

# MPI

## Contents

Allgemein .....	2
Aufruf .....	2
MPI Process Ranks und Size .....	2
Aufbau .....	3
Message Exchange .....	3
MPI_Send .....	3
MPI Send Operation Modes .....	5
MPI_Recv.....	5
Non-blocking Operations .....	6
Global Collective Operations .....	6
MPI_Bcast.....	7
MPI_Scatter .....	7
MPI_Gather .....	8
MPI_Scatterv .....	9
MPI_Allgather.....	9
MPI_Alltoall .....	10
MPI_Reduce.....	10
Vektor-Varianten .....	11
Synchronization .....	12
MPI_Barrier .....	12
MPI_Test und MPI_Wait .....	12
Aufgaben.....	12
AllToAll .....	12
Implementieren .....	12
Tabelle füllen .....	13
Collective Operations manuell implementieren: MPI_Gather .....	13
Lösung .....	13
Kurz .....	14
Code für Gather .....	14
Variable übergeben statt Array in sendbuf oder recvbuf .....	14
Array.....	14

Variable .....	14
Scatter == Send different blocks of array to different proc .....	14
Triangle arrays .....	15
Scatter vs ScatterV .....	15

## Allgemein

### Aufruf

`mpirun -np N PROGRAM ARGUMENTS`

Wobei N - Number of Processes

### MPI Process Ranks und Size

#### MPI Process Ranks



- With `MPI_Comm_rank` a program can retrieve the number of the processing node (rank) it is running on within a communicator
- Depending on the rank the control-flow can branch
  - MIMD: Multiple instruction, multiple data becomes also possible
- With `MPI_Comm_size` the total number of processes in a communicator can be determined

```
int size, my_rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank == 0) {
    ...
}
```

- Both methods return an integer, indicating the success of the operation
  - For that reason, size and rank are returned using the passed pointer
  - Here, we assume that it was successful and do not check the return value

**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!

### Hello World



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

int main(int argc, char** args) {
    int size;
    int myrank;

    MPI_Init(&argc, &args);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    printf("Hello world, I have rank %d out of %d.\n", myrank, size);

    MPI_Finalize();

    return 0;
}
```

Import MPI commands

Initialize MPI

Clean up afterwards

## 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)

## Message Exchange (1)



### ■ Two important MPI communication primitives are ...

#### ■ `MPI_Send` and `MPI_Recv`

- both are *blocking* and *asynchronous*
  - i.e. no synchronous sending/receiving necessary
- `MPI_Send` blocks until the message buffer can be reused
- `MPI_Recv` blocks until the message is received in the buffer *completely*

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

- `buffer`: the initial memory address of the sender's buffer
  - C/C++ uses `void*` for arguments with a "free choice" type
- `count`: number of elements that will be send
- `datatype`: type of buffer's elements
- `dest`: rank of the destination process
- `tag`: "context" of the message (e.g. a conversion ID)
- `comm`: communicator of the process group

Type of data  
needs to be  
specified  
explicitly

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

## MPI Send Operation Modes

### MPI Send Operation Modes



- `MPI_Send`: standard-mode blocking send
- `MPI_Bsend`: buffered-mode blocking send
- `MPI_Ssend`: synchronous-mode blocking send
- `MPI_Rsend`: ready-mode blocking send
- All send operations have the same parameter list
- `MPI_Sendrecv` is blocking
  - But internally parallel in MPI (like threads with `join`)
  - Send buffers and receive buffers must be disjoint
- `MPI_Sendrecv_replace` available with **one** buffer for send and receive
  - Can be seen as "OUTIN" parameter semantics

## MPI\_Recv

### Message Exchange (2)



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

Wildcard possible:  
`MPI_ANY_SOURCE`

Wildcard possible:  
`MPI_ANY_TAG`

- Except for source and status, parameters are identical with the send counterpart
  - source and tag need to match a send operation
  - Fewer datatype elements than count can be received
    - More would be an error
    - `MPI_PROBE` can be used to receive messages of unknown length
  - `MPI_Status` is required because tag and source can be unknown due to wildcards
    - `status.MPI_SOURCE` and `status.MPI_TAG` contain the required information

## Non-blocking Operations

### Non-blocking Operations



- Non-blocking send and receive operations:

```
int MPI_Isend( void* buf, int count, MPI_Datatype type,
               int dest, int tag, MPI_Comm comm,
               MPI_Request* request)
int MPI_Irecv( void* buf, int count, MPI_Datatype type,
               int src, int tag, MPI_Comm comm,
               MPI_Request* request)
```

- I stands for immediate
- request is a pointer to status information about the operation
- Send and receive operations can be checked for completion

```
int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s)
```

- Non-blocking check
- flag set to 1 if operation completed (0 if not yet)

```
int MPI_Wait(MPI_Request* r, MPI_Status* s)
```

- Blocking check

### Non-blocking Receive Example



- Waiting for a message, the receiver can perform other operations:

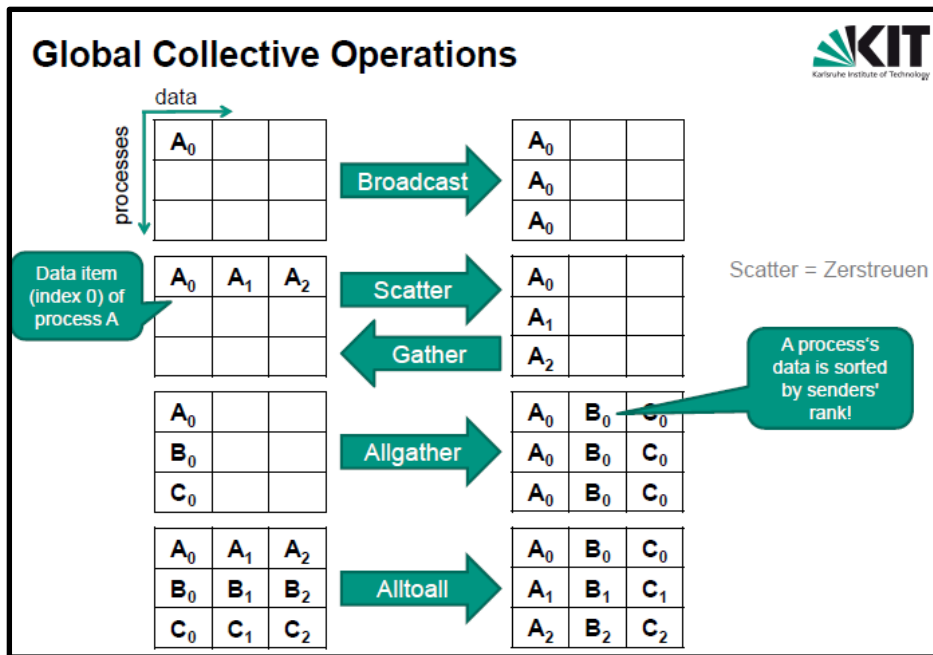
```
int *msg, msglen, flag=0;
MPI_Status status;
MPI_Request request;

...
MPI_Irecv(msg, msglen, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
          MPI_COMM_WORLD, &request);
...

while (flag==0) {
    MPI_Test(&request, &flag, &status);
    // other calculations
}
...
```

## 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);
```

## Broadcasting


  
Karlsruhe Institute of Technology

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

- root is the rank of the message sender
- root uses buffer to *provide* data
- All others (i.e. receivers) use buffer for receiving data
  - Other parameters (count, type, comm) must be identical
- root sends the data to itself, too
  - into its part of *receive* buffer
- And how is a broadcast received?
  - Remember that everybody has to participate in a collective operation

## 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)

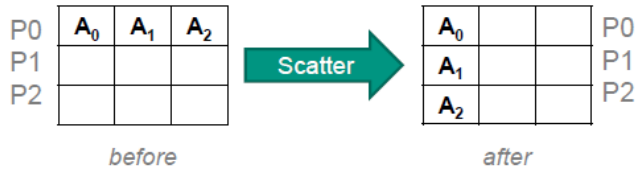
Es werden bei Scatter alle Werte aus `sendbuf` des `roots` geschickt

### MPI\_Scatter



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

- All receivers get equal-sized but *content-different* data  
→ scatter and broadcast are different
- `sendcount` and `recvcount` are the numbers of elements sent to and received by one process and are usually equal




## MPI\_Gather

Sammele Daten von allen Prozessen bei Root


`sendcount`, `recvcount`, `root` - wie bei Scatter

### MPI\_Gather



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

- `root`'s buffer contains collected data *sorted by rank*, including `root`'s own buffer contents
- Receive buffer is ignored by all non-`root` processes
- `recvcount`: number of items received from *each* process
- Recall that `MPI_Scatter` is the inverse of `MPI_Gather`





## Scatter/Gather Example

```
int main(int argc, char** argv){
    // ...
    int total = 0;
    local_array = (int*) malloc(count * sizeof(int));
    if (rank == 0) {
        size = count * numnodes;
        send_array = (int*) malloc(size * sizeof(int));
        back_array = (int*) malloc(numnodes * sizeof(int));
        for (i = 0; i < size; i++) send_array[i]=i;

        MPI_Scatter(send_array, count, MPI_INT, local_array, count,
                   MPI_INT, 0, MPI_COMM_WORLD);
        // ... (each processor sums up his local_array into back_array)
        MPI_Gather(&total, 1, MPI_INT, back_array, 1, MPI_INT, 0,
                  MPI_COMM_WORLD);

        // ...
    }
}
```

e.g. count = 4

Reserve memory for two arrays

Fill send\_array with integer numbers

## MPI\_Scatterv

**sendcounts** i-te entry = n -> process i bekommt n elemente

**displs** i-te entry = x -> process i bekommt n elemente ab array[x] (start point of data stream for process i)

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

## Vector variant of MPI\_Scatter

```
int MPI_Scatterv( void* sendbuf, int* sendcounts, int* displacements,
                  MPI_Datatype sendtype, void* recvbuf, int recvcount,
                  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Allows varying counts for data sent to each process
- **sendcounts**: integer array with the number of elements to send to each process
- **displacements**: integer array, entry *i* specifies the displacement relative to sendbuf from which to take the outgoing data to process *i* (gaps allowed but no overlaps)
- **sendtype**: data type of send buffer elements (handle)
- **recvcount**: number of elements in receive buffer (integer)
- **recvtype**: data type of receive buffer elements (handle)

## 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)

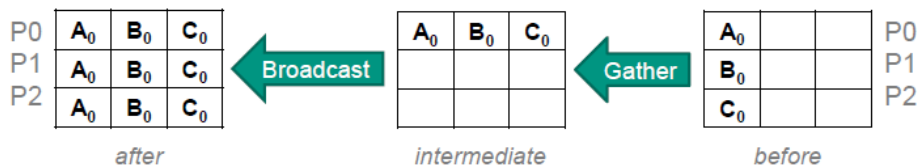
## "Multi-Broadcast" with MPI\_Allgather



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

- Basically this is a gather + broadcast
- Multi-Broadcast: for each process  $p_j$  in `comm`:  $p_j$  collects and sends the same data to all other processes in `comm`
  - At the end, the buffer of each process in `comm` has the same data (incl. its own data) in the same order

- Again, the vector variant is `MPI_Allgatherv`



## MPI\_Alltoall

Prozess mit Rang i bekommt alle i-te Datenpakete

Kann als „Shuffle“ dienen

### MPI\_Alltoall



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

- A sender  $p_s$  sends to receiver  $p_r$  only its  $r$ -th element
- A receiver  $p_r$  stores information from sender  $p_s$  at the position  $s$  in its buffer



- `MPI_Alltoallw` allows separate specification of count, displacement and datatype for each block
- Again, the vector variant is `MPI_Alltoallv`

## MPI\_Reduce

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

**Reduce kann man implementieren als:**

AlltoAll + for-Schleife mit reduce-Operation (als array -> variable) -> Gather

## MPI\_Reduce

data →

P0	A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>
P1	B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>
P2	C <sub>0</sub>	C <sub>1</sub>	C <sub>2</sub>

Reduce

P0	= A <sub>0</sub> +B <sub>0</sub> +C <sub>0</sub>	= A <sub>1</sub> +B <sub>1</sub> +C <sub>1</sub>	= A <sub>2</sub> +B <sub>2</sub> +C <sub>2</sub>
P1			
P2			

```

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

- Applies an operation to the data in sendbuf and stores the result in recvbuf of the root process
- count: number of columns in the output buffer
- op: can be ...
  - logical / bitwise "and" / "or": MPI\_BAND / MPI\_BAND / MPI\_LOR / MPI BOR / ...
  - MPI\_MAX / MPI\_MIN / MPI\_SUM / MPI\_PROD / ...
  - MPI\_MINLOC / MPI\_MAXLOC find local minimum / maximum and return the value of the "causing" rank
  - own operations can also be defined

## MPI\_Reduce: Example

```

int    myrank, numprocs;
double mytime, /* variables used for gathering timing statistics */
       maxtime, mintime, avgtime;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Barrier(MPI_COMM_WORLD); /* synchronize all processes */
mytime = MPI_Wtime(); /* get time just before work section */
/* Do some work */
mytime = MPI_Wtime() - mytime; /* get time just after work section */
/* compute max, min, and average timing statistics */
MPI_Reduce(&mytime, &maxtime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Reduce(&mytime, &mintime, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
MPI_Reduce(&mytime, &avgtime, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (myrank == 0) {
    avgtime /= numprocs;
    printf("Min: %lf Max: %lf Avg: %lf\n", mintime, maxtime, avgtime);
}
  
```

## Vektor-Varianten

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

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

**Scatter -> Gleiche Länge der Blöcke, bei ScatterV -> einstellbare Länge der Blöcke**

# Synchronization

## MPI\_Barrier

### Synchronization



- `MPI_Barrier` blocks until all processes have called it

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

- It is similar to `Thread.join()` in Java
- It makes sure that all processes have reached a certain point
- Replace the simple `printf` in the Hello World program with the following code:

```
int i;  
for (i = 0; i < size; i++) {  
    MPI_Barrier(MPI_COMM_WORLD);  
    if (i == myrank) {  
        printf("Hello World, I have rank %d out of %d.\n", myrank, size);  
    }  
}
```

- What does the code do?

## 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`.

- Send and receive operations can be checked for completion

```
int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s)
```

- Non-blocking check
- flag set to 1 if operation completed (0 if not yet)

```
int MPI_Wait(MPI_Request* r, MPI_Status* s)
```

- Blocking check

## 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

```
1 int size, rank;
2 MPI_Comm_size(MPI_COMM_WORLD, &size);
3 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4
5 int *sendBuffer = malloc(size * sizeof(int));
6 int *recvBuffer = malloc(size * sizeof(int));
7
8 for (int i = 0; i < size; i++) {
9     sendBuffer[i] = i + size * rank;
10 }
11
12 MPI_Alltoall(sendBuffer, 1, MPI_INT, recvBuffer, 1, MPI_INT,
    MPI_COMM_WORLD);
```

- (a) Nehmen Sie an, dass das angegebene Programm von 3 Prozessen ausgeführt wird. [4 Punkte]  
Geben Sie den Inhalt von `sendBuffer` und `recvBuffer` in allen Prozessen nach der Ausführung des Programms an.

Prozess Nr.	sendBuffer (Lösung)	recvBuffer (Lösung)
0	0, 1, 2	0, 3, 6
1	3, 4, 5	1, 4, 7
2	6, 7, 8	2, 5, 8

## Collective Operations manuell implementieren: MPI\_Gather

```
21 MPI_Gather(&firstResult, 1, MPI_DOUBLE,
22     fullBC, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
23 MPI_Bcast(fullBC, valuesPerThread, MPI_DOUBLE, 0, MPI_COMM_WORLD);
24 double secondResult = 0;
25 for (int i = 0; i < valuesPerThread; i++) {
26     secondResult += rowA[i] * fullBC[i];
27 }
28
29 double* result = malloc(sizeof(double) * valuesPerThread);
30 → MPI_Gather(&secondResult, 1, MPI_DOUBLE,
31     result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
32
33 return result;
34 }
```

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 Ergebnis-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);
```

## Scatter == Send different blocks of array to different proc

```
1 void operation(int elementCount, int elements[elementCount]) {
2     int rank;
3     int processCount;
4     MPI_Status status;
5     MPI_Request request;
6
7     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8     MPI_Comm_size(MPI_COMM_WORLD, &processCount);
9
10    int elementsPerProcess = elementCount / processCount;
11    int local[elementsPerProcess];
12    if (rank == 0) {
13        for (int i = 0; i < processCount; i++) {
14            MPI_Isend(&elements[i*elementsPerProcess], elementsPerProcess,
15                    MPI_INT, i, 0, MPI_COMM_WORLD, &request);
16        }
17    }
18    MPI_Recv(local, elementsPerProcess, MPI_INT, 0, 0,
19            MPI_COMM_WORLD, &status);
```

## Triangle arrays

```
14     int received[(rank + 1) * elementsPerProcess];
15
16     for (int i = rank; i < size; i++){
17         MPI_Isend(localElements, elementsPerProcess, MPI_INT, i, 0,
18                 MPI_COMM_WORLD, &request);
19     }
20     for (int i = 0; i <= rank; i++){
21         MPI_Recv(received + (i * elementsPerProcess), elementsPerProcess,
22                 MPI_INT, i, 0, MPI_COMM_WORLD, &status);
23     }
```

Sei: size = 9, 3 processes, localElements = [1,2,3,4,5,6,7,8,9]

Nach Zeile 22:

0: received = [1, 2, 3]

1: received = [1, 2, 3, 4, 5, 6]

2: received = [1, 2, 3, 4, 5, 6, 7, 8, 9]

## Scatter vs ScatterV

**Scatter -> Gleiche Länge der Blöcke, bei ScatterV -> einstellbare Länge**

In der gegebenen Implementierung besteht die Einschränkung, dass die Anzahl der Elemente in digitSequence ein Vielfaches der Anzahl von gestarteten Prozessen sein muss. Erklären Sie kurz, wodurch sich die Notwendigkeit für diese Beschränkung im Bezug auf die gegebene Implementierung ergibt. Beschreiben Sie stichpunktartig, wie Sie die Implementierung ändern müssen, damit das Programm auch ohne diese Einschränkung korrekte Ergebnisse liefert. Gehen Sie jedoch weiterhin davon aus, dass das Programm mit 3 Prozessen gestartet wird. [4 Punkte]

**Beispiellösung:** Die Einschränkung ergibt sich nur durch die Anwendung von MPI\_Scatter in Zeile 13, die die restlichen kollektiven MPI-Operationen immer auf Arrays fixer Länge (3) arbeiten. MPI\_Scatter kann nur Blöcke gleicher Größe an alle Prozesse verteilen. Hierdurch ergibt sich die Einschränkung an die Größe von input auf ein Vielfaches der Anzahl an Prozessen, da nur so alle int-Zahlen in input korrekt auf die Prozesse verteilt werden können. Um diese Einschränkung zu beheben, kann die Vektor-Variante MPI\_Scatterv verwendet werden. MPI\_Scatterv ermöglicht es für jeden Empfänger-Prozess eine unterschiedliche Anzahl an Elementen zu definieren welche dieser zugesendet bekommen soll. Die Parametrisierung von MPI\_Scatterv unterscheidet sich hierbei, unter anderem, dadurch von MPI\_Scatter, dass anstatt einer festen Zahl der zu versendenden Elemente für alle Prozesse, ein Array mit einer Anzahl an zu versendenden Elementen für jeden Prozess übergeben werden kann. Somit muss für jeden Prozess zunächst eine Anzahl an Elementen berechnet werden, so dass die Summe der Anzahlen die Länge von input (inputLength) ergibt.