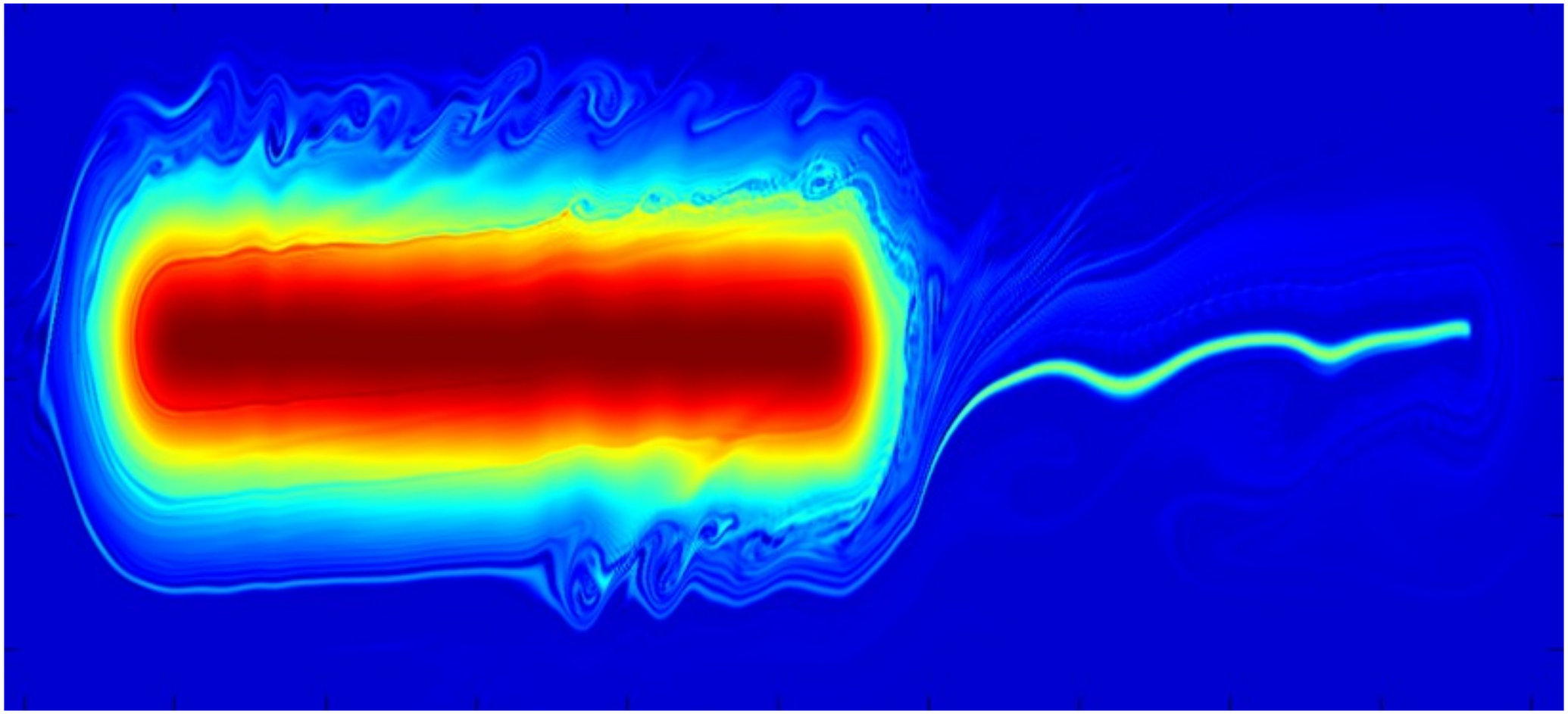
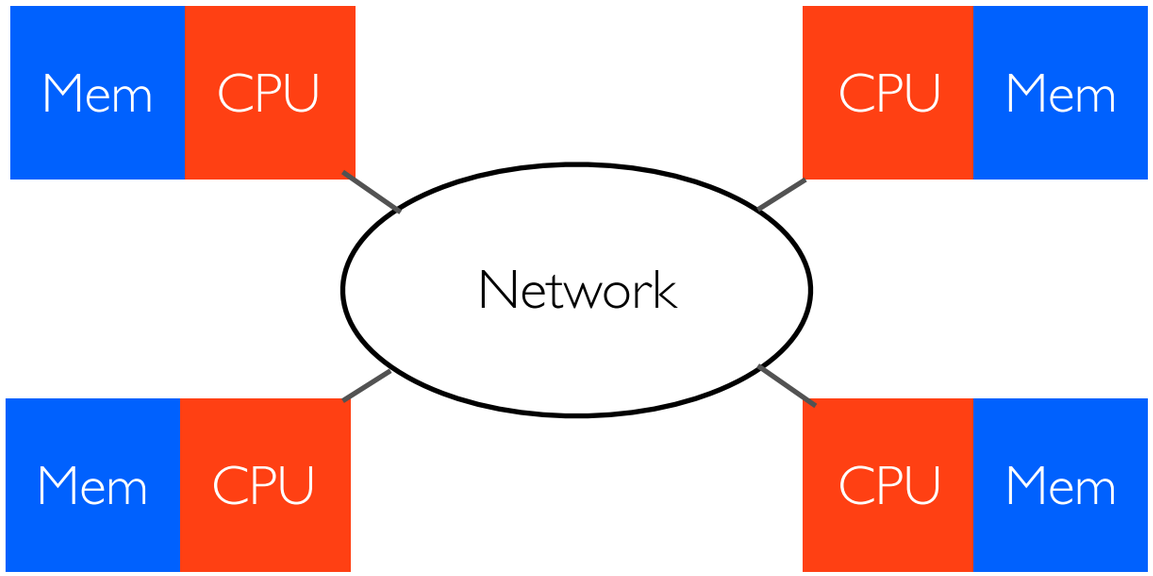
HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF



Chapter 7 - MPI

Message Passing Interface

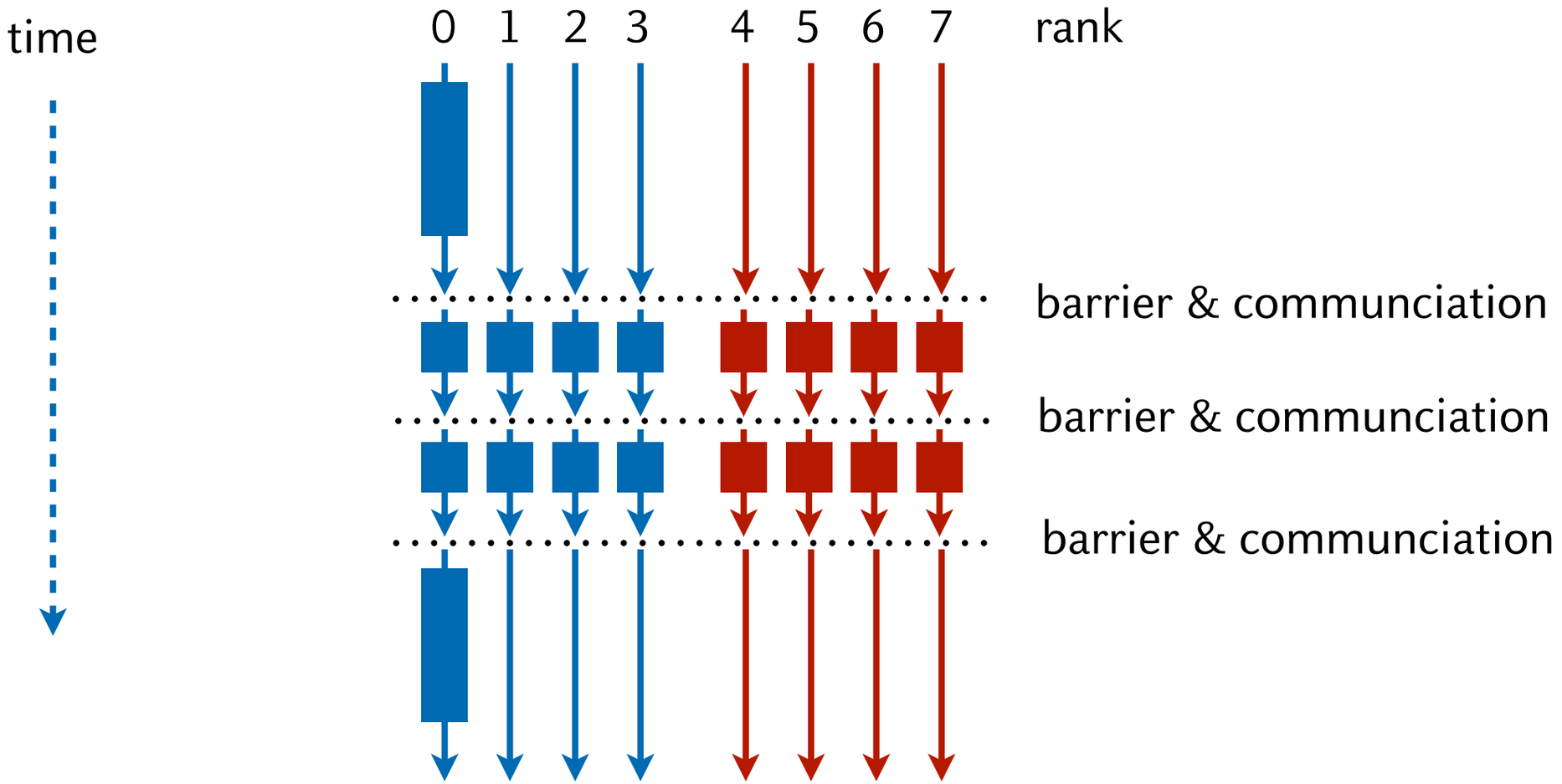
- MPI is a widely used standard for writing parallel *distributed* memory applications



- MPI allows a platform and operating system independent implementation of communication between tasks running on different machines which are interconnected via a network
- MPI defines the interfaces and commands we may use to communicate between tasks (the current standard is MPI 3.0)
- Popular implementations of MPI are OpenMPI and MPICH2.

Schematical workflow of MPI programs

- A MPI program is started on many machines.
- Assume we have two quad-core machines, then we might for example start the same program 8 times, 4 times per machine.



Elementary MPI program

```
#include <stdlib.h>
#include <mpi.h>
#include <iostream>

using namespace std;

int main (int argc, char* argv[]) {

 int myrank;
 int size;

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

 cout << "I am rank " << myrank
      << " out of " << size << " processes." << endl;

 MPI_Finalize();
 exit(0);
}
```

- The simplest MPI program is one that has no communication between ranks at all.
- All MPI programs need to have calls to `MPI_Init()` and `MPI_Finalize()`.
- Very similar to multithreading in OpenMP we can find out how many global tasks there are (`MPI_Comm_rank`), and what is the rank number of the local task (`MPI_Comm_size`).

Output:

```
I am rank 0 out of 4 processes.
I am rank 1 out of 4 processes.
I am rank 2 out of 4 processes.
I am rank 3 out of 4 processes.
```

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Compiling and executing MPI programs

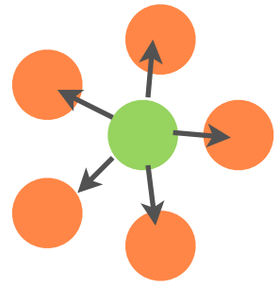
- To compile a MPI program we have two choices:
  - Use the standard g++ compiler call and append all neccessary paths and libraries:
    `g++ mpi_ex.cxx -o mpi_ex -I$HOME/libs/mpi/include -L$HOME/libs/mpi/lib -lmpi`
  - Use compiler wrappers that come along with the MPI implementation.

- Compiler wrappers are nothing else than the standard C++ compiler but they will automatically handle appending all MPI related compiler options.

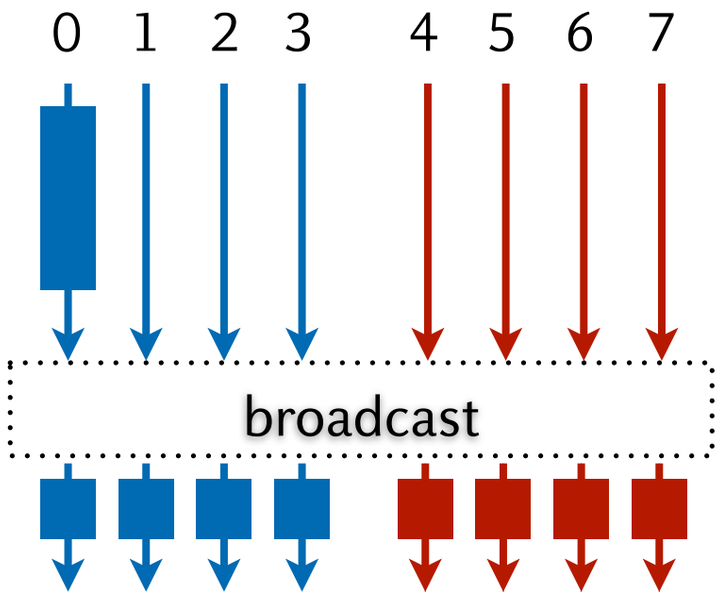
- For C++ the wrapper is called mpicxx. The compiler line will then be
  `mpicxx mpi_ex.cxx -o mpi_ex`

- Once we have the binary we can start it using the tool `mpirun` (also comes along with the MPI distribution):
  `mpirun -np 4 ./mpi_ex`

Collective communication



- One task sends data to all other ranks (*collective communication*)
- This requires that the sending rank sends messages to the network and all other ranks listen to the network
- Example:
  - Rank 0 might read a parameter from the console before we start with the caluclation.
  - Once rank 0 knows the value of the parameter it has to send this value to all other ranks.
  - Since all ranks will get the same value this can be done as a broadcast.
  - Once everybody knows the parameter value, all ranks may start their calculation.



HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Collective communication - Broadcast

- When we want to distribute the same information from one rank to all other ranks we use `MPI_Bcast()`:
- `int MPI_Bcast(void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);`
- All nodes execute the same broadcast command.
- `root` specifies the rank number which is supposed to send data. When the local rank number is different from `root`, then the process knows that it is supposed to receive data.
- The variable type specifies what kind of data will be send during the communication. Example for types are MPI_INT, MPI_DOUBLE, MPI_CHAR.
- The pointer buffer points to the adress in memory where the data is stored. count specifies how many elements of type specified by type will be sent.
- The communicator comm specifies within which set of tasks the communication will take place. For our purposes there will always only be one set called MPI_COMM_WORLD.

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Collective communication - Broadcast

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

using namespace std;

int main (int argc, char* argv[]) {

 int rank, size, N=0;

 // Initialize MPI
 MPI_Init(&argc, &argv);

 // Who am I and how many are we?
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 // Read N from keyboard if you are rank 0
 if(rank==0){
      cout << "N = "; cin >> N;
 }
 // wait here for everybody, in particular rank 0
 MPI_Barrier(MPI_COMM_WORLD);

 // Everybody writes his value of N
 cout << "A: rank = " << rank << ", N =  " << N << endl;

 // Broadcast of N from rank 0 to all other ranks
 MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

 // Everybody writes his value of N
 cout << "B: rank = " << rank << ", N =  " << N << endl;

 MPI_Finalize();
}
```

- `int MPI_Bcast(void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);`

```
mpirun -np 4 ./mpi2
N = 2
A: rank = 0, N =  2
B: rank = 0, N =  2
A: rank = 1, N =  0
B: rank = 1, N =  2
A: rank = 2, N =  0
B: rank = 2, N =  2
A: rank = 3, N =  0
B: rank = 3, N =  2
```

Collective communication - Scatter

- Distributing different values to different ranks is called a scatter operation.
- Scattering the same amount of elements to every rank from root is done via

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

- sendbuf, sendcount, sendtype are only relevant for root.
- recvbug, recvcount, recvtype are relevant for every rank (including root).

Collective communication - Gather

- Collecting information from all ranks to one particular rank is a gather operation.
- We suppose that every rank sents the same amout of data, then gathering is done via

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

- sendbuf, sendcount, sendtype are relevant for every rank
- recvbuf, recvcount, recvtype are relevant for root, which will collect all data into the buffer recvbuf

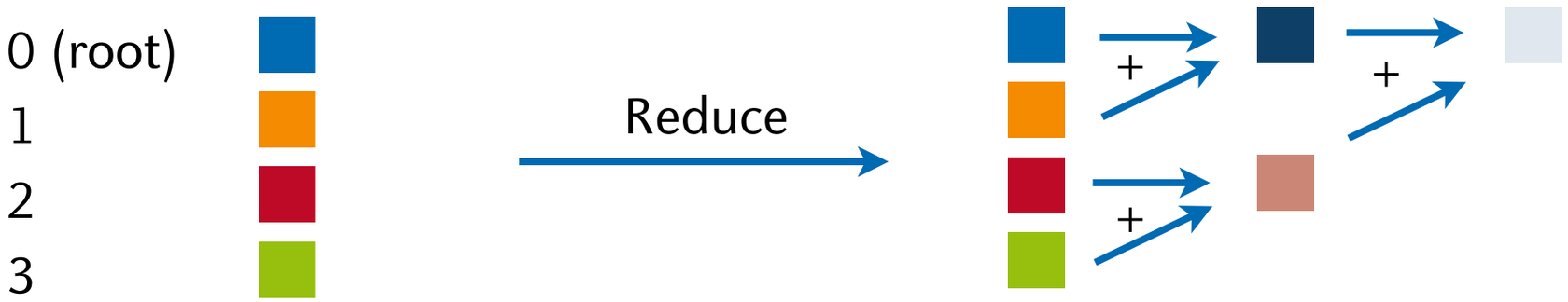
Collective communication - All-Gather

- An All-Gather operation allows us to send the same data to all other ranks while receiving data from all other ranks at the same time. There is no more a distinguished root process.

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

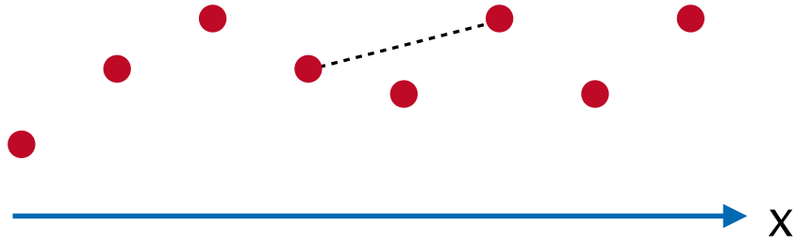
Collective communication - Reduction

- In case that the root process should collect data from the other processes and then do a simple operation like performing a sum over all collected values we can use a special reduction command
- `int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)`
- `MPI_Op` may be e.g. `MPI_MAX`, `MPI_MIN`, `MPI_PROD`, `MPI_SUM`
- A reduction using MPI_SUM might look something like



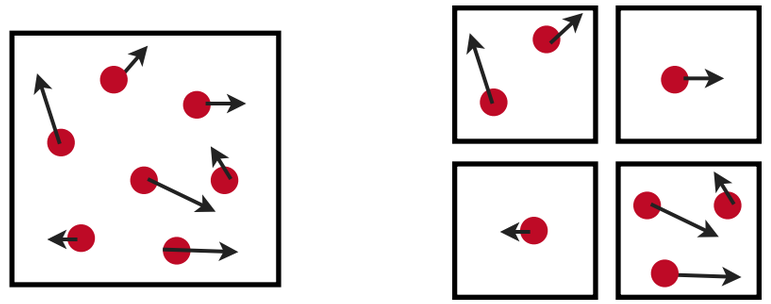
Point-to-point communication

- *Point-to-point communication* refers to communication between two particular MPI tasks, all other tasks take no part in this communication.



- Examples for point-to-point communication:
  - Calculating the derivative of a function, where the global interval is splitted into sub-intervals, each treatet by a different rank
    - Need to exchange points at the boundary

  - Particle motion in a global domain that has been subdivided into sub-domains. Each sub-domain is treated by one rank.
    - Need to pass particles from one rank to the other



HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Point-to-Point communication

- For p2p-communictation sender and receiver have to issue different commands (this is different from collective communication)
- The rank that sends data uses

```
int MPI_Send(void *sendbuf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm);
```

- The rank that should receive the packages runs

```
int MPI_Recv(void *recvbuf, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- In `MPI_Send` (run by the sender), `dest` has to be the rank of the node that should receive the data.
- In `MPI_Recv` (run by the receiver), `source` has to specify from which rank the data will be sent.
- `tag` is number by which we can give the data package a label. The receiver will only accept data which has been sent to him with the correct `tag` number.
- `status` is a special MPI object that can be used for error handling.

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Point-to-point communication

```
#include <stdlib.h>
#include <mpi.h>
#include <iostream>

using namespace std;

int main (int argc, char* argv[]) {

 int rank;
 int size;
 int N=0;
 MPI_Status status;

 // Initialize MPI
 MPI_Init(&argc, &argv);
 // Who am I and how many are we?
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 if (rank==0){
      N = 12345;
      MPI_Send(&N, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
 }
 else if (rank==1)
      MPI_Recv(&N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

 cout << "rank = " << rank << ", N = " << N << endl;

 MPI_Finalize();
 exit(0);
}
```

- `int MPI_Send(void *sendbuf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`
- `int MPI_Recv(void *recvbuf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status);`

```
mpirun -np 4 ./mpi3
rank = 0, N = 12345
rank = 1, N = 12345
rank = 2, N = 0
rank = 3, N = 0
```

Point-to-point communication

- We can easily produce a deadlock in p2p-communication

```
#include <stdlib.h>
#include <mpi.h>
#include <iostream>

using namespace std;

int main (int argc, char* argv[]) {

 int rank;
 int size;
 int N=0;
 MPI_Status status;

 // Initialize MPI
 MPI_Init(&argc, &argv);
 // Who am I and how many are we?
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 N = rank*2;
 MPI_Send(&N, 1, MPI_INT, (rank+1)%size, 0, MPI_COMM_WORLD);
 MPI_Recv(&N, 1, MPI_INT, (rank-1)%size, 0, MPI_COMM_WORLD, &status);

 cout << "rank = " << rank << ", N = " << N << endl;

 MPI_Finalize();
 exit(0);
}
```

Every rank sents data to the process "to the right" in cyclic order

Every rank waits for data from the process "to the left"

MPI_Send() is blocking communication, i.e. the program only commences once sending data is finished. This will not happen since nobody is receiving.

Non-blocking p2p-communication

- MPI_Send() and MPI_Recv() are commands for blocking communication.
- This means that the sending rank is stalled until he receives information that the sent information has been received.
- The receiving rank also stalls until it actually receives the correct data package.
- This kind of communication is also referred to as synchronous communication.

- Blocking communication is mostly inefficient, since one of the tasks might have wait for the other and we acutally have to explicitly wait for the network until everything is transfered.

- Non-blocking communication is asynchronous. We can send data and then continue our work and check later if it has been received. On the other side we can initiate receiving data and do something else while we wait for the data to arrive.
- Non-blocking communication allows to hide the transfer time for data over the network.

Non-blocking p2p-communication

- Non-blocking sending:
```
int MPI_Isend (void* buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request* request)
```
- Non-blocking receive:
```
int MPI_Irecv (void* buf, int count, MPI_Datatype datatype, int source, int tag,
               MPI_Comm comm, MPI_Request* request)
```
- MPI_Isend() and MPI_Irecv() will initiate data transfer. Right after they return we can carry on with our program, while the data is send in the background.
- Before we reach a point in our program where we actually need the transfered values, we have to make sure that sending and receiving is completed. We can check this using the request object
```
int MPI_Test (MPI_Request* request, int* flag, MPI_Status* status)
```
- flag will be 1 when transfer is finished or 0 otherwise.
- We can also wait until the communication is done,
```
int MPI_Wait (MPI_Request* request, MPI_Status* status)
```

Non-blocking p2p-communication

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

using namespace std;

int main (int argc, char* argv[]) {

 int rank, size, N=0;
 MPI_Status status;
 MPI_Request send_request, recv_request;

 // Initialize MPI
 MPI_Init(&argc, &argv);
 // Who am I and how many are we?
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 N = rank*2;
 // Everybody send N to right neighbor
 MPI_Isend(&N, 1, MPI_INT, (rank+1)%(size), 0, MPI_COMM_WORLD, &send_request);
 // And now everybody receive new value of N from left neighbor
 MPI_Irecv(&N, 1, MPI_INT, (rank-1)%(size), 0, MPI_COMM_WORLD, &recv_request);

 // ...right here we could do other things while data is transfered in background

 // wait for all ranks to receive their data if not yet finished
 MPI_Wait(&recv_request, &status);

 cout << "rank = " << rank << ", N = " << N << endl;

 MPI_Finalize();
}
```