Parallel Processing

Winter Term 2024/25

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: December 16, 2024

Parallel Processing

Winter Term 2024/25

4 Parallel Programming with Message Passing

Contents

Parallel Processing

Winter Term 2024/25

09.12.2024

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

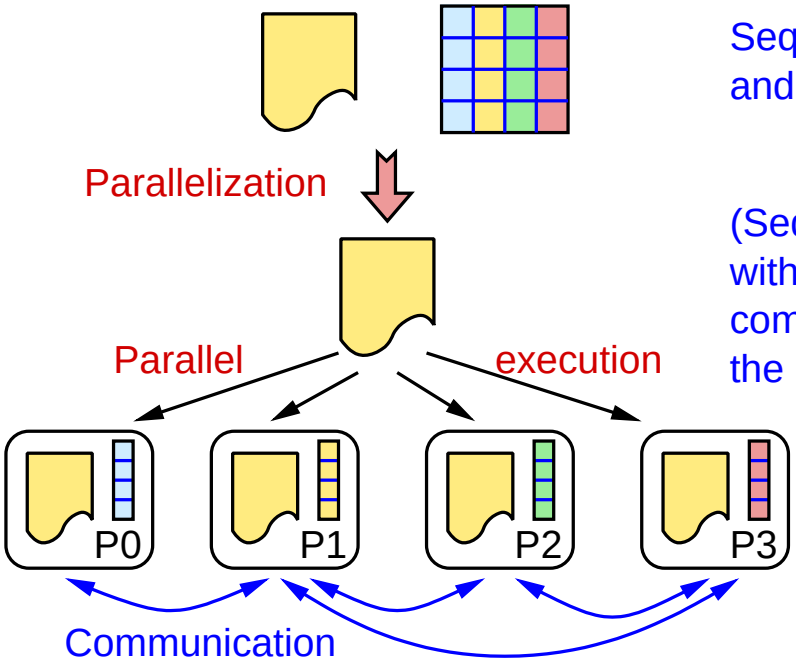
Stand: December 16, 2024

Evaluation

- You all have got an invitation link for the evaluation of this lecture
- Please fill the questionaire **right now**!
  - only evaluate the lecture, the lab is evaluated separately

4.1 Typical approach

Data partitioning with SPMD model



Sequential program and data partitioning

(Sequential) node program with message passing performs computations for a part of the data

Identical copies of the program, executed in parallel using multiple processes

Activities when creating a node program

- Adjustment of array declarations
  - node program stores only a part of the data
  - (assumption: data are stored in arrays)
- Index transformation
  - global index ↔ (process number, local index)
- Work partitioning
  - each process executes the computations on its part of the data
- Communication
  - when a process needs non-local data, a suitable message exchange must be programmed

About communication

- When a process needs data: the owner of the data must send them explicitly
  - exception: one-sided communication (☞ **4.11**)
- Communication should be merged as much as possible
  - one large message is better than many small ones
  - however, data dependences must not be violated

Sequential execution

```
a[1] = ...;
a[2] = ...;
a[3] = a[1]+...;
a[4] = a[2]+...;
```

Parallel execution

Process 1

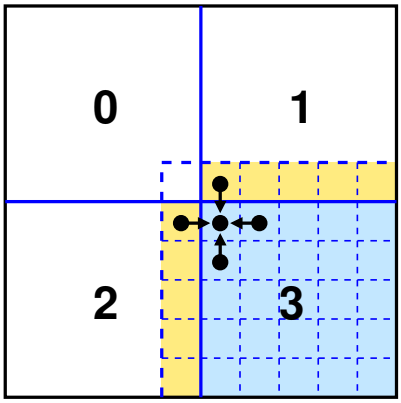
```
a[1] = ...;
a[2] = ...;
send(a[1],a[2]);
```

Process 2

```
recv(a[1],a[2]);
a[3] = a[1]+...;
a[4] = a[2]+...;
```

About communication ...

- Often the node program allocates an overlapping buffer region (**ghost region** / **ghost cells**) for non-local data
- Example: Jacobi iteration



Partitioning of the matrix into 4 parts

Each process allocates an additional row/column at the borders of its sub−matrix

Data exchange at the end of each iteration

4.2 MPI (*Message Passing Interface*)

History and background

- At the beginning of the parallel computer era (late 1980's):
  - many different communication libraries (NX, PARMACS, PVM, P4, ...)
  - parallel programs are not easily portable
- Definition of an informal standard by the MPI forum
  - 1994: MPI-1.0
  - 1997: MPI-1.2 and MPI-2.0 (considerable extensions)
  - 2009: MPI 2.2 (clarifications, minor extensions)
  - 2012/15: MPI-3.0 und MPI-3.1 (considerable extensions)
  - documents at `http://www.mpi-forum.org/docs`
- MPI only defines the API (i.e., the programming interface)
  - different implementations, e.g., MPICH2, OpenMPI, ...

Programming model

- Distributed memory, processes with message passing
- SPMD: one program code for all processes
  - but different program codes are also possible
- MPI-1: static process model
  - all processes are created at program start
    - program start is standardized since MPI-2
  - MPI-2 also allows to create new processes at runtime
- MPI is thread safe: a process is allowed to create additional threads
  - hybrid parallelization using MPI and OpenMP is possible
- Program terminates when all its processes have terminated

4.3 MPI Core routines

- MPI-1.2 has 129 routines (and MPI-2 even more ...)
- However, often only 6 routines are sufficient to write relevant programs:
  - `MPI_Init` – MPI initialization
  - `MPI_Finalize` – MPI cleanup
  - `MPI_Comm_size` – get number of processes
  - `MPI_Comm_rank` – get own process number
  - `MPI_Send` – send a message
  - `MPI_Recv` – receive a message

MPI_Init

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

INOUT	**argc**	Pointer to **argc** of **main()**
INOUT	**argv**	Pointer to **argv** of **main()**
Result		**MPI_SUCCESS** or error code

- Each MPI process must call `MPI_Init`, before it can use other MPI routines
- Typically:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ...
```

- `MPI_Init` may also ensure that all processes receive the command line arguments

MPI_Finalize

```
int MPI_Finalize()
```

- Each MPI process must call `MPI_Finalize` at the end
- Main purpose: deallocation of resources
- After that, no other MPI routines must be used
  - in particular, no further `MPI_Init`
- `MPI_Finalize` does **not** terminate the process!

MPI_Comm_size

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

IN	**comm**	Communicator
OUT	**size**	Number of processes in **comm**

- Typically: `MPI_Comm_size(MPI_COMM_WORLD, &nprocs)`
  - returns the number of MPI processes in `nprocs`

MPI_Comm_rank

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

IN	**comm**	Communicator
OUT	**rank**	Number of processes in **comm**

- Process number (“rank”) counts upward, starting at 0
  - only differentiation of the processes in the SPMD model

Communicators

- A communicator consists of
  - a process group
    - a subset of all processes of the parallel application
  - a communication context
    - to allow the separation of different communication relations (☞ 4.7)
- There is a predefined communicator `MPI_COMM_WORLD`
  - its process group contains all processes of the parallel application
- Additional communicators can be created as needed (☞ 4.7)

MPI_Send

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

IN	**buf**	(Pointer to) the data to be sent (send buffer)
IN	**count**	Number of data elements (of type **dtype**)
IN	**dtype**	Data type of the individual data elements
IN	**dest**	Rank of destination process in communicator **comm**
IN	**tag**	Message tag
IN	**comm**	Communicator

- Specification of data type: for format conversions
- Destination process is always relative to a communicator
- *Tag* allows to distinguish different messages (or message types) in the program

MPI_Send ...

- `MPI_Send` blocks the calling process, until all data has been read from the send buffer
  - send buffer can be reused (i.e., modified) immediately after `MPI_Send` returns
- The MPI implementation decides whether the process is blocked until
  a) the data has been copied to a system buffer, or
  b) the data has been received by the destination process.
  - in some cases, this decision can influence the correctness of the program! (☞ slide 323)

MPI_Recv

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

OUT	**buf**	(Pointer to) receive buffer
IN	**count**	Buffer size (number of data elements of type **dtype**)
IN	**dtype**	Data type of the individual data elements
IN	**source**	Rank of source process in communicator **comm**
IN	**tag**	Message tag
IN	**comm**	Communicator
OUT	**status**	Status (among others: actual message length)

- Process is blocked until the message has been completely received and stored in the receive buffer

MPI_Recv ...

- `MPI_Recv` only receives a message where
  - sender,
  - message tag, and
  - communicator

  match the parameters
- For source process (sender) and message tag, wild-cards can be used:
  - `MPI_ANY_SOURCE`: sender doesn't matter
  - `MPI_ANY_TAG`: message tag doesn't matter

MPI_Recv ...

- Message must not be larger than the receive buffer
  - but it may be smaller; the unused part of the buffer remains unchanged
- From the return value `status` you can determine:
  - the sender of the message: `status.MPI_SOURCE`
  - the message tag: `status.MPI_TAG`
  - the error code: `status.MPI_ERROR`
  - the actual length of the received message (number of data elements): `MPI_Get_count(&status, dtype, &count)`

Simple data types (`MPI_Datatype`)

MPI	C/C++	MPI	C/C++
`MPI_CHAR`	`char`	`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_SHORT`	`short`	`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_INT`	`int`	`MPI_UNSIGNED`	`unsigned int`
`MPI_LONG`	`long`	`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_FLOAT`	`float`		
`MPI_DOUBLE`	`double`	`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	Byte with 8 bits	`MPI_PACKED`	Packed data*

*☞ 4.10

Example: typical MPI program skeleton (☞ 04/rahmen.cpp)

```
#include <iostream>
#include <mpi.h>
using namespace std;

int main (int argc, char **argv)
{
    int i;
    int myrank, nprocs;
    int namelen;
    char name[MPI_MAX_PROCESSOR_NAME];

    /* Initialize MPI and set the command line arguments */
    MPI_Init(&argc, &argv);

    /* Determine the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```


```
    /* Determine the own rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    /* Determine the node name */
    MPI_Get_processor_name(name, &namelen);

    /* flush is used to enforce immediate output */
    cout << "Process " << myrank << "/" << nprocs
         << "started on " << name << "\n" << flush;

    cout << "-- Arguments: ";
    for (i = 0; i<argc; i++)
        cout << argv[i] << " ";
    cout << "\n";

    /* finish MPI */
    MPI_Finalize();

    return 0;
}
```

Starting MPI programs: `mpiexec`

- `mpiexec -n 3 myProg arg1 arg2`
  - starts `myProg arg1 arg2` with 3 processes
  - the specification of the nodes to be used depends on the MPI implementation and the hardware/OS plattform
- Starting the example program using MPICH:
  `mpiexec -n 3 -machinefile machines ./rahmen a1 a2`
- Output:

```
Process 0/3 started on bslab02.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 2/3 started on bslab03.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 1/3 started on bslab06.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
```


Example: ping pong with messages (☞ 04/pingpong.cpp)

```
int main (int argc, char **argv)
{
  int i, passes, size, myrank;
  char *buf;
  MPI_Status status;
  double start, end;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  passes = atoi(argv[1]); // Number of repetitions
  size = atoi(argv[2]);   // Message length
  buf = new char[size];
```

```
  if (myrank == 0) {      /* Proccess 0 */
    start = MPI_Wtime(); // Get the current time
    for (i=0; i<passes; i++) {
      /* Send a message to process 1, tag = 42 */
      MPI_Send(buf, size, MPI_CHAR, 1, 42, MPI_COMM_WORLD);
      /* Wait for the answer, tag is not relevant */
      MPI_Recv(buf, size, MPI_CHAR, 1, MPI_ANY_TAG,
               MPI_COMM_WORLD, &status);
    }
    end = MPI_Wtime(); // Get the current time
    cout << "Time for one message: "
         << ((end - start) * 1e6 / (2 * passes)) << "us\n";
    cout << "Bandwidth: "
         << (size*2*passes/(1024*1024*(end-start))) << "MB/s\
  }
```


```
  else {   /* Process 1 */
    for (i=0; i<passes; i++) {
      /* Wait for the message from process 0, tag is not relevant */
      MPI_Recv(buf, size, MPI_CHAR, 0, MPI_ANY_TAG,
               MPI_COMM_WORLD, &status);
      /* Send back the answer to process 0, tag = 24 */
      MPI_Send(buf, size, MPI_CHAR, 0, 24, MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();
  return 0;
}
```

Example: ping pong with messages ...

- Results (on the XEON cluster):
  - `mpiexec -n 2 ... ./pingpong 1000 1`
    ```
    Time for one message: 50.094485 us
    Bandwidth: 0.019038 MB/s
    ```
  - `mpiexec -n 2 ... ./pingpong 1000 100`
    ```
    Time for one message: 50.076485 us
    Bandwidth: 1.904435 MB/s
    ```
  - `mpiexec -n 2 ... ./pingpong 100 1000000`
    ```
    Time for one message: 9018.934965 us
    Bandwidth: 105.741345 MB/s
    ```
- (Only) with large messages the bandwidth of the interconnection network is reached
  - XEON cluster: 1 GBit/s Ethernet (≙ 119.2 MB/s)

Additional MPI routines in the examples:

`int MPI_Get_processor_name(char *name, int *len)`

OUT	**`name`**	Pointer to buffer for node name
OUT	**`len`**	Length of the node name
Result		**`MPI_SUCCESS`** or error code

- The buffer for node name should have the length `MPI_MAX_PROCESSOR_NAME`

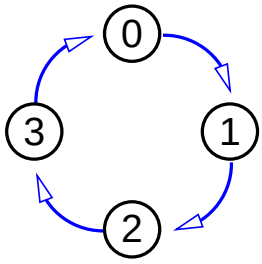
`double MPI_Wtime()`

Result	Current wall clock time in seconds

- for timing measurements
- in MPICH2: time is synchronized between the nodes

Example: sending in a closed cycle (☞ 04/ring.cpp)

```
int a[N];
...
MPI_Send(a, N, MPI_INT, (myrank+1) % nprocs,
         0, MPI_COMM_WORLD);
MPI_Recv(a, N, MPI_INT,
         (myrank+nprocs-1) % nprocs,
         0, MPI_COMM_WORLD, &status);
```



- Each process first attempts to send, before it receives
- This works **only if** MPI buffers the messages
- But `MPI_Send` can also block until the message is received
  - deadlock!

Example: sending in a closed cycle (correct)

- Some processes must first receive, before they send

```
int a[N];
...
if (myrank % 2 == 0) {
   MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
   MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
}
else {
   MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
   MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
}
```

- Better: use non-blocking operations

Non-blocking communication

- `MPI_Isend` and `MPI_Irecv` return immediately
  - before the message actually has been sent / received
  - result: request object (`MPI_Request`)
  - send / receive buffer must bot be modified / used, until the communication is completed
- `MPI_Test` checks whether communication is completed
- `MPI_Wait` blocks, until communication is completed
- Allows to overlap communication and computation
- can be “mixed” with blocking communication
  - e.g., send usgin `MPI_Send`, receive using `MPI_Irecv`

Example: sending in a closed cycle with `MPI_Irecv`

(☞ `04/ring2.cpp`)

```
int sbuf[N];
int rbuf[N];
MPI_Status status;
MPI_Request request;
...
// Set up the receive request
MPI_Irecv(rbuf, N, MPI_INT, (myrank+nprocs-1) % nprocs, 0,
          MPI_COMM_WORLD, &request);
// Sending
MPI_Send(sbuf, N, MPI_INT, (myrank+1) % nprocs, 0,
         MPI_COMM_WORLD);
// Wait for the message being received
MPI_Wait(&request, &status);
```

Notes for slide 326:

MPI offers many different variants for point-to-point communiction:

- For sending, there are four modes:
  - **synchronous**: send operation blocks, until message is received
    - rendez-vous between sender and receiver
  - **buffered**: message will be buffered by the sender
    - application must allocate and register the buffer
  - ***ready***: the programmer must guarantee that the receiver process already waits for the message (allows optimized sending)
  - **standard**: MPI decides whether synchronous or buffered
    - in this case, MPI provides the buffer itself
- In addition: sending can be blocking or non-blocking
- For receiving of messages: only blocking and non-blocking variant

- The following table summarizes all routines:

		synchronous	asynchronous
Sending	synchronous	**MPI_Ssend()**	**MPI_Issend()**
	buffered	**MPI_Bsend()**	**MPI_Ibsend()**
	ready	**MPI_Rsend()**	**MPI_Irsend()**
	standard	**MPI_Send()**	**MPI_Isend()**
Receiving		**MPI_Recv()**	**MPI_Irecv()**

- In addition, MPI also has a routine `MPI_Sendrecv`, which allows to send and receive at the same time, without the possibility of a deadlock. Using this function, the example from (☞ `04/ring1.cpp`) looks like:

```
int sbuf[N];
int rbuf[N];
MPI_Status status;
...

MPI_Sendrecv(sbuf, N, MPI_INT, (myrank+1) % nprocs, 0,
             rbuf, N, MPI_INT, (myrank+nprocs-1) % nprocs, 0,
             MPI_COMM_WORLD, &status);
```

- When using `MPI_Sendrecv`, send and receive buffer must be different, when using `MPI_Sendrecv_replace` the send buffer is overwritten with the received message.

4.6 Tutorial: Working with MPI (MPICH2/OpenMPI)

Available MPI implementations

- e.g., MPICH2 (Linux), OpenMPI 1.10.3
- Portable implementations of the MPI-2 standard

Compiling MPI programs: `mpic++`

- `mpic++ -o myProg myProg.cpp`
- Not a separate compiler for MPI, but just a script that defines additional compiler options:
  - include und linker paths, MPI libraries, ...
  - option `-show` shows the invocations of the compiler

Running MPI programs: `mpiexec`

- `mpiexec -n 3 myProg arg1 arg2`
  - starts `myProg arg1 arg2` with 3 processes
  - `myProg` must be on the command search path or must be specified with (absolute or relative) path name
- On which nodes do the processes start?
  - depends on the implementation and the platform
  - in MPICH2 (with Hydra process manager): specification is possible via a configuration file:
    `mpiexec -n 3 -machinefile machines myProg arg1 arg2`
    - configuration file contains a list of node names, e.g.:
      `bslab01` ← start one process on `bslab03`
      `bslab05:2` ← start two processes on `bslab05`

Debugging

- MPICH2 and OpenMPI support `gdb` and `totalview`
- Using `gdb`:
  - `mpiexec -enable-x -n ... xterm -e gdb myProg`
    - instead of `xterm`, you may (have to) use other console programs, e.g., `konsole` or `mate-terminal`
  - for each process, a `gdb` starts in its own console window
  - in `gdb`, start the process with `run` *args...*
- Prerequisite: compilation with debugging information
  - `mpic++ -g -o myProg myProg.cpp`

Performance Analysis using Scalasca

- In principle, in the same way as for OpenMP
- Compiling the program:
  - `scalasca -instrument mpic++ -o myprog myprog.cpp`
- Running the programms:
  - `scalasca -analyze mpiexec -n 4 ... ./myprog`
  - creates a directory `scorep_myprog_4_sum`
    - `4` indicates the number of processes
    - directory must not previously exist; delete it, if necessary
- Interactive analysis of the recorded data:
  - `scalasca -examine scorep_myprog_4_sum`

Parallel Processing

Winter Term 2024/25

16.12.2024

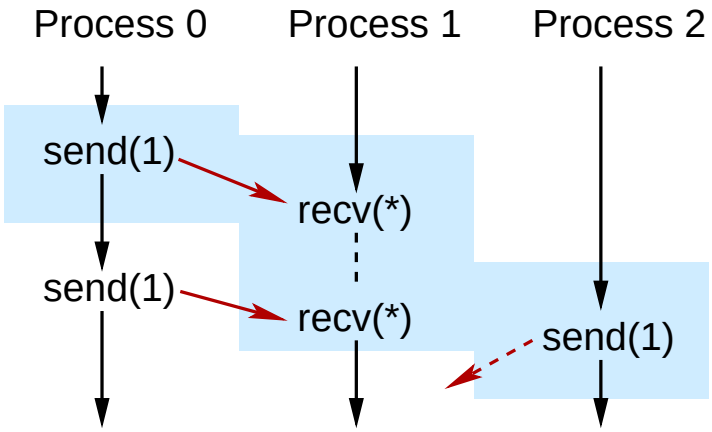
Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: December 16, 2024

4.7 Communicators

(Animated slide)

Motivation: problem of earlier communication libraries



: Code in a parallel (e.g. numerical) library

If process 2 is 'late' for some reason: communication fails!

- Message tags are not a reliable solution
  - tags might be chosen identically by chance!
- Required: different communication contexts

4.7 Communicators ...

- Communicator = process group + context
- Communicators support
  - working with process groups
    - task parallelism
    - coupled simulations
    - collective communication with a subset of all processes
  - communication contexts
    - for parallel libraries
- A communicator represents a communication domain
  - communication is possible only within the same domain
    - no wild-card for communicator in `MPI_Recv`
  - a process can belong to several domains at the same time

Creating new communicators

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

- Collective operations (☞ 4.8)
  - all processes in `comm` must execute them concurrently
- `MPI_Comm_dup` creates a copy with a new context
- `MPI_Comm_split` splits `comm` into several communicators
  - one communicator for each value of `color`
  - as the result, each process receives the communicator to which it was assigned
  - `key` determines the order of the new process ranks

Example for MPI_Comm_split

- *Multi-physics code*: air pollution
  - one half of the processes computes the airflow
  - the other half computes chemical reactions
- Creation of two communicators for the two parts:

```
MPI_Comm_split(MPI_COMM_WORLD, myrank%2, myrank, &comm)
```

Process	`myrank`	Color	Result in `comm`	Rank in C_0	Rank in C_1
P0	0	0	C_0	0	–
P1	1	1	C_1	–	0
P2	2	0	C_0	1	–
P3	3	1	C_1	–	1

4.8 Collective operations

- Collective operations in MPI
  - must be executed concurrently by all processes of a process group (a communicator)
  - are blocking
  - do not neccessarily result in a global (barrier) synchronisation, however
- Collective synchronisation and communication functions
  - barriers
  - reductions (communication with aggregation)
  - global communication: broadcast, scatter, gather, ...

Notes for slide 335:

Note that “concurrently” (German: ”‘nebenläufig”’) does not mean that the operations must be executed at the same time, or in an overlapping way. It just means that (1) all processes in the communicator execute the operation and (2) there is no synchonization that enforces any restriction on the ordering of the operations. (In other words: it must be **possible** that the operations can be executed at the same time, but this is not required)

MPI_Barrier

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

- Barrier synchonization of all processes in `comm`
- With message passing, barriers are actually not really necessary
  - synchonization is achieved by message exchange
- Reasons for barriers:
  - more easy understanding of the program
  - timing measurements, debugging output
  - console input/output ??
  - MPI-2: MPI I/O, one-sided communication

Reduction: MPI_Reduce

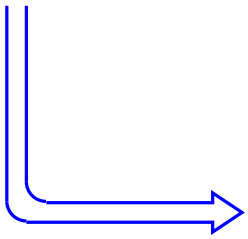
```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype dtype,
               MPI_Op op, int root,
               MPI_Comm comm)
```

- Each element in the receive buffer is the result of a reduction operation (e.g., the sum) of the corresponding elements in the send buffer
- `op` defines the operation
  - predefined: minimum, maximum, sum, product, AND, OR, XOR, ...
  - in addition, user defined operations are possible, too

Example: summing up an array

Sequential

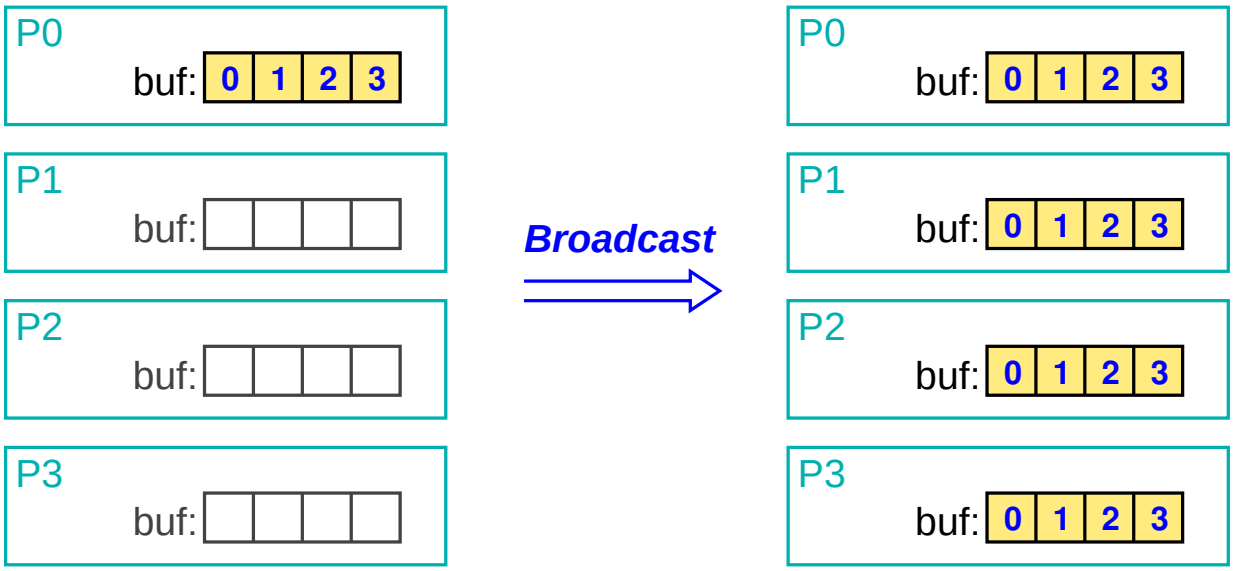
```
s = 0;
for (i=0;i<size;i++)
  s += a[i];
```



Parallel

```
local_s = 0;
for (i=0;i<local_size;i++)
   local_s += a[i];

MPI_Reduce(&local_s, &s,
           1, MPI_INT,
           MPI_SUM,
           0, MPI_COMM_WORLD);
```


Collective communication: broadcast



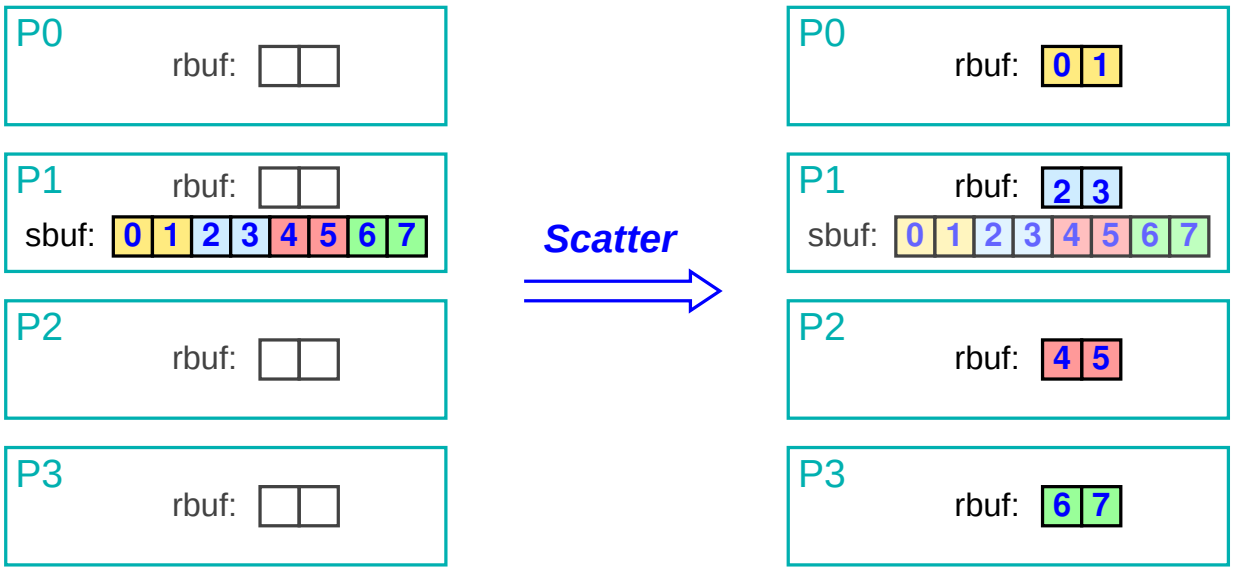
MPI_Bcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype,
              int root, MPI_Comm comm)
```

IN **root** Rank of the sending process

- Buffer is sent by process `root` and reveived by all others
- Collective, blocking operation: no tag necessary
- `count`, `dtype`, `root`, `comm` must be the same in all processes

Collective communication: scatter



MPI_Scatter

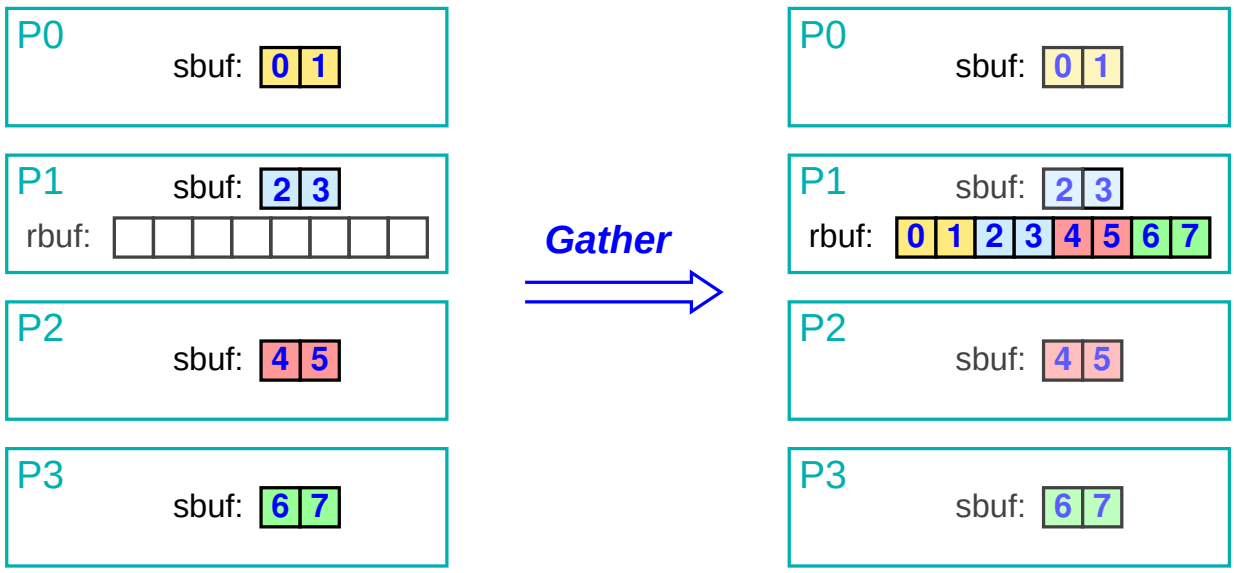
```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

- Process `root` sends a part of the data to each process
  - including itself
- `sendcount`: data length for each process (not the total length!)
- Process `i` receives `sendcount` elements of `sendbuf` starting from position `i` * `sendcount`
- Alternative `MPI_Scatterv`: length and position can be specified individually for each receiver

Notes for slide 342:

- A problem that may arise when using `MPI_Scatter` is that the the data cannot be distributed evenly, e.g., if an array with 1000 elements should be distributed to 16 processes.
- In `MPI_Scatterv`, the argument `sendcount` is replaced by two arrays `sendcounts` and `displacements`
  - process `i` then receives `sendcounts[i]` elements of `sendbuf`, starting at position `displacements[i]`

Collective communication: gather



MPI_Gather

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

- All processes send `sendcount` elements to process `root`
  - even `root` itself
- Important: each process must sent the same amount of data
- `root` stores the data from process `i` starting at position `i` * `recvcount` in `recvbuf`
- `recvcount`: data length for each process (not the total length!)
- Alternative `MPI_Gatherv`: analogous to `MPI_Scatterv`

Example: multiplication of vector and scalar (☞ 04/vecmult.cpp)

```
double a[N], factor, local_a[LOCAL_N];
... // Process 0 reads a and factor from file
MPI_Bcast(&factor, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(a, LOCAL_N, MPI_DOUBLE, local_a, LOCAL_N,
            MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0; i<LOCAL_N; i++)
   local_a[i] *= factor;
MPI_Gather(local_a, LOCAL_N, MPI_DOUBLE, a, LOCAL_N,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);
... // Process 0 writes a into file
```

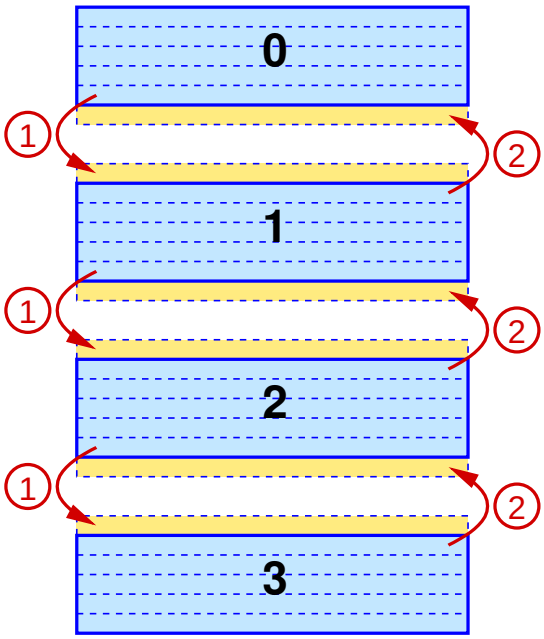
- **Caution:** LOCAL_N must have the same value in all processes!
  - otherwise: use MPI_Scatterv / MPI_Gatherv (☞ 04/vecmult3.cpp)

More collective communication operations

- MPI_Alltoall: all-to-all broadcast (☞ **2.8.5**)
- MPI_Allgather and MPI_Allgatherv: at the end, all processes have the gathered data
  - corresponds to a gather with subsequent broadcast
- MPI_Allreduce: at the end, all processes have the result of the reduction
  - corresponds to a reduce with subsequent broadcast
- MPI_Scan: prefix reduction
  - e.g., using the sum: process i receives the sum of the data from processes 0 up to and including i

(Animated slide)

Gerneral approach



0. Matrix with temperature values
1. Distribute the matrix into stripes
   Each process only stores a part of the matrix
2. Introduce ghost zones
   Each process stores an additional row at the cutting edges
3. After each iteration the ghost zones are exchanged with the neighbor processes
   E.g., first downwards (1), then upwards (2)

Gerneral approach ...

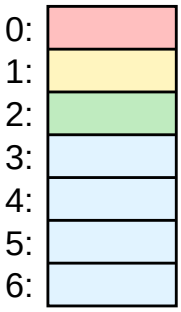
```
int nprocs, myrank;
double a[LINES][COLS];
MPI_Status status;

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

// Step 1: Send downwards, recieve from above
if (myrank != nprocs-1)
   MPI_Send(a[LINES-2], COLS, MPI_DOUBLE, myrank+1, 0,
            MPI_COMM_WORLD);
if (myrank != 0)
   MPI_Recv(a[0], COLS, MPI_DOUBLE, myrank-1, 0,
            MPI_COMM_WORLD, &status);
```

Distribution of data

- For a uniform distribution of an array of length n to np processes:
  - $\mathbf{size}(p) = (n + p) \div np$
  - $\mathbf{start}(p) = \sum_{i=0}^{p-1} \mathbf{size}(i)$
    $= n \div np \cdot p + \max(p - (np - n \bmod np), 0)$
  - process p receives $\mathbf{size}(p)$ elements starting at index $\mathbf{start}(p)$
- This results in the following index transformation:
  - $\mathbf{tolocal}(i) = (p, i - \mathbf{start}(p))$
    with $p \in [0, np - 1]$ such that $0 \leq i - \mathbf{start}(p) < \mathbf{size}(p)$
  - $\mathbf{toglobal}(p, i) = i + \mathbf{start}(p)$
- In addition, you have to consider the ghost zones for Jacobi and Gauss/Seidel!

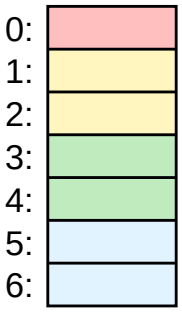
Notes for slide 349:

As a motivation for the formula $\mathbf{size}(p) = (n + p) \div np$, consider the simple example of $n = 7$ and $np = 4$:



7 ÷ 4 = 1

Processes 0 – 2 each get one element, process 3 gets the rest.



7 + p ÷ 4 = 1 | 2 | 2 | 2

Process 0 gets one element, processes 1 – 3 get two elements each.

When `nprocs` contains the number of processes and `myrank` is the rank of the MPI process, the following code will compute the start row (`start`) and the number of rows (`size`) for the current process:

```
size = (n + myrank) / nprocs;
start = n / nprocs * myrank;
if (myrank > nprocs - n % nprocs)
    start += myrank - (nprocs - n % nprocs);
```

Note that after this computation, you will have to modify these numbers a little, since you also have to account for the ghost rows.

Distribution of computation

- In general, using the *owner computes* rule
  - the process that writes a data element also performs the corresponding calculations
- Two approaches for technically realizing this:
  - index transformation and conditional execution
    - e.g., when printing the verification values of the matrix:

```
if ((x-start >= 0) && (x-start < size))
   cout << "a[" << x << "]=" << a[x-start] << "\n";
```

  - adjustment of the bounds of the enclosing loops
    - e.g., during the iteration or when initializing the matrix:

```
for (i=0; i<size; i++)
   a[i] = 0;
```


Notes for slide 350:

For the initialization of the border values it is the easiest method to use conditional execution. So the original loop

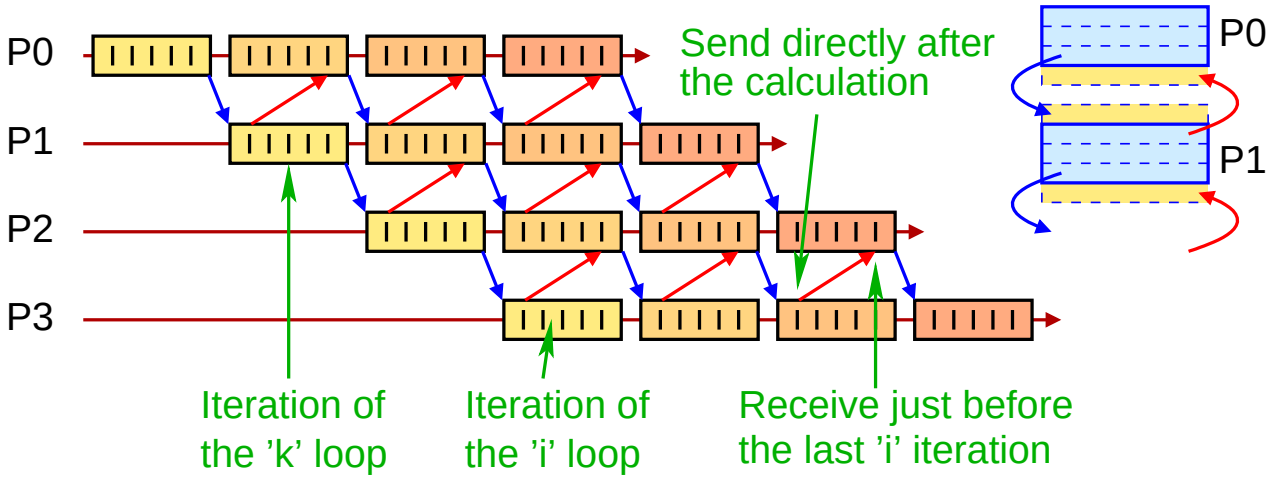
```
for (i=0; i<n; i++) {
    double x = (double)i / (n-1);
    a[i][0]       = x;
    a[n-1-i][n-1] = x;
    a[0][i]       = x;
    a[n-1][n-1-i] = x;
}
```

becomes

```
for (i=0; i<n; i++) {
    double x = (double)i / (n-1);
    if ((i-start >= 0) && (i-start < size))
        a[i-start][0]       = x;
    if ((n-1-i-start >= 0) && (n-1-i-start < size))
        a[n-1-i-start][n-1] = x;
    if ((0-start >= 0) && (0-start < size))
        a[0-start][i]       = x;
    if ((n-1-start >= 0) && (n-1-start < size))
        a[n-1-start][n-1-i] = x;
}
```

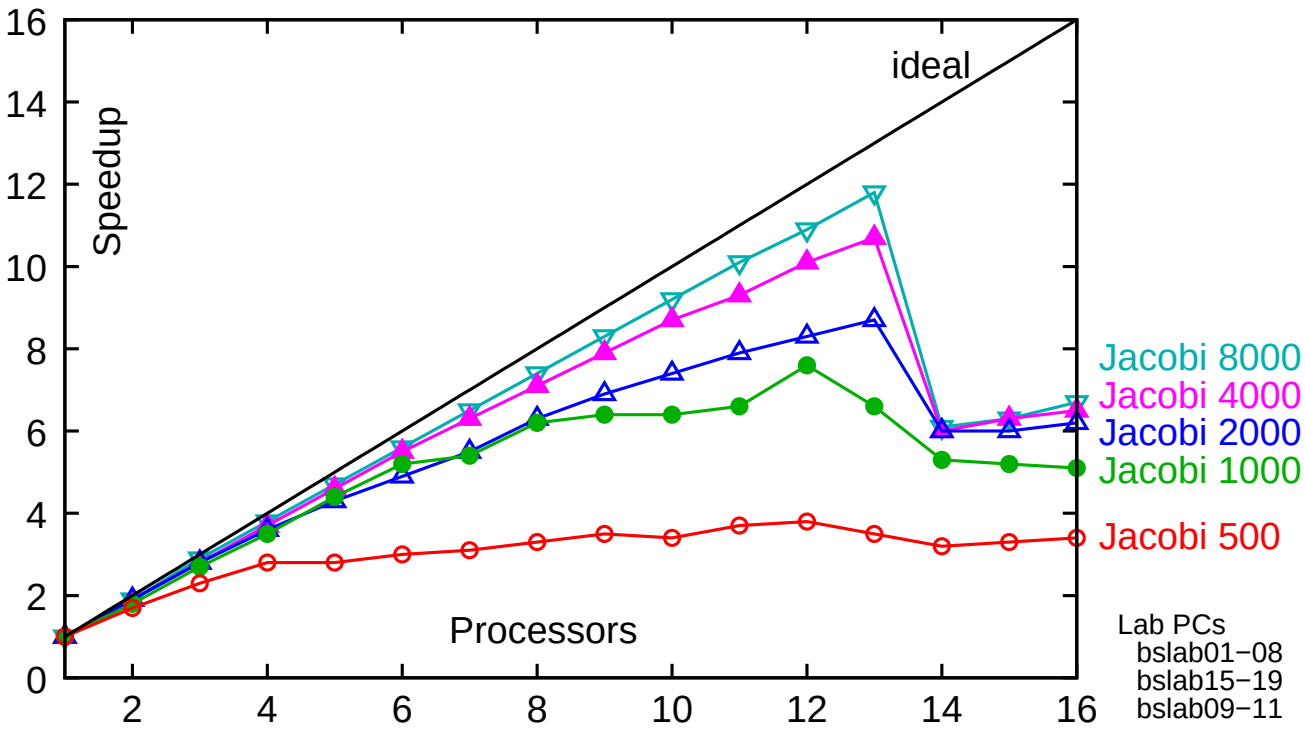
On the parallelization of the Gauss/Seidel method

➥ Similar to the pipelined parallelization with OpenMP (☞ **3.3**)



(Animated slide)

Obtained speedup for different matrix sizes



4.10 Complex data types in messages

- So far: only arrays can be send as messages
- What about complex data types (e.g., structures)?
  - z.B. `struct bsp { int a; double b[3]; char c; };`
- MPI offers two mechanisms
  - **packing and unpacking** the individual components
    - use `MPI_Pack` to pack components into a buffer one after another; send as `MPI_PACKED`; extract the components again using `MPI_Unpack`
  - **derived data types**
    - `MPI_Send` gets a pointer to the data structure as well as a description of the data type
    - the description of the data type must be created by calling MPI routines

Notes for slide 353:

Example for packing and unpacking using `MPI_Pack` and `MPI_Unpack`:

```
// C structure (or likewise C++ object), which should be sent
struct bsp { int a; double b[3]; char c; } str;

char buf[100];     // buffer, must be parge enough!!
int pos;           // position in the buffer
...

pos = 0;
MPI_Pack(&str.a, 1, MPI_INT, buf, 100, &pos, MPI_COMM_WORLD);
MPI_Pack(&str.b, 3, MPI_DOUBLE, buf, 100, &pos, MPI_COMM_WORLD);
MPI_Pack(&str.c, 1, MPI_CHAR, buf, 100, &pos, MPI_COMM_WORLD);
MPI_Send(buf, pos, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
...

MPI_Recv(buf, 100, MPI_PACKED, 1, 0, MPI_COMM_WORLD, &status);
pos = 0;
MPI_Unpack(buf, 100, &pos, &str.a, 1, MPI_INT, MPI_COMM_WORLD);
MPI_Unpack(buf, 100, &pos, &str.b, 3, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buf, 100, &pos, &str.c, 1, MPI_CHAR, MPI_COMM_WORLD);
```

The MPI standard requires that a message always must be packed as shown in successive calls to `MPI_Pack` (pack unit), where buffer, buffer length and communicator are identical.

In this way, the standard allows that an implementation also packs a header into the message (e.g., for an architecture tag). For this, information from the communicator may be used, if required.

4.10 Complex data types in messages ...

Derived data types

- MPI offers constructors, which can be used to define own (derived) data types:
  - for contiguous data: `MPI_Type_contiguous`
    - allows the definition of array types
  - for non-contiguous, strided data: `MPI_Type_vector`
    - e.g., for a column of a matrix or a sub-matrix
  - for other non-contiguous data: `MPI_Type_indexed`
  - for structures: `MPI_Type_create_struct`
- After a new data type has been created, it must be “announced”: `MPI_Type_commit`
- After that, the data type can be used like a predefined data type (e.g., MPI_INT)

MPI_Type_vector: non-contiguous arrays

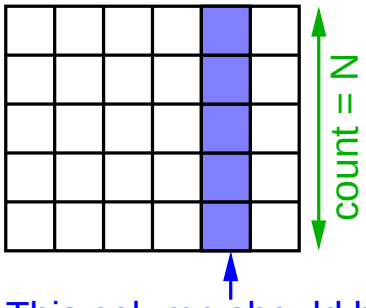
```
int MPI_Type_vector(int count, int blocklen, int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

IN	**count**	Number of data blocks
IN	**blocklen**	Length of the individual data blocks
IN	**stride**	Distance between successive data blocks
IN	**oldtype**	Type of the elements in the data blocks
OUT	**newtype**	Newly created data type

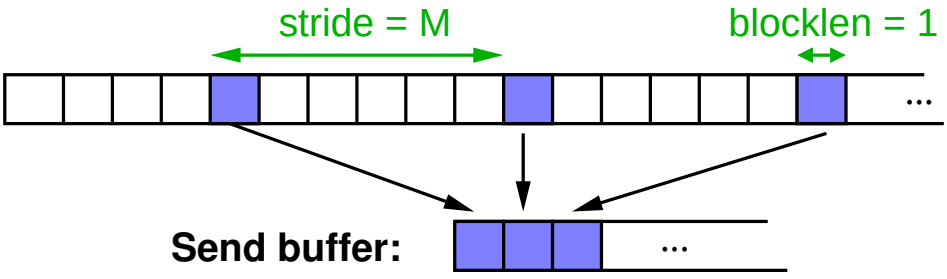
- Summarizes a number of data blocks (described as arrays) into a new data type
- However, the result is more like a new **view** onto the existing data than a new data **type**

Example: transferring a column of a matrix

Matrix: a[N][M]



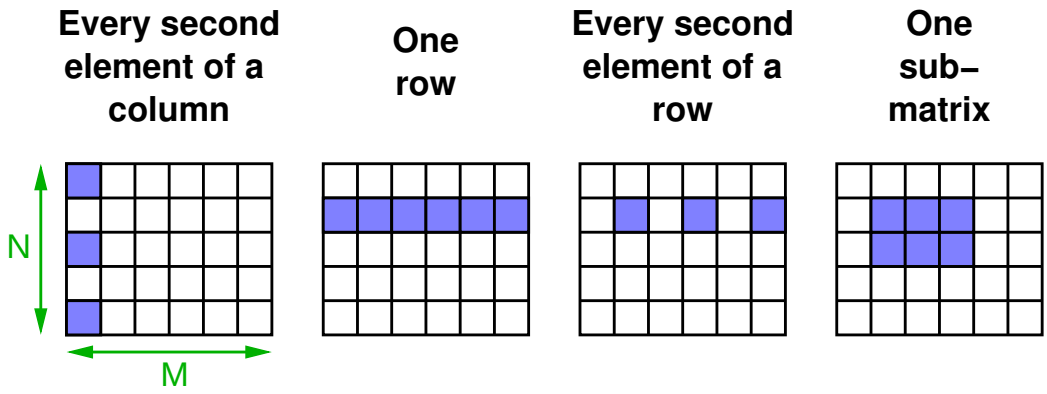
Memory layout of the matrix:




```
MPI_type_vector(N, 1, M, MPI_INT, &column);
MPI_Type_commit(&column);
// Transfer the column
if (rank==0) MPI_Send(&a[0][4], 1, column, 1, 0, comm);
else MPI_Recv(&a[0][4], 1, column, 0, 0, comm, &status);
```

Notes for slide 356:

Additional options of `MPI_Type_vector`



count	N / 2	1	M	M / 2	2
blocklen	1	M	1	1	3
stride	2 * M	x	1	2	M

356-1

4.10 Complex data types in messages ...

Remarks on `MPI_Type_vector`

- The receiver can use a different data type than the sender
- It is only required that the number of elements and the sequence of their types is the same in the send and receive operations
- Thus, e.g., the following is possible:
  - sender transmits a column of a matrix
  - reciever stores it in a one-dimensional array

```
int a[N][M], b[N];
MPI_type_vector(N, 1, M, MPI_INT, &column);
MPI_Type_commit(&column);
if (rank==0) MPI_Send(&a[0][4], 1, column, 1, 0, comm);
else MPI_Recv(b, N, MPI_INT, 0, 0, comm, &status);
```

Notes for slide 357:

Strided arrays that have been created using `MPI_Type_vector` can usually transmitted as efficient as contiguous arrays (i.e., with stride 1) with modern network interface cards. These cards support the transmission of non-contiguous memory areas in hardware.

357-1

4.10 Complex data types in messages ...

How to select the best approach

- Homogeneous data (elements of the same type):
  - contiguous (stride 1): standard data type and `count` parameter
  - non-contiguous:
    - stride is constant: `MPI_Type_vector`
    - stride is irregular: `MPI_Type_indexed`
- Heterogeneous data (elements of different types):
  - large data, often transmitted: `MPI_Type_create_struct`
  - few data, rarely transmitted: `MPI_Pack` / `MPI_Unpack`
  - structures of variable length: `MPI_Pack` / `MPI_Unpack`

Parallel Processing

Winter Term 2024/25

13.01.2025

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: December 16, 2024

4.11 Further concepts

- Topologies
  - the application's communication structure is stored in a communicator
    - e.g., cartesian grid
  - allows to simplify and optimize the communication
    - e,g,. "send to the left neighbor"
    - the communicating processes can be placed on neighboring nodes
- Dynamic process creation (since MPI-2)
  - new processes can be created at run-time
  - process creation is a collective operation
  - the newly created process group gets its own `MPI_COMM_WORLD`
    - communication between process groups uses an *intercommunicator*

4.11 Further concepts ...

- One-sided communication (since MPI-2)
  - access to the address space of other processes
  - operations: read, write, atomic update
  - weak consistency model
    - explicit *fence* and *lock*/*unlock* operations for synchronisation
  - useful for applications with irregular communication
    - one process alone can execute the communication
- Parallel I/O (since MPI-2)
  - processes have individual views to a file
    - specifiec by an MPI data type
  - file operations: individual / collective, private / shared file pointer, blocking / non-blocking

4.12 Summary

- Basic routines:
  - `Init`, `Finalize`, `Comm_size`, `Comm_rank`, `Send`, `Recv`
- Complex data types in messages
  - `Pack` and `Unpack`
  - user defined data types
    - also for non-contiguous data (e.g., column of a matrix)
- Communicators: process group + communication context
- Non-blocking communication: `Isend`, `Irecv`, `Test`, `Wait`
- Collective operations
  - `Barrier`, `Bcast`, `Scatter(v)`, `Gather(v)`, `Reduce`, ...