden, wie ihre blockierenden Gegenstücke
- eine nicht-blockierende Operation direkt gefolgt von einem passenden `MPI_Wait` ist gleichbedeutend mit der entsprechenden blockierenden Operation

Überlappung von Kommunikation und Berechnung

- ob Kommunikation und Berechnung wirklich überlappen hängt von der Implementierung ab
- nicht auf allen Plattformen sind nicht-blockierende Operationen deutlich schneller als blockierende
 - ⇒ interner Overhead für die Abwicklung nicht-blockierender Kommunikation ist höher
- wenn es die Hardware erlaubt, kann die Performance durch die Überlappung deutlich verbessert werden

Faustregel:

- die Kommunikation sollte so **früh** wie möglich angestoßen werden
- die abschließende Synchronisation/Beendigung der Kommunikation sollte so **spät** wie möglich stattfinden

Unterscheidung

Synchron / asynchron

bin ich synchron mit dem Empfänger?

ja → synchron

nein → asynchron

Blockierend / nicht-blockierend

kann ich während der Kommunikation auf den Puffer zugreifen?

ja → nicht-blockierend

nein → blockierend

Kommunikation abbrechen

C

```
int MPI_Cancel( MPI_Request* request);
```

Fortran

```
MPI_Cancel( request, ierror )  
TYPE(MPI_Request), INTENT(IN) :: request  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- bricht eine unerledigte nicht-blockierende Sende- oder Empfangsoperation ab

Input:

request: Request-Handle

Bemerkungen:

- sollte nur in Ausnahmefällen verwendet werden, da teuer
- direkte Rückkehr aus dem Funktionsaufruf (möglicherweise bevor die Kommunikation tatsächlich abgebrochen wurde)
- auch abgebrochene Sende- und Empfangsoperationen müssen mit `MPI_Wait` oder `MPI_Test` korrekt beendet werden
- entweder der Kommunikationsabbruch **oder** die -ausführung wird erfolgreich durchgeführt, aber nicht beides
- beim Abbruch von gepufferter Sendeoperation, wird der benötigte Pufferspeicher wieder freigegeben

Kommunikation abbrechen

C

```
int MPI_Test_cancelled( const MPI_Status* status, int* flag);
```

Fortran

```
MPI_Test_cancelled( status, flag, ierror )
```

```
TYPE(MPI_Status), INTENT(IN) :: status  
LOGICAL, INTENT(OUT) :: flag  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- testet, ob die entsprechende Sende- oder Empfangsoperation erfolgreich abgebrochen wurde

Input:

status: Status-Objekt

Output:

flag: TRUE, wenn die Operation abgebrochen wurde; FALSE, wenn nicht

Bemerkung:

- wurde die Kommunikation erfolgreich abgebrochen, sind alle Informationen im Status-Objekt undefiniert

Prozesse abbrechen

C

```
int MPI_Abort( MPI_Comm comm, int errorcode);
```

Fortran

```
MPI_Abort( comm, errorcode, ierror )  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, INTENT(IN) :: errorcode  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- bricht alle Prozesse im Kommunikator `comm` ab

Input:

- `comm`: Kommunikator, im welchem die Prozesse abgebrochen werden sollen
- `errorcode`: Fehlercode, der an die aufrufende Umgebung zurückgegeben wird

Bemerkung:

- in den meisten MPI-Implementierungen verhält sich `MPI_Abort` so, als ob `MPI_Abort(MPI_COMM_WORLD, errorcode)` gerufen wurde, und bricht alle Prozesse ab

Übung

Übung 4:

Nicht-blockierende Punkt-zu-Punkt Kommunikation



Einführung in die Parallelprogrammierung

MPI Teil 4: Kollektive Kommunikation

Annika Hagemeier

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich



Inhalt

- Eigenschaften der kollektiven Kommunikation
- Blockierende kollektive Kommunikation
 - Synchronisation (`MPI_Barrier`)
 - One-To-All
 - Broadcast (`MPI_Bcast`)
 - `MPI_Scatter`
 - `MPI_Scatterv`
 - All-To-One
 - `MPI_Gather`
 - `MPI_Gatherv`
 - Reduktion (`MPI_Reduce`)
 - All-To-All
 - `MPI_Allgather`
 - `MPI_Alltoall`
- Nicht-blockierende kollektive Kommunikation

Kollektive Operationen

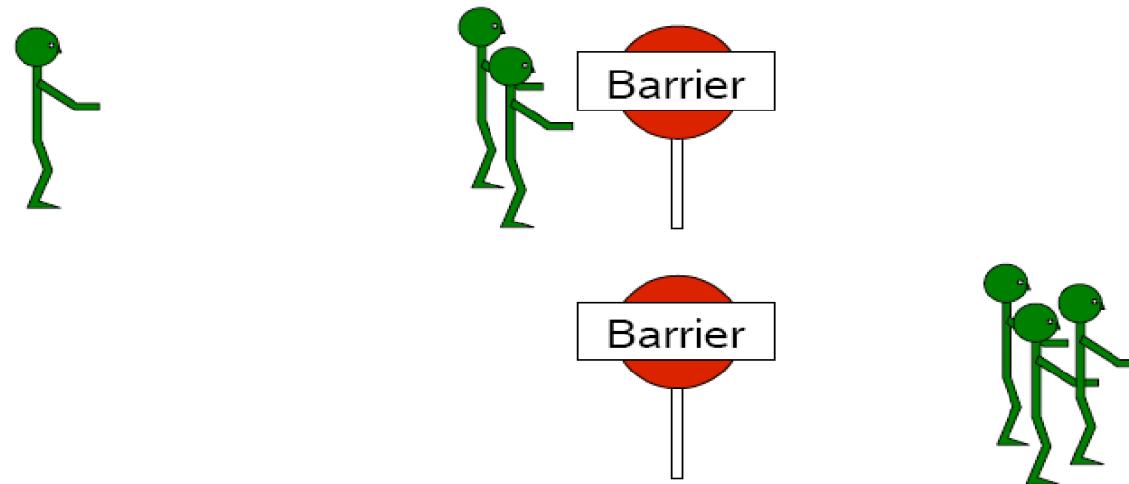
- **Kommunikation betrifft eine Gruppe von Prozessen**
- **alle** Prozesse innerhalb des Kommunikators **müssen** die Funktion **aufrufen**
- **Kategorien:**
 - **One-To-All**
 - **ein** Prozess berechnet das Ergebnis, **alle** Prozesse empfangen das Ergebnis (z. B. Broadcast, Scatter)
 - **All-To-One**
 - **alle** Prozesse berechnen das Ergebnis, **ein** Prozess empfängt das Ergebnis (z. B. Gather, globale Reduktion)
 - **All-To-All**
 - **alle** Prozesse berechnen das Ergebnis, **alle** Prozesse empfangen das Ergebnis (z. B. Allgather)
 - **Andere Operationen**
 - z.B. Synchronisation (Barrieren)

Eigenschaften kollektiver Operationen

- sind **einem** Kommunikator zugeordnet
- Synchronisation kann oder kann nicht stattfinden
- kollektive Operationen können **blockierend** oder **nicht-blockierend** (seit MPI 3.0) sein
- **kein Mischen** mit Punkt-zu-Punkt-Kommunikation möglich
- keine Tags
- Empfangspuffer müssen genau die gleiche Größe haben wie die entsprechenden Sendepuffer
- **Vorteil:** Kann von MPI optimal an die Architektur angepasst werden.

Synchronisation von Prozessen

- manchmal ist es notwendig die Prozesse eines Kommunikators zu synchronisieren, so dass jeder Prozess warten muss, bis alle Prozesse des Kommunikators eine bestimmte Stelle im Programm erreicht haben
- sollte jedoch nur in Ausnahmefällen verwendet werden, da die Zeit des Wartens nicht produktiv ist
- sinnvoll bei Zeitmessungen, Debugging und Profiling



Barrier

C

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

Fortran

```
MPI_Barrier( comm, ierror )
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

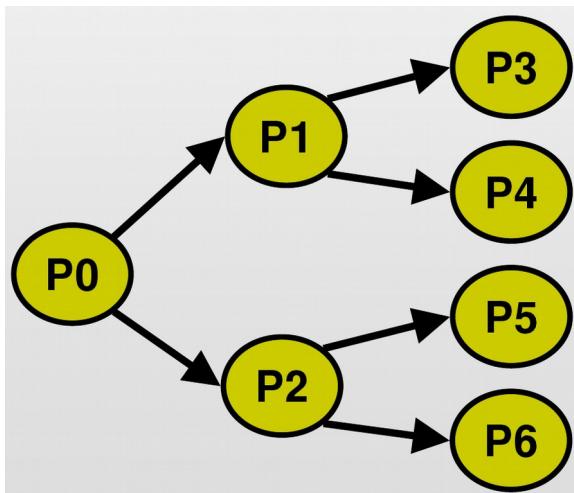
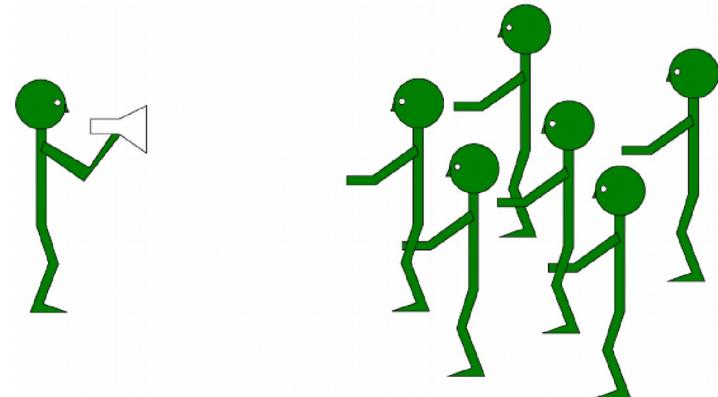
- einfachste kollektive Operation ohne Datenaustausch
- explizite Synchronisation zwischen Prozessen
 - zu beachten: die Zeit, die in `MPI_Barrier` oder jeder anderen Art von Synchronisation verbracht wird, ist nicht produktiv
 - sollte nur dann verwendet werden, wenn globale Synchronisation (in einem Kommunikator) nötig ist
 - durch Funktionen der Punkt-zu-Punkt Kommunikation werden einzelne Prozesse implizit synchronisiert
- ein Prozess kann erst aus der Funktion zurückkehren, wenn alle anderen Prozesse des Kommunikators die Funktion aufgerufen haben

Input:

`comm:` Kommunikator in dem kommuniziert wird

Broadcast

- **One-To-All-Operation:** der Prozess mit dem Rang root sendet eine Nachricht an alle Prozesse im Kommunikator
- alle Prozesse im Kommunikator müssen die Funktion aufrufen, ansonsten Verklemmung



- wie die Daten intern verschickt werden hängt von der jeweiligen MPI-Implementierung ab
- oftmals wird jedoch ein baumartiger Algorithmus verwendet um die Daten zu Blöcken von Prozessen zu schicken

Broadcast

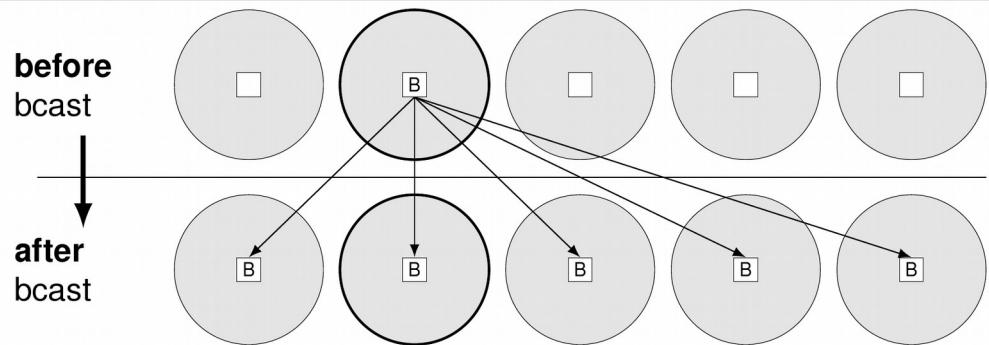
C

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

Fortran

```
MPI_Bcast( buffer, count, datatype, root, comm, ierror )  
  
TYPE(*), DIMENSION(..) :: buffer  
INTEGER, INTENT(IN) :: count, root  
TYPE(MPI_Datatype), INTENT(IN) :: datatype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- sendet eine Nachricht vom Prozess `root` an alle anderen Prozesse des angegebenen Kommunikators



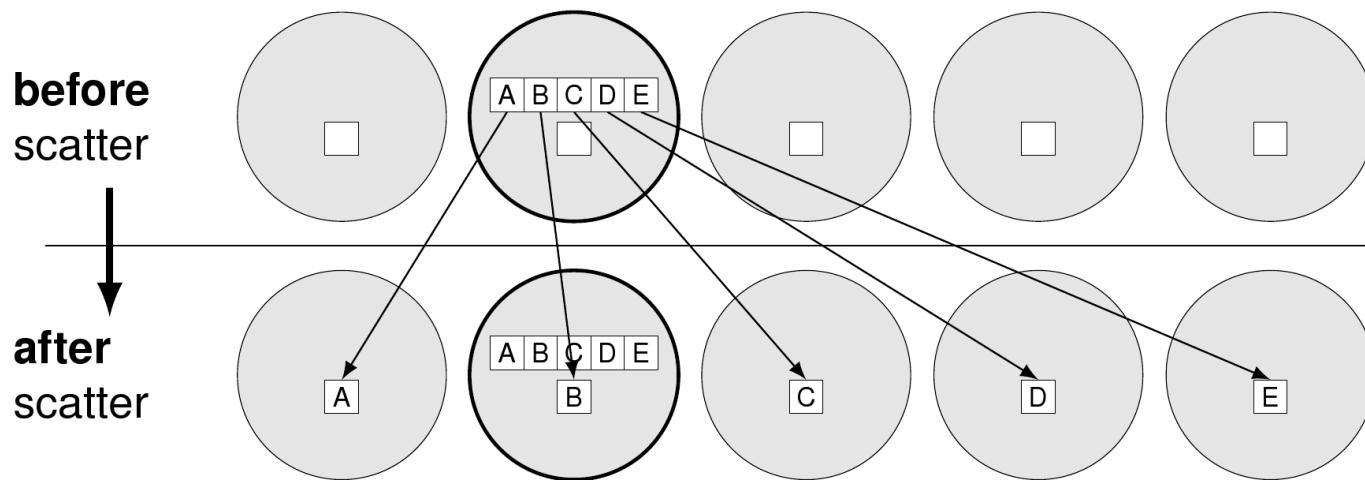
Inout:

<code>buffer:</code>	Startadresse des Datenpuffers
<code>count:</code>	Anzahl der Elemente im Puffer
<code>datatype:</code>	Datentyp der Pufferelemente
<code>root:</code>	Rang des Prozesses, der die Daten versendet
<code>comm:</code>	Kommunikator

Motivation: Scatter

Problem: Ein Vektor der Länge n soll gleichmäßig auf p Prozesse (n ist durch p teilbar) aufgeteilt werden.

Lösung: MPI_Scatter



- der Vektor wird in p gleichgroße Teile der Länge n/p unterteilt
- das i-te Teilstück des Vektors wird an den i-ten Prozess gesendet

Scatter

C

```
int MPI_Scatter( const void* sendbuf, int sendcount,  
                  MPI_Datatype sendtype, void* recvbuf, int recvcount,  
                  MPI_Datatype recvtype, int root, MPI_Comm comm );
```

Fortran

```
MPI_Scatter( sendbuf, sendcount, sendtype, recvbuf, recvcount,  
              recvtype, root, comm, ierror )  
  
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf  
TYPE(*), DIMENSION(..) :: recvbuf  
INTEGER, INTENT(IN) :: sendcount, recvcount, root  
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- verteilt Daten vom Prozess `root` auf alle Prozesse in der Gruppe, so dass alle Prozesse gleichgroße Anteile erhalten

Input:

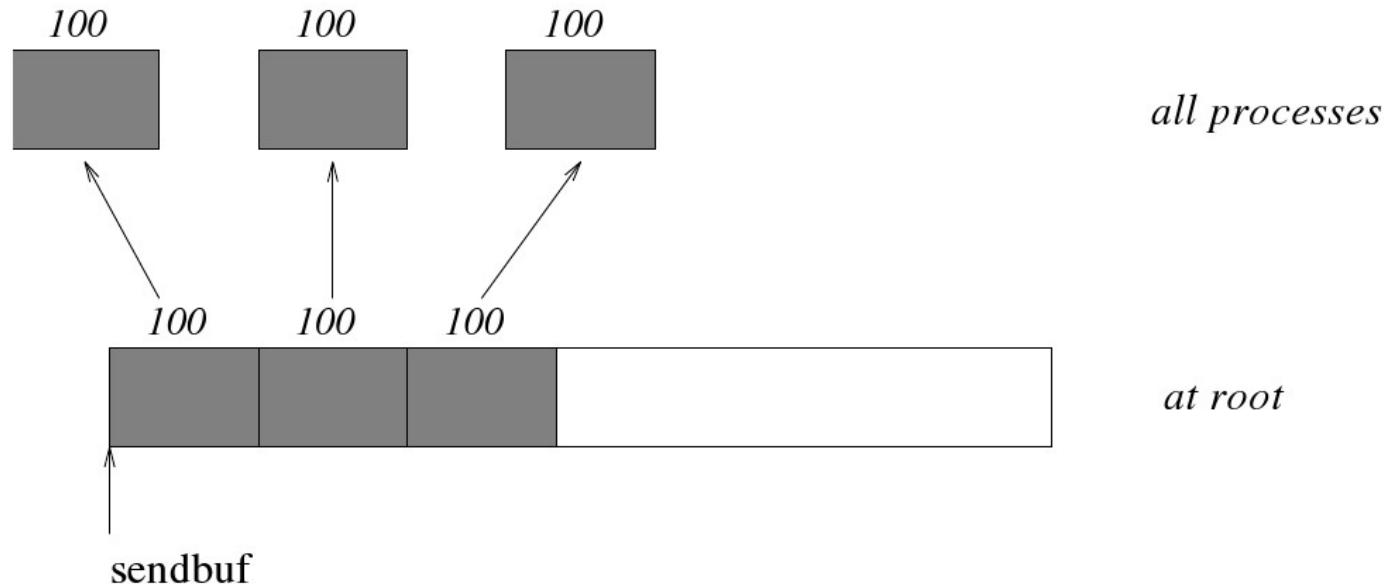
`sendbuf`: Startadresse des Sendepuffers (nur für `root` signifikant)
`sendcount`: Anzahl der Elemente, die jeder Prozess bekommen soll
`sendtype`: Datentyp der Elemente in `sendbuf`
`recvcount`: Anzahl der Elemente im Empfangspuffer
`recvtype`: Datentyp der Elemente in `recvbuf`
`root`: Rang des Prozesses, der die Daten versendet
`comm`: Kommunikator

Output:

`recvbuf`: Startadresse des Empfangspuffers (auch für `root`)

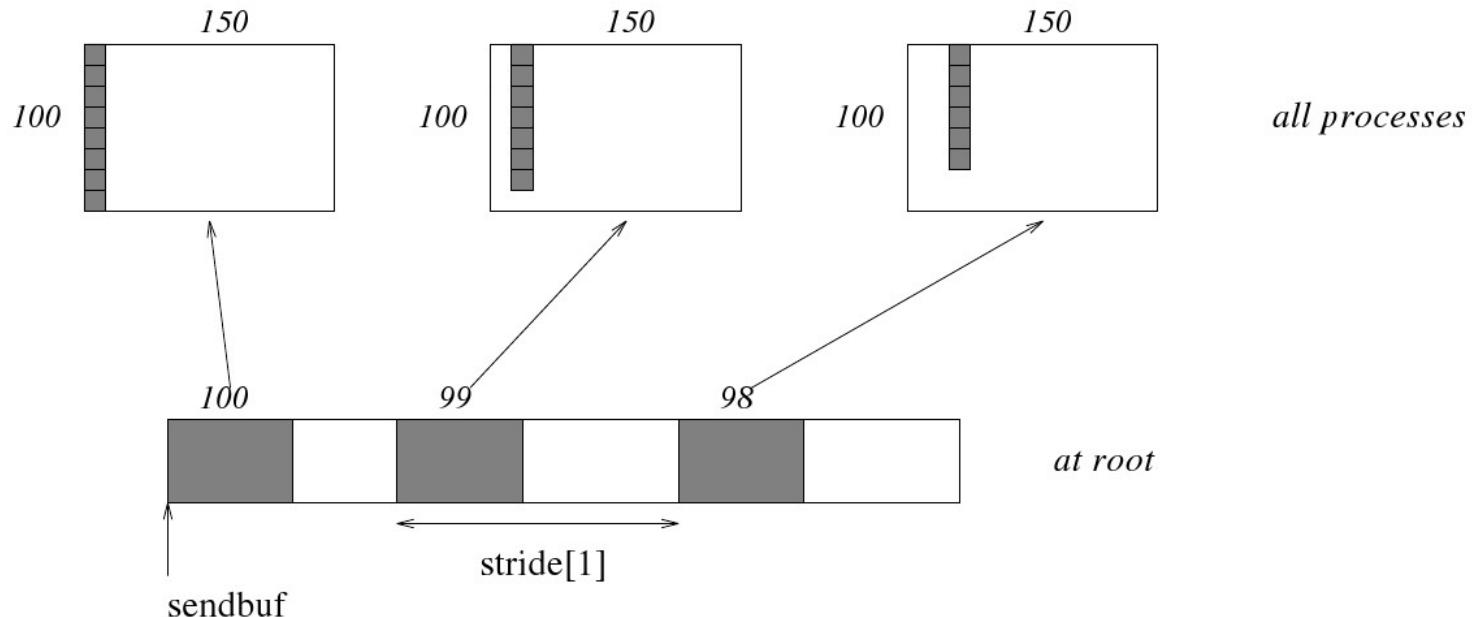
Beispiel: Scatter

```
MPI_Comm comm=MPI_COMM_WORLD;  
int size, my_rank, *sendbuf, root=0, rbuf[100];  
...  
MPI_Comm_size(comm, &size);  
if(my_rank == root)  
    sendbuf = (int *) malloc(size * 100 * sizeof(int));  
...  
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



Motivation: Scatterv

- MPI_Scatter nur anwendbar, wenn jeder Prozess die gleiche Anzahl von Daten erhält (der Vektor ist glatt durch die Anzahl der Prozesse teilbar)
- MPI_Scatter nicht geeignet, wenn:
 - Prozesse eine unterschiedliche Anzahl von Daten erhalten sollen
 - die zu verteilenden Daten nicht hintereinander im Speicher liegen



Scatterv

C

```
int MPI_Scatterv( const void* sendbuf, const int sendcounts[],  
                  const int displs[], MPI_Datatype sendtype, void* recvbuf, int  
                  recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm );
```

Fortran

```
MPI_Scatterv( sendbuf, sendcounts, displs, sendtype, recvbuf,  
               recvcount, recvtype, root, comm, ierror )
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf  
TYPE(*), DIMENSION(..) :: recvbuf  
INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root  
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Input:

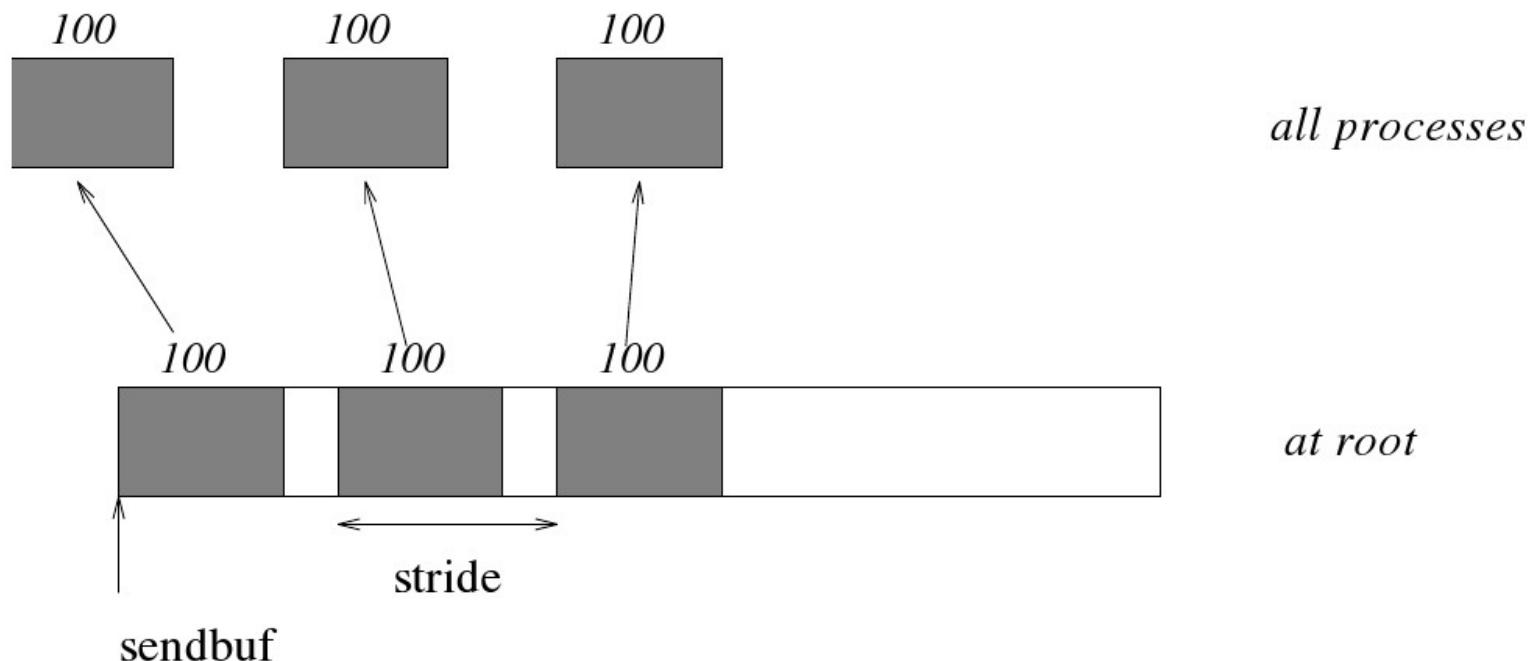
- sendbuf: Startadresse des Sendepuffers (nur für `root` signifikant)
sendcounts: Anzahl der Elemente, die jeder Prozess bekommen soll (`sendcounts[i]` gibt an, wieviele Elemente Prozess i bekommt)
displs: Displacement relativ zu `sendbuf` (`displs[i]` gibt an, wo die Daten für Prozess i liegen)
sendtype: Datentyp der Elemente in `sendbuf`
recvcount: Anzahl der Elemente im Empfangspuffer
recvtype: Datentyp der Elemente in `recvbuf`
root: Rang des Prozesses, der die Daten versendet
comm: Kommunikator

Output:

- recvbuf: Startadresse des Empfangspuffers (auch für `root`)

Beispiel: Scatterv

- der Masterprozess `root` verteilt Blöcke mit je 100 Integer-Werten an alle Prozessoren
- die Teile liegen jedoch nicht hintereinander im Speicher sondern haben einen Stride von 120



Beispiel: Scatterv (Implementierung)

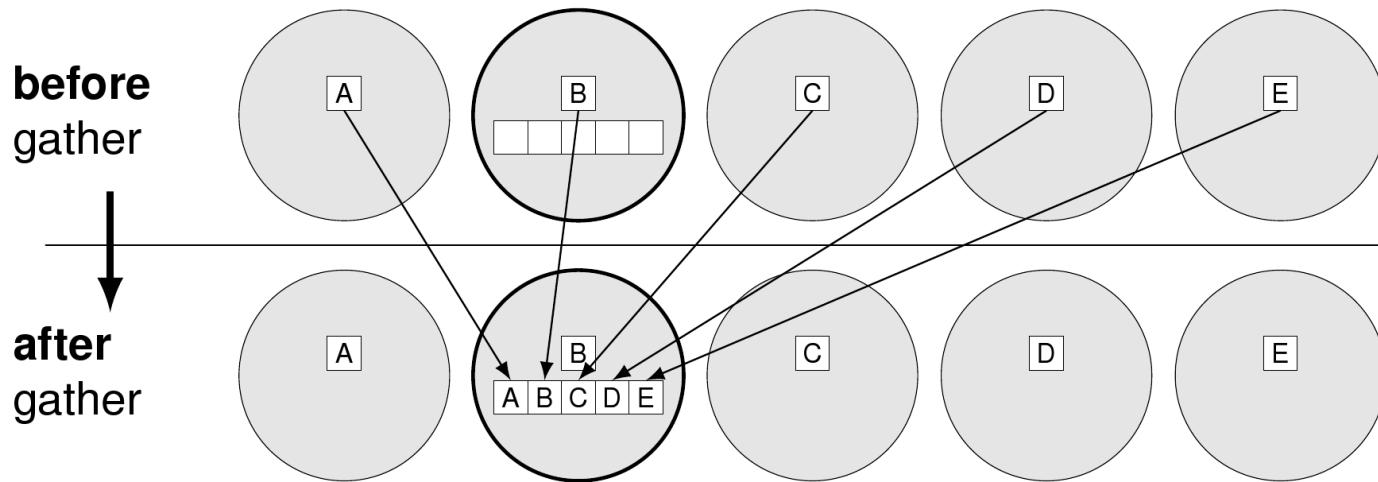
```
MPI_Comm comm=MPI_COMM_WORLD;
int size, rank, *sendbuf, root=0, rbuf[100];
int i, *displs, *counts, stride=120;
...
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
if (rank == root)
    sendbuf = (int *) malloc(size * stride * sizeof(int));
...
displs = (int *) malloc(size * sizeof(int));
counts = (int *) malloc(size * sizeof(int));
for(i=0; i<size; ++i)
    displs[i] = i*stride;
    counts[i] = 100;
}

MPI_Scatterv(sendbuf, counts, displs, MPI_INT, rbuf,
                counts[rank], MPI_INT, root, comm);
```

Motivation: Gather

Problem: Die Elemente eines Vektors der Länge n liegen gleichmäßig verteilt auf p Prozessen (n ist durch p teilbar) vor und sollen beim Root-Prozess gesammelt werden.

Lösung: MPI_Gather



- im Empfangspuffer werden die Daten in der Reihenfolge der Ränge der Prozesse abgelegt; d. h. zuerst die Daten von Prozess 0, dann die Daten von Prozess 1 usw.
- ein Puffer von ausreichender Größe muss bereit gestellt werden (Verantwortung des Programmierers)

Gather

C

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

Fortran

```
MPI_Gather( sendbuf, sendcount, sendtype, recvbuf, recvcount,  
            recvtype, root, comm, ierror )  
  
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf  
TYPE(*), DIMENSION(..) :: recvbuf  
INTEGER, INTENT(IN) :: sendcount, recvcount, root  
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- sammelt die Daten, die in einer Prozessgruppe verteilt sind, ein

Input:

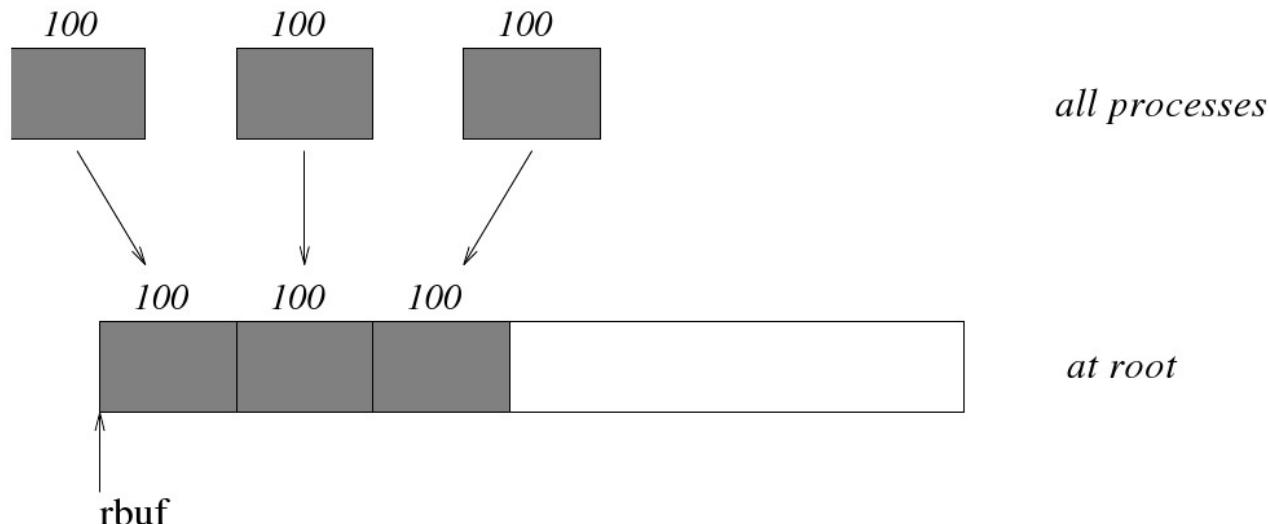
sendbuf:	Startadresse des Sendepuffers
sendcount:	Anzahl der Elemente im Sendepuffer
sendtype:	Datentyp der Elemente in sendbuf
recvcount:	Anzahl der Elemente, die jeder einzelne Prozess sendet
recvtype:	Datentyp der Elemente in recvbuf
root:	Rang des Prozesses, der die Daten empfängt
comm:	Kommunikator

Output:

recvbuf:	Startadresse des Empfangspuffers (nur für root signifikant)
----------	---

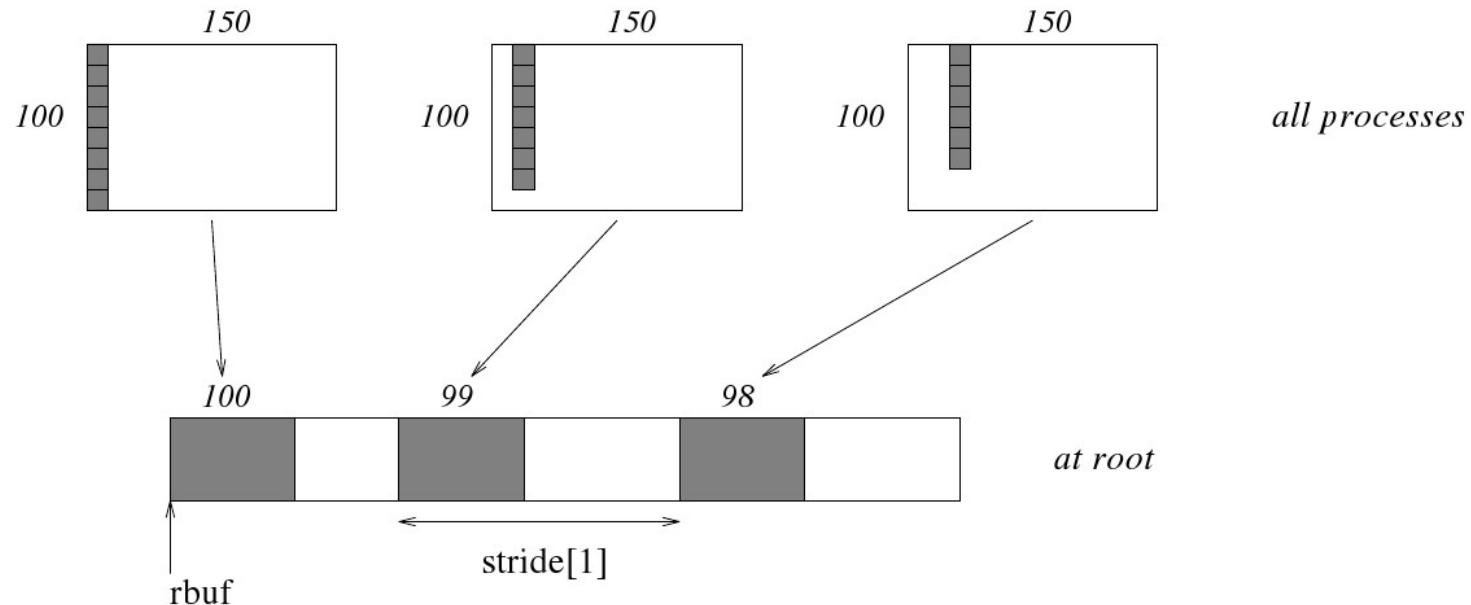
Beispiel: Gather

```
MPI_Comm comm=MPI_COMM_WORLD;  
int size, my_rank, sbuf[100], root=0, *rbuf;  
...  
MPI_Comm_size(comm, &size);  
MPI_Comm_rank(comm, &my_rank);  
if(my_rank == root)  
    rbuf = (int *) malloc(size * 100 * sizeof(int));  
...  
MPI_Gather(sbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



Motivation: Gatherv

- MPI_Gather nur anwendbar, wenn jeder Prozess die gleiche Anzahl von Daten an den Root-Prozess sendet
- MPI_Gather nicht geeignet, wenn:
 - Prozesse eine unterschiedliche Anzahl von Daten an den Root-Prozess schicken
 - die eingesammelten Daten beim Root-Prozess nicht hintereinander im Speicher liegen sollen



Gatherv

C

```
int MPI_Gatherv( const void* sendbuf, int sendcount, MPI_Datatype  
                 sendtype, void* recvbuf, const int recvcounts[], const int  
                 displs[], MPI_Datatype recvtype, int root, MPI_Comm comm );
```

Fortran

```
MPI_Gatherv( sendbuf, sendcount, sendtype, recvbuf, recvcounts,  
             displs, recvtype, root, comm, ierror )  
  
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf  
TYPE(*), DIMENSION(..) :: recvbuf  
INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root  
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Input:

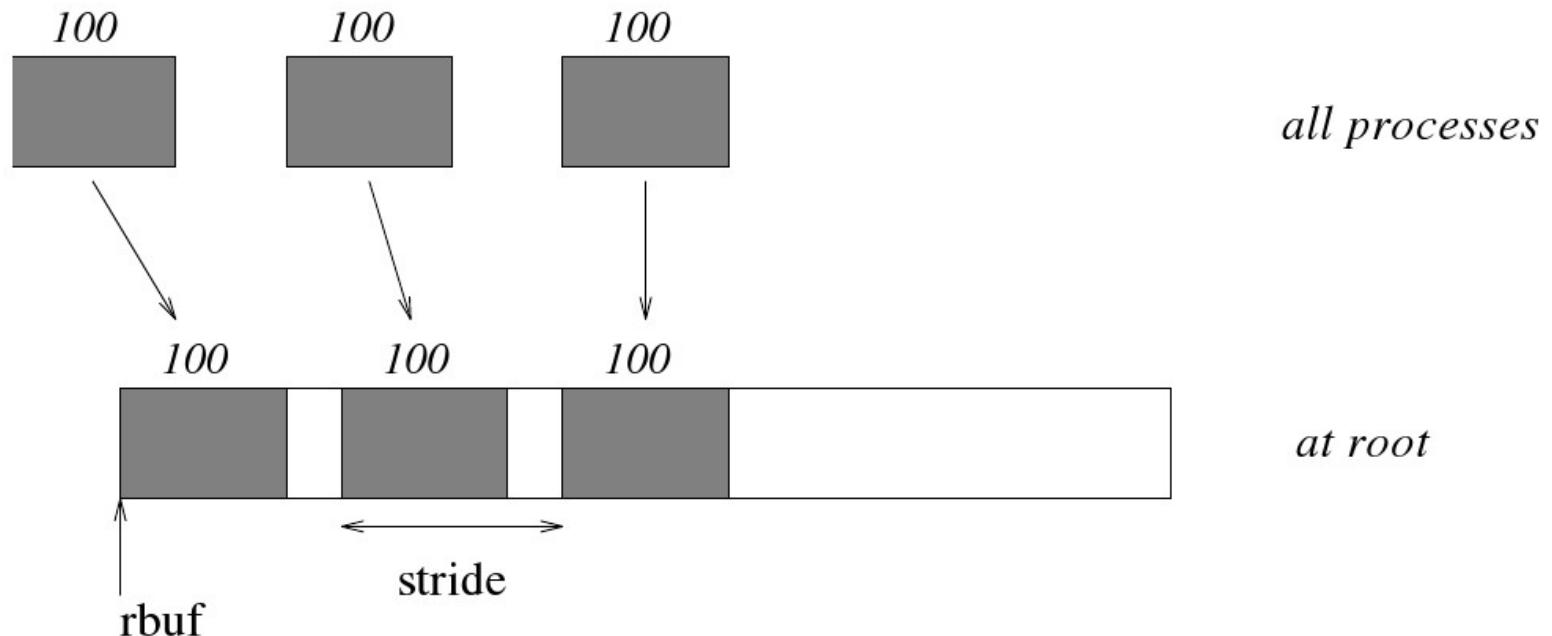
- sendbuf: Startadresse des Sendepuffers
sendcount: Anzahl der Elemente im Sendepuffer
sendtype: Datentyp der Elemente in sendbuf
recvcounts: Anzahl der Elemente, die jeder einzelne Prozess sendet
(recvcounts[i] gibt an, wieviele Elemente Prozess i sendet)
displs: Displacement relativ zu recvbuf (displs[i] gibt an, an welche Adresse die Daten von Prozess i zu schreiben sind)
recvtype: Datentyp der Elemente in recvbuf
root: Rang des Prozesses, der die Daten empfängt
comm: Kommunikator

Output:

- recvbuf: Startadresse des Empfangspuffers (auch für root)

Beispiel: Gatherv

- sammelt die Daten, die auf alle Prozesse im Kommunikator verteilt sind, ein
- jeder Prozess besitzt 100 Elemente des gesamten Vektors
- die Blöcke der Prozesse werden beim Root-Prozess mit einem Stride von 120 im Empfangspuffer gespeichert



Beispiel: Gatherv (Implementierung)

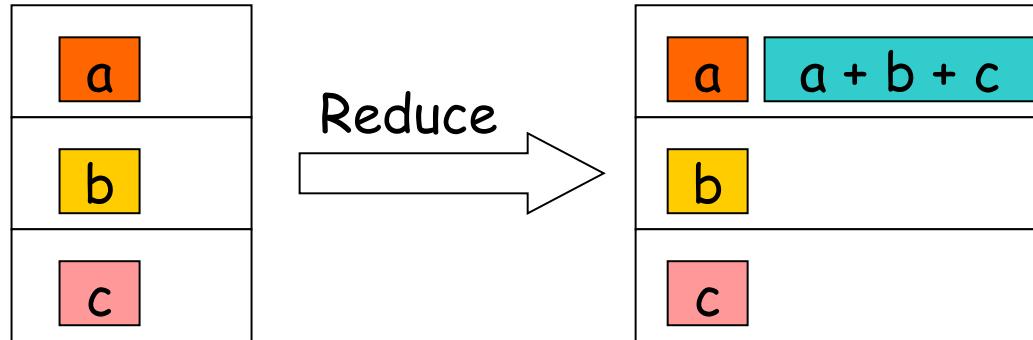
```
MPI_Comm comm=MPI_COMM_WORLD;
int size, rank, sbuf[100], root=0, *rbuf;
int i, *displs, *counts, stride=120;
...
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
if (rank == root)
    rbuf = (int *) malloc(size * stride * sizeof(int));
...
displs = (int *) malloc(size * sizeof(int));
counts = (int *) malloc(size * sizeof(int));
for(i=0; i<size; ++i)
    displs[i] = i*stride;
    counts[i] = 100;
}

MPI_Gatherv(sbuf, counts[rank], MPI_INT, rbuf, counts, displs,
            MPI_INT, root, comm);
```

Motivation: Reduktion

Problem: Jeder Prozess hat ein Teilergebnis berechnet. Die Teilergebnisse müssen über eine Operation miteinander verknüpft werden um das Endergebnis zu erhalten.

Lösung: MPI_Reduce



- jeder der drei Prozessoren hat ein Teilergebnis berechnet
- das Endergebnis ergibt sich durch Addition aller Teilergebnisse
- das Endergebnis liegt nach der Reduktion dem Root-Prozess vor

Reduktion

C

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

```
MPI_Reduce( sendbuf, recvbuf, count, datatype, op, root, comm,
             ierror )
  TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
  TYPE(*), DIMENSION(..) :: recvbuf
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Op), INTENT(IN) :: op
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- führt die globale Operation `op` aus, der Prozess `root` erhält das Resultat
- die globale Operation wird für jedes einzelne Element des Sendepuffers ausgeführt; das erste Element im Empfangspuffer enthält also das Ergebnis der Operation über den ersten Elementen aller Prozesse usw.

Input:

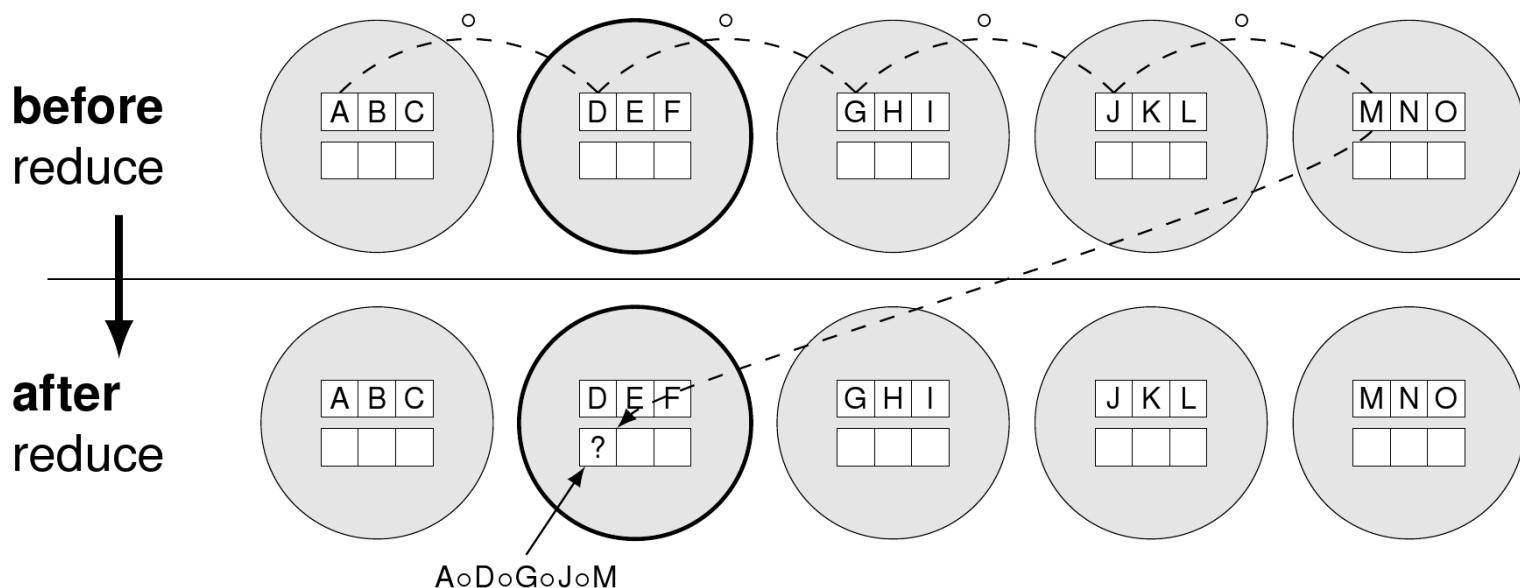
`sendbuf`: Startadresse des Sendepuffers
`count`: Anzahl der Elemente im Sendepuffer
`datatype`: Datentyp der Elemente in `sendbuf`
`op`: auszuführende Operation
`root`: Rang des Prozesses, der das Ergebnis bekommen soll
`comm`: Kommunikator

Output:

`recvbuf`: Startadresse des Puffers, der das Ergebnis enthalten soll

Reduktion: Funktionsweise

- führt die globale Operation op aus
- der Prozess `root` erhält das Resultat
- die globale Operation wird für jedes einzelne Element des Sendepuffers ausgeführt; das erste Element im Empfangspuffer enthält also das Ergebnis der Operation über den ersten Elementen aller Prozesse usw.



Vordefinierte Reduktionsoperationen

Operationshandle	Funktion
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Summe
MPI_PROD	Produkt
MPI_BAND	logisches UND
MPI_BOR	bitweises UND
MPI_LOR	logisches ODER
MPI_BXOR	bitweises ODER
MPI_LXOR	logisches exklusives ODER
MPI_MAXLOC	Maximum + Index des maximalen Wertes
MPI_MINLOC	Minimum + Index des minimalen Wertes

Benutzerdefinierte Reduktionsoperationen

- neben der Verwendung von vordefinierten Reduktionsoperationen besteht in MPI auch die Möglichkeit, eigene Reduktionsoperationen zu definieren
- Bedingungen an den benutzerdefinierten Operator \square :
 - muss assoziativ sein
 - muss die Operation $\vec{A} \square \vec{B}$ berechnen
- der definierte Operator kann in MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter, MPI_Scan und MPI_Exscan verwendet werden

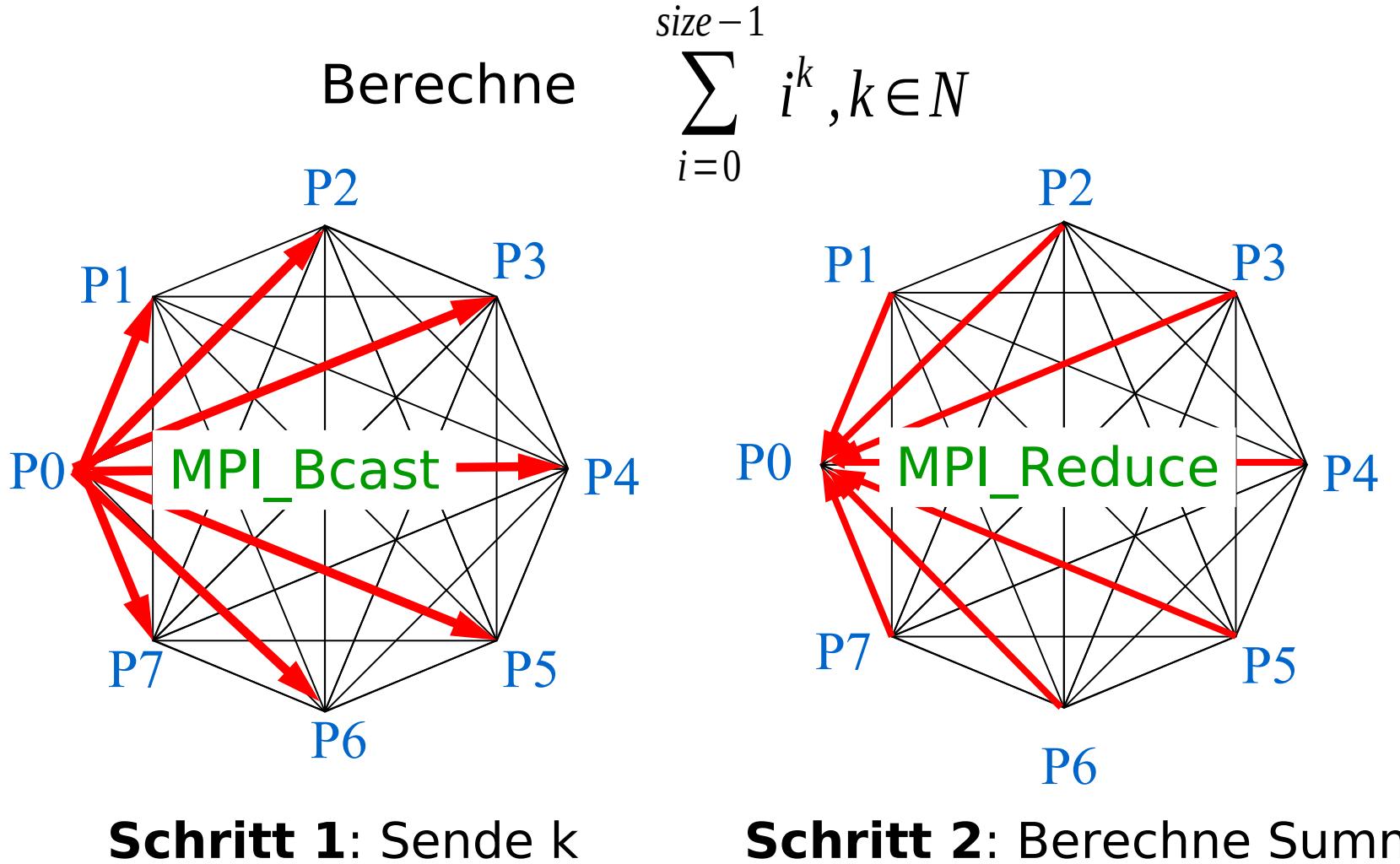
```
int MPI_Op_create( MPI_User_function* func, int commute, MPI_Op *op )
```

- definiert die Funktion `func` als Reduktionsoperator `op`
- `commute` gibt an, ob der Operator kommutativ ist

```
int MPI_Op_free( MPI_Op *op )
```

- löscht den Operator `op` und setzt ihn auf `MPI_OP_NULL`

Beispiel: Reduktion



Beispiel: Reduktion (Implementierung)

```
int my_rank, k, i;
double res, sum;                                /* Das Ergebnis */
...                                              /* Init, Comm_Rank */
if(my_rank == 0) { k=5;} /* Prozess 0 initialisiert k */

/* Schritt 1: Sende/Empfange k*/
MPI_Bcast(&k,1,MPI_INT,0,MPI_COMM_WORLD);
res = (double) my_rank;

/* Berechne i^k */
if(k == 0) res = 1.0;
else
    for(i=0;i<k;++i)
        res *= (double) my_rank;

/* Schritt 2: Berechne Summe */
MPI_Reduce(&res, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if(my_rank == 0) printf(„Das Ergebnis ist %lf\n“,sum);

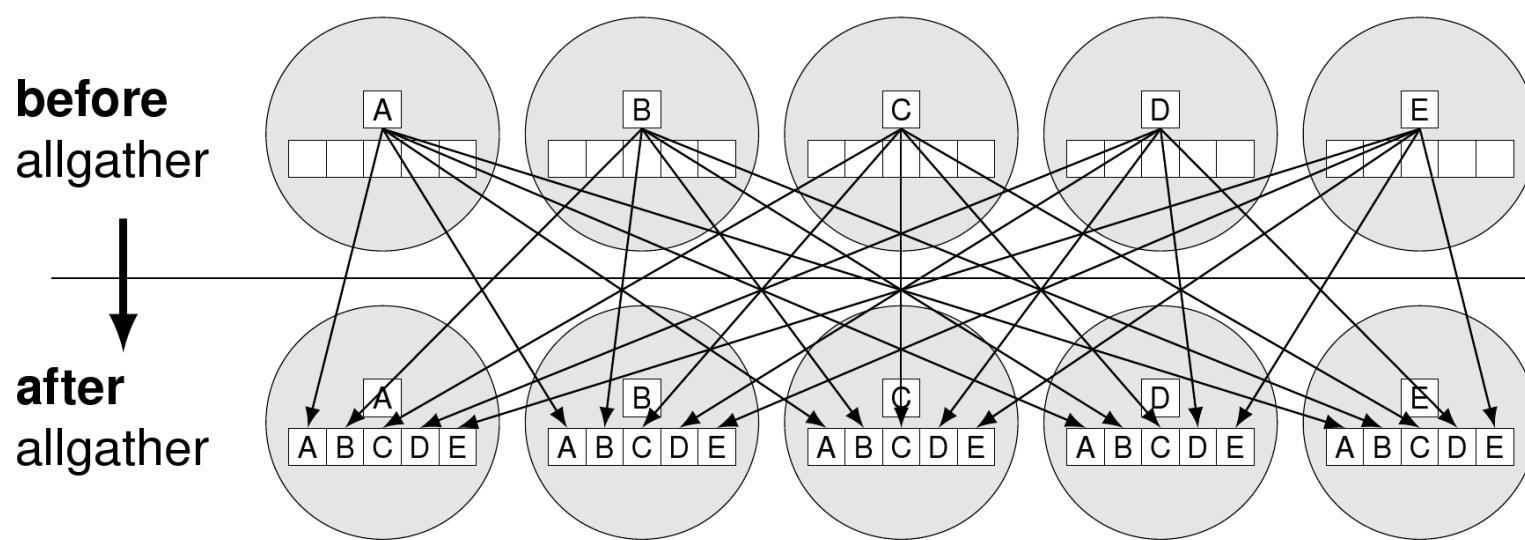
MPI_Finalize();
```

Varianten der Reduktion

- **MPI_Allreduce**:
 - nach Abschluss der Operation stehen die Daten allen Prozessen zur Verfügung
 - entspricht einem `MPI_Reduce(sbuf, rbuf, ...)` gefolgt von `MPI_Bcast(rbuf, ...)`
- **MPI_Reduce_scatter**:
 - Ergebnisvektor wird via Scatter an alle Prozesse verteilt
- **MPI_Scan**:
 - Präfixreduktion, d. h. das Ergebnis von Prozess i ergibt sich durch Reduktion der Werte von Prozess $0, \dots, i$
- **MPI_Exscan**:
 - das Ergebnis von Prozess 0 ist undefined
 - das Ergebnis von Prozess 1 ist der Wert im Sendepuffer von Prozess 0
 - für alle Prozesse $i > 1$ ist das Ergebnis die Reduktion der Werte der Prozesse $0, \dots, i-1$

Allgather

- sammelt Daten, die auf die Prozesse eines Kommunikators verteilt sind, ein und verteilt das Resultat an alle Prozesse in der Gruppe
- nach Abschluss der Operation stehen die Daten allen Prozessen zur Verfügung
- entspricht einem MPI_Gather (sbuf, rbuf, ...) gefolgt von MPI_Bcast (rbuf, ...)



Allgather

C

```
int MPI_Allgather( const void* sendbuf, int sendcount,  
                   MPI_Datatype sendtype, void* recvbuf, int recvcount,  
                   MPI_Datatype recvtype, MPI_Comm comm);
```

Fortran

```
MPI_Allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount,  
                recvtype, comm, ierror )  
  
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf  
TYPE(*), DIMENSION(..) :: recvbuf  
INTEGER, INTENT(IN) :: sendcount, recvcount  
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- sammelt Daten, die in einer Prozessgruppe verteilt sind, ein und verteilt das Resultat an alle Prozesse in der Gruppe.

Input:

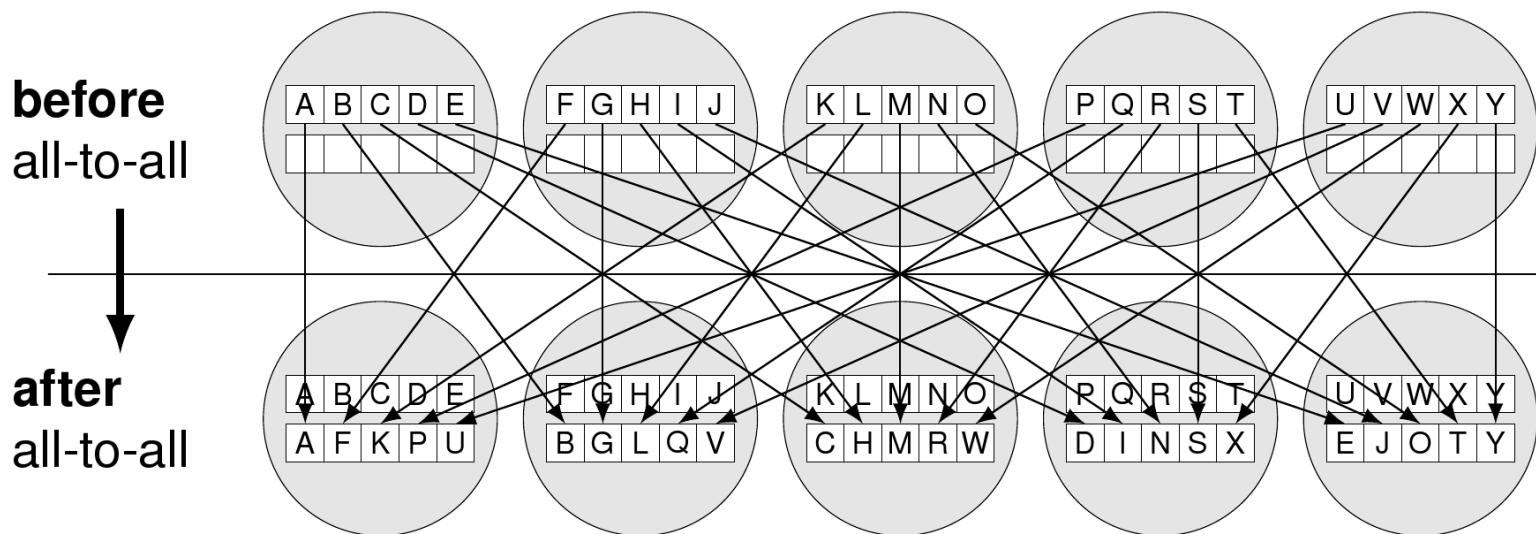
sendbuf: Startadresse des Sendepuffers
sendcount: Anzahl der Elemente im Sendepuffer
sendtype: Datentyp der Elemente in sendbuf
recvcount: Anzahl der Elemente, die jeder einzelne Prozess sendet
recvtype: Datentyp der Elemente in recvbuf
comm: Kommunikator

Output:

recvbuf: Startadresse des Empfangspuffers

Alltoall

- MPI_Alltoall ist eine Erweiterung von MPI_Allgather
- jeder Prozess sendet dabei unterschiedliche Daten zu den anderen Prozessen
- der j-te Block, der von Prozess i gesendet wird, wird von Prozess j empfangen und an der Stelle für den i-ten Block im Empfangspuffer abgespeichert
- die Blöcke müssen dabei gleich groß sein und hintereinander im Speicher liegen



Alltoall

C

```
int MPI_Alltoall( const void* sendbuf, int sendcount,
                  MPI_Datatype sendtype, void* recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm);
```

Fortran

```
MPI_Alltoall( sendbuf, sendcount, sendtype, recvbuf, recvcount,
                recvtype, comm, ierror )

TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- verteilt Daten von jedem Prozess einer Gruppe auf alle anderen Prozesse

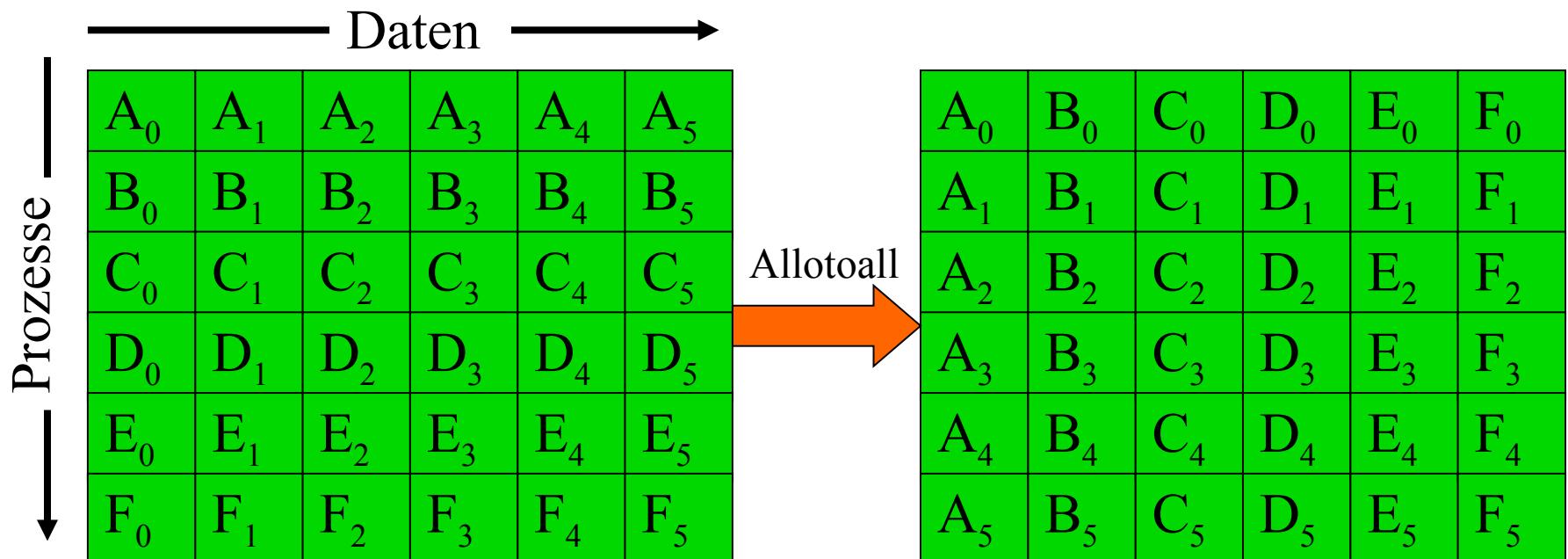
Input:

sendbuf: Startadresse des Sendepuffers
sendcount: Anzahl der Elemente im Sendepuffer
sendtype: Datentyp der Elemente in sendbuf
recvcount: Anzahl der Elemente, die jeder einzelne Prozess empfängt
recvtype: Datentyp der Elemente in recvbuf
comm: Kommunikator

Output:

recvbuf: Startadresse des Empfangspuffers

Beispiel: Alltoall



Kategorien der **blockierenden** kollektiven Kommunikation

- **One-To-All:**

- MPI_Bcast
- MPI_Scatter, MPI_Scatterv

- **All-To-One:**

- MPI_Gather, MPI_Gatherv
- MPI_Reduce

- **All-To-All:**

- MPI_Allgather, MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw
- MPI_Allreduce, MPI_Reduce_scatter
MPI_Reduce_scatter_block

- **Andere:**

- MPI_Barrier
- MPI_Scan, MPI_Exscan

Nicht-blockierende kollektive Kommunikation (1)

- Nicht-blockierende kollektive Kommunikation kombiniert die potenziellen Vorteile von nicht-blockierender Punkt-zu-Punkt-Kommunikation mit der optimierten Implementierung kollektiver Operationen
- Das Prinzip der nicht-blockierenden kollektiven Kommunikation ist ähnlich dem Prinzip der nicht-blockierenden Punkt-zu-Punkt-Kommunikation
- Der Aufruf einer nicht-blockierenden kollektiven Funktion initiiert eine kollektive Operation, die mit einem separaten Aufruf abgeschlossen werden **muss**
- Einmal initiiert, verläuft die Operation im Hintergrund unabhängig von anderen Berechnungen oder anderer Kommunikation
- Nicht-blockierende kollektive Operationen können so mögliche Synchronisationseffekte abschwächen

Nicht-blockierende kollektive Kommunikation (2)

- Zusätzlich zur Überlappung von Kommunikation und Berechnung ermöglicht die nicht-blockierende kollektive Kommunikation die Durchführung kollektiver Operationen auf überlappenden Kommunikatoren, welche bei blockierender Kommunikation zu Deadlocks führen würde
- Wie bei nicht-blockierender Punkt-zu-Punkt-Kommunikation kehren alle nicht-blockierenden kollektiven Aufrufe sofort zurück, unabhängig vom Status der Kommunikation
- Nach der Initiierung darf auf die verwendeten Puffer nicht mehr zugegriffen werden, bevor die Kommunikation nicht beendet wurde
- Der Aufruf liefert ein Request-Objekt zurück, welches als Eingabeparameter für die Funktionen zur Beendigung der Kommunikation dient
- Für nicht-blockierende kollektive Kommunikation existieren dieselben Funktionen zur Beendigung der Kommunikation wie bei nicht-blockierender Punkt-zu-Punkt-Kommunikation

Abschluss der Kommunikation

- jede nicht blockierende Operation **muss** korrekt beendet werden bevor man auf die übertragenen Daten zugreift oder den Kommunikationspuffer wiederverwendet
- **2 Möglichkeiten:**
 - **WAIT** → **blockierend**: blockiert, bis die Kommunikation beendet ist
 - nützlich, bei Zugriff auf übertragene Daten oder Wiederverwendung der Kommunikationspuffer
 - **TEST** → **nicht blockierend**: gibt abhängig vom Kommunikationsstatus TRUE oder FALSE zurück
 - ist nicht blockierend
 - nützlich, wenn man sich nur über den Status der Kommunikation informieren will, und die übertragenen Daten oder die Kommunikationspuffer nicht benötigt

Nicht-blockierende kollektive Kommunikation (3)

- Die **Fertigstellung** einer Operation bezieht sich auch hier auf den **lokalen** Prozess
- d.h. Fertigstellung einer Operation bedeutet
 - Die verwendeten Puffer können gefahrlos, ohne Informationsverlust, wiederverwendet werden
- Fertigstellung einer Operation bedeutet **nicht**
 - Dass die anderen Prozesse ihre Operation ebenfalls beendet haben, bzw. überhaupt schon begonnen haben
 - Dass andere nicht-blockierende kollektive Operationen, die evtl. davor geposted wurden auch beendet sind
- Anders als bei Punkt-zu-Punkt-Kommunikation können nicht-blockierende kollektive Operationen **nicht** mit blockierenden Operationen kombiniert werden
 - Alle Prozesse müssen **alle** kollektiven Operationen in der selben Weise und in der selben Reihenfolge aufrufen (innerhalb eines Kommunikators)

Kategorien der nicht-blockierenden kollektiven Kommunikation

- **One-To-All:**

- MPI_I**bcast**
- MPI_I**scatter**, MPI_I**scatternv**

- **All-To-One:**

- MPI_I**gather**, MPI_I**gatherv**
- MPI_I**reduce**

- **All-To-All:**

- MPI_I**allgather**, MPI_I**allgatherv**
- MPI_I**alltoall**, MPI_I**alltoallv**, MPI_I**alltoallw**
- MPI_I**allreduce**, MPI_I**reduce_scatter**,
MPI_I**reduce_scatter_block**

- **Andere:**

- MPI_I**barrier**
- MPI_I**scan**, MPI_I**exscan**

Fehlerhaftes Beispiel

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

    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Send(buf2, count, type, 1, tag, comm);
}

else if (my_rank == 1) {

    MPI_Recv(buf2, count, type, 0, tag, comm, &status);
    MPI_Bcast(buf1, count, type, 0, comm);

}
...
```

- Dieses Beispiel führt zu einem Deadlock!
- Prozess 0 blockiert beim Bcast solange, bis Prozess 1 das Bcast erreicht
- Prozess 1 wird das Bcast jedoch nicht erreichen, da Prozess 1 beim Recv blockiert, bis die Daten von Prozess 0 empfangen wurden

Richtiges Beispiel

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

    MPI_Ibcast(buf1, count, type, 0, comm, &request) ;
    MPI_Send(buf2, count, type, 1, tag, comm) ;
}

else if (my_rank == 1){

    MPI_Recv(buf2, count, type, 0, tag, comm, &status) ;
    MPI_Ibcast(buf1, count, type, 0, comm, &request) ;
}

MPI_Wait(&request, &status) ;
...
```

Übung

Übung 5: **Kollektive Kommunikation**



Einführung in die Parallelprogrammierung

MPI Teil 5: Abgeleitete Datentypen

Annika Hagemeier

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich



Inhalt

- MPI-Datentypen
- Arten von abgeleiteten Datentypen
 - Contiguous
 - Vector
 - Indexed
 - Struct
 - Subarray
- Erstellen eines abgeleiteten Datentyps
 - Type Map
 - Berechnung des Displacements
 - Commit und Free
 - Size und Extent

MPI Datentypen

- Beschreibung des Buffer Layouts im Hauptspeicher für Senden und Empfangen
- Basis Datentypen
 - erlauben das Versenden von Skalaren oder Felder eines Datentyps
 - Daten müssen in einem zusammenhängendem Bereich des Hauptspeichers liegen
- Problem: Versenden von nicht-zusammenhängenden Daten oder gemischten Datentypen, z.B.:
 - Teil-Blöcke einer Matrix
 - Datenstrukturen

Nicht zusammenhängende Daten

- **Beispiel:**

```
double array[DIM] ;
```

wollen die Werte `array[0]`, `array[5]`, `array[10]` ...
verschicken



- **Lösung mit Basistypen:**

- mehrfaches Versenden und Empfangen von Teilen der Nachricht -> *Langsam und aufwändig*
- Daten in zusammenhängenden Buffer kopieren und diesen versenden -> *Memory, CPU-Zeit*

Gemischte Datentypen

- **Beispiel:**

```
struct buff_layout {  
    int    i[3];  
    double d[5];  
} buffer;
```

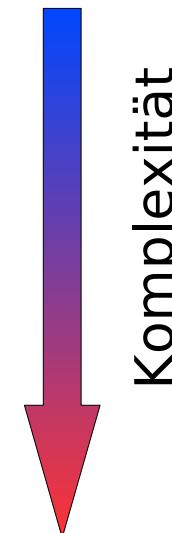


- **Lösung mit Basisdatentypen:**
 - mehrfaches Versenden und Empfangen von Teilen gleichen Typs -> *Langsam und aufwändig*
 - Versenden als MPI_BYTE und Länge mit sizeof
-> *Portabilität*
- **bessere Lösung: Abgeleitete Datentypen**

Datentyp-Konstruktoren

Passenden Datentyp-Konstruktor auswählen:

- `MPI_Type_contiguous`
- `MPI_Type_vector`
- `MPI_Type_indexed`
- `MPI_Type_indexed_block`
- `MPI_Type_create_struct`
- `MPI_Type_create_subarray`
- `MPI_Type_create_darray`



Komplexität

!

Es sollte stets der einfachste abgeleitete Datentyp verwendet werden, der den Anforderungen entspricht. Je komplexer der Datentyp, desto langsamer ist das Handling.

!

Abgeleiteter Datentyp Contiguous

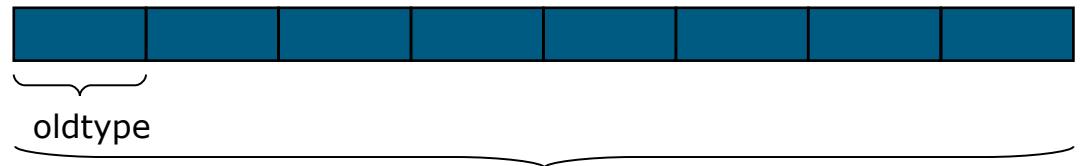
C

```
int MPI_Type_contiguous( int count, MPI_Datatype oldtype,
                         MPI_Datatype* newtype );
```

Fortran

```
MPI_Type_contiguous( count, oldtype, newtype, ierror )
  INTEGER, INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: oldtype
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- einfachste Art einen Datentyp abzuleiten
- besteht aus einer Anzahl gleichartiger Elemente eines vorhandenen Datentyps die hintereinander im Speicher angeordnet werden



Inout:

count: Anzahl der Wiederholungen (nichtnegativ)
oldtype: alter Datentyp

Output:

newtype: neuer Datentyp

Abgeleiteter Datentyp Vector

C

```
int MPI_Type_vector( int count, int blocklength, int stride,  
                      MPI_Datatype oldtype, MPI_Datatype* newtype );
```

Fortran

```
MPI_Type_vector( count, blocklength, stride, oldtype, newtype,  
                  ierror )  
  
INTEGER, INTENT(IN) :: count, blocklength, stride  
TYPE(MPI_Datatype), INTENT(IN) :: oldtype  
TYPE(MPI_Datatype), INTENT(OUT) :: newtype  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- ist ein allgemeinerer Konstruktor
- ermöglicht das Replizieren eines Datentyps in gleichgroße Blöcke, die nicht hintereinander im Speicher liegen müssen
- der Abstand zwischen den Blöcken ist ein Vielfaches des ursprünglichen Datentyps `oldtype`

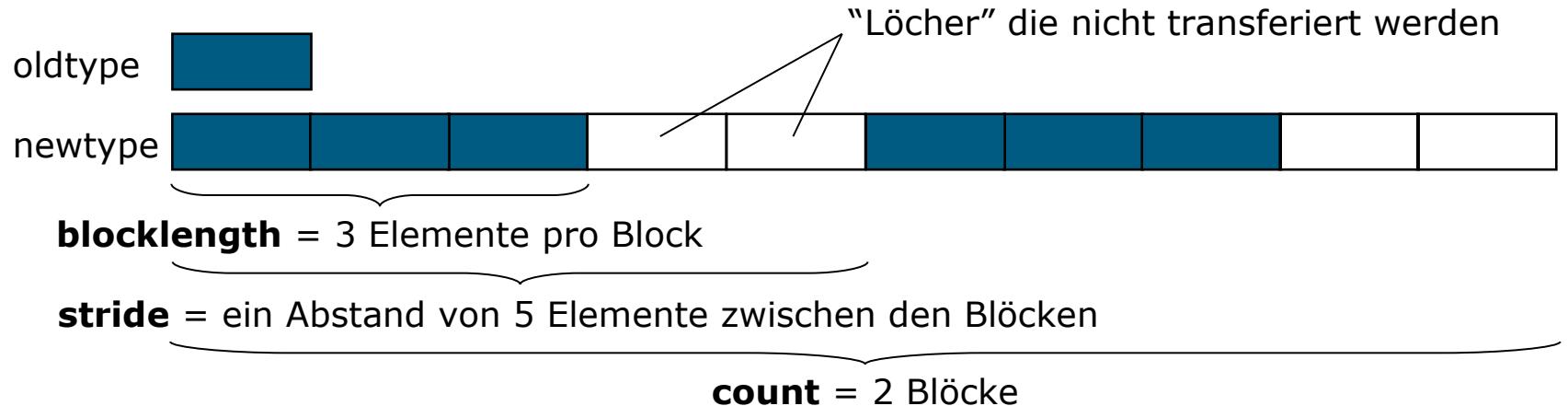
Inout:

`count`: Anzahl der Blöcke (nichtnegativ)
`blocklength`: Anzahl der Elemente pro Block (nichtnegativ)
`stride`: Anzahl der Elemente zwischen den Blockanfängen
`oldtype`: alter Datentyp

Output:

`newtype`: neuer Datentyp

Beispiel: Vector



```
int count, blocklength, stride;  
MPI_Datatype newtype;  
...  
count = 2;  
blocklength = 3;  
stride = 5;  
...  
MPI_Type_vector( count, blocklength, stride, MPI_FLOAT, &newtype);
```

Abgeleiteter Datentyp Indexed

C

```
int MPI_Type_indexed( int count, const int* blocklengths[],  
                      const int displs[], MPI_Datatype oldtype,  
                      MPI_Datatype* newtype );
```

Fortran

```
MPI_Type_indexed( count, blocklengths, displs, oldtype,  
                    newtype, ierror )  
  
INTEGER, INTENT(IN) :: count, blocklengths(count), displs(count)  
TYPE(MPI_Datatype), INTENT(IN) :: oldtype  
TYPE(MPI_Datatype), INTENT(OUT) :: newtype  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- allgemeiner als MPI_Type_vector
- jeder Block kann eine unterschiedliche Anzahl Elemente enthalten
- das Displacement der Blöcke kann ebenfalls unterschiedlich sein

Inout:

count: Anzahl der Blöcke (nichtnegativ), gibt außerdem die Anzahl der Einträge in blocklengths und displs an

blocklengths: Anzahl der Elemente pro Block (Vektor, nichtnegativ)

displs: Anzahl der Bytes zwischen den Blockanfängen (Vektor)

oldtype: alter Datentyp

Output:

newtype: neuer Datentyp

Abgeleiteter Datentyp Struct

C

```
int MPI_Type_create_struct( int count, const int blocklengths[],  
                           const MPI_Aint displs[], const MPI_Datatype oldtypes[],  
                           MPI_Datatype* newtype );
```

Fortran

```
MPI_Type_create_struct( count, blocklengths, displs, oldtypes,  
                        newtype, ierror )  
  
  INTEGER, INTENT(IN) :: count, blocklengths(count)  
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(count)  
  TYPE(MPI_Datatype), INTENT(IN) :: oldtypes(count)  
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- allgemeinste Form eines Konstruktors
- jeder Block enthält gleichartige Elemente eines Datentyps
- die einzelnen Blöcke können jedoch verschiedene Datentypen, Längen und Blockanfänge haben

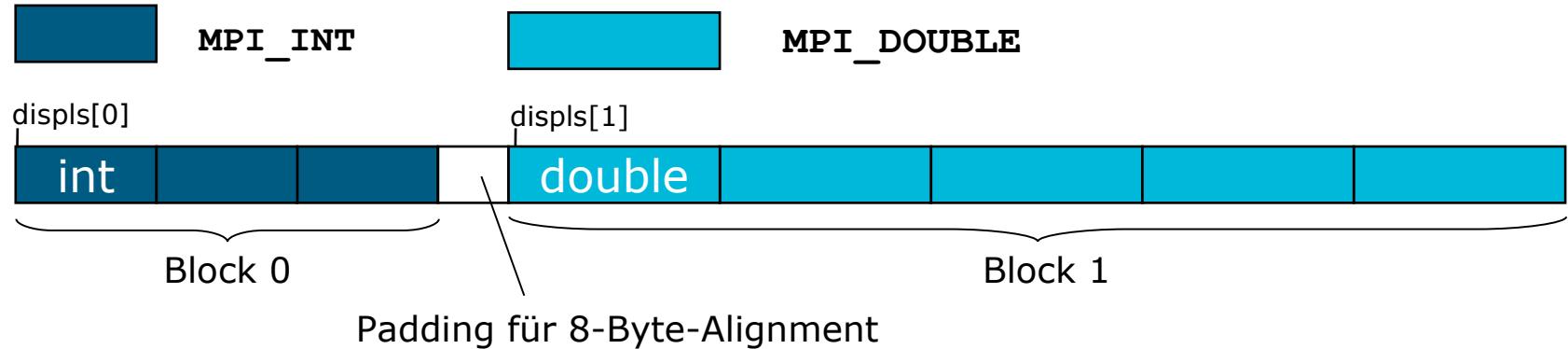
Inout:

count: Anzahl der Blöcke (nichtnegativ), gibt außerdem die Anzahl der Einträge in blocklengths, displs und oldtypes an
blocklengths: Anzahl der Elemente pro Block (Vektor, nichtnegativ)
displs: Anzahl der Bytes zwischen den Blockanfängen (Vektor)
oldtypes: Datentypen der Elemente jedes Blocks

Output:

newtype: neuer Datentyp

Beispiel: Struct



```
int count=2;
int blocklengths[]={3,5};
MPI_Aint displs[]={0,0};
MPI_Datatype oldtypes[]={MPI_INT, MPI_DOUBLE};
MPI_Datatype newtype;
...
displs[1] = ...;
MPI_Type_create_struct( count, blocklengths, displs, oldtypes,
&newtype);
```

Abgeleiteter Datentyp Subarray

C

```
int MPI_Type_create_subarray( int ndims, const int sizes[],  
                             const int subsizes[], const int starts[], int order,  
                             MPI_Datatype oldtype, MPI_Datatype* newtype);
```

Fortran

```
MPI_Type_create_subarray( ndims, sizes, subsizes, starts, order,  
                           oldtype, newtype, ierror )  
  
  INTEGER, INTENT(IN) :: ndims, sizes(ndims), subsizes(ndims),  
                       starts(ndims), order  
  TYPE(MPI_Datatype), INTENT(IN) :: oldtype  
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Erzeugt ein n-dimensionales Subarray eines n-dimensionalen Arrays

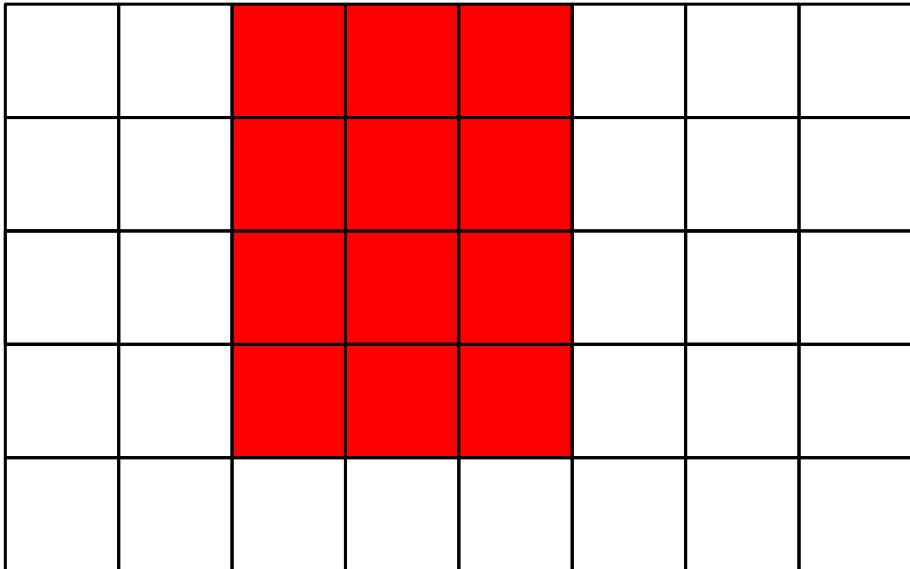
Inout:

ndims: Anzahl der Dimensionen des Arrays
sizes: Anzahl der Elemente des ganzen Arrays in jeder Dimension
(Vektor, positiv)
subsizes: Anzahl der Elemente des Subarrays in jeder Dimension (Vektor, positiv)
starts: Startkoordinaten des Subarrays in jeder Dimension (Vektor, nicht-negativ)
order: `MPI_ORDER_C` oder `MPI_ORDER_FORTRAN`
oldtype: Datentyp der Elemente

Output:

newtype: neuer Datentyp

Beispiel: Subarray



```
int ndims=2;
int sizes={5,8};
int subsizes={4,3};
int starts={0,2};
MPI_Datatype newtype;
```

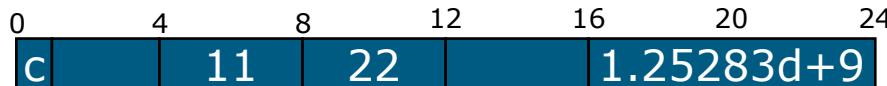
```
MPI_Type_create_subarray( ndims, sizes, subsizes, starts,
                           MPI_ORDER_C, MPI_INT, &newtype);
```

Erstellen eines abgeleiteten Datentyps: Type Map

- Datentypen werden durch ihre Type Map beschrieben
 - eine Liste von Basis-Datentypen
 - Liste von Offsets relative zum Beginn des Datentyps (displacement)Offsets können positiv, Null oder negativ sein

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

- Beispiel:



basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

Rechnen mit Adressen

Neu in
MPI 3.1

- Aus Portabilitätsgründen sollten Berechnungen mit absoluten Adressen nicht mit intrinsischen Operatoren („+“ oder „-“) durchgeführt werden

C `MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp);`

Fortran `MPI_Aint_add(base, disp)`

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp

- neue absolute Adresse := absolute Adresse + relatives Displacement

C `MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2);`

Fortran `MPI_Aint_diff(addr1, addr2)`

INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2

- relatives Displacement := absolute Adresse 1 - absolute Adresse 2

Inout:

base:	Basis-Adresse
disp:	Displacement
addr1:	Adresse des Minuenden
addr2:	Adresse des Subtrahenden

Berechnung des Displacements

C

```
int MPI_Get_address( const void* location, MPI_Aint* address );
```

Fortran

```
MPI_Get_address( location, address, ierror )  
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location  
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die absolute Adresse eines Speicherobjektes

Input:

location: Speicherobjekt

Output:

address: absolute Adresse des Speicherobjektes

Beispiel:



```
MPI_Aint displs[]={0,0};  
MPI_Aint addr0, addr1;  
...  
MPI_Get_address(&buffer.i[0], &addr0);  
MPI_Get_address(&buffer.d[0], &addr1);  
displs[1] = MPI_Aint_diff(addr1, addr0);  
/* MPI 3.0 und früher: displs[1] = addr1 - addr0; */
```

Commit und Free

C

```
int MPI_Type_commit( MPI_Datatype* datatype );
```

Fortran

```
MPI_Type_commit( datatype, ierror )
```

```
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- der Datentyp muss vor dem ersten Gebrauch bekannt gemacht werden
- MPI_Type_commit meldet den abgeleiteten Datentyp datatype an

Inout:

datatype: Datentyp, der bekannt gemacht werden soll

C

```
int MPI_Type_free( MPI_Datatype* datatype );
```

Fortran

```
MPI_Type_free( datatype, ierror )
```

```
TYPE(MPI_Datatype), INTENT(INOUT) :: datatype  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

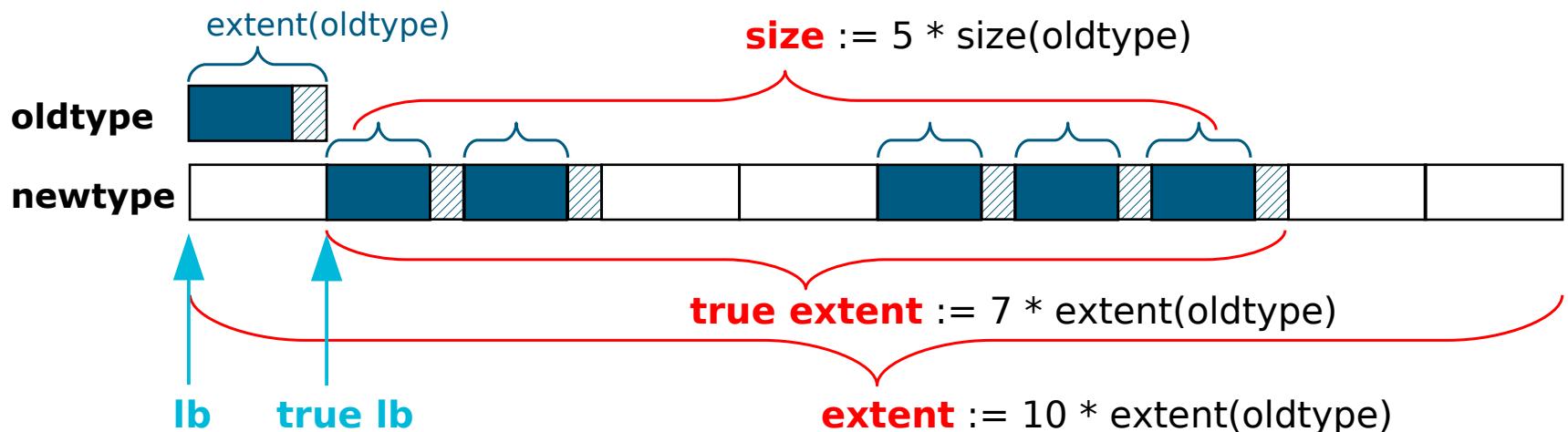
- markiert den abgeleiteten Datentyp datatype als deallokiert
- nach dem Deallokieren kann der Datentyp nicht mehr verwendet werden

Inout:

datatype: Datentyp, der deallokiert werden soll

Definition Size und Extent

- **Size** = Anzahl der Bytes, die transferiert werden (Größe der Daten ohne Lücken)
- **Extent** = Abstand vom ersten bis zum letzten Byte (Größe inklusive aller Lücken)
- **True Extent** = Abstand vom ersten bis zum letzten Byte (ohne Lücken am Anfang und Ende)
- Basis Datentypen: size = extent = number of bytes used by the compiler



Abfrage von Size

C `int MPI_Type_size(MPI_Datatype datatype, int* size);`

Fortran `MPI_Type_size(datatype, size, ierror)`

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die Größe der Einträge eines Datentyps in Bytes
- entspricht der Größe einer Nachricht, die mit diesem Datentyp kreiert werden würde

Input:

`datatype`: Datentyp

Output:

`size`: Größe der Einträge des Datentyps in Bytes

! Der `sizeof()`-Operator liefert lediglich die Größe des MPI-Handels, nicht die Größe des Datentyps.

Abfrage von Extent

C

```
int MPI_Type_get_extent( MPI_Datatype datatype, MPI_Aint* lb,  
                        MPI_Aint* extent );
```

Fortran

```
MPI_Type_get_extent( datatype, lb, extent, ierror )
```

```
  TYPE(MPI_Datatype), INTENT(IN) :: datatype  
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die untere Schranke und Ausdehnung eines Datentyps in Bytes
- Extent = Abstand vom ersten bis zum letzten Byte (inklusive aller Lücken)

Input:

datatype: Datentyp

Output:

lb: untere Schranke des Datentyps

extent: Ausdehnung des Datentyps in Bytes

Abfrage von True Extent

C

```
int MPI_Type_get_true_extent( MPI_Datatype datatype,
                             MPI_Aint* true_lb, MPI_Aint* true_extent );
```

Fortran

```
MPI_Type_get_true_extent( datatype, true_lb, true_extent,
                           ierror )
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die untere Schranke und Ausdehnung eines Datentyps in Bytes
- True Extent = Abstand vom ersten bis zum letzten Byte (ohne Lücken an Anfang und Ende)

Input:

datatype: Datentyp

Output:

true_lb: untere Schranke des Datentyps

true_extent: Ausdehnung des Datentyps in Bytes

Beispiel: Gemischte Datentypen

```
struct buff_layout {  
    int    i[3];  
    double d[5];  
} buffer;
```

Compiler

```
types[0] = MPI_INT;  
blocklengths[0] = 3;  
displacements[0] = 0;  
types[1] = MPI_DOUBLE;  
blocklengths[1] = 5;  
displacements[1] = ...;  
MPI_Type_create_struct(2,  
                      blocklengths,  
                      displacements, types,  
                      &buff_datatype);  
MPI_Type_commit(&buff_datatype);
```

`MPI_Send(&buffer, 1, buff_datatype, ...)`



Beispiel: Gemischte Datentypen (Implementierung)

```
struct buff_layout {
    int    i[3];
    double d[5];
} buffer;
...
MPI_Datatype types[] = {MPI_INT, MPI_DOUBLE};
MPI_Datatype newtype;
int lengths[] = {3, 5};
MPI_Aint displs[] = {0, 0};
MPI_Aint addr0, addr1;
...
MPI_Get_address(&buffer.i[0], &addr0);
MPI_Get_address(&buffer.d[0], &addr1);
displs[1] = MPI_Aint_diff(addr1, addr0);
MPI_Type_create_struct(2, lengths, displs, types, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(&buffer, 1, newtype, dest, 4711, MPI_COMM_WORLD);
...
MPI_Type_free(&newtype);
```

Weiteres zu Datentypen

- **Performance**
 - abgeleitete Datentypen erlauben das Versenden von weniger Nachrichten oder kleinerer Datenmengen
 - Performance ist abhängig von der MPI Implementierung
 - weniger Kopieraktionen notwendig
 - abgeleitete Datentypen können MPI-IO effizienter machen
- **weitere Funktionen:**
 - **`MPI_Type_create_resized`**: Anpassung des Extents
 - **`MPI_Type_create_indexed_block`**: wie `MPI_Type_indexed` nur mit konstanter Blocklänge
 - **`MPI_Type_create_darray`**: Teilmatrix für eine kartesische Prozessortopologie

Übung

Übung 6:

Abgeleitete Datentypen



Einführung in die Parallelprogrammierung

MPI Teil 6: Gruppen, Kontexte, Kommunikatoren

Annika Hagemeier

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich

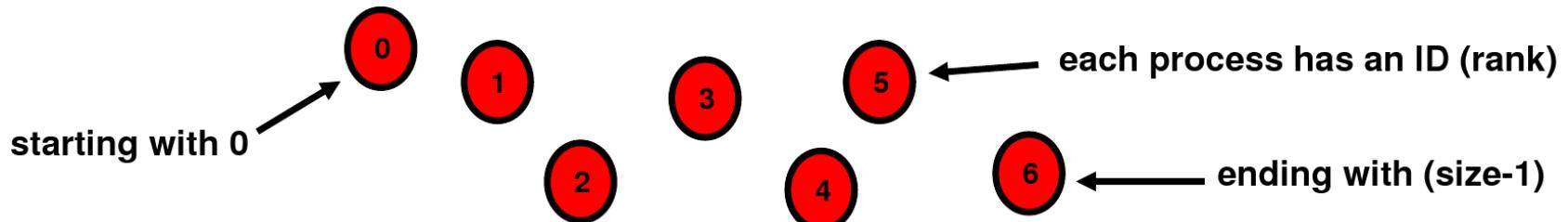


Inhalt

- Gruppen
- Kontext
- Kommunikator
- Kommunikator und Kontext
- Gruppen
 - Management
 - Konstruktoren
- Kommunikatoren
 - Management
 - Konstruktoren
- MPI_Comm_split

Gruppen

- eine Gruppe ist eine **geordnete Menge von Prozessen**
- ein Prozess kann zu einer oder mehreren Gruppen gehören
- jeder Prozess besitzt einen **eindeutigen Rang** (beginnend bei 0) innerhalb der Gruppe
- definiert den Raum für Prozessnamen (Ränge) für die Punkt-zu-Punkt-Kommunikation und legt fest, welche Prozesse in eine kollektive Operation einbezogen werden
- eine Gruppe ist ein **dynamisches Objekt**, welches während der Programmausführung erzeugt und gelöscht werden kann
- **vordefinierte Gruppen:**
 - MPI_GROUP_EMPTY → eine leere Gruppe
 - MPI_GROUP_NULL → Wert für eine ungültige Gruppe

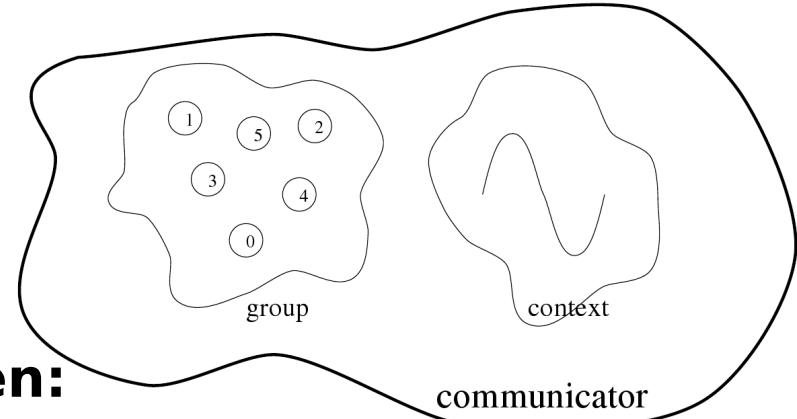


Kontext

- ein Kontext definiert die **Eigenschaft** eines Kommunikators
- ermöglicht **Aufteilung/Trennung** des Kommunikationsraums (stellt einen sicheren Tag-Raum bereit, welcher von jeglicher Kommunikation außerhalb nicht gestört werden kann)
- Nachrichten, die in einem Kontext gesendet werden, können nicht in einem anderen Kontext empfangen werden (Analogie: Radiofrequenzen)
- ein Kontext ist kein explizites MPI-Objekt, sondern lediglich ein **Teil eines Kommunikators**
- ein Kontext kann als ein zusätzliches Tag zur Kennzeichnung von Nachrichten angesehen werden
- **Motivation:**
 - parallele Bibliotheken
 - unerledigte Punkt-zu-Punkt Kommunikation

Kommunikator

- die Begriffe **Gruppe** und **Kontext**, werden zu einem Objekt zusammengefügt: dem **Kommunikator**
- jegliche Art von Kommunikation in MPI ist **nur** über einen Kommunikator möglich
- kann neben Prozessgruppe und Kontext noch Attribute und einen Errorhandler besitzen
- ein Kommunikator bietet eine „private“ Kommunikationsdomäne
- ein Kommunikator ist ein dynamisches Objekt, das während der Ausführung erzeugt und gelöscht werden kann
- **vordefinierte Kommunikatoren:**
 - `MPI_COMM_WORLD` → enthält alle MPI-Prozesse
 - `MPI_COMM_SELF` → enthält nur den lokalen Prozess
 - `MPI_COMM_NULL` → Wert für ungültigen Kommunikator



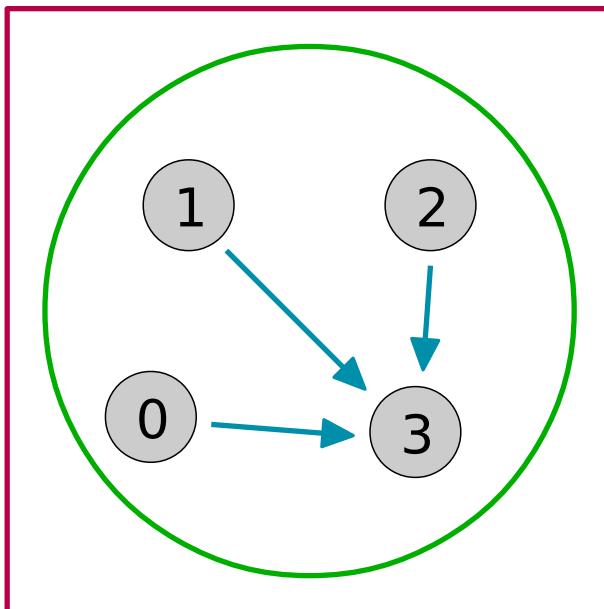
Kommunikator und Kontext

- untrennbar, da ein Kommunikator lediglich ein Handle für einen Kontext ist
- jeder Kommunikator hat einen eindeutigen Kontext
- jeder Kontext hat einen eindeutigen Kommunikator
- Kommunikator: zentrales Objekt der Kommunikation in MPI
 - jede Funktion zur MPI-Kommunikation benötigt einen Kommunikator als Argument
 - ⇒ jede MPI-Kommunikation erfolgt in einem bestimmten Kontext
 - ⇒ zwei MPI-Prozesse können nur kommunizieren wenn sie sich im selben Kontext befinden
 - ⇒ Nachrichten, die in einem Kontext gesendet werden, können nicht in einem anderen Kontext empfangen werden

Intra- und Inter-Kommunikatoren

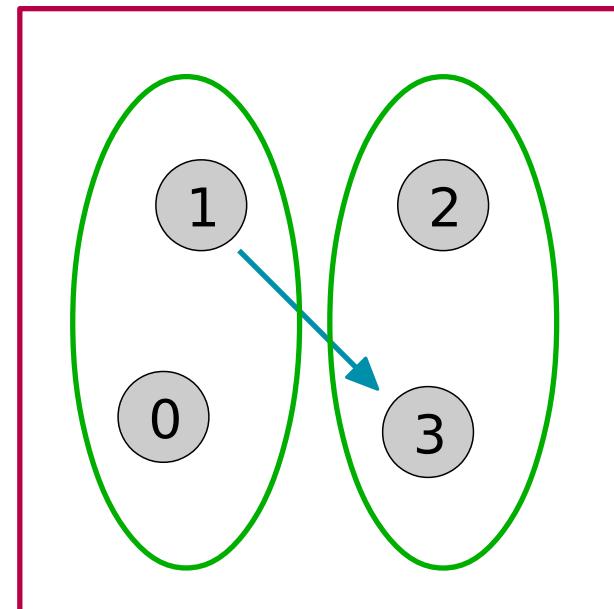
Intra-Kommunikator:

kommuniziert **innerhalb** einer Gruppe



Inter-Kommunikator:

kommuniziert **zwischen** zwei Gruppen (Domänen)



① Prozess

○ Gruppe

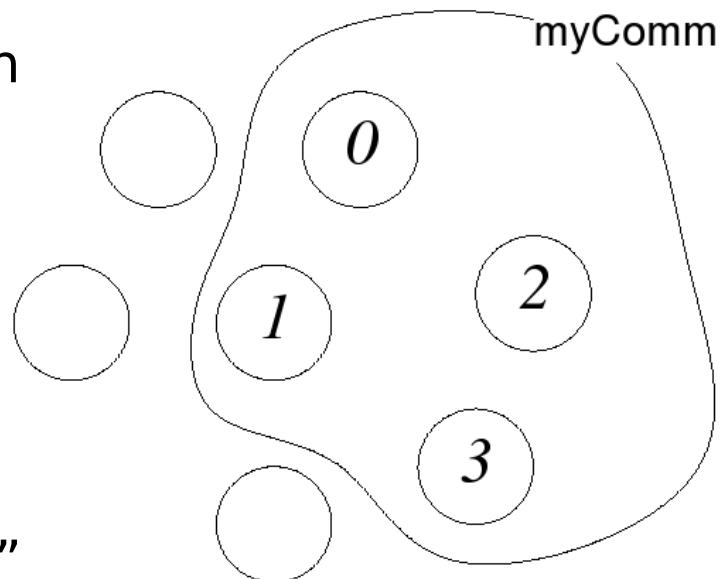
□ Kommunikator

→ Kommunikation

Werden hier nur Intra-Kommunikatoren behandeln!

Wozu verschiedene Kommunikatoren?

- bessere Gruppierung der Prozesse
- Separation der Kommunikation in Bibliotheken
 - **parallele Bibliotheken** haben oft ihr eigenes internes Kommunikationsmuster, durch Kontexte können Funktionen aufgerufen werden ohne vorherige Synchronisation der Prozesse und ohne dass die Kommunikation in „anderen“ Kommunikatoren gestört wird



Gruppenmanagement (1)

- Gruppen werden stets aus bereits existierenden Gruppen zusammengestellt
 - als Basisgruppe kann jede Gruppe, die einem Kommunikator (z.B. MPI_COMM_WORLD) zugrunde liegt, genutzt werden
- Gruppenkonstruktoren arbeiten **lokal** auf dem rufenden Prozess
 - für diesen Vorgang ist keine Kommunikation zwischen Prozessen notwendig, da es auf der Basis von Gruppen keine gemeinsamen Operationen gibt
- jeder Prozess kann beliebige Gruppen für sich definieren, auch dann, wenn er selbst nicht in der Gruppe enthalten ist

Gruppenmanagement (2)

- Gruppen können unabhängig von Kommunikatoren manipuliert werden, dienen aber nur der Kommunikation

```
int MPI_Group_size( MPI_Group group, int* size );
```

- liefert die Größe einer Gruppe

```
int MPI_Group_rank( MPI_Group group, int* rank );
```

- liefert den Rang eines Prozesses in einer Gruppe

```
int MPI_Group_translate_ranks( MPI_Group group1, int n,
                               int* ranks1, MPI_Group group2, int* ranks2 );
```

- liefert zu den Prozessen mit den Rängen `ranks1` in `group1` die entsprechenden Ränge in `group2`

```
int MPI_Group_compare( MPI_Group group1, MPI_Group group2,
                       int* result );
```

- Vergleicht zwei Gruppen miteinander
- Ergebnisse: `MPI_IDENT`, `MPI_SIMILAR`, `MPI_UNEQUAL`

Gruppenkonstruktoren (1)

- mit Hilfe der Gruppenkonstruktoren lassen sich neue Gruppen aus bereits bestehenden Gruppen erzeugen

```
int MPI_Comm_group( MPI_Comm comm, MPI_Group* group );
```

- liefert die zu `comm` gehörende Gruppe

```
int MPI_Group_union( MPI_Group group1, MPI_Group group2,  
                     MPI_Group* newgroup );
```

- vereint zwei Gruppen zu einer Gruppe

```
int MPI_Group_intersection( MPI_Group group1,  
                           MPI_Group group2, MPI_Group* newgroup );
```

- erzeugt aus dem Schnitt der beiden Gruppen eine neue Gruppe

```
int MPI_Group_difference( MPI_Group group1,  
                         MPI_Group group2, MPI_Group* newgroup );
```

- erzeugt aus der Differenz der beiden Gruppen eine neue Gruppe

Gruppenkonstruktoren (2)

```
int MPI_Group_incl( MPI_Group group, int n, int* ranks,  
                    MPI_Group* newgroup );
```

- fasst die ersten n in ranks angegebenen Prozesse in group zu einer neuen Gruppe zusammen

```
int MPI_Group_excl( MPI_Group group, int n, int* ranks,  
                    MPI_Group* newgroup );
```

- newgroup ergibt sich, indem die ersten n in ranks angegebenen Prozesse aus der Gruppe entfernt werden

```
int MPI_Group_range_incl( MPI_Group group, int n,  
                          int ranges[][3], MPI_Group* newgroup );
```

- fasst die in ranges spezifizierten Prozesse aus group zu einer neuen Gruppe zusammen
- ranges ist ein Array von Integer-Tripeln der Form (erster Rang, letzter Rang, Stride)

Gruppenkonstruktoren (3)

```
int MPI_Group_range_excl( MPI_Group group, int n,
                           int ranges[][][3], MPI_Group* newgroup );
```

- `newgroup` ergibt sich, indem die mit `ranges` spezifizierten Prozesse aus der Gruppe entfernt werden
- `ranges` ist ein Array von Integer-Tripeln der Form (erster Rang, letzter Rang, Stride)

```
int MPI_Group_free( MPI_Group* group );
```

- deallokiert ein Gruppenobjekt
- das Gruppenhandle wird auf `MPI_GROUP_NULL` gesetzt

Kommunikatormanagement (1)

- das Erzeugen eines Kommunikators ist eine **globale** Operation
 - jeder Prozess, der Mitglied in einem Kommunikator werden soll, muss den Konstruktor aufrufen, und erhält somit ein Handle für den neuen Kommunikator
- MPI erlaubt, dass auch Prozesse, die nicht Mitglied werden, den Konstruktor rufen
 - sie erhalten anstelle eines gültigen Handles `MPI_COMM_NULL` zurück

Kommunikatormanagement (2)

- betrachten hier nur Intra-Kommunikatoren
- Zugriff auf Kommunikatorinformationen ist lokal

```
int MPI_Comm_size( MPI_Comm comm, int* size );
```

- liefert die Anzahl der Prozesse in einem Kommunikator

```
int MPI_Comm_rank( MPI_Comm comm, int* rank );
```

- liefert den Rang eines Prozesses in einem Kommunikator

```
int MPI_Comm_compare( MPI_Comm comm1, MPI_Comm comm2,  
                      int* result );
```

- Vergleicht zwei Kommunikatoren miteinander
- Ergebnisse:

- MPI_IDENT: Kommunikatoren sind identisch
- MPI_CONGRUENT: Kommunikatoren unterscheiden sich nur im Kontext
- MPI_SIMILAR: die Prozessoren sind identisch, unterscheiden sich aber in der Reihenfolge der Ränge
- MPI_UNEQUAL: in allen anderen Fällen

Kommunikatorkonstruktoren

- Operationen zum Anlegen von Kommunikatoren sind **kollektiv!**

```
int MPI_Comm_dup( MPI_Comm comm, MPI_Comm* newcomm );
```

- kopiert comm nach newcomm

```
int MPI_Comm_create( MPI_Comm comm, MPI_Group group,
                     MPI_Comm* newcomm );
```

- erzeugt einen neuen Kommunikator mit den Prozessen in group
(alle Prozesse in comm müssen mitmachen)

```
int MPI_Comm_split( MPI_Comm comm, int color, int key,
                    MPI_Comm* newcomm );
```

- erzeugt disjunkte Untergruppen von comm
- Prozesse mit der gleichen color gehören zur selben Untergruppe
- key bestimmt die Ordnung innerhalb der Gruppe

```
int MPI_Comm_free( MPI_Comm* comm );
```

- deallokiert einen Kommunikator
- das Kommunikatorhandle wird auf MPI_COMM_NULL gesetzt

Erzeugen von Kommunikatoren

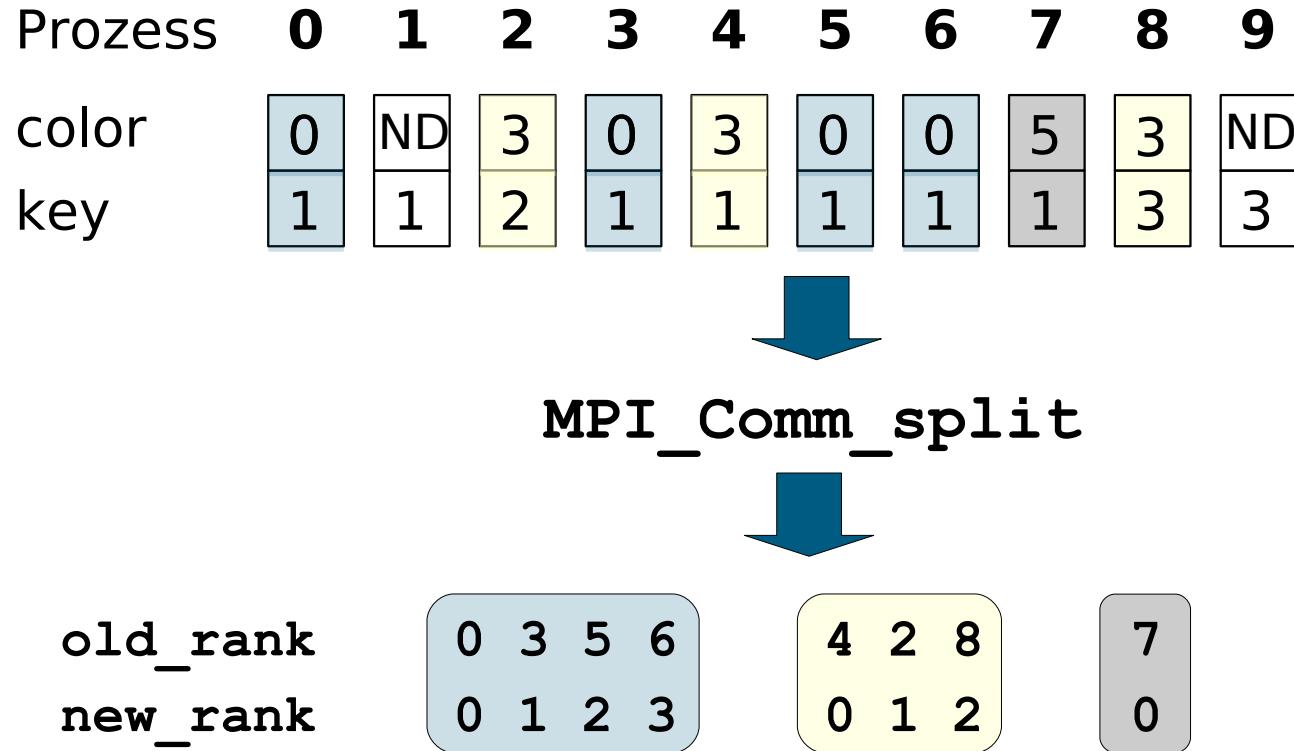
1. mit Hilfe von Gruppen:

- eine Prozessgruppe aus einem bestehenden Kommunikator extrahieren mit
 - `MPI_Comm_group`
- die Gruppe mit Hilfe von MPI-Gruppenkonstruktoren modifizieren
 - z.B. `MPI_Group_incl`, `MPI_Group_excl`
- einen neuen Kommunikator aus der neuen Gruppe erzeugen mit
 - `MPI_Comm_create`

2. aus einem bestehenden Kommunikator heraus:

- einen Kommunikator kopieren mit `MPI_Comm_dup`
- einen Kommunikator in eine Gruppe von neuen Kommunikatoren aufsplitten mit `MPI_Comm_split`

Beispiel: MPI_Comm_split



- ND = MPI_UNDEFINED
- Prozess 1 und 9 erhalten als Ergebnis MPI_COMM_NULL

Übung 7:

Gruppen, Kontexte, Kommunikatoren



Einführung in die Parallelprogrammierung

MPI Teil 7: Virtuelle Topologien

Annika Hagemeier

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich



Inhalt

- Prozess-Topologien
- Kartesische Topologien
 - Dimensionen bestimmen
 - Kartesische Topologie erzeugen
 - Informationen abfragen
 - kartesischer Shift-Operator
- Graph-Topologien
 - Graph-Topologie erzeugen
 - Nachbar-Operatoren
- Weitere Operatoren

Prozess Topologien (1)

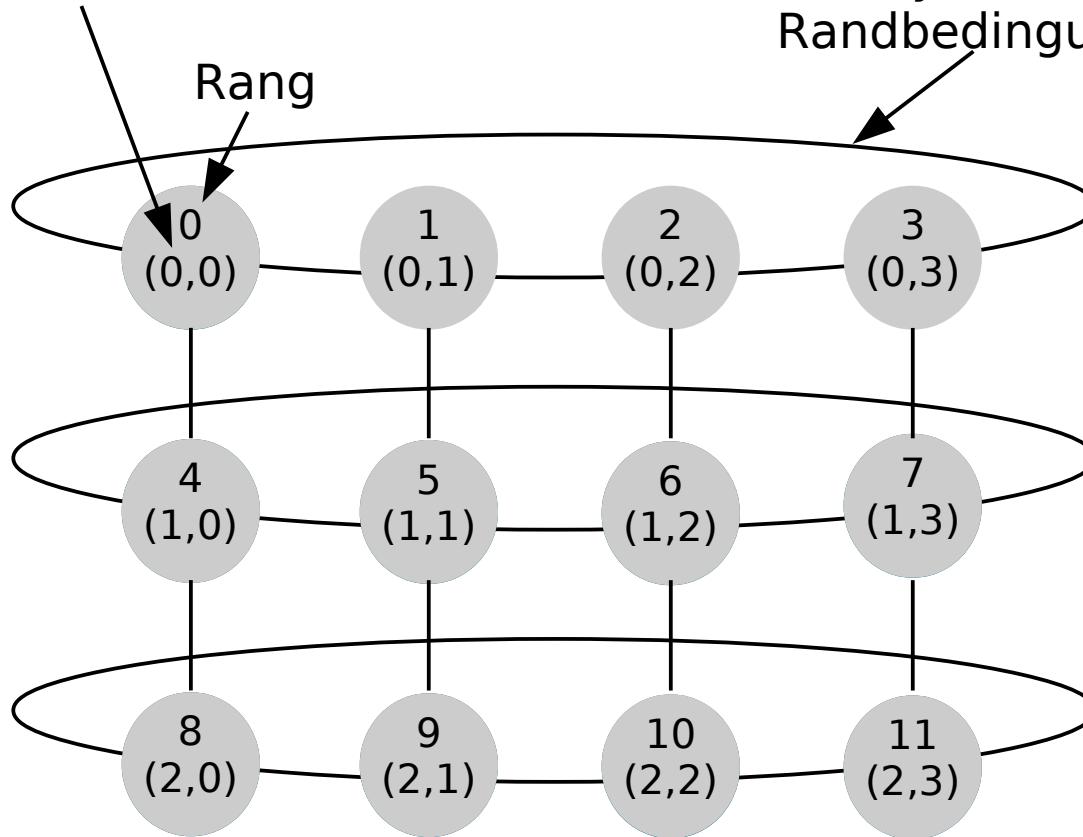
- MPI unterstützt Prozess-Topologien auf Graph-Basis und kartesische Topologien
- Topologie kann **virtuell** sein, aber auch der **Topologie der Hardware** entsprechen
- **virtuelle Topologie**: Benennung/Nummerierung der Prozesse, so dass sie mit der Kommunikationsstruktur übereinstimmen
- **Ziele:**
 - effizientes Programmieren
 - Ausnutzung der Topologie der verwendeten Hardware (Hypercube, 3D-Torus,...)
 - komfortable Benennung der Prozesse, passend zur Kommunikationsstruktur
 - Vereinfachung des Quellcodes
- Kommunikation ist auch außerhalb der Topologie möglich, d.h. jeder Prozess kann mit jedem kommunizieren.

Beispiel zweidimensionaler Zylinder

kartesische Prozesskoordinate

Rang

zyklische
Randbedingungen



Prozess Topologien (2)

- eine virtuelle Topologie markiert die hauptsächliche Kommunikationsstruktur innerhalb eines Kommunikators
- jeder Prozess kann jedoch trotzdem noch mit jedem anderen Prozess kommunizieren
- eine virtuelle Topologie ist mit einem Kommunikator verbunden:
 - wird eine Topologie auf einem bestehenden Kommunikator erzeugt, wird automatisch ein neuer Kommunikator erzeugt und zurückgegeben
 - zur Verwendung der Topologie muss der neue Kommunikator benutzt werden

Kartesische Topologie

- die Prozesse werden in einem n -dimensionalen Quader angeordnet, so dass ein Prozess mit seinen jeweiligen Nachbarn in jeder Dimension direkt verbunden ist
- **Erzeugen von kartesischen Topologien:**
 - vor der Benutzung muss die Topologie erzeugt werden
 - für kartesische Topologien gibt es die Hilfsfunktion `MPI_Dims_create`, sie versucht eine gewisse Anzahl Prozesse bestmöglichst auf eine gegebene Anzahl von Dimensionen aufzuteilen
 - eine kartesische Topologie wird mit der Funktion `MPI_Cart_create` erzeugt
 - die beteiligten Prozesse werden einem neuen Kommunikator zugeordnet, wobei die beteiligten Prozesse nicht explizit angegeben werden können, sondern nur ein Kommunikator, aus dem der neue Kommunikator erzeugt wird

Dimensionen bestimmen (1)

C `int MPI_Dims_create(int nnodes, int ndims, int* dims);`

Fortran `MPI_Dims_create(nnodes, ndims, dims, ierror)`

```
INTEGER, INTENT(IN) :: nnodes, ndims  
INTEGER, INTENT(INOUT) :: dims(ndims)  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert eine ausgeglichene Aufteilung einer Anzahl von Prozessen auf eine kartesisches Struktur

Input:

`nnodes`: Anzahl der Prozesse, die verteilt werden sollen

`ndims`: Anzahl der Dimensionen der kartesischen Struktur

Inout:

`dims`: Array der Größe `ndims`, welches die Anzahl der Prozesse in jeder Dimension angibt

Dimensionen bestimmen (2)

Bemerkungen zu `dims`:

- `dims[n]` = Anzahl der Prozesse in der Dimension n
- das Feld kann als `dims`-Argument in `MPI_Cart_create()` verwendet werden.
- in `dims` werden nur diejenigen Elemente neu berechnet, die vor dem Aufruf Null waren
- Werte > 0 werden als gesetzt betrachtet und bleiben unverändert, Einträge < 0 sind fehlerhaft.

Beispiel:

<code>dims vorher</code>	Funktionsaufruf	<code>dims nachher</code>
(0, 0)	<code>MPI_Dims_create(6, 2, dims)</code>	(3, 2)
(0, 0)	<code>MPI_Dims_create(7, 2, dims)</code>	(7, 1)
(0, 3, 0)	<code>MPI_Dims_create(6, 3, dims)</code>	(2, 3, 1)
(0, 3, 0)	<code>MPI_Dims_create(7, 3, dims)</code>	Fehler!

Kartesischen Kommunikator erzeugen

C

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int* dims,  
                    const int* periods, int reorder, MPI_Comm* comm_cart);
```

Fortran

```
MPI_Cart_create( comm_old, ndims, dims, periods, reorder,  
                  comm_cart, ierror )  
  
TYPE(MPI_Comm), INTENT(IN) :: comm_old  
INTEGER, INTENT(IN) :: ndims, dims(ndims)  
LOGICAL, INTENT(IN) :: periods(ndims), reorder  
TYPE(MPI_Comm), INTENT(OUT) :: comm_cart  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- erzeugt einen neuen Kommunikator mit kartesischer Topologie (z. B. Gitter, Ring, Hypercube)

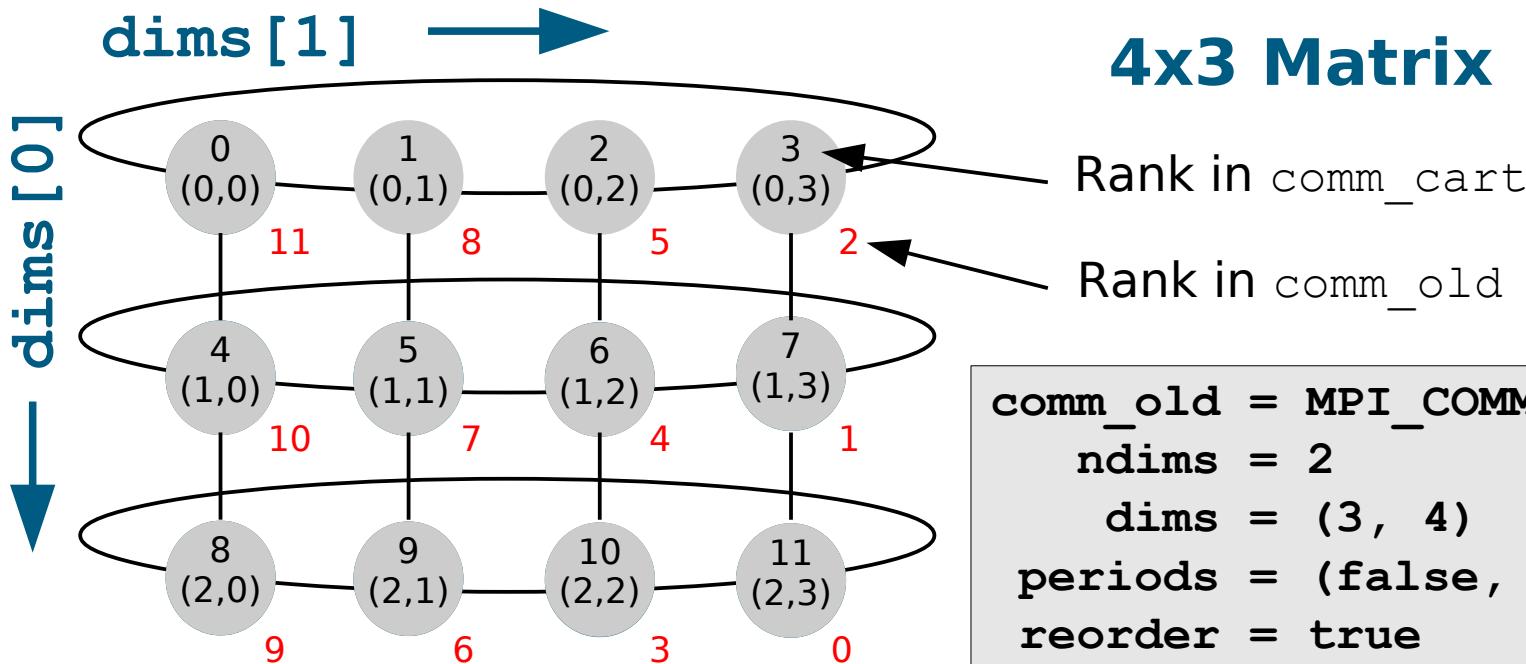
Input:

comm_old: Eingabekommunikator
ndims: Anzahl der Dimensionen der kartesischen Struktur
dims: Array der Größe ndims, welches die Anzahl der Prozesse in jeder Dimension angibt
periods: logisches Array der Größe ndims, gibt an, ob die Struktur in der Dimension n periodisch (true) oder nicht periodisch (false) ist
reorder: gibt an, ob MPI die Ränge neu vergeben darf (true) oder ob die Ordnung von comm_old beizubehalten ist (false)

Output:

comm_cart: neuer Kommunikator mit kartesischer Topologie

Beispiel zweidimensionaler Zylinder



- Ränge in `comm_old` und `comm_cart` können unterschiedlich sein, da `reorder=true`
- Die Nummerierung der Ränge erfolgt zeilenweise (wie in C)
- Achtung: die Dimension 0 entspricht dem Element **ganz rechts** im Koordinaten-Tupel

Informationen über kart. Topologien abfragen

```
int MPI_Cartdim_get ( MPI_Comm comm, int *ndim );
```

- Anzahl der Dimensionen einer kartesischen Topologie auf einem Kommunikator bestimmen

```
int MPI_Cart_get ( MPI_Comm comm, int maxdim, int *dims,  
                   int *periods, int *coords )
```

- Topologieinformation, mit welcher eine kartesische Topologie erzeugt wurde, bestimmen

```
int MPI_Cart_rank( MPI_Comm comm, const int *coords,  
                   int *rank )
```

- Rang eines Prozesses bestimmen, dessen kartesische Koordinaten bekannt sind

```
int MPI_Cart_coords( MPI_Comm comm, int rank,  
                     int maxdims, int *coords )
```

- die logischen Koordinaten eines Prozesses in einer kartesischen Topologie aus dem Rang des Prozesses ermitteln

Kartesischer Shift-Operator

C

```
int MPI_Cart_shift( MPI_Comm comm, int direction, int disp,  
                    int* rank_source, int* rank_dest);
```

Fortran

```
MPI_Cart_shift( comm, direction, disp, rank_source,  
                 rank_dest, ierror )
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm_old  
INTEGER, INTENT(IN) :: direction, disp  
INTEGER, INTENT(OUT) :: rank_source, rank_dest  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die Ränge der in Richtung `direction` `disp` entfernten Prozesse zurück (nützlich z.B. bei Gebietszerlegung)
- leichte Ermittlung der Nachbarn bei kartesischen Kommunikatoren

Input:

`comm`: Kommunikator mit kartesischer Struktur
`direction`: Dimension, in welcher Daten transportiert werden sollen
`disp`: gibt an, wie weit verschoben werden soll (`disp < 0` : abwärts verschieben; `disp > 0` : aufwärts verschieben)

Output:

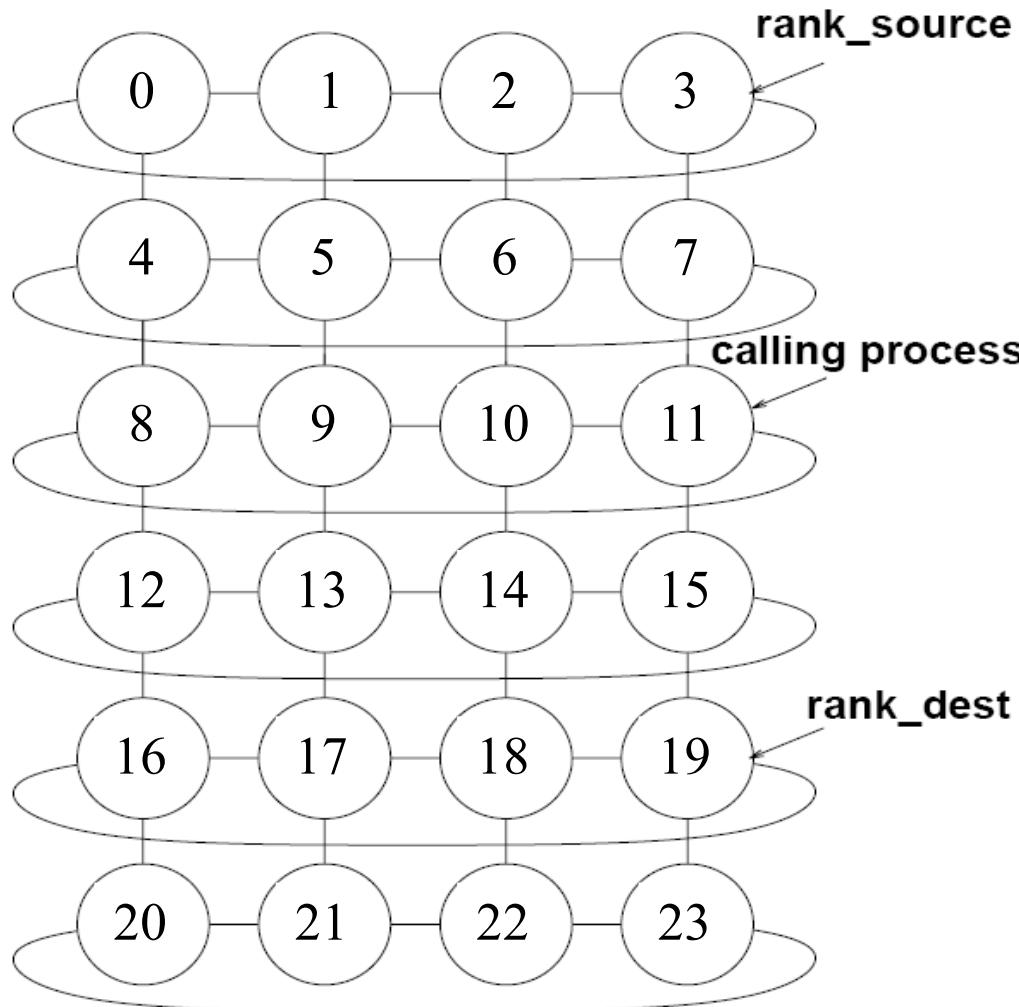
`rank_source`: Prozess, von dem der rufende Prozess Daten empfängt
`rank_dest`: Prozess, an den der rufende Prozess Daten sendet

Kartesischer Shift-Operator

Bemerkung:

- Ist die Dimension, in der verschoben wird, nicht periodisch, so liefert `MPI_Cart_shift()` für Quell- und Zielprozesse, die außerhalb des Intervalls 0 bis n liegen, `MPI_PROC_NULL`

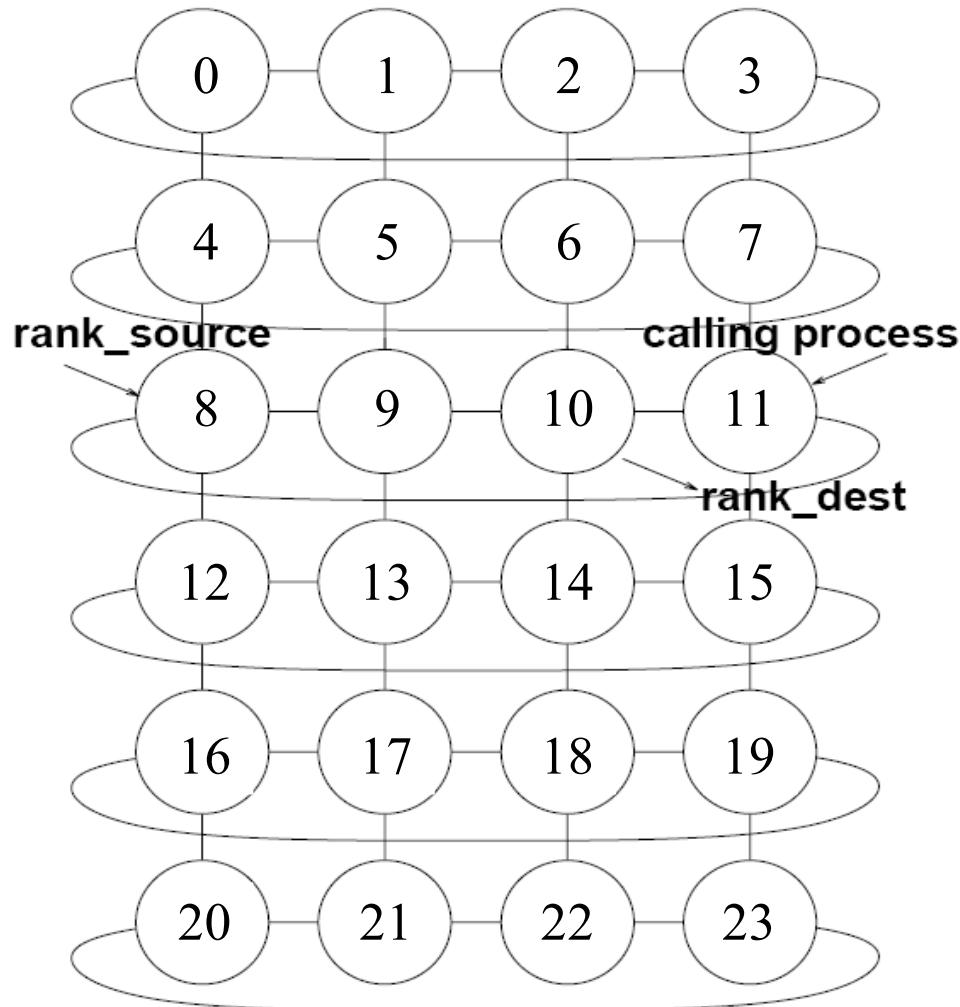
Kartesischer Shift-Operator (Beispiel 1)



MPI_Cart_shift
wird von Prozess 11
aufgerufen:

- direction=0
- disp=2
- rank_source=3
- rank_dest=19

Kartesischer Shift-Operator (Beispiel 2)



MPI_Cart_shift
wird von Prozess 11
aufgerufen:

- direction=1
- disp=-1
- rank_source=8
- rank_dest=10

Graph-Topologie

- die Prozesse werden auf die Knoten eines Graphen abgebildet
- die Kanten stellen virtuelle Verbindungen zwischen den Knoten dar, über sie läuft die Kommunikation
- **Erzeugen von Graph-Topologien:**
 - vor der Benutzung muss die Topologie erzeugt werden
 - eine Graph-Topologie wird mit der Funktion **MPI_Graph_create** erzeugt
 - die beteiligten Prozesse werden einem neuen Kommunikator zugeordnet, wobei die beteiligten Prozesse nicht explizit angegeben werden können, sondern nur ein Kommunikator, aus dem der neue Kommunikator erzeugt wird

Graph-Topologie erzeugen (1)

C

```
int MPI_Graph_create( MPI_Comm comm_old, int nnodes, const int*  
                     index, const int* edges, int reorder, MPI_Comm* comm_graph);
```

Fortran

```
MPI_Graph_create( comm_old, nnodes, index, edges, reorder,  
                   comm_graph, ierror )  
  
TYPE(MPI_Comm), INTENT(IN) :: comm_old  
INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)  
LOGICAL, INTENT(IN) :: reorder  
TYPE(MPI_Comm), INTENT(OUT) :: comm_graph  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- erzeugt einen neuen Kommunikator mit Graph-Topologie

Input:

comm_old: Eingabekommunikator
nnodes: Anzahl der Knoten im Graph
index: Array der Größe nnodes, index[i] gibt an, wieviele Nachbarn die ersten i Knoten insgesamt haben
edges: Liste der Nachbarknoten (hat soviele Einträge, wie der Graph Kanten hat), zuerst alle Nachbarn von Prozess 0, dann alle Nachbarn von Prozess 1 usw.
reorder: gibt an, ob MPI die Ränge neu vergeben darf (true) oder ob die Ordnung von comm_old beizubehalten ist (false)

Output:

comm_graph: neuer Kommunikator mit Graph-Topologie

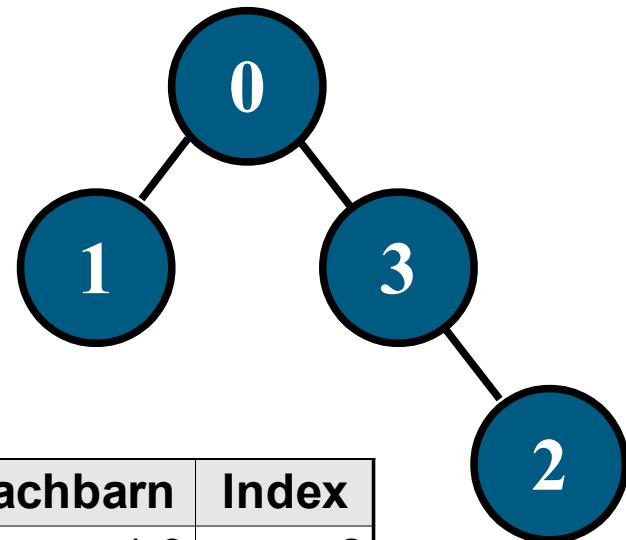
Graph-Topologie erzeugen (2)

Bemerkungen:

- enthält `comm_old` mehr Prozesse als für den Graphen benötigt werden, erhalten überzählige Prozesse in `comm_graph` den Wert `MPI_COMM_NULL` zurück
⇒ sie sind damit nicht im neuen Kommunikator enthalten
- der Funktionsaufruf von `MPI_Graph_create()` führt zu einem Fehler, wenn der Graph mehr Prozesse benötigt als in `comm_old` enthalten sind

Beispiel: Graph-Topologie

```
int nnodes = 4;  
int index[4] = { 2, 3, 4, 6 };  
int edges[6] = { 1,3, 0,  
                 3, 0,2 };  
int reorder = true;  
MPI_Comm comm_graph;
```



Knoten	Nachbarn	Index
0	1,3	2
1	0	3
2	3	4
3	0,2	6

```
MPI_Graph_create(MPI_COMM_WORLD, nnodes, index,  
                  edges, reorder, &comm_graph );
```

Nachbar-Operatoren für Graphen (1)

C

```
int MPI_Graph_neighbors_count( MPI_Comm comm, int rank,
                               int* nneighbors);
```

Fortran

```
MPI_Graph_neighbors_count( comm, rank, nneighbors, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(IN) :: rank
  INTEGER, INTENT(OUT) :: nneighbors
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die Anzahl der Nachbarknoten zu einem Knoten in einer Graph-Topologie

Input:

comm: Kommunikator mit Graph-Topologie
rank: Rang zur Identifikation des Prozesses

Output:

nneighbors: Anzahl der Nachbarknoten des angegebenen Prozesses rank

Nachbar-Operatoren für Graphen (2)

C

```
int MPI_Graph_neighbors( MPI_Comm comm, int rank,  
                        int maxneighbors, int* neighbors);
```

Fortran

```
MPI_Graph_neighbors( comm, rank, maxneighbors, neighbors,  
                      ierror)  
  
TYPE(MPI_Comm), INTENT(IN) :: comm  
INTEGER, INTENT(IN) :: rank, maxneighbors  
INTEGER, INTENT(OUT) :: neighbors(maxneighbors)  
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- liefert die Ränge der Nachbarknoten zu einem Knoten in einer Graph-Topologie

Input:

comm: Kommunikator mit Graph-Topologie
rank: Rang zur Identifikation des Prozesses
maxneighbors: Anzahl der Nachbarknoten des Prozesses rank

Output:

neighbors: Feld der Größe maxneighbors mit den Rängen der Nachbarknoten des Prozesses rank

Weitere Operatoren

Für topologische Kommunikatoren stehen noch weitere Operatoren zur Verfügung, die hier nicht weiter betrachtet werden:

- **Funktionen zum Erzeugen von Topologien:**
 - `MPI_Cart_sub` (Teilstruktur einer kartesischen Topologie)
- **Funktionen zum Abfragen von Topologien:**
 - `MPI_Topo_test` (Topologietyp)
 - `MPI_Graphdims_get` (Zahl der Knoten und Kanten)
 - `MPI_Graph_get` (vollständige Graphinformation)
- **Low-level - Funktionen:** Ermitteln einer "optimalen" Plazierung der Prozesse auf der Hardware
 - `MPI_Cart_map`
 - `MPI_Graph_map`

Übung 8:

Virtuelle Topologien

