Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

MPI – Message Passing Interface

Summer Term 2023 | 31.05.2023

Barry Linnert

Objectives of Today's Lecture

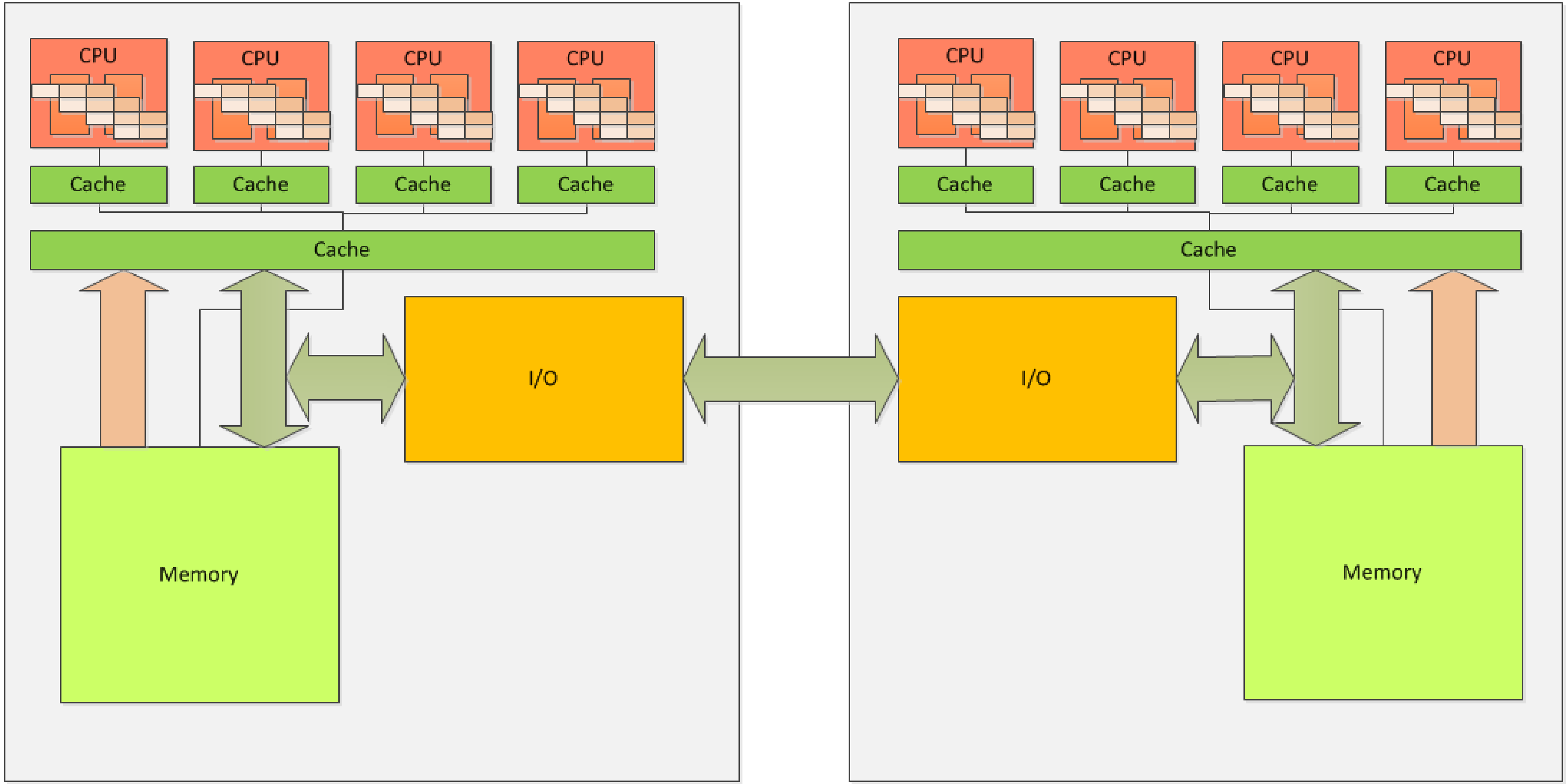
- Introduction to MPI

Freie Universität Berlin

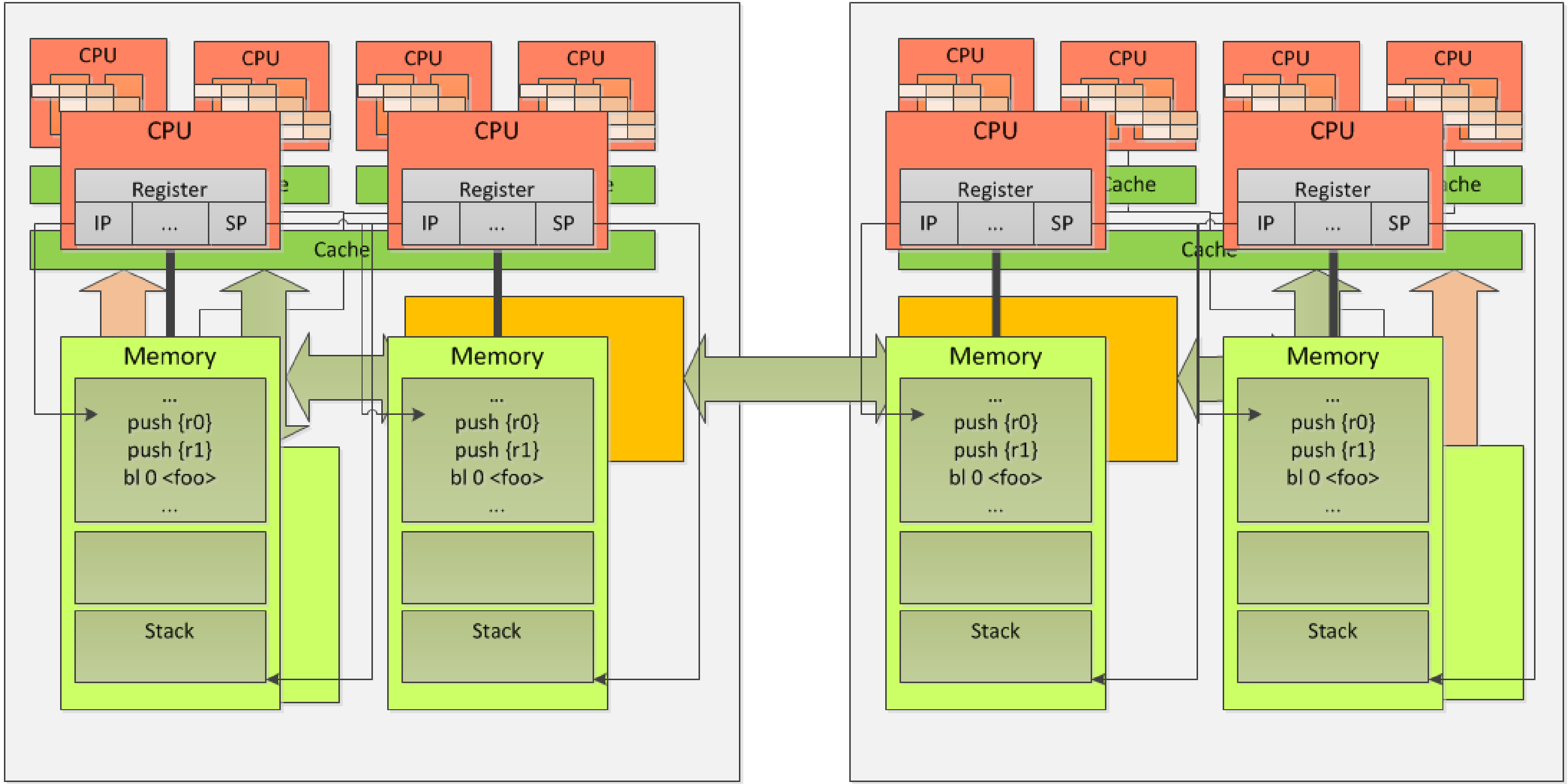
Concepts of Non-sequential and Distributed Programming

MPI

Machine Model



Machine and Execution Model



MPI – Overview

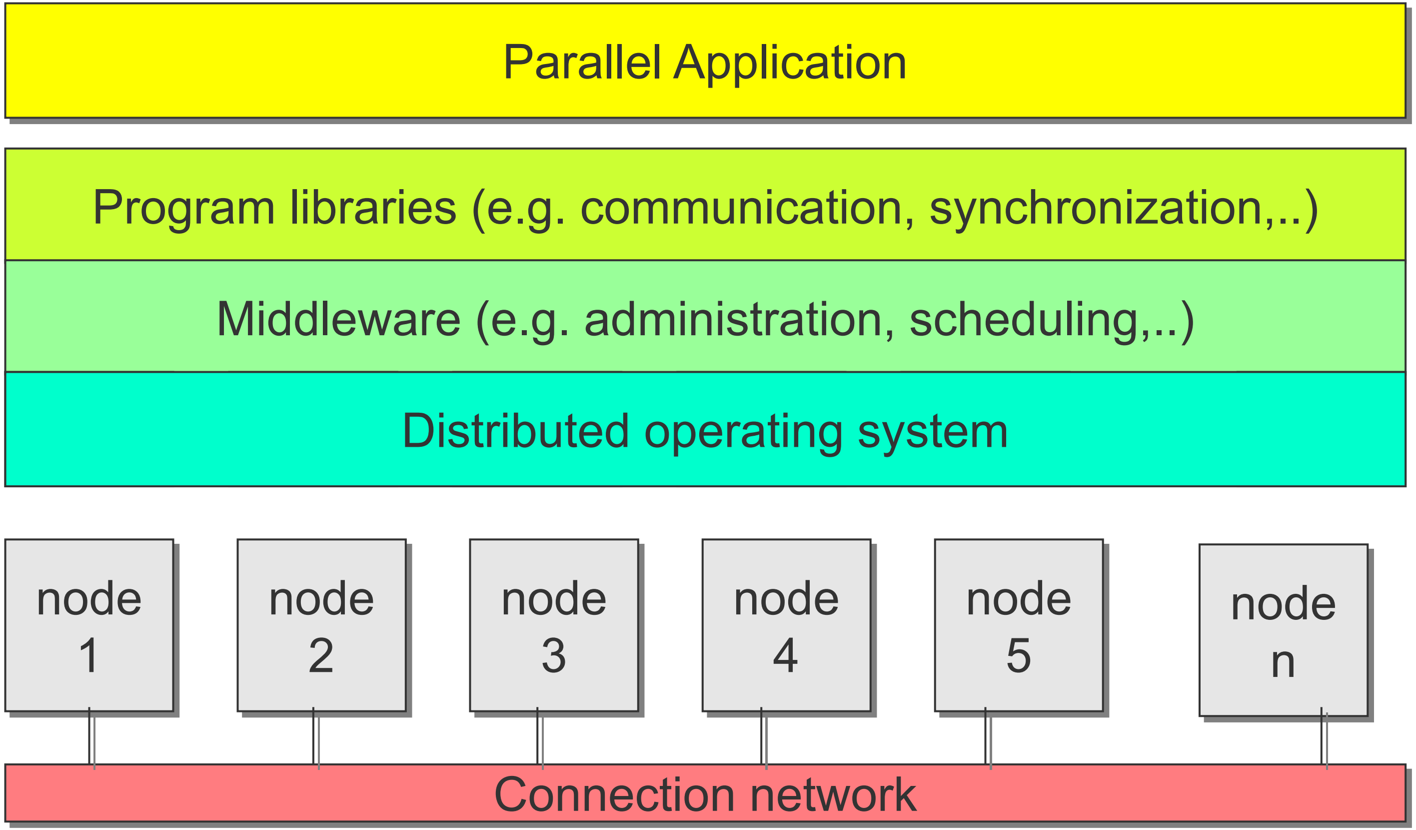
- MPI stands for Message Passing Interface.
- MPI is the standard for message passing programming in parallel programming and especially in high performance computing (HPC).
- It is basically a library for functions supporting process or thread interaction.
- Additionally, MPI comes with an runtime environment to control the process or thread interaction.
- There is a primary support for the dominating programming languages used in HPC, such as C, C++ and Fortran.
- Nowadays other languages as C#, Java and Python are supported or are able to import the library, too.

MPI – Overview II

- MPI introduces support for a variety of network technologies, especially support for high performance networks.

- MPI can be used as infrastructure to follow the Design Methodology by Ian Foster.

- Different implementations of the MPI standard are available.
- Free versions are:
  - MPICH - https://www.mpich.org
  - OpenMPI - www.open-mpi.org


Parallel Application
Program libraries (e.g. communication, synchronization,..)
Middleware (e.g. administration, scheduling,..)
Distributed operating system
node 1
node 2
node 3
node 4
node 5
node n
Connection network

Compiling and Launching an MPI Program

- To compile an MPI program the environment is set and the corresponding compiler is run with:

```
mpicc -o test test.c
```

- The MPI program is started:

```
mpirun -machinefile Machinefile -np 4 test
```

  - `-machinefile Machinefile` – The file `Machinefile` contains the nodes the program should start it's processes on. With recent versions of the MPI library together with the number of MPI processes to be started on the node.
  - `-np` – Gives the number of MPI processes to be started.
  - `mpirun` uses remote or secure shell (`ssh` or `rsh`) to log in to the nodes and to create the processes on the node.

Machinefile

- The content of the machinefile may look like this:

```
node00:2
node01:2
```

  - Two nodes are given. Each with two MPI processes to be started (2 CPUs).
  - In previous implementations of the MPI standard the machinefile only had to contain the nodes (the name the nodes can be reached using rsh or ssh) without the amount of processes to be started on the node.

MPI_Init and MPI_Finalize

- Using C/C++ the provided MPI functions have to be included by:

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

```
MPI_Init (&argc, &argv);
```

  - Initialize the MPI runtime environment with the arguments that were passed to the program.
  - Thus, the arguments are distributed among all of the processes started in this specific MPI runtime environment for the program.

```
MPI_Finalize ();
```

  - Shut down the MPI runtime environment and release all of the connected resources.

MPI_Send

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

- Sending a message to a receiving process.
- The function blocks the process until the message buffer `smessage` is avaiable again. Usually, the run-time system copies the message into a system buffer, but this has not to be implemented.

MPI_Send

- The parameters for `MPI_Send` are:

- `smessage` – pointer to the buffer in which the message to be sent is located,
- `count` – amount of elements of type `datatype` to be sent,
- `datatype` – type of elements to be sent, all elements of a message must have the same type,
- `dest` – number of the process to which the message is to be sent,
- `tag` – tag for the message, allows the received process to distinguish different messages from the same sender,
- `comm` – communicator that describes the group of processes that can exchange messages.

MPI_Recv

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

- Receiving a message from a sending process.
- The function blocks the process until the message is stored at buffer `rmessage`.

MPI_Recv

- The parameters for `MPI_Recv` are:
  - `rmessage` – pointer to the buffer in which the message to be received should be stored,
  - `count` – limit for the amount of elements to be received,
  - `datatype` – type of elements to be received,
  - `source` – number of the process from which the message is to be received,
  - `tag` – tag for the message,
  - `comm` – communicator that describes the group of processes that can exchange messages,
  - `status` – data structure containing information about the transfer of message received.

MPI_Recv

Some constants may be useful in order to control the `MPI_Recv` function:

`source == MPI_ANY_SOURCE`

- Receiving a message from any sending process.

`tag == MPI_ANY_TAG`

- Receiving a message with any tag.

After receiving a message, the following information about the message are available:

`status.MPI_SOURCE` – sender process

`status.MPI_TAG` – marking the received message

`status.MPI_ERROR` – error code

MPI_Get_count

```
int MPI_Get_count ( MPI_Status *status,
                    MPI_Datatype datatype,
                    int *count_ptr);
```

- Returns the actual size of the received message with the MPI status `status` in `count_ptr`.

MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	(packed data in bytes)
MPI_BYTE	(a untyped byte)
MPI	(general)

Freie Universität Berlin

MPI_Comm

- The communicator indicates the group of processes involved in the communication.

- The predefined communicator `MPI_COMM_WORLD` includes all processes of the program started by `mpirun`.

MPI_Comm_rank

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

- Returns the rank of the process (as unique ID) within the process group of the communicator `comm`.

MPI_Comm_size

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

- Returns the number of processes within the process group of the communicator `comm` and stores it in `size`.

```c
// simple program with MPI
// Rauber, Ruenger: Parallele und vert. Prg.
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
  int my_rank, p, source, dest, tag = 0;


  MPI_Init (&argc, &argv);





  MPI_Finalize ();
  return 0;
}
```

source code: 11-00.c

Freie Universität Berlin

```c
// simple program with MPI
// Rauber, Ruenger: Parallele und vert. Prg.
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
  int my_rank, p, source, dest, tag = 0;


  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &p);





  MPI_Finalize ();
  return 0;
}
```

source code: 11-00.c

```c
// simple program with MPI
// Rauber, Ruenger: Parallele und vert. Prg.
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
  int my_rank, p, source, dest, tag = 0;
  char msg[20];

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  if (my_rank == 0) {
    strcpy (msg, "Hello ");

  }


  MPI_Finalize ();
  return 0;
}
```

source code: 11-00.c

```c
// simple program with MPI
// Rauber, Ruenger: Parallele und vert. Prg.
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
  int my_rank, p, source, dest, tag = 0;
  char msg[20];
  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  if (my_rank == 0) {
    strcpy (msg, "Hello ");
    MPI_Send (msg, strlen (msg) + 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
  }
  if (my_rank == 1)
    MPI_Recv (msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  MPI_Finalize ();
  return 0;
}
```

source code: 11-00.c

```c
// simple program with MPI
// Rauber, Ruenger: Parallele und vert. Prg.
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
  int my_rank, p, source, dest, tag = 0;
  char msg[20];
  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  if (my_rank == 0) {
    strcpy (msg, "Hello ");
    MPI_Send (msg, strlen (msg) + 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
  }
  if (my_rank == 1)
    MPI_Recv (msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  MPI_Finalize ();
  return 0;
}
```

source code: 11-00.c

Messages Order

- The sequence of the sent messages between the sender and receiver process is ensured on the receiver side.
  - Messages in a communication sequence do not overtake each other.

- But: The preservation of the sequence does not apply if further processes are involved!

Messages Order II

```c
// Example not preserving the message order
// Rauber, Ruenger: Parallele und vert. Prg.

MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Send (sendbuf1, count, MPI_INT, 2, tag, comm);
  MPI_Send (sendbuf2, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
  MPI_Recv (recvbuf1, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send (recvbuf1, count, MPI_INT, 2, tag, comm);
}
else if (my_rank ==2) {
  MPI_Recv (recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
  MPI_Recv (recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status);
}
```

The MPI runtime environment has no control about the execution of the operations of this MPI process.

source code: 11-01.c

Deadlocks

- Messages (which are waited for) can be considered as resources.

- Thus, parallel programs with message passing have to face the challenges regarding the use of resources and the possibility of deadlocks (see lecture about deadlocks), too.

w

for

Example for a Deadlock

```
// Example for a program with a deadlock
// Rauber, Ruenger: Parallele und vert. Prg.

MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

source code: 11-02.c

Example for a Deadlock depending on the MPI environment

```c
// Example for a program with a deadlock depending on the
// implementation of the MPI environment
// Rauber, Ruenger: Parallele und vert. Prg.

MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
```

source code: 11-03.c

Freie Universität Berlin

Example without Deadlock

```
// Example without deadlock
// Rauber, Ruenger: Parallele und vert. Prg.

MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
  MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
  MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
  MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

source code: 11-04.c

MPI_Sendrecv

```
int MPI_Sendrecv (  void *sendbuf, int sendcount,
                    MPI_Datatype senddatatype, int dest,
                    int sendtag,
                    void *recvbuf, int recvcount,
                    MPI_Datatype recvdatatype, int source,
                    int recvtag,
                    MPI_Comm comm,
                    MPI_Status *status);
```

– Combines a send and a receive operation.

MPI_Sendrecv

- `sendbuf` – pointer to the buffer in which the message to be sent is located
- `recvbuf` – pointer to the buffer in which the message to be received should be stored
- Send and receive buffers must not overlap!

- `sendcount/recvcount` – number of elements to be sent / received
- `senddatatype/recvdatatype` – type of elements to be sent / received
- `dest/source` – number of the process to which the message is to be sent/from which the message is to be received
- `sendtag/recvtag` – tags for the messages

MPI_Sendrecv

- `comm` – communicator that describes the group of processes that can exchange messages
- `status` – data structure containing information about the message actually received

MPI_Sendrecv_replace

```
int MPI_Sendrecv_replace ( void *buffer,
                           int count,
                           MPI_Datatype datatype,
                           int dest,
                           int sendtag,
                           int source,
                           int recvtag,
                           MPI_Comm comm,
                           MPI_Status *status);
```

- Combined send and receive operation with only one, share buffer for the send and the receive operation.

MPI_Isend

```
int MPI_Isend (      void *buffer,
                     int count,
                     MPI_Datatype datatype,
                     int dest,
                     int tag,
                     MPI_Comm comm,
                     MPI_Request *request);
```

- Asynchronous (non-blocking) sending of a message to a receiver process.
- In order to check if the operation is finished and the buffer can be used again `MPI_Wait()` can be used.

MPI_Isend

- `buffer` – pointer to the buffer in which the message to be sent is located
- `count` – number of elements of type datatype to be sent
- `datatype` – type of elements to be sent, all elements of a message must have the same type
- `dest` – number of the process to which the message is to be sent
- `tag` – message marking, allows the received process to distinguish different messages from the same sender
- `comm` – communicator that describes the group of processes that can exchange messages
- `request` – status information about the execution status of the message transmission (not directly accessible)

MPI_Irecv

```
int MPI_Irecv (      void *buffer,
                     int count,
                     MPI_Datatype datatype,
                     int source,
                     int tag,
                     MPI_Comm comm,
                     MPI_Request *request);
```

- Receiving a message from a sender process in an asynchronous (non-blocking) operation.

MPI_Irecv

- `buffer` – pointer to the buffer in which the message to be received should be stored
- `count` – limit for the amount of elements to be received
- `datatype` – Type of elements to be received
- `source` – number of the process from which the message is to be received
- `tag` – tag of the message to be received
- `comm` – communicator that describes the group of processes that can exchange messages
- `request` – status information about the execution status

MPI_Test

```
int MPI_Test (       MPI_Request *request,
                     int *flag,
                     MPI_Status *status);
```

– The status of a message passing operation provided by `MPI_Request` can be transferred to `MPI_Status` and `flag` using `MPI_Test()`.

Freie Universität Berlin

MPI_Test

- `request` – status information about the execution status of the send or receive operation,
- `flag` – is true (1) if the asynchronous (non-blocking) operation is completed, otherwise false (0),
- `status` – data structure containing information about the message .

MPI_Wait

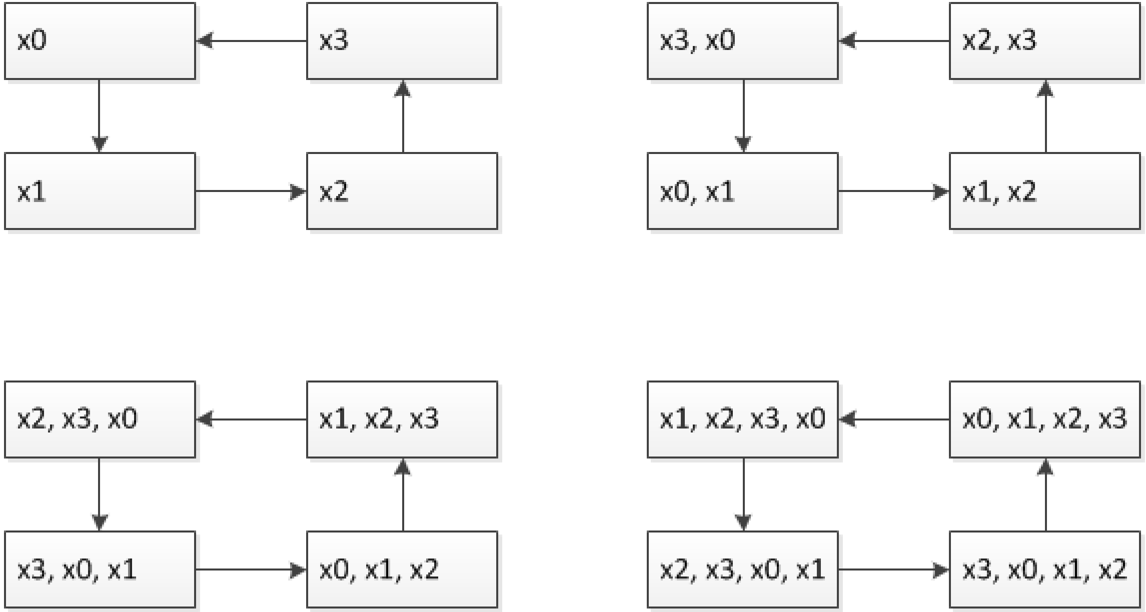
```
int MPI_Wait (      MPI_Request *request,
                    MPI_Status *status);
```

- Blocks the process until the (asynchronous, non-blocking) operation associated with request is completed.
- `MPI_Wait()` returns when the data has been copied at the receiver into the buffer or into a buffer of the MPI runtime environment.
- Ensures that the buffers (on the sender side) or buffer contents (on the receiver process side) can be used (again).

MPI_Wait

- `request` – status information about the status of the execution of the send or receive operation and to determine the operation,
- `status` – data structure containing information about the message.

Example: Gathering Data using Ring



```c
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;









  for (i = 0; i < p-1; i++) {





  }
}
```

source code: 11-05.c

```
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;


  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);




  for (i = 0; i < p-1; i++) {




  }
}
```

source code: 11-05.c

```c
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;


  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < blocksize; i++)
    y[i + my_rank * blocksize] = x[i];


  for (i = 0; i < p-1; i++) {




  }
}
```

source code: 11-05.c

```c
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;


  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < blocksize; i++)
    y[i + my_rank * blocksize] = x[i];
  succ = (my_rank + 1) % p;
  pred = (my_rank - 1 + p) % p;
  for (i = 0; i < p-1; i++) {



  }
}
```

source code: 11-05.c

```
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;


  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < blocksize; i++)
    y[i + my_rank * blocksize] = x[i];
  succ = (my_rank + 1) % p;
  pred = (my_rank - 1 + p) % p;
  for (i = 0; i < p-1; i++) {
    send_offset = ((my_rank - i + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p) * blocksize;



  }
}
```

source code: 11-05.c

```c
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;


  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < blocksize; i++)
    y[i + my_rank * blocksize] = x[i];
  succ = (my_rank + 1) % p;
  pred = (my_rank - 1 + p) % p;
  for (i = 0; i < p-1; i++) {
    send_offset = ((my_rank - i + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p) * blocksize;
    MPI_Send (y + send_offset, blocksize, MPI_FLOAT, succ, 0, MPI_COMM_WORLD);


  }
}
```

source code: 11-05.c

```
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;
  MPI_status status;

  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < blocksize; i++)
    y[i + my_rank * blocksize] = x[i];
  succ = (my_rank + 1) % p;
  pred = (my_rank - 1 + p) % p;
  for (i = 0; i < p-1; i++) {
    send_offset = ((my_rank - i + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p) * blocksize;
    MPI_Send (y + send_offset, blocksize, MPI_FLOAT, succ, 0, MPI_COMM_WORLD);
    MPI_Recv (y + recv_offset, blocksize, MPI_FLOAT, pred, 0, MPI_COMM_WORLD,
         &status);
  }
}
```

source code: 11-05.c

```c
// simple MPI program to gather data
// Rauber, Ruenger: Parallele und vert. Prg.

void Gather_ring (float x[], int blocksize, float y[]) {
  int i, p, my_rank, succ, pred, send_offset, recv_offset;
  MPI_status status;

  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i = 0; i < blocksize; i++)
    y[i + my_rank * blocksize] = x[i];
  succ = (my_rank + 1) % p;
  pred = (my_rank - 1 + p) % p;
  for (i = 0; i < p-1; i++) {
    send_offset = ((my_rank - i + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p) * blocksize;
    MPI_Send (y + send_offset, blocksize, MPI_FLOAT, succ, 0, MPI_COMM_WORLD);
    MPI_Recv (y + recv_offset, blocksize, MPI_FLOAT, pred, 0, MPI_COMM_WORLD,
         &status);
  }
}
```

source code: 11-05.c

```
/// simple MPI program to gather data with asynchr. comm.
// Rauber, Ruenger: Parallele und vert. Prg.
void Gather_ring (float x[], int blocksize, float y[])
{
  int i, p, my_rank, succ, pred, send_offset, recv_offset;



  // get rank and size and init buffers

  for (i = 0; i < p-1; i++) {




    send_offset = ((my_rank - i - 1 + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 2 + p) % p) * blocksize;


  }
}
```

source code: 11-06.c

```c
/// simple MPI program to gather data with asynchr. comm.
// Rauber, Ruenger: Parallele und vert. Prg.
void Gather_ring (float x[], int blocksize, float y[])
{
  int i, p, my_rank, succ, pred, send_offset, recv_offset;
  MPI_status status;
  MPI_Request send_request, recv_request;

  // get rank and size and init buffers

  for (i = 0; i < p-1; i++) {
    MPI_Isend (y + send_offset, blocksize, MPI_FLOAT, succ, 0, MPI_COMM_WORLD,
          &send_request);


    send_offset = ((my_rank - i - 1 + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 2 + p) % p) * blocksize;
    MPI_Wait (&send_request, &status);

  }
}
```

source code: 11-06.c

```c
/// simple MPI program to gather data with asynchr. comm.
// Rauber, Ruenger: Parallele und vert. Prg.
void Gather_ring (float x[], int blocksize, float y[])
{
  int i, p, my_rank, succ, pred, send_offset, recv_offset;
  MPI_status status;
  MPI_Request send_request, recv_request;

  // get rank and size and init buffers

  for (i = 0; i < p-1; i++) {
    MPI_Isend (y + send_offset, blocksize, MPI_FLOAT, succ, 0, MPI_COMM_WORLD,
         &send_request);
    MPI_Irecv (y + recv_offset, blocksize, MPI_FLOAT, pred, 0, MPI_COMM_WORLD,
         &recv_request);
    send_offset = ((my_rank - i - 1 + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 2 + p) % p) * blocksize;
    MPI_Wait (&send_request, &status);
    MPI_Wait (&recv_request, &status);
  }
}
```

source code: 11-06.c

```c
// simple MPI program to gather data with asynchr. comm.
// Rauber, Ruenger: Parallele und vert. Prg.
void Gather_ring (float x[], int blocksize, float y[])
{
  int i, p, my_rank, succ, pred, send_offset, recv_offset;
  MPI_status status;
  MPI_Request send_request, recv_request;

  // get rank and size and init buffers

  for (i = 0; i < p-1; i++) {
    MPI_Isend (y + send_offset, blocksize, MPI_FLOAT, succ, 0, MPI_COMM_WORLD,
         &send_request);
    MPI_Irecv (y + recv_offset, blocksize, MPI_FLOAT, pred, 0, MPI_COMM_WORLD,
         &recv_request);
    send_offset = ((my_rank - i - 1 + p) % p) * blocksize;
    recv_offset = ((my_rank - i - 2 + p) % p) * blocksize;
    MPI_Wait (&send_request, &status);
    MPI_Wait (&recv_request, &status);
  }
}
```

source code: 11-06.c

MPI_Ssend

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

– (Explicitly) synchronous, blocking sending of a message to a receiver process.

MPI_Issend

```
int MPI_Issend (     void *buffer,
                     int count,
                     MPI_Datatype datatype,
                     int dest,
                     int tag,
                     MPI_Comm comm,
                     MPI_Request *request);
```

- Non-blocking sending of a message in synchronous mode.
- `MPI_Wait()` returns only after the data has been copied into the buffer at the receiver.

MPI_Bsend

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

– Sending a message to a receiver process and copying the message to a communication buffer.

MPI_Ibsend

```
int MPI_Ibsend (     void *buffer,
                     int count,
                     MPI_Datatype datatype,
                     int dest,
                     int tag,
                     MPI_Comm comm,
                     MPI_Request *request);
```

– Asynchronous (non-blocking) sending of a message to a receiver process with copying of the message into a communication buffer.

MPI_Buffer_attach

```
int MPI_Buffer_attach (    void *buffer,
                           int buffersize);
```

- Setting up a communication buffer for buffered message transmission. The size of the buffer is specified in bytes via `buffersize`.
- The size of the buffer must be sufficient for the buffered message, otherwise the corresponding communication operation cannot be executed.

MPI_Buffer_detach

```
int MPI_Buffer_detach (    void *buffer,
                           int *buffersize);
```

- Releases a communication buffer for buffered message transmission. The size of the buffer is specified in bytes via `buffersize`.
- The function only returns when all messages in the buffer have been delivered.

Concepts of Non-sequential and Distributed Programming

NEXT LECTURE

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

MPI Group Communication and MPI-2

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2023)