MPI

Contents

[Allgemein 1](#_Toc68000797)

[MPI Process Ranks 1](#_Toc68000798)

[Message Exchange 2](#_Toc68000799)

[MPI\_Send 2](#_Toc68000800)

[MPI Send Operation Modes 3](#_Toc68000801)

[MPI\_Recv 4](#_Toc68000802)

[Non-blocking Operations 4](#_Toc68000803)

[Global Collective Operations 5](#_Toc68000804)

[MPI\_Bcast 5](#_Toc68000805)

[MPI\_Scatter 6](#_Toc68000806)

[MPI\_Gather 7](#_Toc68000807)

[MPI\_Allgather 7](#_Toc68000808)

[MPI\_Alltoall 8](#_Toc68000809)

[MPI\_Reduce 9](#_Toc68000810)

[Vektor-Varianten 9](#_Toc68000811)

[Synchronization 10](#_Toc68000812)

[MPI\_Barrier 10](#_Toc68000813)

[MPI\_Test und MPI\_Wait 10](#_Toc68000814)

[Aufgaben 10](#_Toc68000815)

[AllToAll 10](#_Toc68000816)

[Implementieren 10](#_Toc68000817)

[Tabelle füllen 11](#_Toc68000818)

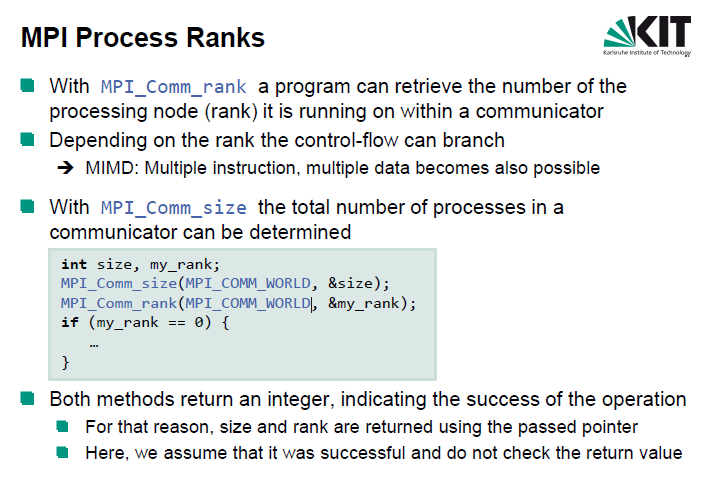
# Allgemein

## Aufruf

mpirun –np N PROGRAM ARGUMENTS

Wobei N - Number of Processes

## MPI Process Ranks und Size



### Rank

Unique number (Identifier) of the current process.

Root hat immer rank 0.

### Size

the total number of processes

## Communicator

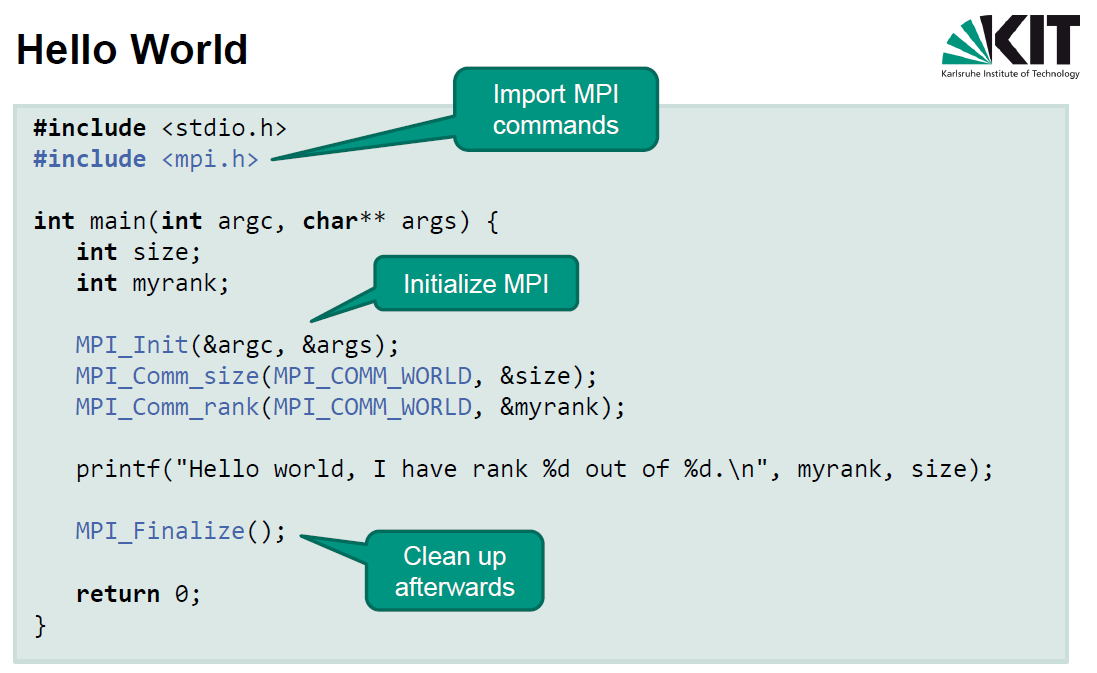
**MPI\_COMM\_WORLD** - default Communicator -> i.e. the collection of all processes

## Lib

**#include <mpi.h>**

## Aufbau

**MPI\_Init** und **MPI\_Finalize** nicht vergessen!



# Message Exchange

**Alle Collective Operations können mit dem Send und Receive implementiert werden!**

## MPI\_Send

**buf :** initial address of send buffer (choice)

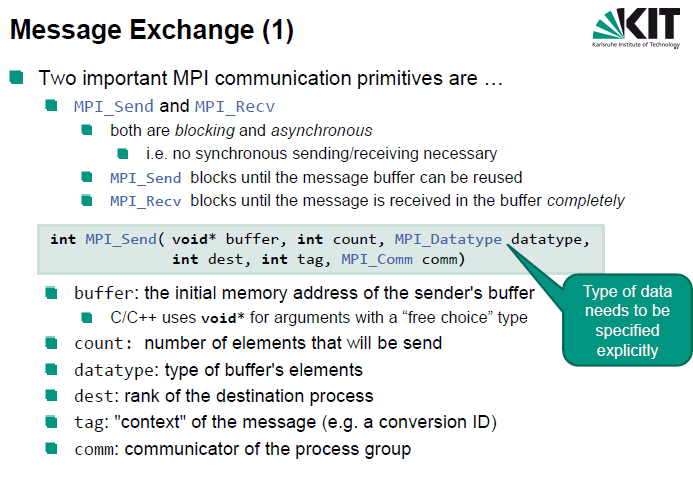
**count :** number of elements in send buffer (nonnegative integer)

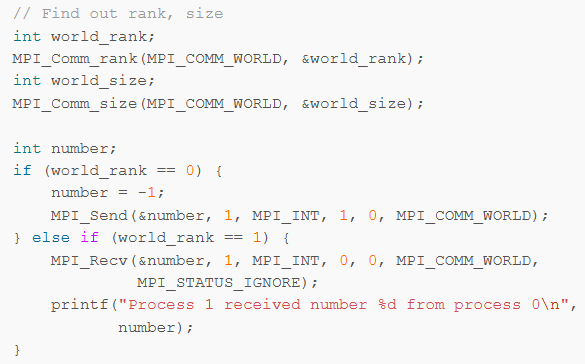
**datatype :** datatype of each send buffer element (handle)

**dest :** rank of destination (integer)

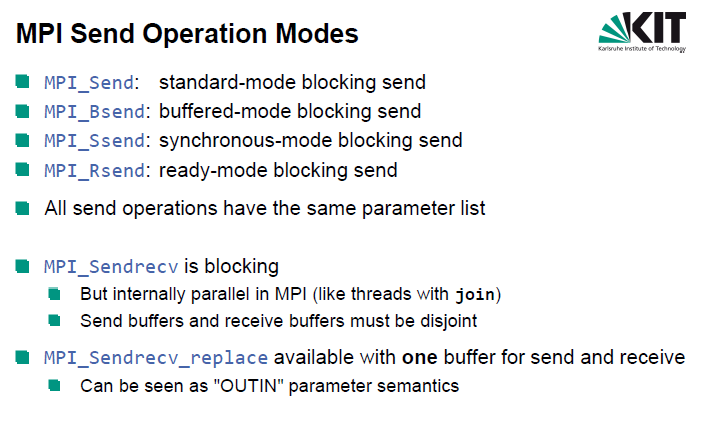
**tag :** message tag (integer)

Tag braucht man in der Prüfung nie (einfach 0 schreiben)

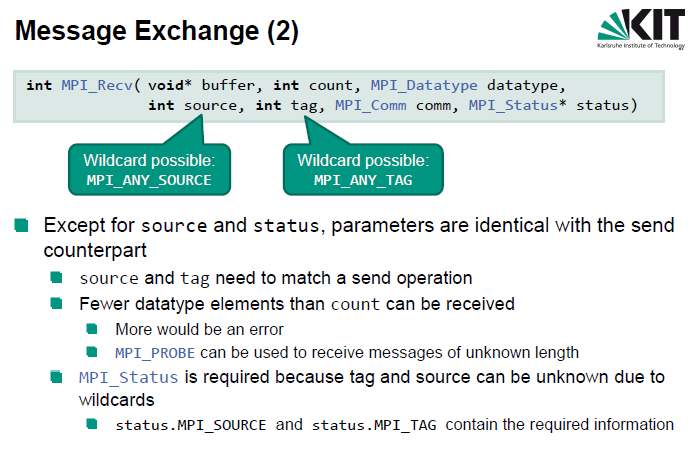




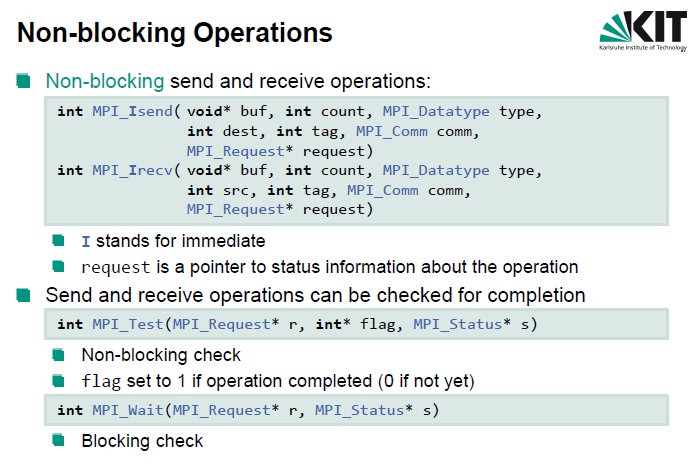
### MPI Send Operation Modes

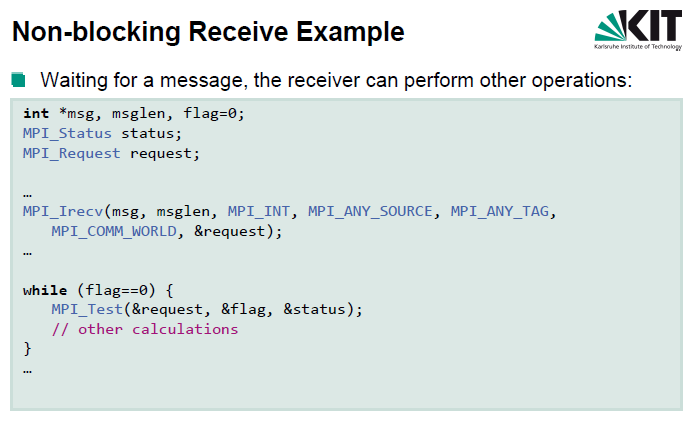


## MPI\_Recv



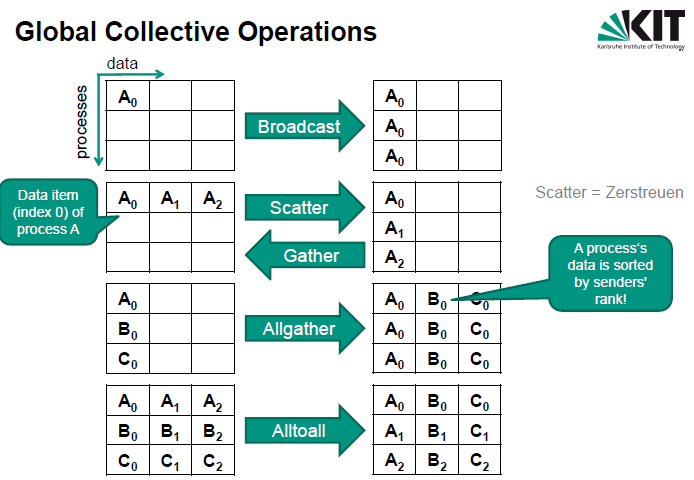
## Non-blocking Operations





# Global Collective Operations

**Alle Collective Operations können mit dem Send und Receive implementiert werden!**



## MPI\_Bcast

Root send his data from buffer to all processes

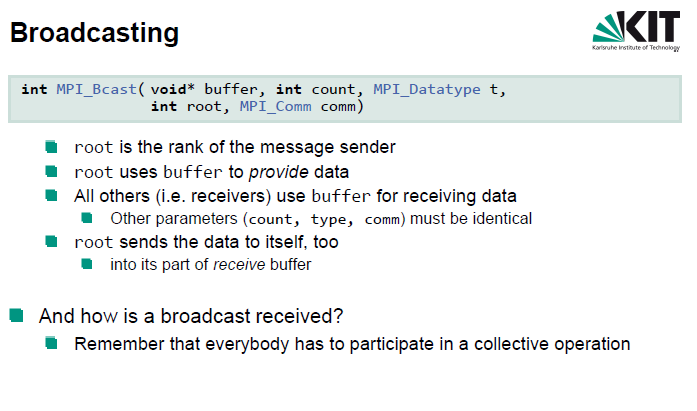
MPI\_Comm comm;

int array[100];

int root=0;

...

MPI\_Bcast( array, 100, MPI\_INT, root, comm);



## MPI\_Scatter

“Streue” Daten von root an alle Prozesse

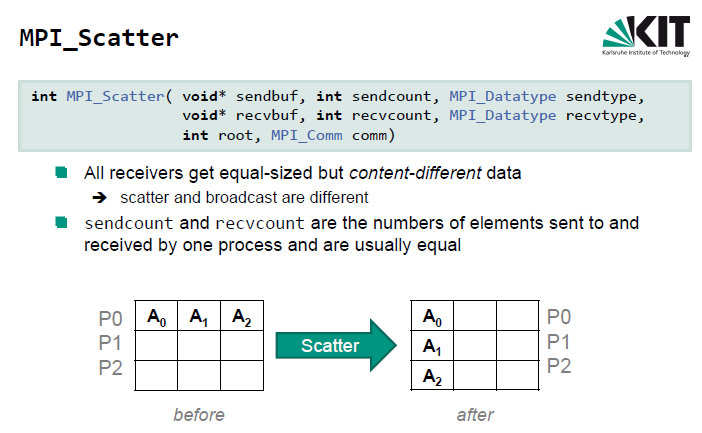
Root ist hier Rang vom Sender, muss nicht 0 sein

**sendcount:** number of elements sent to each process (integer, significant only at root)

**recvcount:**number of elements in receive buffer (integer)

**root:** rank of sending process (integer)

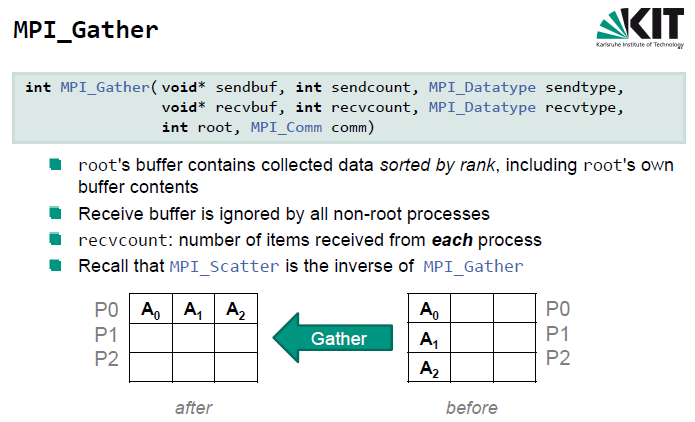
**Es werden bei Scatter alle Werte aus sendbuf geschickt**

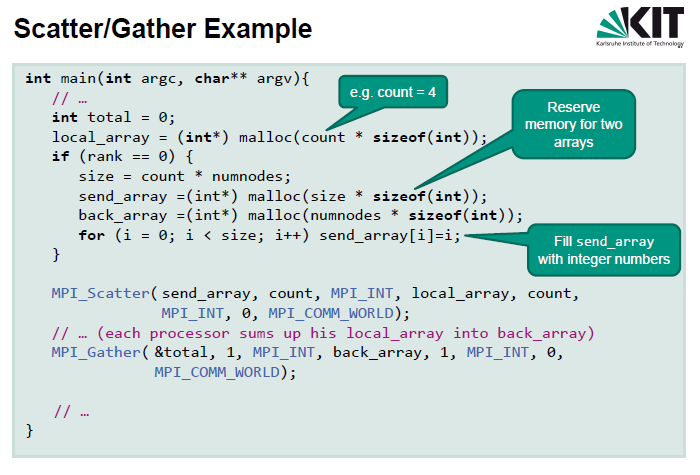


## MPI\_Gather

Sammele Daten von allen Prozessen bei Root

sendcount, revcount, root - wie bei Scatter





## MPI\_Allgather

Daten sammeln mit Gather, dann die Kopien der „ganzen“ Daten verteilen

**sendbuf:** starting address of send buffer (choice)

**sendcount:** number of elements in send buffer (integer)

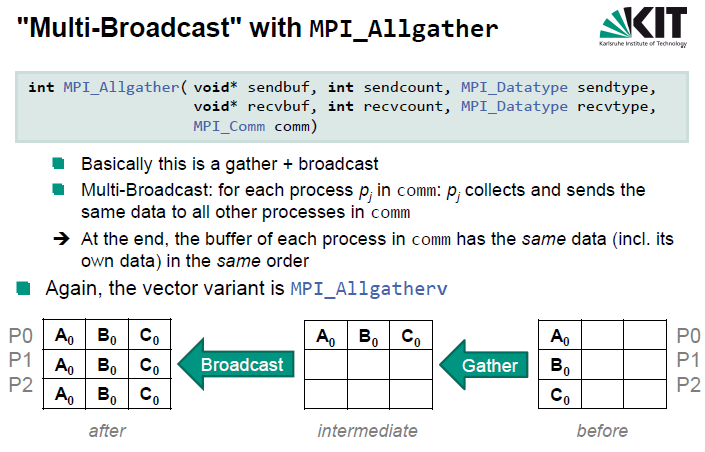
**sendtype:** data type of send buffer elements (handle)

**recvcount:** number of elements received from any process (integer)

**recvtype:** data type of receive buffer elements (handle)

**comm:** communicator (handle)

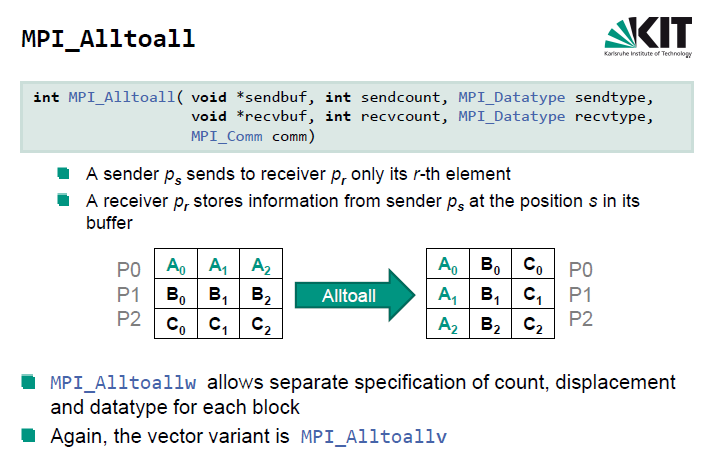
(fast immer gilt: sendtype=recvtype, sendcount=recvcount)



## MPI\_Alltoall

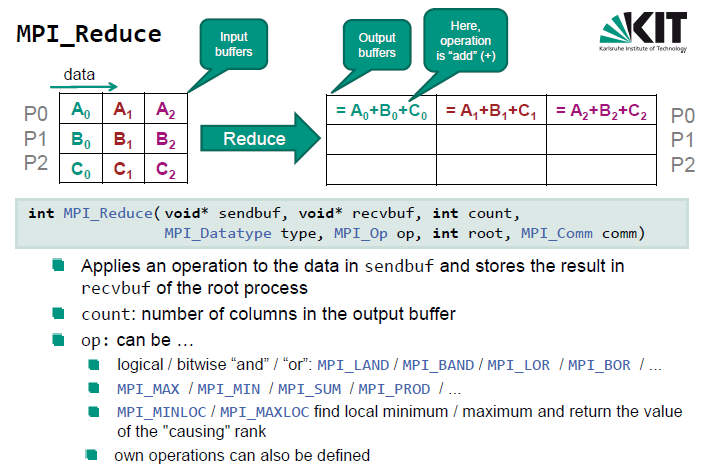
Prozess mit Rang i bekommt alle i-te Datenpakete

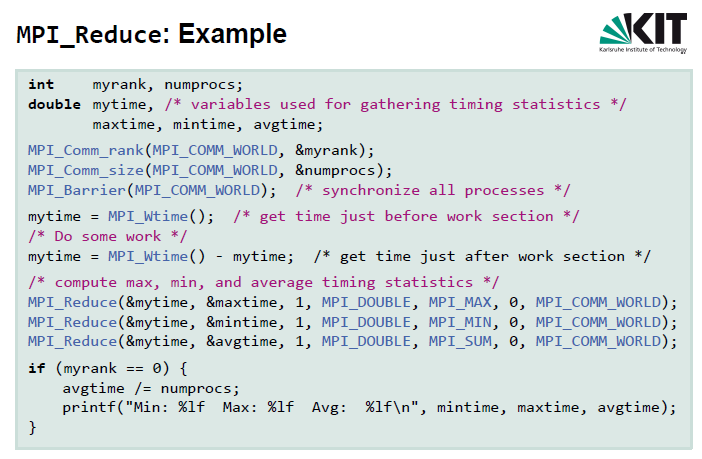
Kann als „Shuffle“ dienen



## MPI\_Reduce

Root bekommt die Daten von allen Prozessen + Reduce mit einer math. Funktion (wie bei fold)





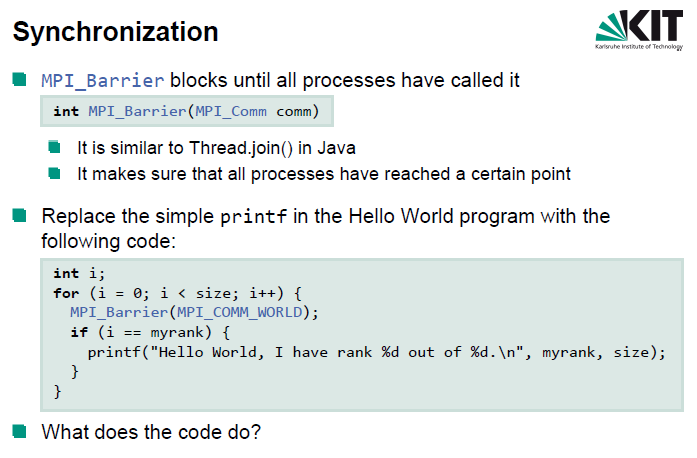
## Vektor-Varianten

Alle MPI-Befehle haben auch eine Vektor-Variante, die mit \*v endet:

MPI\_Gatherv, MPI\_Scatterv, MPI\_Allgatherv...

# Synchronization

## MPI\_Barrier

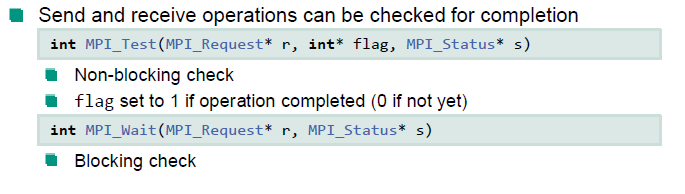


## MPI\_Test und MPI\_Wait

MPI\_Wait Waits for an MPI request to complete

request: [in] request (handle)

status: [out] status object (Status). May be MPI\_STATUS\_IGNORE.



# Aufgaben

## AllToAll

### Implementieren

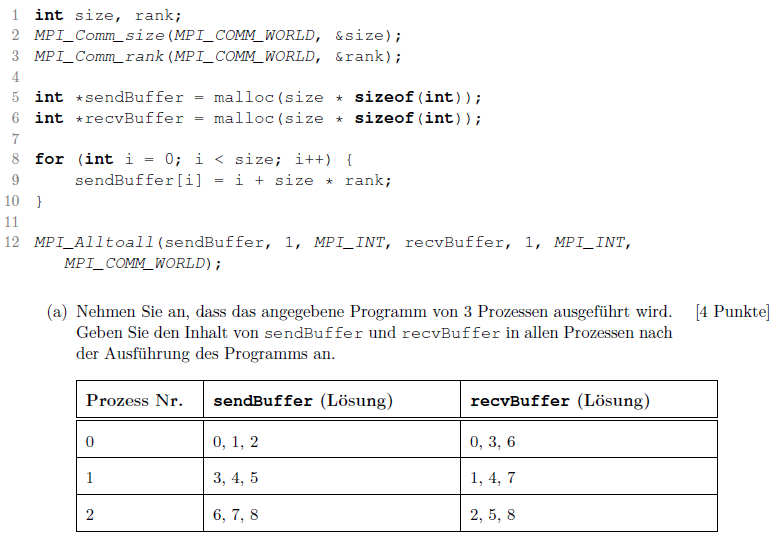
for (int sender = 0; sender < size; sender++) {

MPI\_Scatter(sendBuffer, 1, MPI\_INT, recvBuffer + sender, 1,

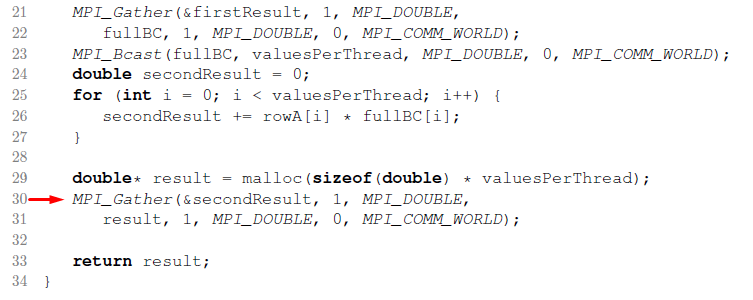
MPI\_INT, sender, MPI\_COMM\_WORLD);

}

### Tabelle füllen



## Collective Operations manuell implementieren: MPI\_Gather



Aufgabe: Erläutern Sie kurz, wie Sie die Zeilen 30 bis 31 *ohne* die Verwendung von kollektiven

MPI-Operationen ausdrücken können. Gehen Sie insbesondere auf das

Verhalten des Root-Prozesses ein.

### Lösung

Die MPI\_Gather-Operation kann, wie alle kollektiven Operationen, durch eine

Menge einzelner Sende- und Empfangsoperationen ausgedrückt werden. In diesem Fall müsste

jeder Prozess eine MPI\_Send-Operation mit seinem lokalen Ergebnis starten, und der Root-

Prozess müsste für jeden vorhandenen Prozess eine MPI\_Recv-Operation starten, um den Wert

zu empfangen. Damit der Root-Prozess auch an sich selbst versenden kann, muss MPI\_ISend

verwendet werden. Alternativ kann auch der Wert des Root-Prozesses direkt in das Ergbnis-

Array geschrieben werden und nur für alle anderen Prozesse eine (normale) Sendeoperation

verwendet werden.

Kurz

Alle Collective Operations können mit dem MPI\_Send und MPI\_Recv manuell implementiert werden. Da Root auch an sich die Nachricht schicken muss, muss man MPI\_ISend verwenden (MPI\_Send verursacht deadlock in diesem Fall)

### Code für Gather

MPI\_Status status;

MPI\_Request request;

**for** (**int** i = 0; i < valuesPerThread; i++) {

**MPI\_Isend** (&secondResult, 1, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, &request);

**if** (rank == 0) {

**MPI\_Recv** (&result[i], 1, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, &status);

}

}

## Variable übergeben statt Array in sendbuf oder recvbuf

Scatter, Gather... brauchen ein Pointer -> Für Variable einfach ein Link übergeben

### Array

send\_array=(int\*) malloc(size\* sizeof(int));

...

MPI\_Scatter(send\_array, count, MPI\_INT, local\_array, count, MPI\_INT, 0, MPI\_COMM\_WORLD);

### Variable

send\_value=42

MPI\_Scatter(&send\_value, 1, MPI\_INT, local\_array, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);