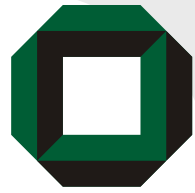
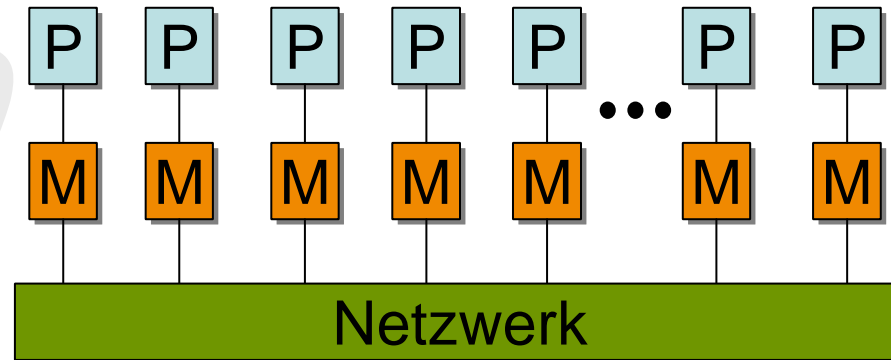


# Programmieren mit MPI

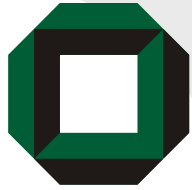
- MPI steht für: **M**essage **P**assing **I**nterface Standard.
- Spezifiziert die Schnittstelle einer Bibliothek für Nachrichtenausch.
- Standardisierung durch ein Gremium, in dem:
  - Hersteller v. Hardware (Supercomputer, Kommunikationshardware),
  - Software-Firmen, Universitäten, sowie
  - Anwender vertreten sind.
- Sehr verbreitet. Verfügbar für nahezu alle Parallel- und Supercomputer.
- Auf COTS-Rechnerbündeln kommt zumeist eine der beiden frei verfügbaren Implementierungen LAM oder MPICH zum Einsatz.
  - Je nach verwendeter Hardware (Myrinet, Infiniband, etc.) werden angepasste oder erweiterte Version verwendet.



# Programmiermodell



- **SPMD**: Single Program Multiple Data
- Dasselbe Programm läuft auf allen Knoten.
  - Programm ist in einer sequentiellen Sprache geschrieben (C, C++, Fortran).
  - Alle Variablen sind prozessor- bzw. prozesslokal.
  - Kommunikation findet stets über Bibliotheksaufrufe statt (Nachrichtenaustausch).



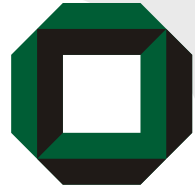
# Hello-World in MPI

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

int main (int argc, char **argv)
{
    int rank, size;

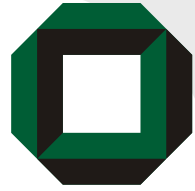
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello World.\n My rank is %d (out of %d)\n",
           rank, size);
    MPI_Finalize();
    return 0;
}
```

<b>Übersetzen:</b>	mpicc -o hello hello.c
<b>Aufruf:</b>	mpirun -np 4 ./hello



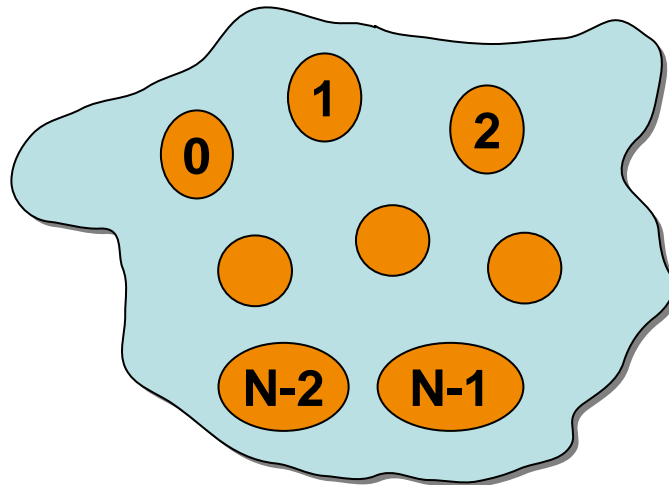
# Nachrichtenaustausch

- Nachrichten sind Datenpakete, die zwischen den Prozessen einer parallelen Anwendung ausgetauscht werden.
- Das Kommunikationssystem benötigt dazu folgende Information:
  - Senderkennung,
  - Speicherort der Quelldaten,
  - Datentyp (Integer, Float, strukturierte Daten, ...),
  - Anzahl der Datenelemente beim Sender,
  - Empfängererkennung (ein oder mehrere Empfänger),
  - Speicherort der Zieldaten,
  - Größe des Empfangsbereiches.

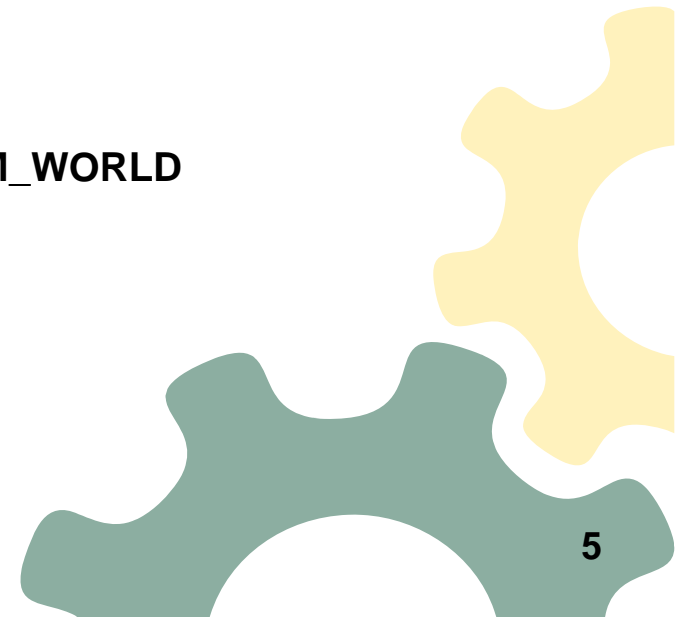


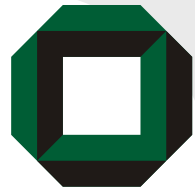
# Adressierung (1)

- MPI verwendet zur Adressierung seiner Prozesse sog. Kommunikatoren.
- Ein **Kommunikator** ist eine eindeutige Kommunikationsumgebung (**Nachrichtenkontext**) für eine **Prozessgruppe**, deren Elemente von 0 bis N-1 durchnummeriert sind.
- Der vordefinierte Standardkommunikator ist **MPI\_COMM\_WORLD**, der alle gestarteten Prozesse enthält.



MPI\_COMM\_WORLD

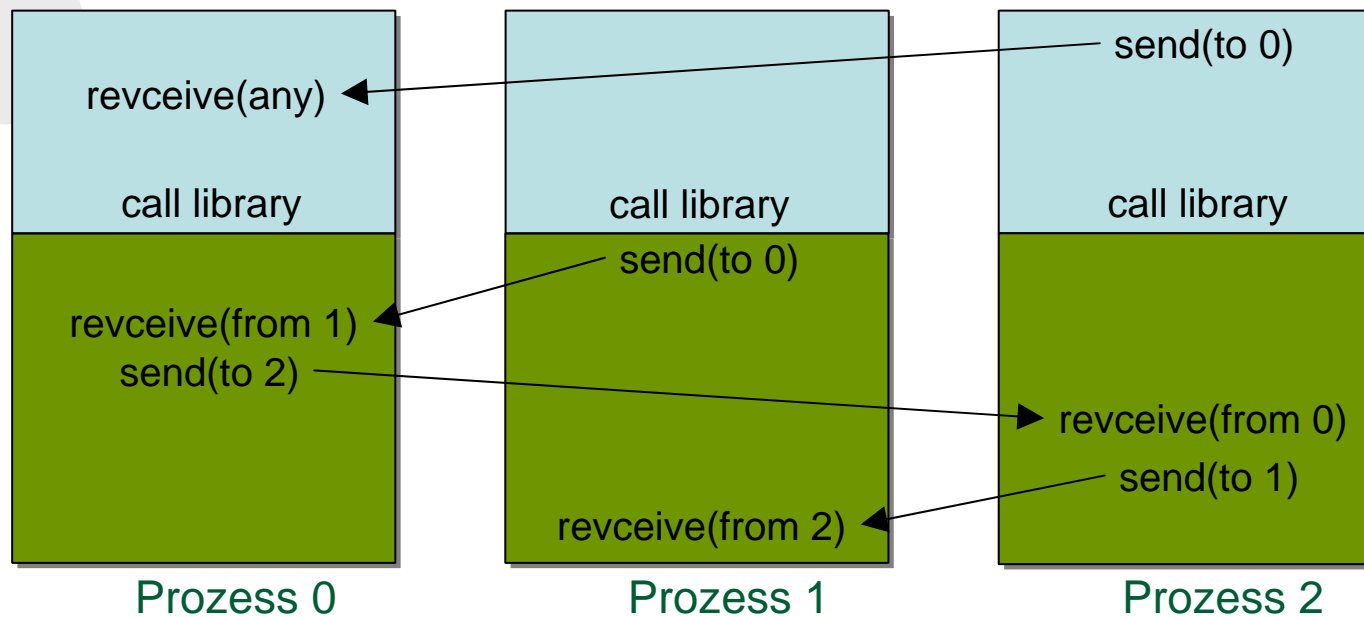




# Adressierung (2)

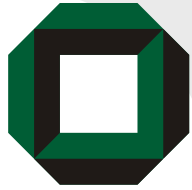
Kommunikator = Prozessgruppe + Kontext

Bessere Unterstützung von Subroutinen und Bibliotheken bzgl. Koordination mit Hauptprogramm.

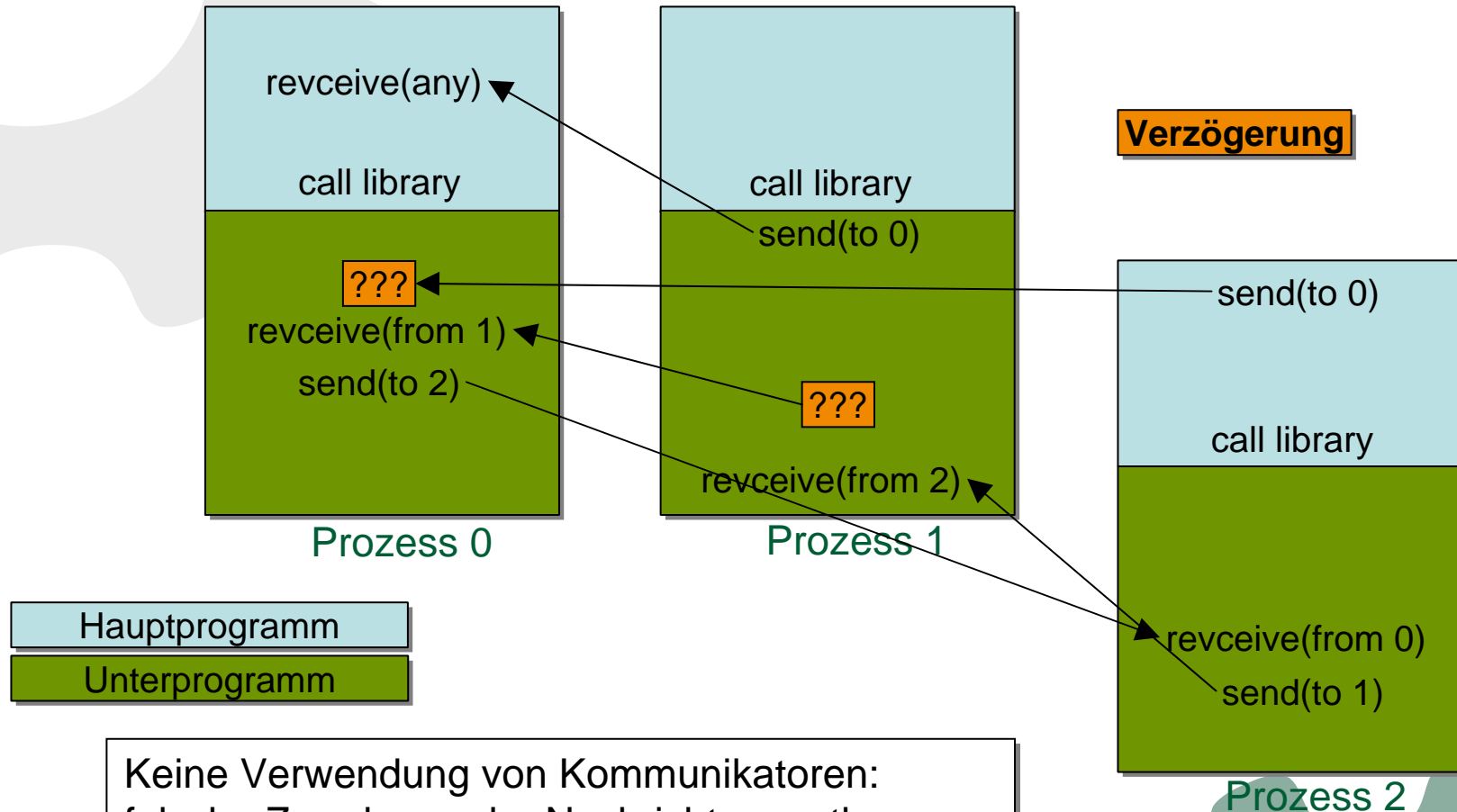


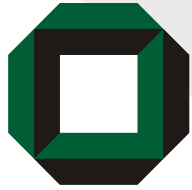
Hauptprogramm

Subroutine

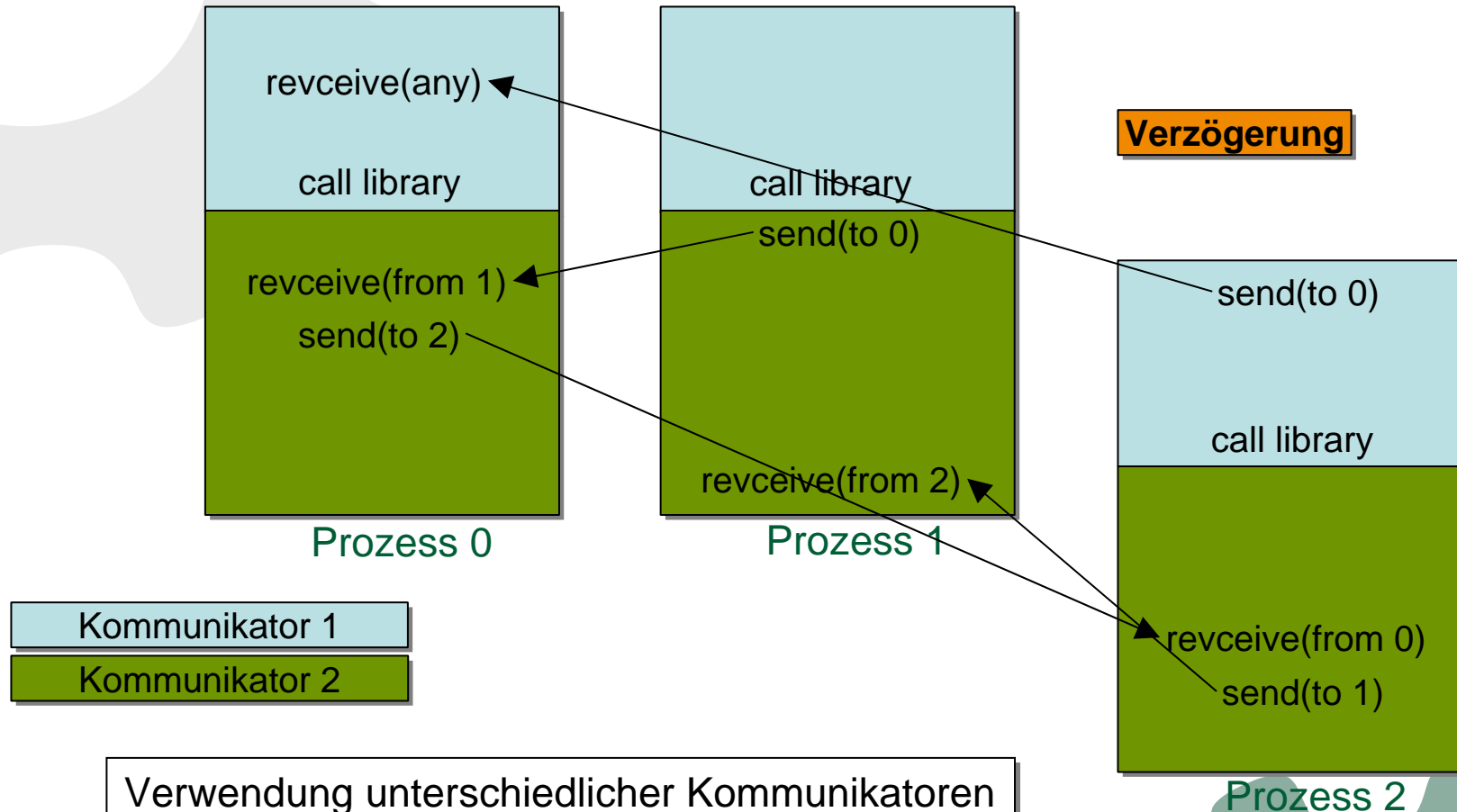


# Adressierung (3)

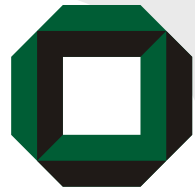




# Adressierung (4)

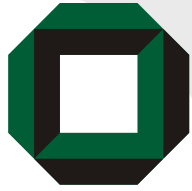


Verwendung unterschiedlicher Kommunikatoren durch die Anwendung und die Bibliothek.



# Kommunikatoren, Gruppen, Kontexte und virtuelle Topologien

- **Kommunikator:**  
Eindeutige Kommunikationsumgebung. Enthält zwingend einen Kontext und eine Gruppe, evtl. auch eine virtuelle Topologie.  
Wird in allen Kommunikationsoperationen als Referenz (Handle) verwendet.
- **(Prozess-) Gruppe:**  
Definiert die Teilnehmer einer Kommunikation. Wird zur Konstruktion von Kommunikatoren verwendet.
- **Kontext:**  
Definiert die (eindeutige) Umgebung, in der eine Kommunikation stattfindet. Interne Kennung im Kommunikator, nicht vom Programm manipulierbar.
- **virtuelle Topologie:**  
Spezielle Anordnung von Prozessnummern in einer Gruppe, die eine bestimmte Topologie widerspiegeln (s. spätere Folien).



# Kommunikatoroperationen

- Erfragen der Größe der zugehörigen Prozessgruppe und Ordnungsnummer (Rang) darin:

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

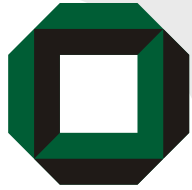
- Kommunikator duplizieren (erzeugt neuen Kontext):  
`MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)`

- Neuen Kommunikator erzeugen:  
`MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)`

- Kommunikator teilen:  
`MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)`

Alle Prozesse, die den gleichen Wert für `color` spezifizieren, sind nach der Operation in derselben Gruppe.

- Kommunikator löschen:  
`MPI_Comm_free(MPI_Comm *comm)`
- Die Aufrufe zum Erzeugen/Löschen eines Kommunikators müssen immer von allen Prozessen der Prozessgruppe des alten Kommunikators aufgerufen werden!

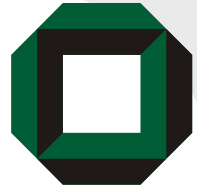


# Gruppenoperationen

- Gruppenobjekt aus Kommunikator extrahieren  
`MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`
- Größe der Gruppe erfragen  
`MPI_Group_size(MPI_Group group, int *size)`
- Eigenen Rang bestimmen (liefert MPI\_UNDEFINED, falls nicht Mitglied)  
`MPI_Group_rank(MPI_Group group, int *rank)`
- Gruppe modifizieren  
`MPI_Group_union(MPI_Group group1, MPI_Group group2,  
MPI_Group *newgroup)`  
`MPI_Group_intersection(MPI_Group group1, MPI_Group group2,  
MPI_Group *newgroup)`  
`MPI_Group_difference(MPI_Group group1, MPI_Group group2,  
MPI_Group *newgroup)`  
`MPI_Group_incl(MPI_Group group, int n, int ranks[],  
MPI_Group *newgroup)`  
`MPI_Group_excl(MPI_Group group, int n, int ranks[],  
MPI_Group *newgroup)`

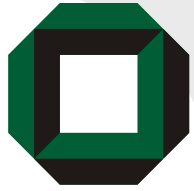
Die neue Gruppe enthält die Prozesse mit den aufgezählten  
(bzw. die nicht aufgezählten) Rangnummern.

- Gruppe löschen: `MPI_Group_free(MPI_Group *group)`
- Die Funktionen zum Erzeugen neuer Gruppen müssen immer von allen Mitgliedern der alten Gruppe aufgerufen werden!



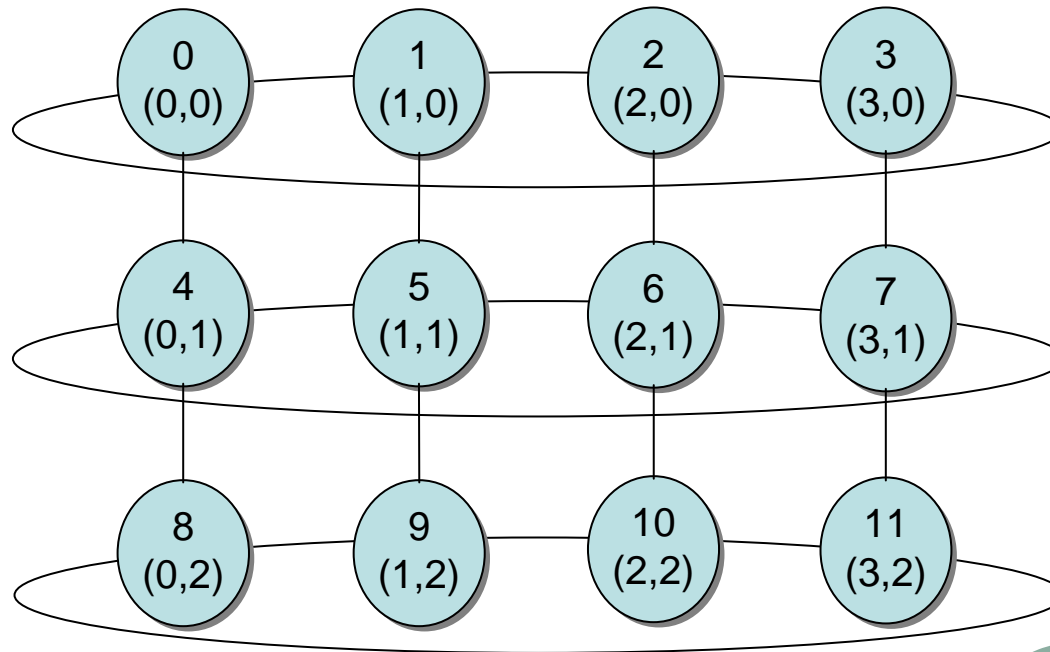
# Virtuelle Topologien (1)

- Eine virtuelle Topologie ist eine an einen Kommunikator geknüpfte Zusatzinformation, und zwar
- die Abbildung von Prozessnummern auf einen am Problem orientierten Namensraum und umgekehrt.
- Methoden zur Erzeugung / zum Umgang mit virtuellen Topologien:
  - Allgemein: Konstruktion eines Graphen:
    - Knoten sind Prozesse.
    - Kanten sind Kommunikationspfade.
  - Spezialfall: Kartesische Koordinaten: Ringe, Tori, zwei- oder höherdimensionale Gitter

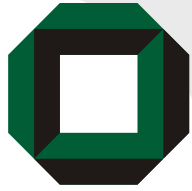


## Virtuelle Topologien (2): kartesische Koordinaten

- `MPI_Cart_create(MPI_Comm comm, int ndims, int dims[], int periods[], int reorder, MPI_Comm *newcomm)`

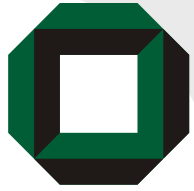


`ndims=2, dims[]={4,3}, periods[]={true, false}`



## Virtuelle Topologien (3): Parameter von `MPI_Cart_create()`

- **comm**: Alter Kommunikator.
- **ndims**: Anzahl der Dimensionen des Gitters/Torus.
- **dims**: Adresse eines Feldes mit **ndims** Einträgen, die die Größe der jeweiligen Dimension angeben.
- **periods**: Adresse eines Feldes mit **ndims** Einträgen, die angeben, ob die jeweilige Dimension periodisch geschlossen ist (1), oder nicht (0).
  - Dies hat Auswirkungen z.B. bei `MPI_Cart_shift()`, wo bei nicht geschlossenen Dimensionen `MPI_PROC_NULL` als Kommunikationspartner zurückgegeben werden kann.
- **reorder**: Gibt an, ob MPI die Reihenfolge der Ränge der beteiligten Prozesse beibehalten soll (0) oder ob es umnummerieren darf, um z.B. eine bessere Abbildung von Prozessen auf die Kommunikationshardware vorzunehmen.
- **newcomm**: Adresse des neuen Kommunikators, der dann die virtuelle Topologie enthält (Ausgabeparameter).



## Virtuelle Topologien (4): Nutzung der kartesischen Koordinaten

- Abbildung der Ordnungsnummer auf Koordinaten

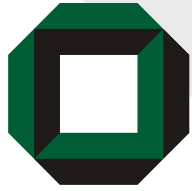
```
MPI_Cart_coords(MPI_Comm comm, int rank,  
int ndims, int *coords[])
```

- Abbildung der Koordinaten auf die Ordnungsnummer

```
MPI_Cart_rank(MPI_Comm comm, int coords[],  
int *rank)
```

- Ermittlung des Rangs eines Nachbarprozesses entlang einer bestimmten Dimension, mit Abstand `displ`

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



# Allgemeines zu Kommunikation (1)

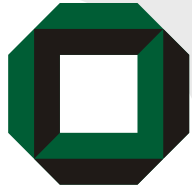
## Jeweils aus Sicht des aufrufenden Prozesses:

### • Blockierende Kommunikationsoperation

- die Rückkehr der Kontrolle zum aufrufenden Prozess bedeutet, dass alle Ressourcen (z.B. Puffer), die für den Aufruf benötigt werden, erneut für andere Operationen genutzt werden können.
- Alle durch den Aufruf ausgelösten Zustandsveränderungen des aufrufende Prozesses finden vor der Rückkehr der Kontrolle statt.

### • Nicht blockierende Kommunikationsoperation

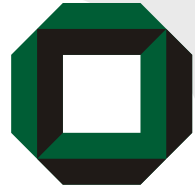
- die aufgerufene Kommunikationsanweisung gibt die Kontrolle zurück,
  - bevor die durch sie ausgelöste Operation vollständig abgeschlossen ist und
  - bevor eingesetzte Ressourcen (z.B. Puffer) wieder benutzt werden dürfen.
- Die ausgelöste Operation ist erst dann wieder vollständig abgeschlossen, wenn
  - alle Zustandsänderungen dieser Operation für den die Prozedur aufrufenden Prozess sichtbar sind und
  - alle Ressourcen wieder verwendet werden können.
- Testen mit *MPI\_Test()* oder Warten mit *MPI\_Wait()*.



# Allgemeines zu Kommunikation (2)

## Lokale und nicht-lokale Aufrufe

- Ein *lokaler* Aufruf
  - kehrt immer zurück, unabhängig davon, welche Operationen andere Prozesse durchführen.  
**Natürlich** kann es je nach Art der Operation lange dauern, bis der Aufruf zurückkehrt. Der Zeitpunkt der Rückkehr kann durch das Verhalten anderer Prozesse beeinflusst werden.
- Ein *nicht-lokaler* Aufruf
  - hängt von anderen Prozessen ab. Wenn dort eine bestimmte Bedingung nicht eintritt, kehrt er unter Umständen nie zurück.



# Allgemeines zu Kommunikation (3)

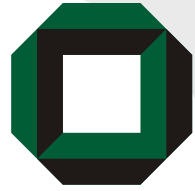
Aus globaler Sicht:

- **Synchrone Kommunikation**

- Die eigentliche Übertragung einer Nachricht findet nur statt, wenn Sender und Empfänger zur *gleichen* Zeit an der Kommunikation teilnehmen.

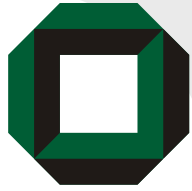
- **Asynchrone Kommunikation**

- Der Sender kann Daten versenden, ohne sicher zu sein, dass der Empfänger bereit ist, die Daten zu empfangen.



# Punkt-zu-Punkt-Kommunikation

- Einfachste Form des Datenaustausches zwischen 2 Prozessen
- Genau 2 Prozesse beteiligt: Sender + Empfänger
  - Beide müssen Kommunikationsanweisung explizit durchführen
  - MPI\_Send / MPI\_Recv
    - Davon gibt es verschiedene Varianten



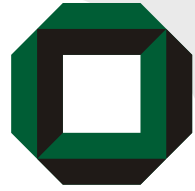
# Senden und Empfangen (1)

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

- `buf`: Adresse des Puffers mit den zu sendenden Daten
- `count`: Anzahl der zu sendenden Elemente (nicht die Länge der Daten in Bytes !!)
- `datatype`: Typ der Elemente
- `dest`: Adresse (Rang) des Empfängers
- `tag`: Paketkennung (vom Benutzer frei wählbar)
- `comm`: Kommunikator

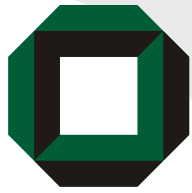
- `MPI_Recv (`  
    `void *buf, int count,`  
    `MPI_Datatype datatype,`  
    `int source, int tag,`  
    `MPI_Comm comm,`  
    `MPI_Status *status)`

- `buf`: Adresse des Puffers für die zu empfangenen Daten
- `count`: Anzahl der zu empfangenen Datenelemente (nicht die Länge der Daten in Bytes !!)
- `datatype`: Typ der Elemente
- `source`: Adresse (Rang) des Senders (oder `MPI_ANY_SOURCE`)
- `tag`: Paketkennung (oder `MPI_ANY_TAG`)
- `comm`: Kommunikator
- `status`: Statusinformation und Fehlercodes (Rückgabewert)



## Senden und Empfangen (2)

- **MPI\_Send()** und **MPI\_Recv()**
  - Sind **blockierende, asynchrone** Operationen
    - Recv kann gestartet werden, wenn Send noch nicht gestartet
      - Recv blockiert, bis sein Empfangspuffer Nachricht enthält
    - Send kann gestartet werden, wenn Recv noch nicht gestartet
      - Send blockiert, bis Sendepuffer wieder verwendet werden kann
- Das tatsächliche Verhalten des Senders sowie Puffermechanismen hängen von spezifischer MPI-Implementierung ab.
  - Achtung: Fehlerquelle für Verklemmungen (Deadlocks)



# Senden und Empfangen (3)

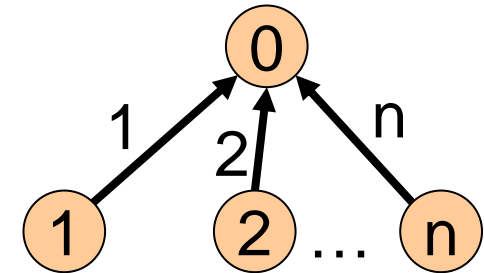
## Hello World v2 in MPI

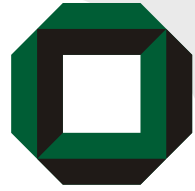
```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char **argv){
    int rank, size, i, buf;
    MPI_Status status;

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

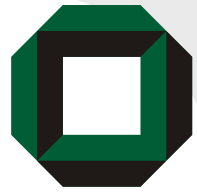
    if(rank == 0){ /* Master process */
        for(i=1; i<size; i++){
            MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &status)
            printf("Got message from node %d\n",buf);
        }
    } else { /* Client process */
        MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```





## Senden und Empfangen (4)

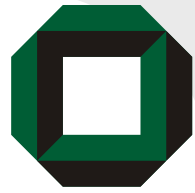
- Es gibt vier verschiedene Modi für Sendeoperationen:
  - *Standard Send (send)*.
  - *Buffered Send (b`send`)*
  - *Synchronous Send (s`send`)*
  - *Ready Send (r`send`)*
- In allen Fällen kann der Sendepuffer wieder verwendet werden, wenn diese Aufrufe zurückkehren.



# Senden und Empfangen (5)

## Standard Send

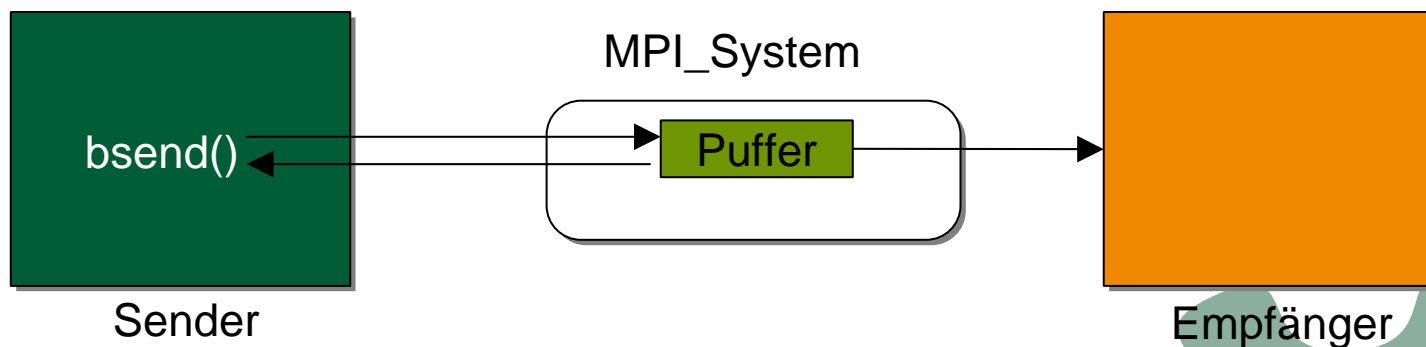
- Allgemeine Sendeoperation.
- Ob die Nachricht bei *Standard Send* gepuffert wird oder nicht, liegt an der jeweiligen Implementierung.
  - Wird im Sendeknoten gepuffert, kehrt die Sendeoperation sofort zurück.  
(Dann ist der Aufruf *lokal*.)
  - Wird nicht gepuffert, muss unter Umständen gewartet werden, bis ein Empfangspuffer beim Empfangsknoten verfügbar wird oder bis der Empfänger seine Empfangsoperation startet.  
(Dann ist der Aufruf *nicht-lokal*.)
- Das Verhalten beim *Standard Send* kann insbesondere auch von der Nachrichtengröße abhängen.

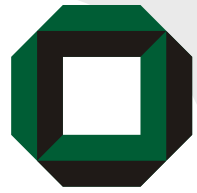


# Senden und Empfangen (6)

## Gepuffertes Senden

- Bei einer **gepufferten** Sendeoperation trägt das MPI-System des Sendeknotens dafür Sorge, dass die Nachricht irgendwo zwischengespeichert wird.
- Bei `bSEND()` muss der Puffer dem MPI-System vorher durch den expliziten Aufruf von `buf_attach()` zur Verfügung gestellt werden.
- Eine gepufferte Sendeoperation ist eine **lokale Operation**, d.h. sie wird immer beendet, ganz gleich, ob eine passende Empfangsoperation im Zielknoten vorliegt oder nicht.
- Ein Pufferüberlauf wird als Fehler behandelt.

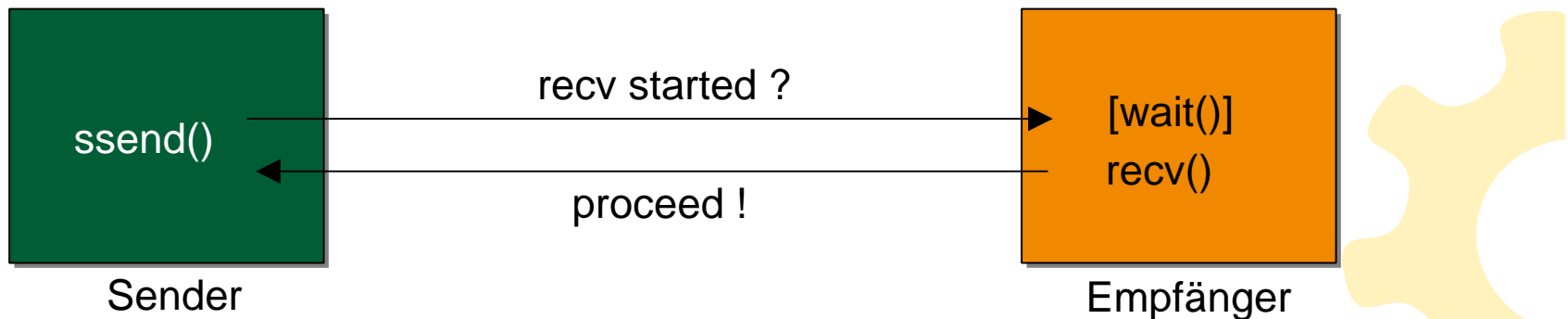


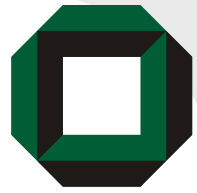


# Senden und Empfangen (7)

## Synchrones Senden

- Eine *synchrone* Sendeoperation kehrt erst dann zurück, wenn der Empfänger einen passende Empfangsauftrag abgesetzt hat.  
Es muss aber noch kein Byte der Nachricht beim Empfänger angekommen sein.
- Der Aufruf ist *nicht-lokal*, d.h. ob (und wann) er zurückkehrt, hängt vom Verhalten des Senders ab.

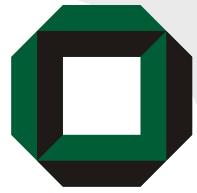




# Senden und Empfangen (8)

## Ready Send

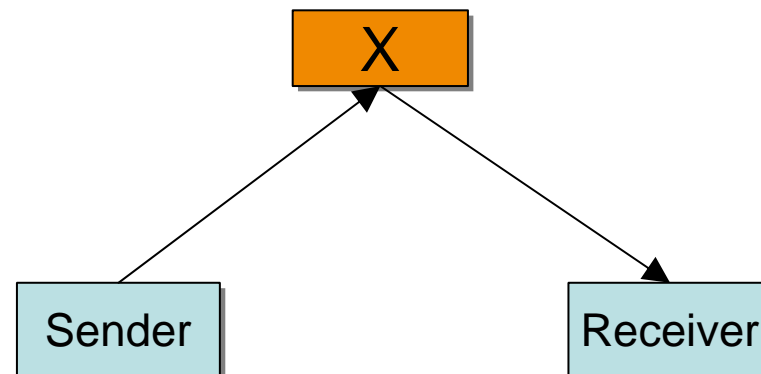
- *Ready Send* entspricht *Standard Send*, aber das MPI-System des Senders geht davon aus, dass beim Empfänger ein passender Empfangsauftrag bereits vorliegt.
  - Andernfalls ist das Verhalten der Anwendung undefiniert (Blockieren, Verlust der Nachricht,...).  
Es wird dann kein Fehler signalisiert.
- In einem korrekten Programm kann jedes *Ready* durch ein *Standard Send* ersetzt werden, ohne die Semantik zu verändern.
- Die Information, dass der Sender schon bereit ist, kann vom MPI-System zur Leistungssteigerung verwendet werden.
  - Einsparung zweier Rendez-vous-Nachrichten.

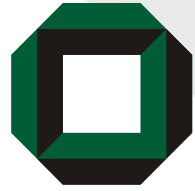


# Senden und Empfangen (9) SendReceive Operation

- Datenaustausch (gleichzeitiges Senden und Empfangen):

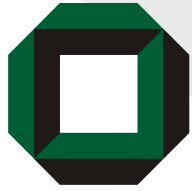
```
MPI_Sendrecv(sendbuf, sendcount,  
             sendtype, dest, sendtag,  
             recvbuf, recvcount,  
             recvtype, source, recvtag,  
             comm, &status);
```





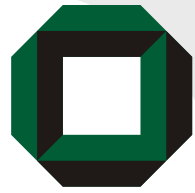
# Nicht-blockierende Operationen (1)

- Auftrennen der Aufrufe zum Senden und Empfangen:
  - *Sendeaufruf* → *Sendestart + Abschluss*
  - *Empfangsaufruf* → *Empfangsstart + Abschluss*
- Zwischen Start und Abschluss der Kommunikationsoperation können **andere Aufgaben** vom Prozess ausgeführt werden, die Kommunikation wird von MPI (bzw. dem Kommunikationssystem, meist von der Hardware) **asynchron** ausgeführt.
- Auswahl blockierend/nicht-blockierend und Modus sind **orthogonal**:
  - Bei Initiieren der Sendeoperation wird der Modus festgelegt:  
*isend()*, *ibsend()*, *irsend()*, *issend()*
- Dagegen nur ein Aufruf zum Initiieren der Empfangsoperation:  
*irecv()*  
**unabhängig** vom Sendemodus und davon, ob synchron oder asynchron gesendet wurde!



## Nicht-blockierende Operationen (2)

- Die *ixsend()* und *irecv()* Aufrufe (mit  $x = \text{„b“}, \text{„s“}, \text{„r“}$  oder  $\text{„“}$ ) selbst sind **lokal**, d.h. kehren gewöhnlich sofort zurück.
- Auf die Beendigung der Operation kann mit *MPI\_Wait()* gewartet werden. Ob dieser Aufruf (bei einer Sendeoperation) **lokal** ist oder nicht, hängt vom **gewählten Modus** ab.
- Mit *MPI\_Test()* kann überprüft werden, ob die Operation bereits abgeschlossen ist (lokaler Aufruf).
- Jeder asynchrone Operation muss zu Ende geführt werden, d.h. es ist entweder
  - *MPI\_Test()* solange aufzurufen, bis sie abgeschlossen ist, oder
  - *MPI\_Wait()* aufzurufen.
  - Auch die gemischte Variante: *MPI\_Test()*, *MPI\_Test()*, ..., *MPI\_Wait()* ist möglich.
- Bei *MPI\_Test()* und *MPI\_Wait()* wird die gemeinte Operation durch ein Handle identifiziert, das von den *ixsend()* bzw. *irecv()* zurückgegeben wird.



# Empfangstests

- Idee: Schau nach, ob es eine Nachricht gibt, die einem bestimmten Muster entspricht (kann danach je nach Status behandelt werden).

- **Blockierender Empfangstest:**

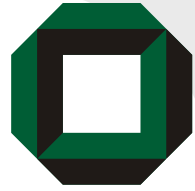
```
MPI_Probe(int source, int tag,  
          MPI_Comm comm, MPI_Status *status)
```

- Wartet, bis eine Nachricht verfügbar ist, die mit einem MPI\_Recv mit gleichem **source** und **tag** empfangen würde (nicht-lokaler Aufruf).

- **nicht blockierender Empfangstest:**

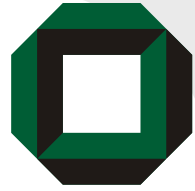
```
MPI_Iprobe(int source, int tag,  
           MPI_Comm comm, MPI_Status *status)
```

- Testet, ob eine Nachricht verfügbar ist, die mit einem MPI\_Recv mit gleichem **source** und **tag** empfangen würde (lokaler Aufruf).



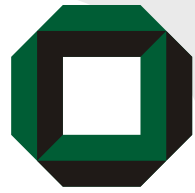
# MPI Datentypen (für C)

- **MPI\_CHAR, MPI\_UNSIGNED\_CHAR**
- **MPI\_SHORT, MPI\_UNSIGNED\_SHORT**
- **MPI\_INT, MPI\_UNSIGNED**
- **MPI\_LONG, MPI\_UNSIGNED\_LONG**
- **MPI\_FLOAT**
- **MPI\_DOUBLE**
- **MPI\_LONG\_DOUBLE**
- **MPI\_BYTE**
- **MPI\_PACKED**
- abgeleitete Datentypen (derived data types)



# Abgeleitete Datentypen (1)

- Idee: ein allgemeiner Datentyp besteht aus zwei Komponenten:
  - einer Sequenz von Basistypen
  - einer Sequenz von Abständen (gerechnet von einer Basisadresse *buf* an).
- Dieses Komponentenpaar wird als **Typemap** mit der Signatur **Typesig** bezeichnet:
$$\text{Typemap} = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$
$$\text{Typesig} = \{type_0, \dots, type_{n-1}\}$$
- **Typemap** und Basisadresse *buf* beschreiben einen Kommunikationspuffer.



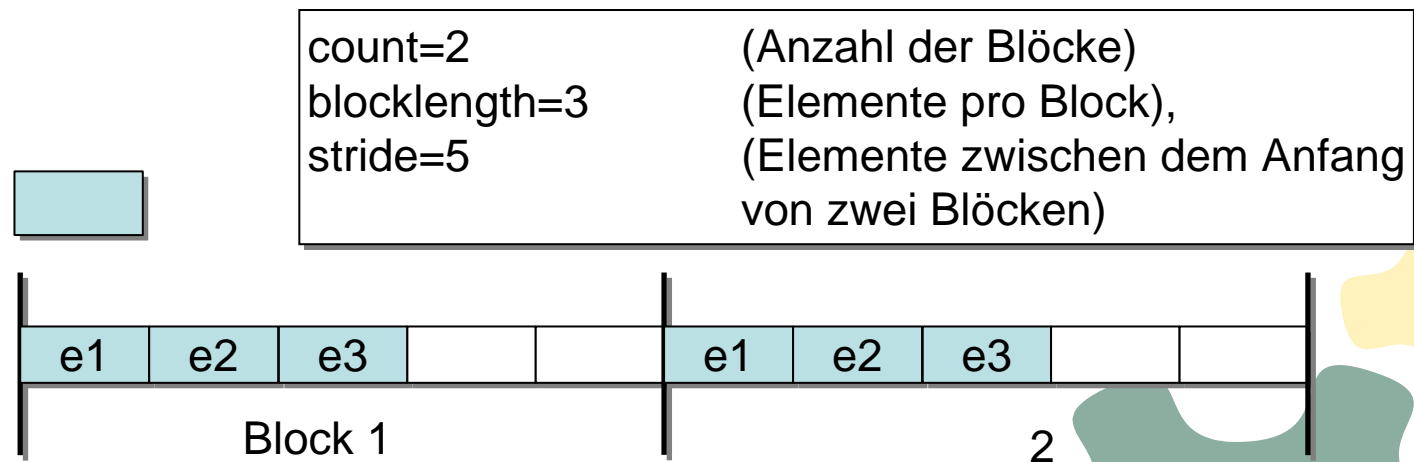
## Abgeleitete Datentypen (2): „Contiguous“ und „Vector“

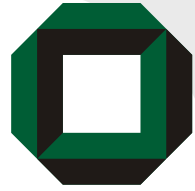
- `MPI_Type_contiguous(int count,  
MPI_Type oldtype, MPI_Type *newtype)`

**Beispiel:** RGB Farbtyp mit 3 Ganzzahlen

```
MPI_Type_contiguous(3, MPI_INT, &RGB_Color);
```

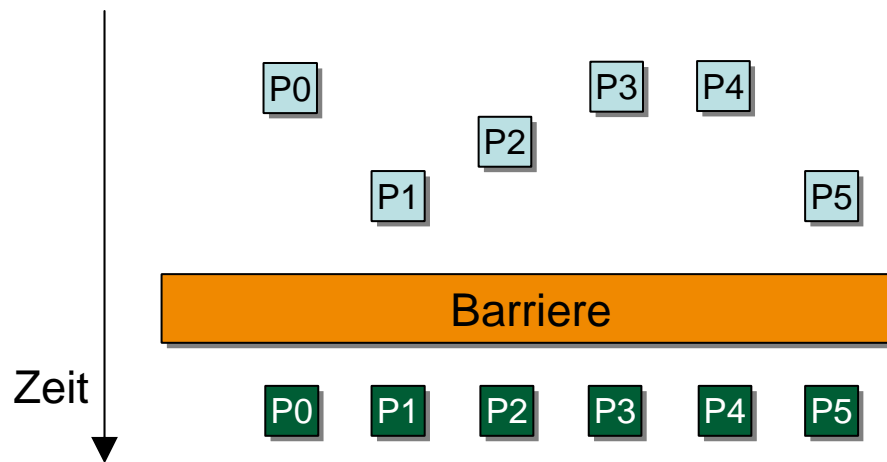
- `MPI_Type_vector(int count, int blocklength,  
int stride, MPI_Type oldtype, MPI_Type *newtype)`

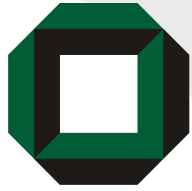




# Kollektive Operationen (1)

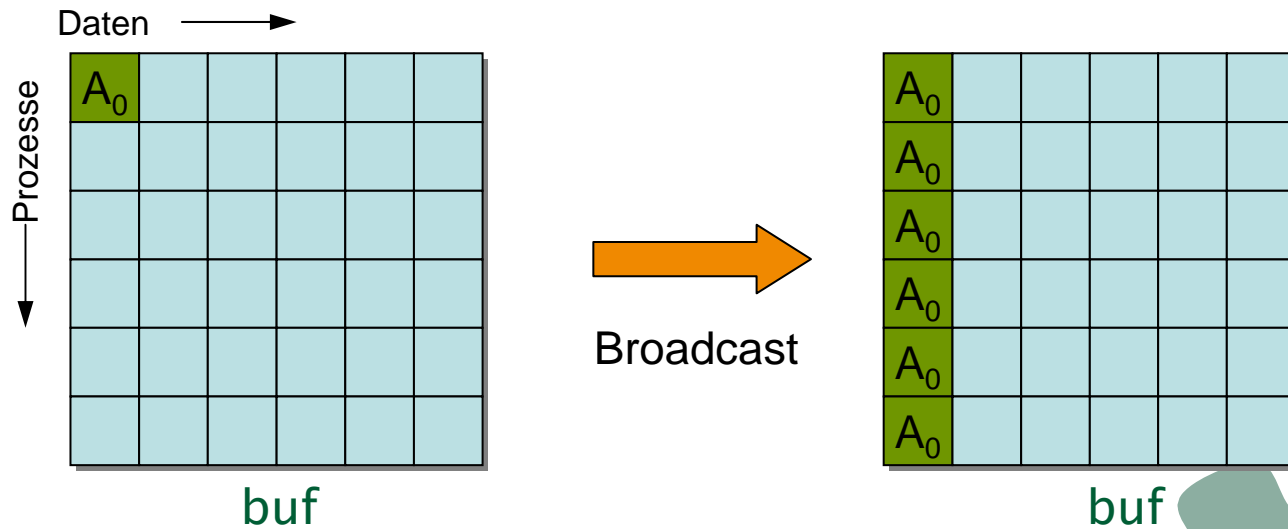
- Synchronisation einer Prozessgruppe:  
`MPI_Barrier(MPI_Comm comm)`
  - **Achtung:** Eine Barriere garantiert nur folgendes:  
Kein Prozess verlässt die Barriere, bevor nicht alle anderen Prozesse die Barriere betreten haben.  
Das heißt, dass nicht alle Prozesse die Barriere notwendigerweise zum gleichen Zeitpunkt verlassen.

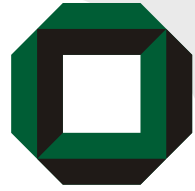




# Kollektive Operationen (2)

- Broadcast: Verteilen der Daten von einem zu vielen anderen Prozessen:  
`MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

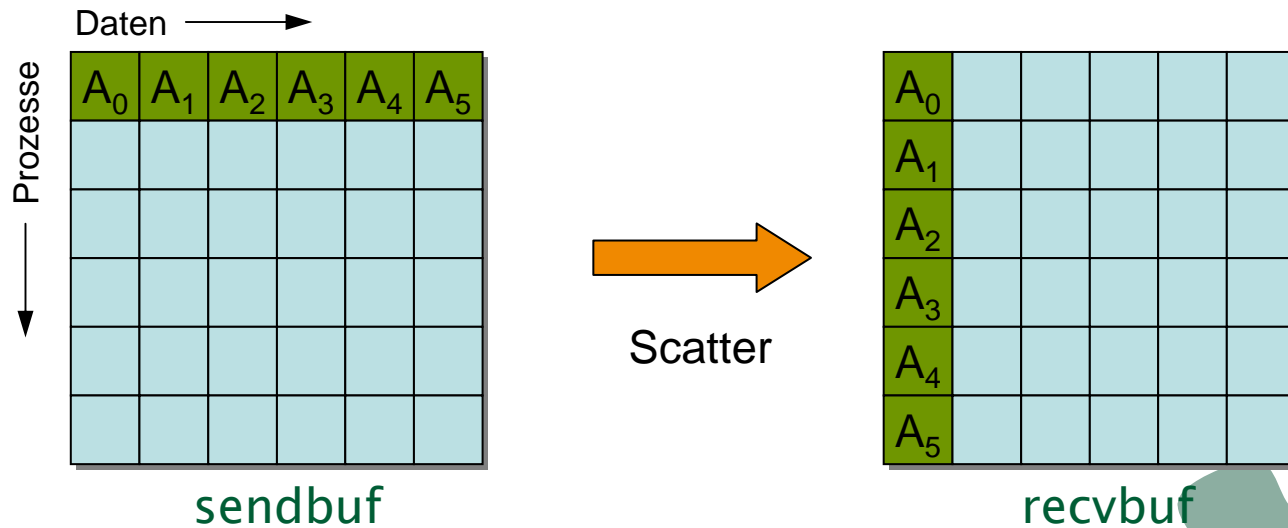


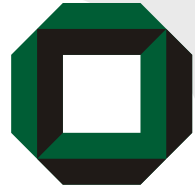


# Kollektive Operationen (3)

- Verteilen von Elementen aus einem Kommunikationspuffer an verschiedene Prozesse:

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

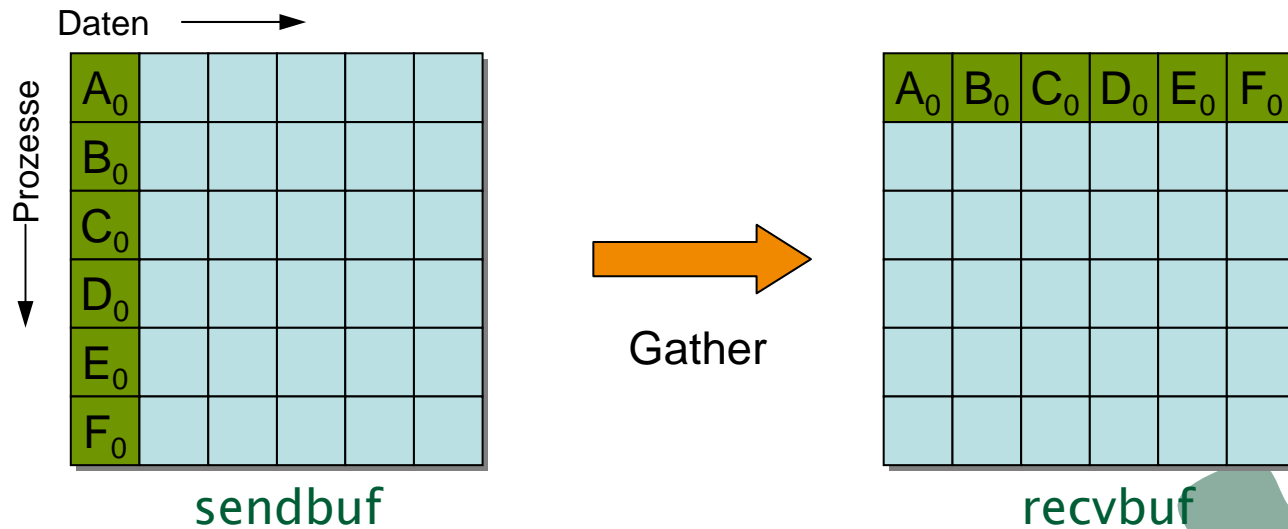


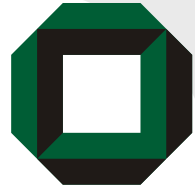


# Kollektive Operationen (4)

- Aufsammeln der Elemente eines Kommunikationspuffers von verschiedenen Prozessen:

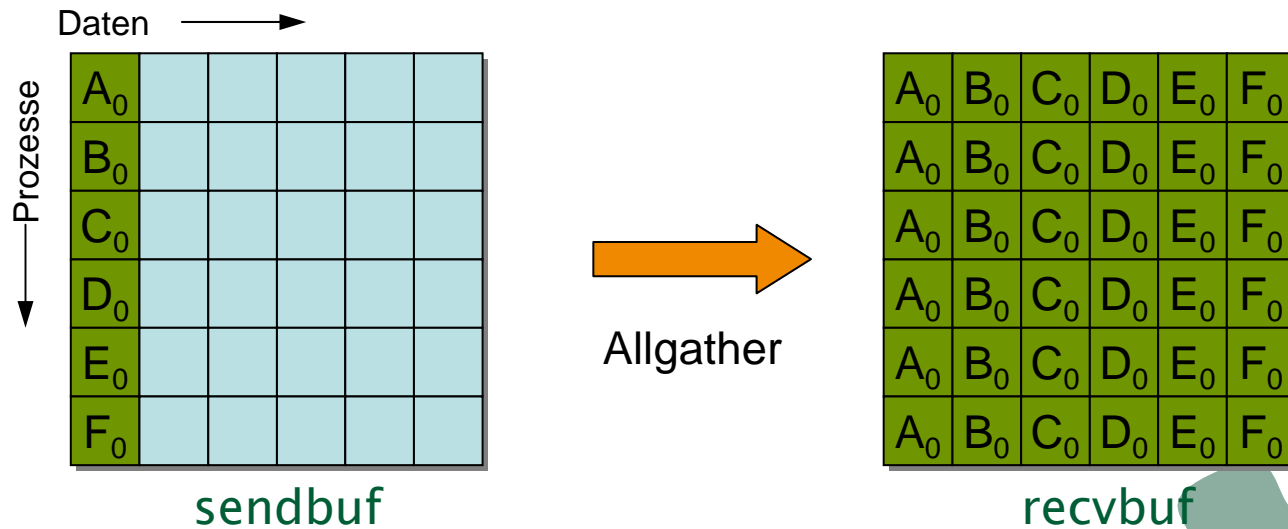
```
MPI_Gather(void *sendbuf, int sendcnt, MPI_Type sendtype,  
          void *recvbuf, int recvcnt, MPI_Type recvtype,  
          int root, MPI_Comm comm)
```

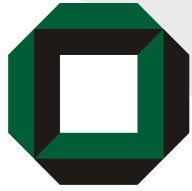




# Kollektive Operationen (5)

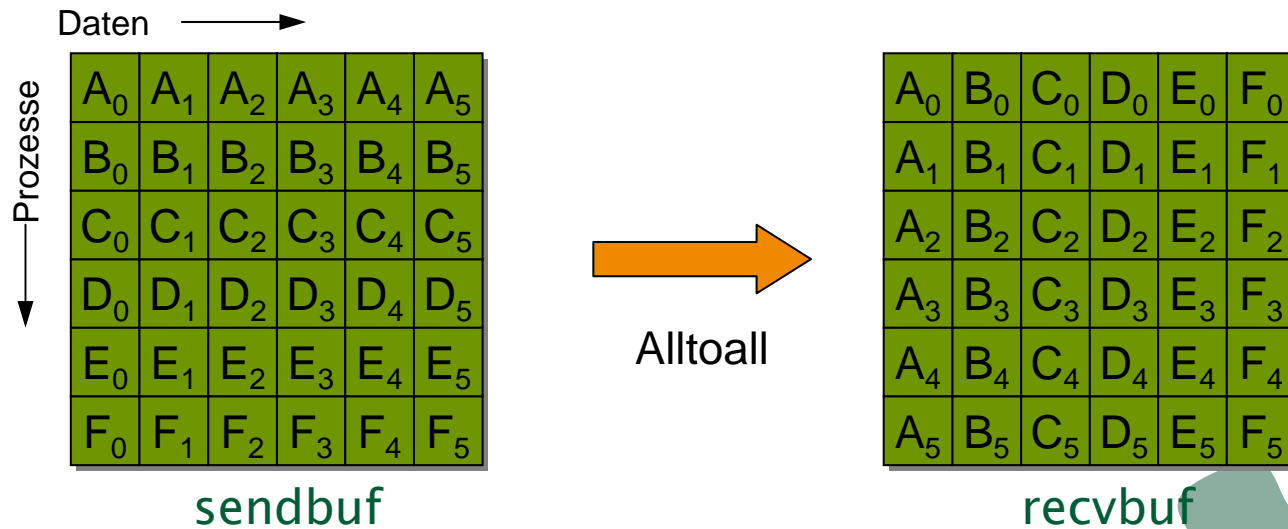
- Elemente von allen werden bei allen gesammelt (wie Gather, aber das Ergebnis findet sich hinterher bei jedem Prozess)  
`MPI_Allgather(void *sbuf, int scnt, MPI_Type stype, void *rbuf, int rcnt, MPI_Type rtype, MPI_Comm comm)`

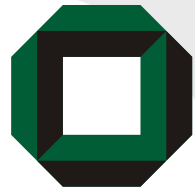




# Kollektive Operationen (6)

- Alle erhalten von allen anderen jeweils ein bestimmtes Element  
`MPI_Alltoall(void *sbuf, int scnt, MPI_Type stype, void *rbuf, int rcnt, MPI_Type rtype, MPI_Comm comm)`



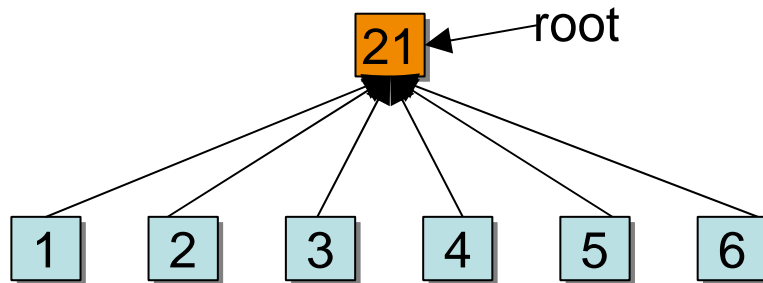


# Reduktionsoperationen (1)

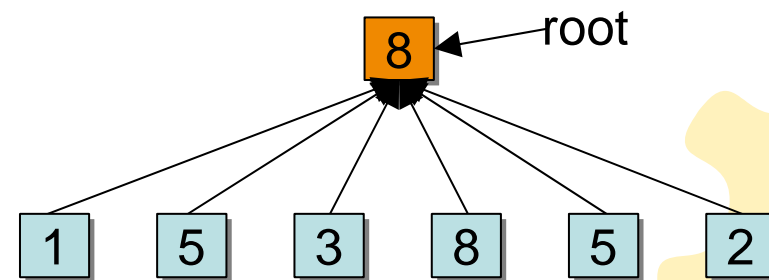
- Globale Operationen auf verteilten Daten:

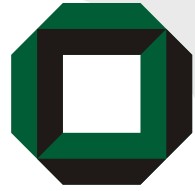
```
MPI_Reduce(void *sendbuf, void *recvbuf,  
           int count, MPI_Type datatype, MPI_Op operator,  
           int root, MPI_Comm comm)
```

op = MPI\_SUM



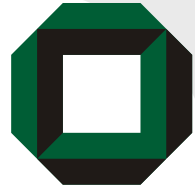
op = MPI\_MAX





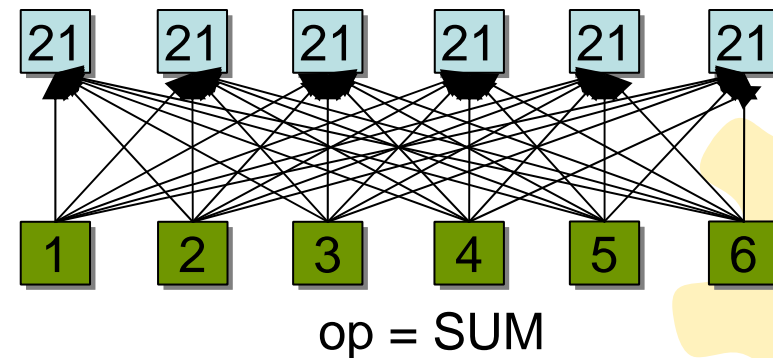
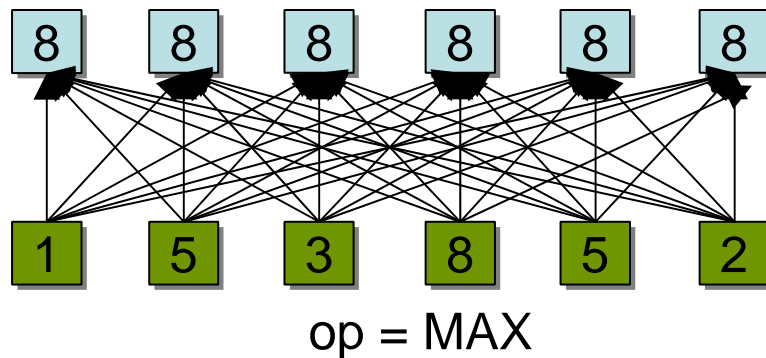
# Reduktionsoperationen (2)

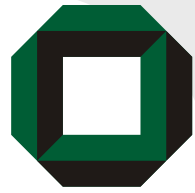
- **Vordefinierte Operatoren:**
  - Maximum, Minimum, Summe, Produkt  
MPI\_MAX, MPI\_MIN, MPI\_SUM, MPI\_PROD
  - Boolesche Operatoren (logische/bitweise Verknüpfung)  
MPI\_BAND, MPI\_BOR  
MPI\_LAND, MPI\_LOR  
MPI\_LXOR, MPI\_BXOR
  - Maximum/Minimum inkl. Position  
MPI\_MAXLOC, MPI\_MINLOC  
(als Datentyp wird ein Paar <Wert, Rang> verwendet)
- **Es sind auch selbstdefinierte (assoziative) Operatoren möglich.**



# Reduktionsoperationen (3)

- Globale Operationen auf verteilten Daten:  
`MPI_Allreduce(void *sendbuf, void *recvbuf,  
int count, MPI_Type datatype  
MPI_Op operator, MPI_Comm comm)`
- Wie Reduce, aber alle erhalten das Ergebnis.

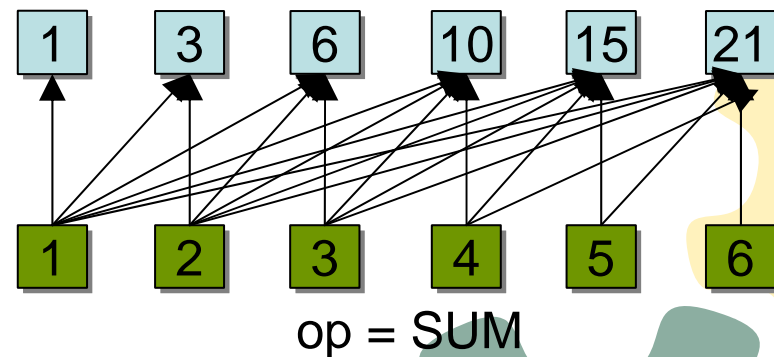
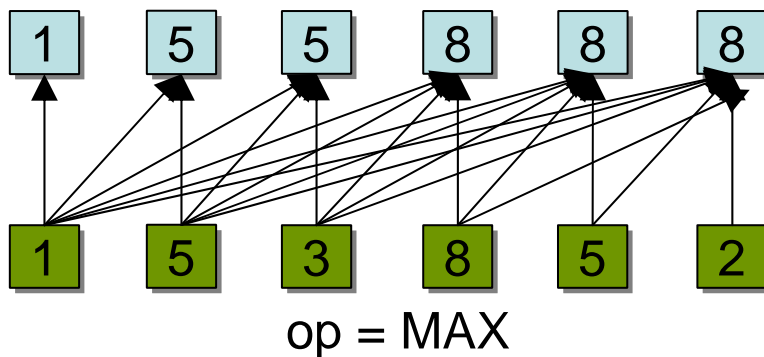


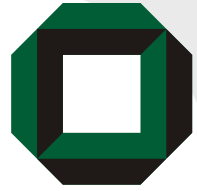


# Reduktionsoperationen (4)

- Globale Operationen auf verteilten Daten mit Zwischenergebnissen (Prozess  $i$  erhält das Ergebnis einer Reduktionsoperation auf den Prozessen  $1\dots i$ , also inklusiv)

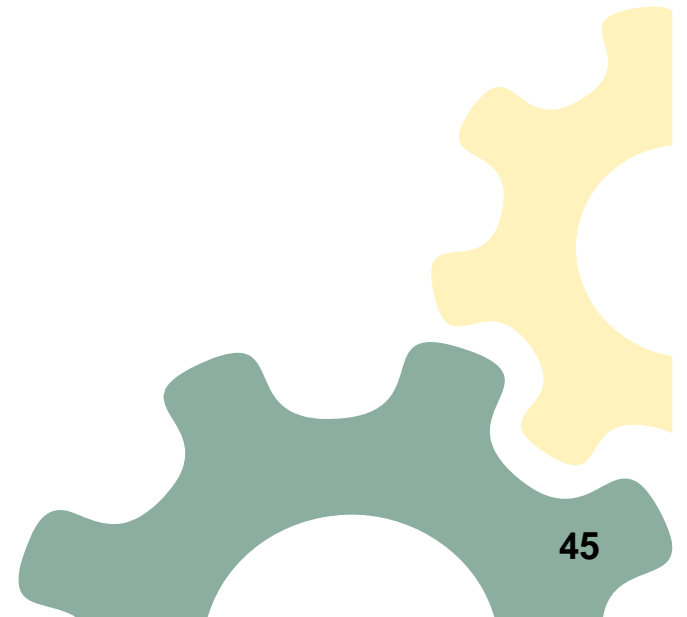
```
MPI_Scan(void *sendbuf, void *recvbuf,  
         int count, MPI_Type datatype,  
         MPI_Op operator, MPI_Comm comm)
```

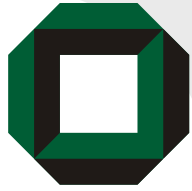




## Beispiel (1)

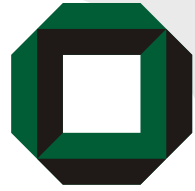
- Gegeben  $n$  Prozesse
- Prozess 1 schickt Nachricht an Prozess 2
- Jeder  $i$ -te Prozess leitet diese weiter an  $i+1$
- Prozess  $n$  gibt Nachricht aus





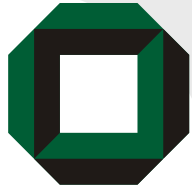
## Beispiel (2)

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv ) {
    int rank, value, size;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    do {
        if (rank == 0) { scanf( "%d", &value );
            MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
        } else {
            MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
            if (rank < size - 1)
                MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
        }
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);
    MPI_Finalize( );
    return 0;
}
```



# Ausblick: MPI-2

- Neue Version des Standards, erweitert MPI-1 in einigen Bereichen
  - Loslösung vom starren Prozessmodell: Starten weiterer Prozesse zur Laufzeit möglich. Mit den neu gestarteten Prozessen kann kommuniziert werden.
  - Kommunikation mit schon laufenden MPI-Anwendungen möglich.
    - Kommunikation über sog. Ports.
    - Finden benannter Ports über eine Registry.
  - Einseitige Kommunikation: Put, Get, Accumulate, sowie Methoden zur Synchronisation,
  - Parallele Datei-Ein- und Ausgabeoperation,
  - Binding für C++,
  - ... weitere Verbesserungen, z.B. bei der Konstruktion von Datentypen.



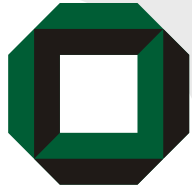
# MPI2: Einseitige Kommunikation

- Mit der kollektiven Operation

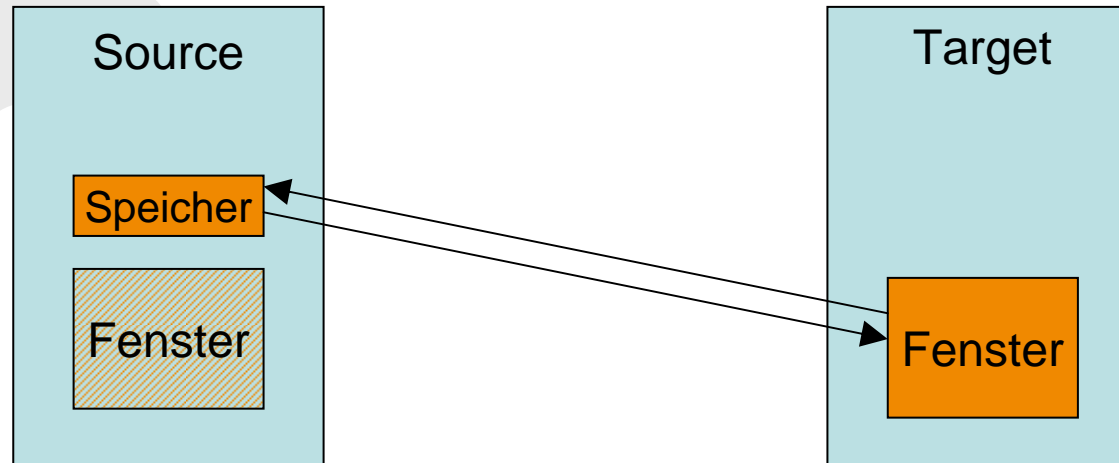
```
MPI_Win_create(void* base, ..., MPI_Communicator  
comm, MPI_Win *win)
```

definiert jeder Prozess aus `comm` einen Speicherbereich, auf den von den anderen Prozessen aus mit `MPI_Put()` oder `MPI_Get()` zugegriffen werden kann.

- `MPI_Put()` und `MPI_Get()` sind einseitige Operationen. Sie werden vom Quellprozess (source) unabhängig vom Zielprozess (target) ausgeführt. Als Zielprozess wird der Prozess bezeichnet, auf dessen Speicherbereich zugegriffen wird.
- Synchronisation: Die Aufrufe zur einseitigen Kommunikation müssen von Aufrufen zur Synchronisation eingerahmt werden. Erst nach Abschluss dieser sind alle Datentransfers abgeschlossen, und die Speicherbereiche enthalten gültige Daten bzw. sind wiederverwendbar.

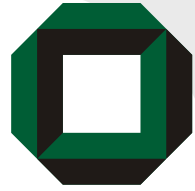


# MPI2: Einseitige Kommunikation



- Ablauf:

- Knoten „source“ greift mittels `get()` und `put()` auf das Fenster des Knotens „target“ zu. Für Quell- bzw. Zieldaten können im Prinzip beliebige Speicherbereiche verwendet werden.
- Knoten „target“ stellt sein Fenster zur Verfügung. Evtl. geänderte Daten sind erst nach Synchronisation sichtbar.



## Weitere Informationen

- Message Passing Interface Forum  
<http://www.mpi-forum.org/>
- Argonne National Lab (Univ. of Chicago)  
<http://www-unix.mcs.anl.gov/mpi/>
- LAM:  
<http://www.lam-mpi.org/>