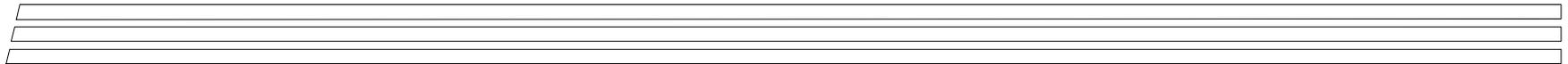


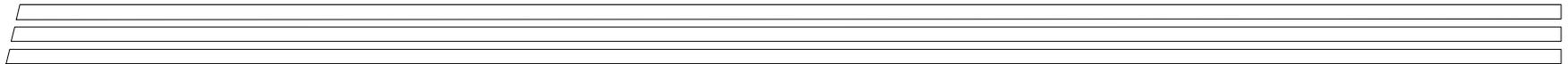
## 7. *MPI - Message Passing Interface*

- **de facto Standard für nachrichten-basierte Programmierung**
  - **für FORTRAN, C und C++**
- => [www.mpi-forum.de](http://www.mpi-forum.de)**



# *MPI Programmiermodell*

- SPMD- Programmstruktur
- 6 Basisfunktionen:
  - `MPI_Init` Initialisierung
  - `MPI_Finalize` Terminierung
  - `MPI_Comm_rank` Prozessid abfragen
  - `MPI_Comm_size` Anzahl Prozesse bestimmen
  - `MPI_Send` Senden
  - `MPI_Recv` Empfangen
- Namenskonventionen:
  - `MPI_` als Präfix für alle Routinen und Konstanten
  - Konstanten in Großbuchstaben
  - FORTRAN nur Großbuchstaben, C Groß- und Kleinbuchstaben



# *Struktur eines MPI Programms in C*

```
#include "mpi.h"
```

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

```
{
```

```
    int my_rank, nprocs
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

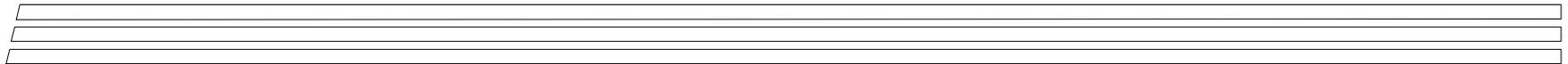
```
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

```
    ...
```

```
    ...
```

```
    MPI_Finalize();
```

```
}
```



# *Initialisierung und Terminierung*

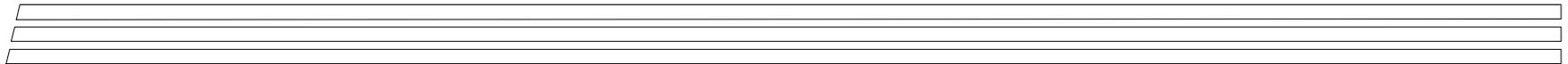
- `MPI_Init (int *argc, char ***argv)`

Adressen der Argumente von main

- initialisiert die MPI-Umgebung
- definiert den sogenannten Kommunikator `MPI_COMM_WORLD`
- Ein Kommunikator definiert
  - eine Gruppe von Prozessen und
  - einen Kommunikationskontext.

`MPI_COMM_WORLD` umfasst alle Prozesse.

- `MPI_Finalize()`
  - beendet MPI.
  - Es sind keine weiteren MPI-Aufrufe möglich.



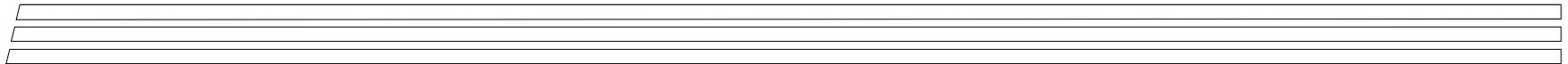


# *Senden*

- **MPI\_Send** (**void \*buf**,  
**int count**,  
**MPI\_Datatype datatype**,  
**int dest**,  
**int tag**,  
**MPI\_Comm comm**)  
} zu übertragende Daten  
Rang des Empfängers  
Kennung  
Kommunikator

ist die Standard-Senderoutine.

„count“ Datenobjekte vom Typ „datatype“ ab Adresse „buf“  
(-> Sendepuffer) werden  
mit der Kennung „tag“ an den Prozess mit Rang „dest“  
innerhalb des Kommunikators „comm“ gesendet.

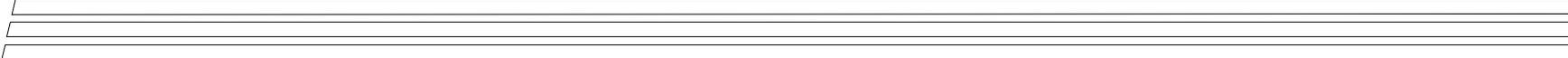






## *MPI-Datentypen*

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



## *Beispielprogramm*

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size < 2) { printf("Need at least 2 processes.\n");
                MPI_Finalize();
                return(1);
            }
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (0 == rank) {
    MPI_Send(buf, BUFSIZE, MPI_INT, 1, 11, MPI_COMM_WORLD);
    printf("rank %d sent message\n", rank);
}
else if (1 == rank) {
    MPI_Recv(buf, BUFSIZE, MPI_INT, 0, 11, MPI_COMM_WORLD,
             &status);
    printf("rank %d received message\n", rank);
}
MPI_Finalize();
```

---

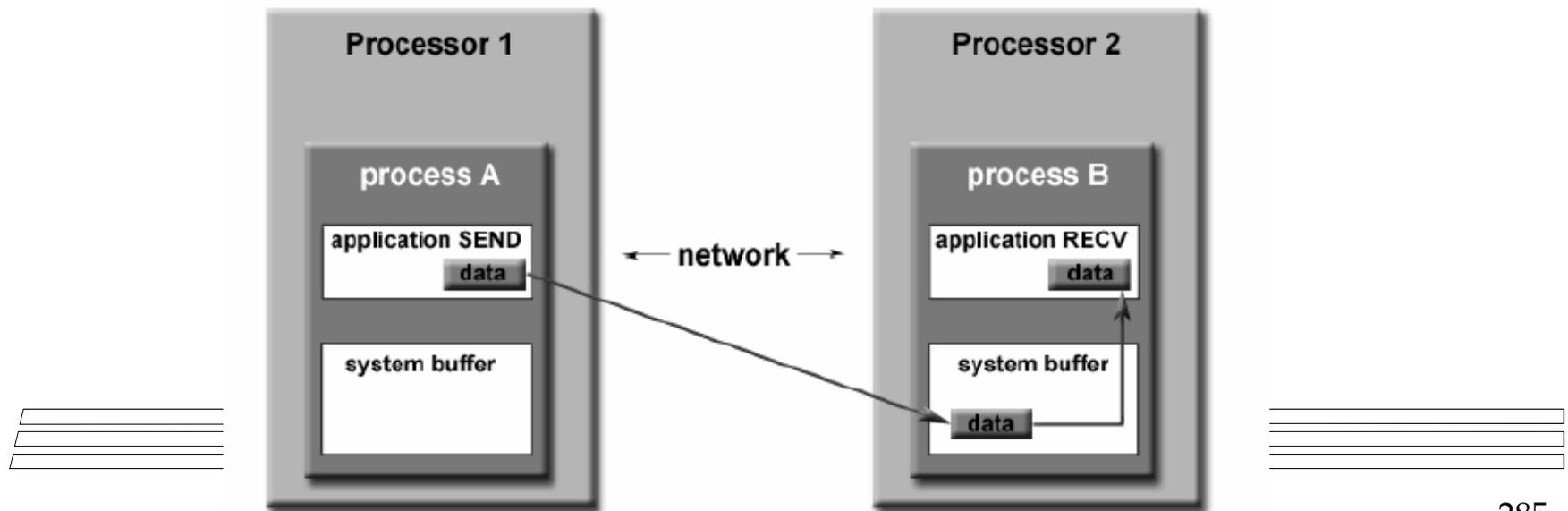
---

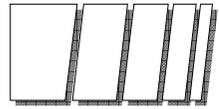
---



# Punkt-zu-Punkt-Kommunikation

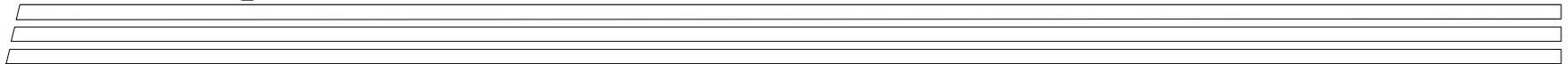
- **MPI unterstützt**
  - Punkt-zu-Punkt-Kommunikation (1:1)
  - kollektive Kommunikation
- **Punkt-zu-Punkt**
  - genau zwei Prozesse, Sender und Empfänger, identifiziert durch ihren Rang innerhalb eines Kommunikators

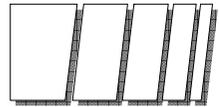




## *Arten von Sendefunktionen*

- **Standard MPI\_Send**
  - blockierende Basis-Sende-Operation
  - Standard lässt konkrete Implementierung offen
- **gepuffert MPI\_Bsend**
  - Nachrichten werden gepuffert, damit das Senden unabhängig vom Empfang abgeschlossen werden kann
  - lokale Operation
  - Bereitstellung von Pufferbereich durch Programmierer erforderlich
- **synchron MPI\_Ssend**
  - Senden endet erst, wenn Empfang der Daten begonnen hat
  - nicht-lokale Operation
- **ready MPI\_Rsend**
  - sofortiges Senden (ohne Zwischenpufferung)
  - Empfängerprozess muss zum Empfang bereit sein
  - lokale Operation





## *Blockierend vs nicht-blockierend*

- **Blockierend bedeutet:**

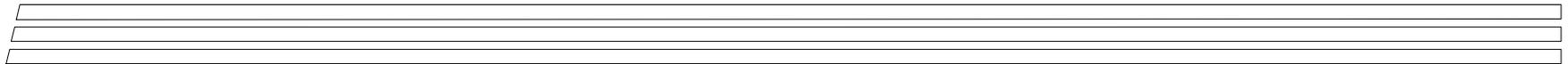
Sendefunktion wird erst beendet, wenn die Nachrichteninitiiierung so weit fortgeschritten ist, dass der (Anwendungs-) Sendepuffer nicht durch eine nachfolgende Anweisung überschrieben werden kann.

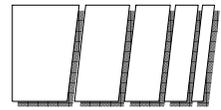
- **Nicht-blockierend bedeutet:**

Der Benutzer ist selbst dafür verantwortlich, dass die abgehende Nachricht nicht korrumpiert wird. Das MPI System wird nur mit der Nachrichtensendung beauftragt.

=> Überlappung von Kommunikation und Berechnung.

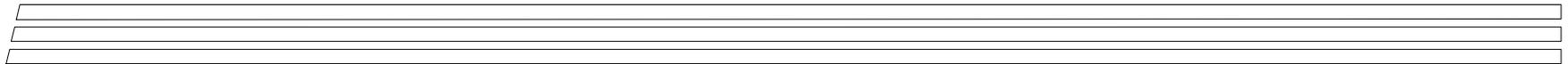
MPI\_Isend - MPI\_Ibsend - MPI\_Irsend - MPI\_Issend

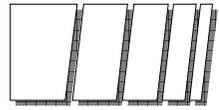




## *Empfangsfunktionen / Pufferverwaltung*

- Es gibt nur eine Empfangsoperation, unabhängig vom Sendemodus, die
  - blockierend (Standard) oder
  - nicht-blockierend (MPI\_Irecv)sein kann.
- Pufferverwaltung für MPI\_Bsend / MPI\_Ibsend
  - MPI\_Buffer\_attach (void \*buffer, int size)  
Bereitstellung von Pufferplatz, size in Bytes
  - MPI\_Buffer\_detach (void \*buffer, int size)  
Freigabe von Pufferplatz





## *Abfrageargument bei nicht-blockierendem Senden und Empfangen*

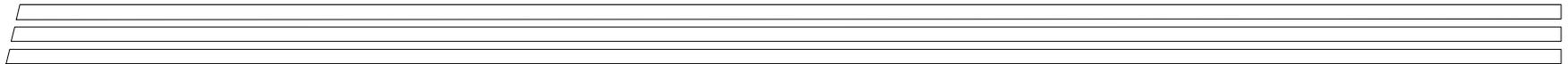
- Bei den nicht-blockierenden Sende- und Empfangsanweisungen kann über ein zusätzliches Argument  
`MPI_Request *request`,  
das zurückgeliefert wird, später getestet werden, ob die Kommunikationsanweisung abgeschlossen ist.
- `MPI_Wait(&request, &status)`  
wartet, bis die zugehörige Kommunikationsanweisung fertig ist.
- `MPI_Test(&request, &flag, &status)`  
zeigt mit `flag`, ob die Kommunikationsanweisung fertig ist oder nicht.

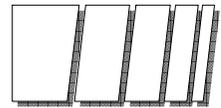
typische Konstellation:

`MPI_Isend/Irecv (..., &request)`

`.../* Berechnungen */`

`MPI_Wait (&request, &status)`





## *Semantik der Punkt-zu-Punkt-Komm.*

- Reihenfolge von Nachrichten mit identischen Umschlägen bleibt erhalten
- Werden passende Sende-/Empfangsanweisungen initiiert, so wird mindestens eine der Anweisungen abgeschlossen.

### Beispiel:

```
MPI_Comm-rank(comm,rank);
```

```
if (rank == 0)
```

```
    { MPI_Bsend (buf1, count, MPI_REAL, 1, tag1, comm);
```

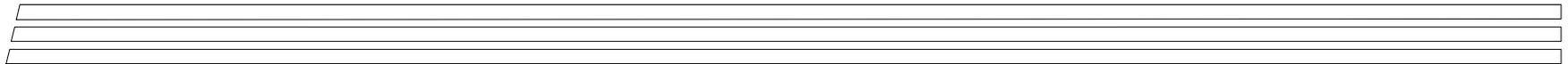
```
      MPI_Ssend (buf2, count, MPI_REAL, 1, tag2, comm); }
```

```
else
```

```
    { MPI_Recv (buf1, count, MPI_REAL, 0, tag2, comm, &status);
```

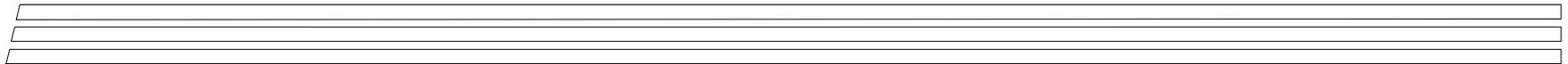
```
      MPI_Recv (buf2, count, MPI_REAL, 0, tag1, comm, &status); }
```

- keine Garantie von Fairness



# *Kollektive Kommunikation*

- Typen
  - globale Barriere
  - globale Datenbewegungen
  - Reduktionsoperationen
- **MPI\_Barrier (MPI\_Comm comm)**  
bewirkt globale Synchronisation aller Prozesse im Kommunikator comm
- **MPI\_Bcast( void \*inbuf, int incnt, MPI\_Datatype intype, int root, MPI\_Comm comm)**  
bewirkt Broadcast von dem Prozess mit Rang root zu allen anderen Prozessen im Kommunikator comm.  
Die ersten drei Argumente sind analog zu send/receive.



# *Aufsammeln und Verteilen von Daten*

- **MPI\_Gather / MPI\_Scatter**

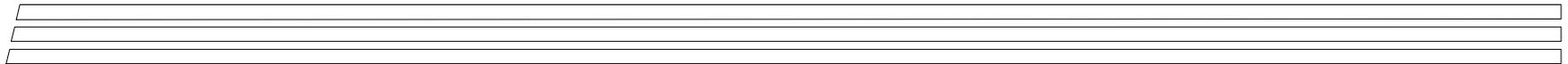
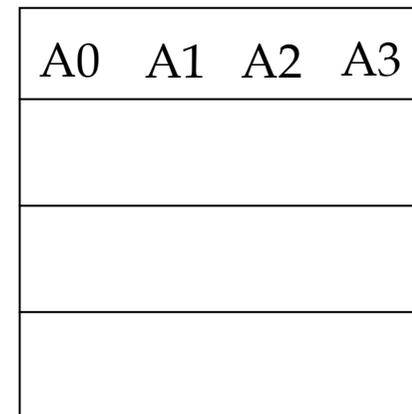
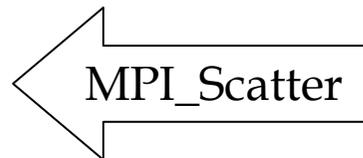
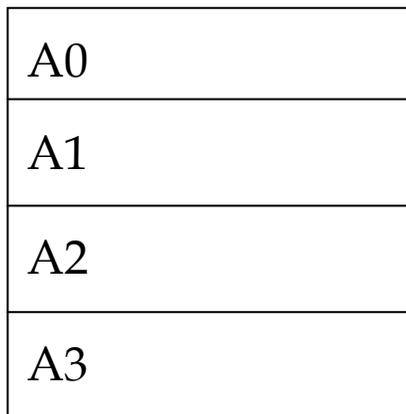
**(void \*inbuf, int incnt, MPI\_Datatype intype,**

Eingabe

**void \*outbuf, int outcnt, MPI\_Datatype outtype,**

Ausgabe

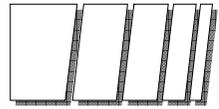
**int root, MPI\_Comm comm)**



# *Reduktionsoperationen*

**MPI\_Reduce / MPI\_Allreduce (void \*inbuf, void \*outbuf, int count, MPI\_Datatype type, MPI\_Op op, int root / \_, MPI\_Comm comm)**

MPI Reduction Operation		C Data Types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

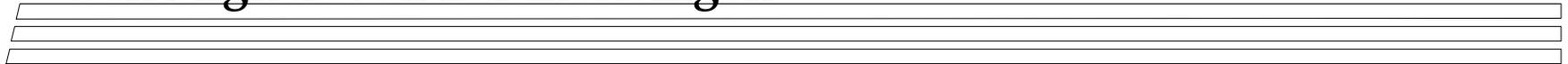


## *Zeitmessungen in MPI*

- **MPI\_Wtime ()** wall clock time  
liefert als Gleitkommazahl mit doppelter Präzision eine Sekundenzahl bzgl. eines festen, d.h. während der Laufzeit eines Prozesses unveränderten Zeitpunkts in der Vergangenheit  
Zeitmessungen erfordern mehrere Aufrufe dieser Routine:

```
{ double starttime, endtime;  
  starttime = double MPI_Wtime();  
  ...  
  endtime = double MPI_Wtime();  
  printf(„That took %f seconds\n“, endtime-starttime); }
```

- **MPI\_WTick()**  
liefert eine Gleitkommazahl, die die Zeit zwischen aufeinanderfolgenden Uhrticks angibt.  
**=> Genauigkeit von Zeitmessungen**



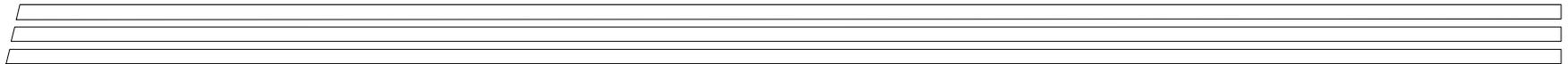
## *MPI\_Sendrecv*

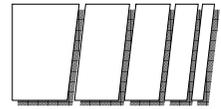
Gekoppeltes Senden und Empfangen:

- Senden und gleichzeitiges Empfangen einer Nachricht
- blockierend bis Sendepuffer frei ist und Empfangspuffer empfangene Daten enthält

**MPI\_Sendrecv**

```
(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
    int dest, int sendtag,  
    void *recvbuf, int recvcount, MPI_Datatype recvtype,  
    int source, int recvtag,  
    MPI_Comm comm, MPI_Status *status)
```





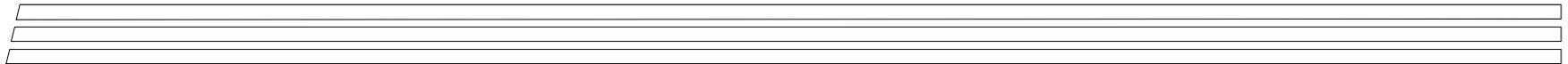
# *Kommunikatoren und Prozessgruppen*

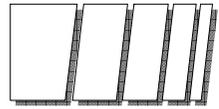
- **Nachrichtenkennungen (tags)**
  - **Unterscheidung und Selektion von Nachrichten**
- **Prozessgruppen**
  - **Einschränkung des Empfangs von Nachrichten und der Anwendung einer kollektiven Kommunikation**

**werden in fast allen Kommunikationsbibliotheken unterstützt.**

## **neu in MPI: Kommunikatoren**

- **bessere Kapselung von Bibliotheksfunktionen**
- **Erleichterung der Arbeit mit Prozessgruppen und virtuellen Topologien**





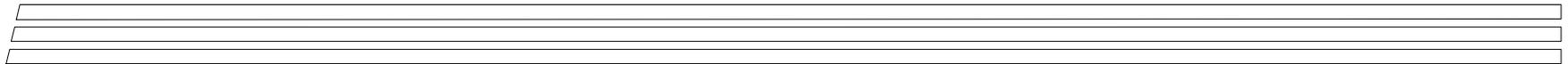
# *Kommunikator-Definition*

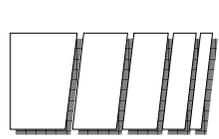
## Ein Kommunikator

- bestimmt eine Gruppe von Prozessen, d.h. eine geordnete Menge von Prozessen mit lokalem Rang  $\in \{0, \dots, \#\text{Prozesse} - 1\}$   
=> Umgebung für Punkt-zu-Punkt- und kollektive Kommunikationen
- definiert einen Kontext für Kommunikationen (implizit)  
=> Einführung separater, sicherer, d.h. sich nicht beeinflussender Universen zum Nachrichtenaustausch, wichtig für Unterprogrammbibliotheken

## vordefinierte Kommunikatoren:

- `MPI_COMM_WORLD` umfasst die Gruppe aller Prozesse
- `MPI_COMM_SELF` enthält nur den Prozess selbst
- `MPI_COMM_NULL` ungültiger Kommunikator

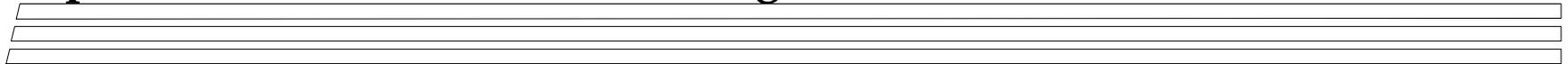


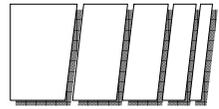


## *Prozessgruppen:*

### *Konstruktion, Analyse und Manipulation*

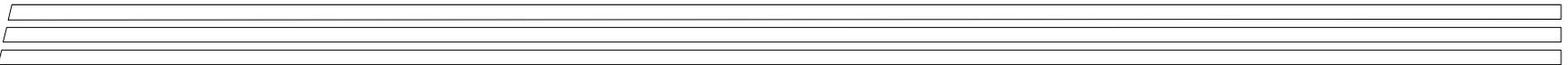
- **MPI\_Comm\_group (MPI\_Comm comm, MPI\_Group \*group)**  
liefert die Prozessgruppe des Kommunikators comm.
- **MPI\_Group\_union/intersection/difference**  
(MPI\_Group group1, MPI\_Group group2, MPI\_Group group3)  
bestimmt die Vereinigung, den Schnitt und die Differenz von Gruppen.  
Vereinigung und Schnitt sind assoziativ, aber wegen der Prozessnummerierung nicht kommutativ.
- **MPI\_GROUP\_EMPTY** ist die leere Prozessgruppe.
- **MPI\_Group\_free (MPI\_Group group)** deallokiert eine Prozessgruppe.  
group wird an MPI\_GROUP\_NULL gebunden.





## Prozessgruppen Fortsetzung

- **MPI\_Group\_size** (MPI\_Group group, int \*size)  
liefert in size die Anzahl der Prozesse einer Gruppe.
- **MPI\_Group\_rank** (MPI\_Group group, int \*rank)  
liefert den Rang eines Prozesses in einer Gruppe und,  
falls der Prozess nicht in der Gruppe ist, MPI\_UNDEFINED.
- **MPI\_Group\_compare**  
(MPI\_Group group1, MPI\_Group group2, int \*result)  
liefert in result:
  - **MPI\_IDENT** bei identischen Gruppen, d.h. dieselben Prozesse  
mit derselben Nummerierung
  - **MPI\_SIMILAR** bei Gruppen mit denselben Prozessen, aber  
unterschiedlicher Nummerierung



- **MPI\_UNEQUAL** sonst.

## *Prozessgruppen Fortsetzung 2*

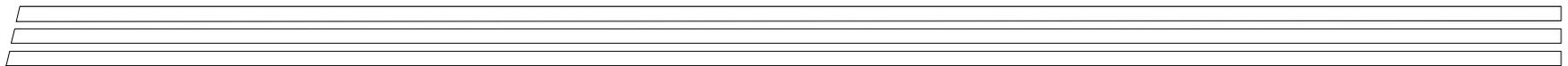
- **MPI\_Group\_incl/excl**  
(MPI\_Group group, int n, int \*ranks, MPI\_Group \*newgroup)  
erzeugt eine neue Gruppe mit n Prozessen,  
die in group die Ränge ranks[0] ... ranks[n-1] haben und  
in newgroup die Ränge 0 ... n-1.

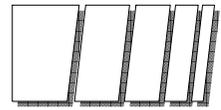
Prozess ranks[i] in group ist Prozess i in newgroup.

Die in ranks angegebenen Ränge müssen in group vorkommen und paarweise verschieden sein.

=> Umordnung von Elementen einer Gruppe

Die excl-Variante streicht die durch ranks angegebenen Prozesse aus der Gruppe.



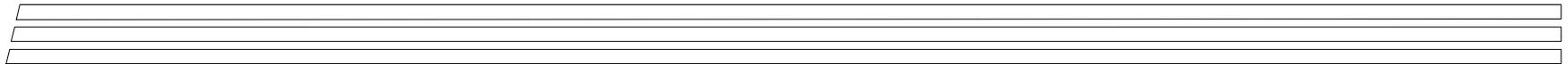


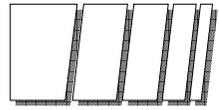
## *Erzeugung /Freigabe von Kommunikatoren*

Neue Kommunikatoren können aus bestehenden Kommunikatoren oder aus Prozessgruppen gebildet werden.

Die folgenden Funktionen sind kollektive Funktionen, d.h. sie müssen von allen Prozessen eines Kommunikators aufgerufen werden.

- **MPI\_Comm\_dup (MPI\_Comm comm, MPI\_Comm \*newcomm)**  
erzeugt einen neuen Kommunikator mit derselben Prozessgruppe, denselben assoziierten Informationen, aber einem neuen Kommunikationskontext.
- **MPI\_Comm\_free (MPI\_Comm \*comm)**  
Deallokation von comm und Rückgabe von MPI\_COMM\_NULL





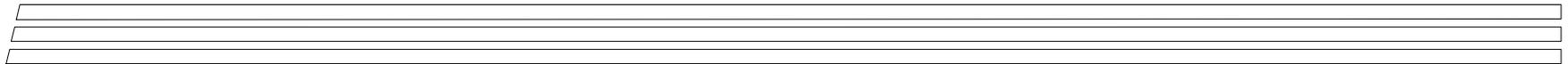
## *Erzeugung von Kommunikatoren aus Prozessgruppen*

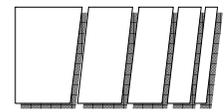
- **MPI\_Comm\_create**

(MPI\_Comm comm, MPI\_Group group, MPI\_Comm \*newcomm)

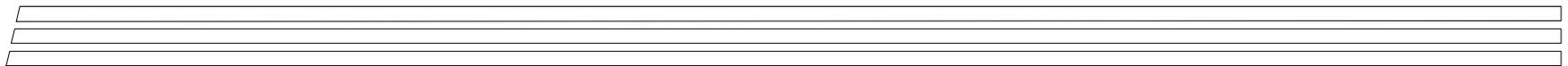
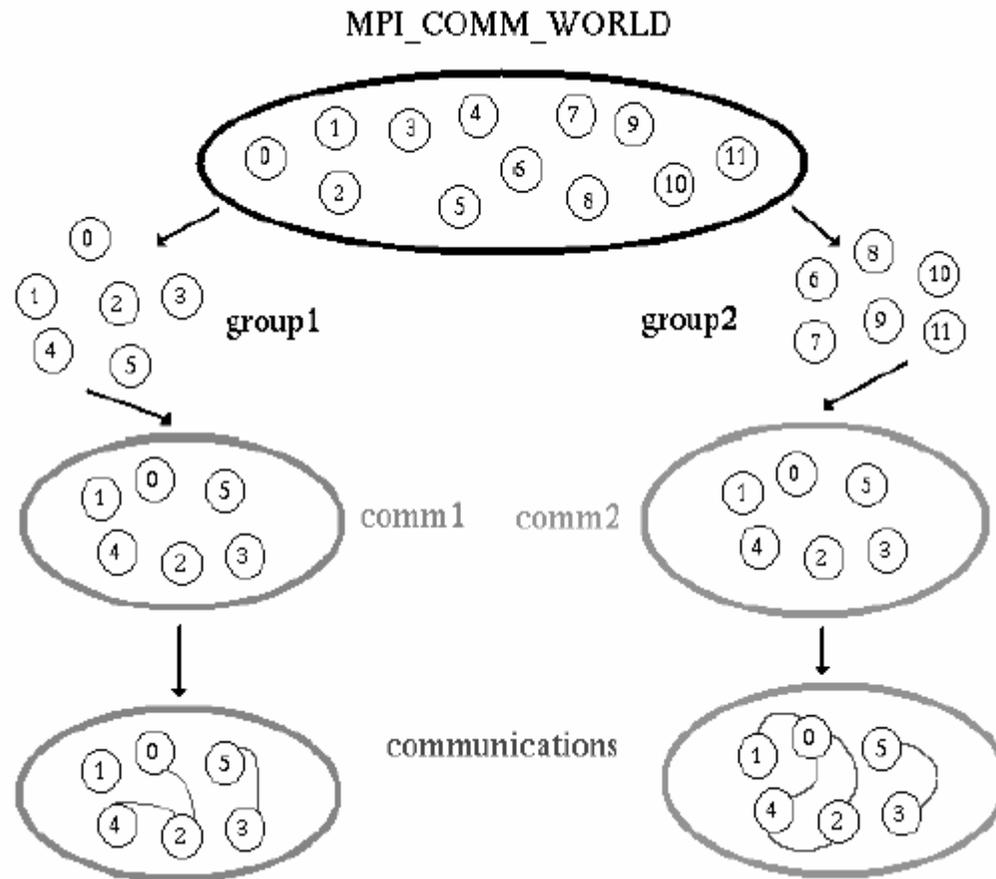
erzeugt einen neuen Kommunikator mit Prozessgruppe group, die Teilmenge der Prozessgruppe von comm sein muss.

- **kollektiver Aufruf**
- **identische group Argumente erforderlich**
- **Prozesse in comm, aber nicht in group, erhalten MPI\_COMM\_NULL zurück.**
- **Synchronisation der Aufrufe nicht erforderlich, aber „Nachrichten aus der Zukunft“ möglich**





# Typischer Aufbau neuer Kommunikatoren



## *Aufteilen von Kommunikatoren*

- **MPI\_Comm\_Split**

(MPI\_Comm comm, int color, int key, MPI\_Comm \*newcomm)

Aufteilung der Prozessgruppe von comm in disjunkte Teilgruppen, je eine pro Wert von color

=> Prozesse mit derselben Farbe sind in derselben Teilgruppe

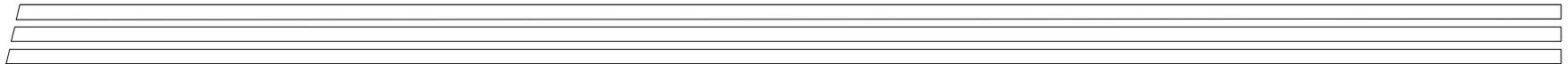
Nummerierung innerhalb der Gruppen gemäß key, bei identischen Schlüsseln gemäß Nummerierung in comm

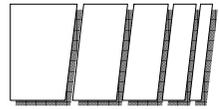
Aufrufe mit der Farbe MPI\_UNDEFINED liefern den Kommunikator MPI\_COMM\_NULL zurück.

Bsp: **MPI\_Comm\_split (comm, 0, 0, newcomm)**

entspricht

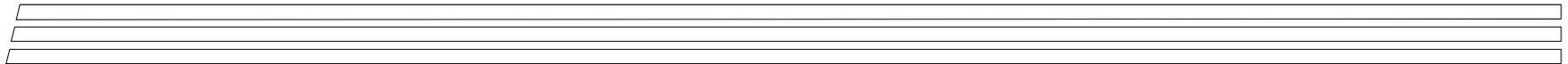
**MPI\_Comm\_dup (comm, newcomm)**

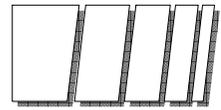




# *Virtuelle Topologien*

- **Unterstützung fester Kommunikationsstrukturen**
  - **effizientere Abbildung von Prozessstrukturen auf physikalischer Zielmaschine**
  - **einfachere, adäquatere Benennung von Prozessen**
  - **Verbesserung der Lesbarkeit von Programmen**
- **zwei Arten:**
  - **kartesische Topologien: Gitter, Würfel, Torus, Hypercube**
  - **Graphtopologien**
- **feste Assoziation mit Kommunikator**
- **Erzeugung virtueller Topologie erzeugt neuen Kommunikator**





## *Erzeugung kartesischer Topologie*

- **MPI\_Cart\_create**

(MPI\_Comm comm\_old, int ndims, [int] dims, [bool] periods,  
bool reorder, MPI\_Comm \*comm\_cart)

assoziiert mit comm\_old eine kartesische Topologie mit ndims Dimensionen und der Ausdehnung dim[i] in Dimension i.

periods[i] = true bedeutet eine zyklische Verbindung in Dim. i.

reorder = false => Prozessränge bleiben unverändert

reorder = true => Umordnung von Prozessen erlaubt,  
um Abbildung auf physikalische Topologie  
zu verbessern.

comm\_cart kann weniger Prozesse enthalten als comm\_old, einige Prozesse erhalten MPI\_COMM\_NULL als Resultat.

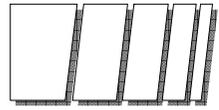
---

---

---

---

Nummerierung beginnt immer bei Null, auch in Fortran.



## Zugriffsroutinen

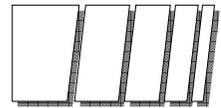
- **MPI\_Cart\_rank** (MPI\_Comm comm, int \*coords, int \*rank)  
liefert in Kommunikator mit kartesischer Topologie den Rang eines Prozesses zu dessen Koordinaten und MPI\_PROC\_NULL bei ungültigen Koordinaten (nicht bei periodischen Strukturen)
- **MPI\_Cart\_coords**  
(MPI\_Comm comm, int rank, int maxdims, int \*coords)  
liefert Koordinaten zu Prozessrang.  
maxdims gibt Dimension des Arrays coords an.
- **MPI\_Cart\_shift**  
(MPI\_Comm comm, int direction, int disp,  
int \*rank\_source, int \*rank\_dest)  
liefert Ränge der Partnerprozesse für Shift-Operationen.

---

---

---

directions bestimmt die Dimension für Shift, disp die Schrittweite.

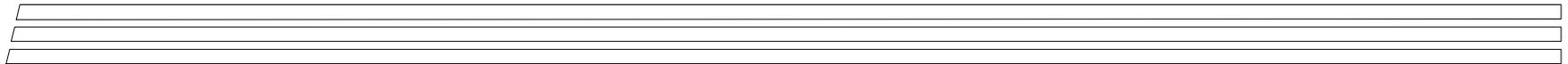


## *Bestimmung balancierter Gitterstrukturen*

- **MPI\_Dims\_create (int nnodes, int ndims, int \*dims)**  
bestimmt zu nnodes Prozessen eine balancierte kartesische Topologie mit ndims Dimensionen

**Im Feld dims können Felder vorbelegt werden.**

**nnodes muss Vielfaches von dem Produkt aller Werte dims[i] mit dims[i]  $\neq$  0 sein.**

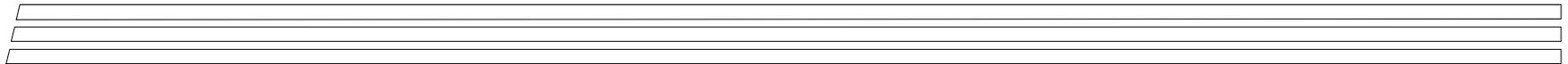


## *Interkommunikatoren*

- **Interkommunikatoren dienen der Kommunikation zwischen disjunkten Prozessgruppen. Sie erlauben nur Punkt-zu-Punkt-Kommunikationen.**
- **MPI\_Intercomm\_create ( local\_comm, local\_leader, peer\_comm, remote\_leader, tag, newintercomm)**

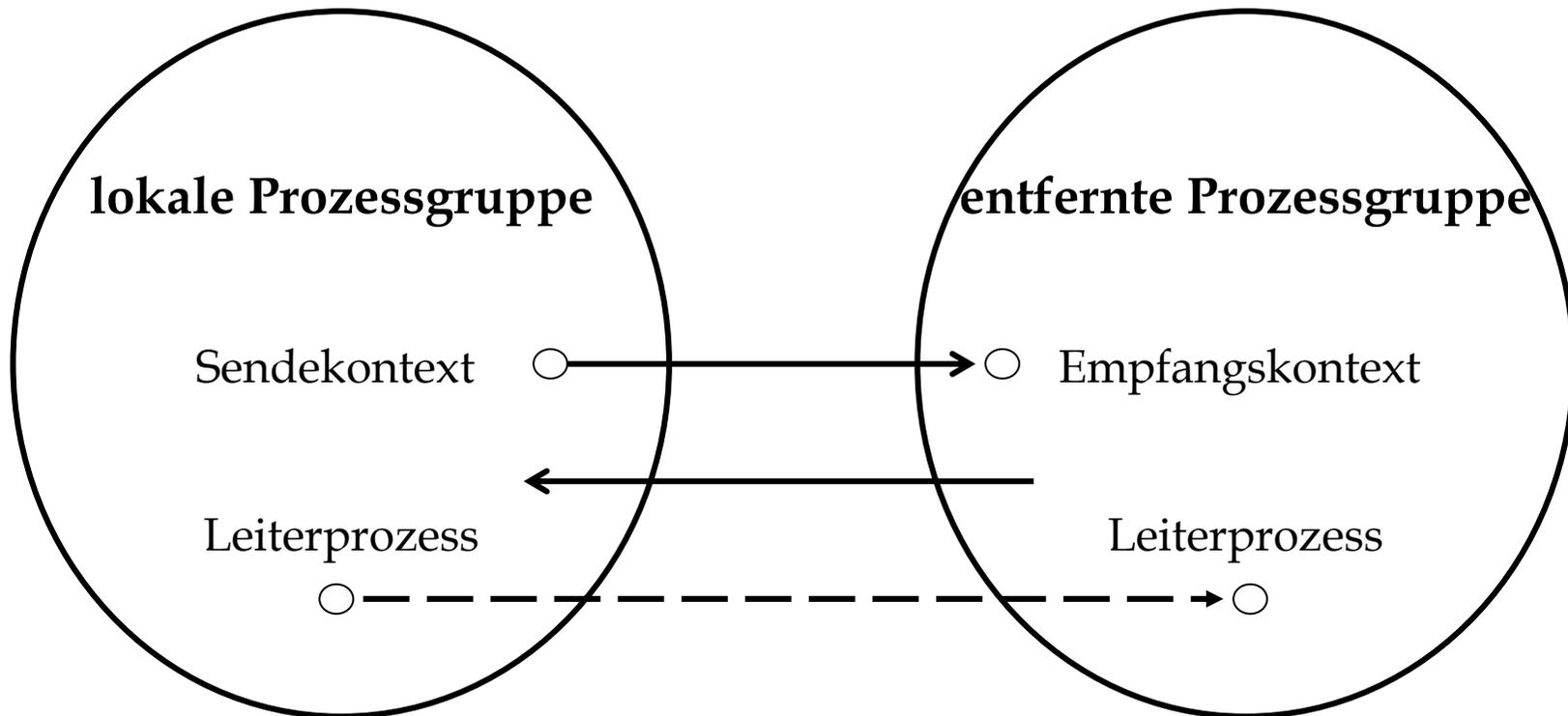
**muss kollektiv in beiden Gruppen aufgerufen werden.  
Alle Prozesse müssen sich jeweils auf einen Leiter einigen.**

**peer-comm ist ein übergeordneter Kommunikator, der beide Gruppen enthalten muss, meist MPI\_COMM\_WORLD oder ein Duplikat.**



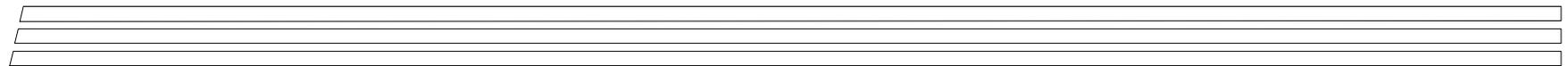
# Interkommunikation

- Der Aufruf von `MPI_Intercomm_create` bewirkt
  - kollektive Kommunikationen in den einzelnen Gruppen
  - Punkt-zu-Punkt-Kommunikation zwischen Leiterprozessen in `peer_comm` mit „tag“



# *Abgeleitete Datentypen*

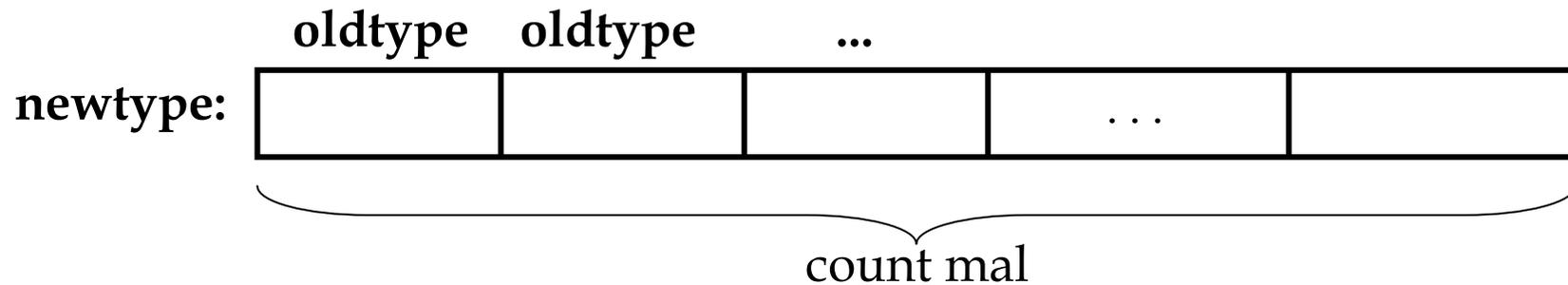
- Erzeugung in zwei Stufen:
  - A - Spezifikation des Datentyps
  - B - Bereitstellung des Datentyps
- ad A: Eine Typabbildung (type map) ist eine Folge von Paaren der Form  
 $\langle \text{type}_i, \text{disp}_i \mid 0 \leq i < n \rangle$   
wobei  $\text{type}_i$  Basistypen und  
 $\text{disp}_i$  ganze Zahlen /relative Adressen (displacements)  
sind.
- Die Folge  $\langle \text{type}_i \mid 0 \leq i < n \rangle$  heißt Typsignatur der Typabbildung.
- Zusammen mit einer Basisadresse  $\text{buf}$  spezifiziert eine Typabbildung einen Kommunikationspuffer mit  $n$  Einträgen. Der  $i$ -te Eintrag beginnt



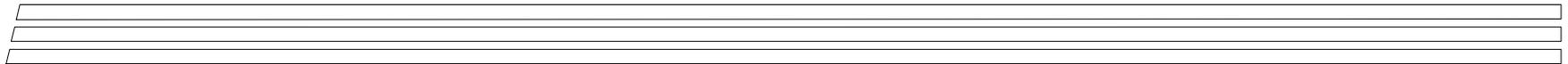
an der Adresse  $\text{buf} + \text{disp}_i$  und hat Typ  $\text{type}_i$ .

# Routinen zur Definition abgeleiteter Typen

- **MPI\_Type\_contiguous (count, oldtype, newtype)**  
bewirkt count-fache ( $\text{count} > 0$ ) Replikation von oldtype zu newtype.



- Die Typüberprüfung beim Empfang von Daten berücksichtigt nur die Typsignatur, d.h. die Folge der Basistypen, und nicht die zugrundeliegenden Typdefinitionen.



# Routinen zur Definition abgeleiteter Typen

- `MPI_Type_vector (count, blocklength, stride, oldtype, newtype)`

`blocklength` spezifiziert die Blocklänge, `stride` die Schrittweite.

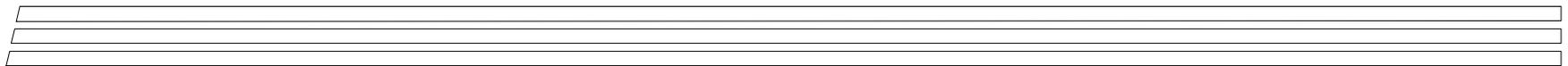
Bsp: `count = 2, blocklength = 3, stride 5`

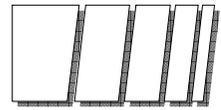
oldtype 

newtype



Schrittweite





## *Bereitstellung von Datentypen*

- **MPI\_Type\_commit (datatype)**  
aktiviert neu erzeugten Datentyp, muss vor Verwendung in Kommunikationsroutinen aufgerufen werden.
- **MPI\_Type\_free (datatype)**  
Freigabe von Datentyp

