Algorithms and Programming IV
MPI – Message Passing Interface

Summer Term 2024 | 27.05.2024
Barry Linnert
Objectives of Today‘s Lecture

• Introduction to MPI
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 its 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 available 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

```c
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 \texttt{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

```c
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

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>(packed data in bytes)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>(a untyped byte)</td>
</tr>
<tr>
<td>MPI</td>
<td>(general)</td>
</tr>
</tbody>
</table>
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

```c
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

```c
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`. 
// simple program with MPI
#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;
}
// simple program with MPI
#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;
}
```c
#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
// simple program with MPI
#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;
}
```c
#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

// Example not preserving the message order

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);
}

source code: 11-01.c

The MPI runtime environment has no control about the execution of the operations of this MPI process.
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.
Example for a Deadlock

// Example for a program with a deadlock

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

// Example for a program with a deadlock depending on the
// implementation of the MPI environment

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
Example without Deadlock

// Example without deadlock

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);
}
MPI_Sendrecv

```c
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

```c
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**

```c
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
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().
**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

```c
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
// simple MPI program to gather data

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
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
// simple MPI program to gather data

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
// simple MPI program to gather data

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++) {
    }

}
// simple MPI program to gather data

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
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);
    }
}
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

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);
    }
}
/// simple MPI program to gather data with asynchr. comm.
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
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
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
// simple MPI program to gather data with asynchr. comm.

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_Buffer_attach

```c
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

```c
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

MPI GROUP COMMUNICATION
MPI_Bcast

```c
int MPI_Bcast ( void *buffer,
               int count,
               MPI_Datatype datatype,
               int root,
               MPI_Comm comm);
```

- (Blocking) broadcast operation to send \((\text{root} == \text{my\_rank})\) or receive \((\text{root} != \text{my\_rank})\) a message to all participating processes of the communicator \(\text{comm}\).
- The broadcast message must be received by all processes with \(\text{MPI\_Bcast()}.\)
- In case several messages were sent the sequence of these will be preserved.
**MPI_Bcast**

- **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,
- **root** – rank of the sending process,
- **comm** – communicator that describes the group of processes that can exchange messages.
MPI_Reduce

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

- Merging the content of different messages in a global reduction operation (accumulation operation) \( \text{op} \) to be stored in a single value in \( \text{recvbuf} \).
- All processes of the communicator \( \text{comm} \) have to send messages for the reduce operation.
- The root process \( \text{root} \) must provide the receive buffer \( \text{recvbuf} \).
MPI_Reduce

- **sendbuf** – pointer to the buffer in which the message to be sent is located
- **recvbuf** – pointer to the buffer in which the messages are stored when received.
- **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
- **op** – reduction operation
- **root** – rank of the receiving process
- **comm** – communicator that describes the group of processes that can exchange messages
## MPI_Reduce – Operations

<table>
<thead>
<tr>
<th>MPI_Reduce operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Returns the maximum element</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Returns the minimum element</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sums the elements</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Multiplies all elements</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Performs a <em>logical and</em> across the elements</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Performs a <em>bitwise and</em> across the bits of the elements</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Performs a <em>logical or</em> across the elements</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Performs a <em>bitwise or</em> across the bits of the elements</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Performs a <em>logical exclusive or</em> across the elements</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Performs a bitw. <em>exclusive or</em> across the bits of the elements</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Returns the max. value and the rank of the proc. that owns it</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Returns the min. value and the rank of the proc. that owns it</td>
</tr>
</tbody>
</table>
MPI_Op_create

```c
int MPI_Op_create ( MPI_User_function *function,
                    int commute,
                    MPI_Op *op);
```

- To define a special reduce operation to be used with `MPI_Reduce()`.
MPI_Op_create

- function – pointer to the function to be applied to the corresponding operation,
- commute – specifies whether the operation is commutative \((\text{commute} = 1)\) or not \((\text{commute} = 0)\),
- op – data type of the operation to be applied.
## Extended MPI Datatypes

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>Combination of C datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_FLOAT_INT</td>
<td>(float, int)</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>(double, int)</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>(long, int)</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>(short, int)</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>(long double, int)</td>
</tr>
<tr>
<td>MPI_2INT</td>
<td>(int, int)</td>
</tr>
</tbody>
</table>
MPI_Type_create_struct

```c
int MPI_Type_create_struct ( 
    int count,
    int array_of_blocklengths[],
    const MPI_Aint array_of_displacements[],
    const MPI_Datatype array_of_types[],
    MPI_Datatype *new_type);
```

- Generation of a new structured data type as a combination of MPI basic types.
MPI_Type_create_struct

- `count` – number of elements in the following arrays,
- `array_of_blocklengths` – specifies the number of elements in each block,
- `array_of_displacements` – specifies the number of bytes to move each block,
- `array_of_types` – specifies the MPI type of the elements of each block,
- `new_type` – new data type.
MPI_Type_contiguous

```c
int MPI_Type_contiguous (  
    int count,  
    MPI_Datatype old_type,  
    MPI_Datatype *new_type);
```

- Creates a contiguous datatype out of an existing datatype.
MPI_Type_commit

int MPI_Type_commit (MPI_Datatype *data_type);

- Commits a datatype to be used by the MPI environment.
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr) {
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real - inout->imag*in->imag;
        c.imag = inout->real*in->imag + inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Compl.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// At this point, the answer, which consists of 100 Complexes,
// resides on root
source code: 12-00.c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr) {
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real - inout->imag*in->imag;
        c.imag = inout->real*in->imag + inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Complex.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );

// At this point, the answer, which consists of 100 Complexes,
// resides on root source code: 12-00.c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
void myProd( Complex *in, Complex *inout,
    int *len, MPI_Datatype *dptr)
{
    int i;
    Complex c;
    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real -
            inout->imag*in->imag;
        c.imag = inout->real*in->imag +
            inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Complex.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE,
    &ctype );

MPI_Type_commit( &ctype );

// At this point, the answer, which
// consists of 100 Complexes,
// resides on root
source code: 12-00.c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr)
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real - inout->imag*in->imag;
        c.imag = inout->real*in->imag + inout->imag*in->real;
        *inout = c;
        inout++; in++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Complex.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE,
    &ctype );
MPI_Type_commit( &ctype );

// create the complex-product user-op
MPI_Op_create( myProd, True, &myOp );

// At this point, the answer, which
// consists of 100 Complexes,
// resides on root

source code: 12-00.c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr)
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real –
                 inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                 inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Complex.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE,
                     &ctype );
MPI_Type_commit( &ctype );

// create the complex-product user-op
MPI_Op_create( myProd, True, &myOp );

MPI_Reduce( a, answer, 100, ctype,
            myOp, root, comm );

// At this point, the answer, which
// consists of 100 Complexes,
// resides on root
source code: 12-00.c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
void myProd( Complex *in, Complex *inout, 
    int *len, MPI_Datatype *dptr) 
{
    int i;
    Complex c;

    for (i=0; i< *len; ++i) {
        c.real = inlet->real*in->real - 
                 inlet->imag*in->imag;
        c.imag = inlet->real*in->imag + 
                 inlet->imag*in->real;
        *inout = c; 
        in++; inout++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Complex...
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE, 
    &ctype );
MPI_Type_commit( &ctype );

// create the complex-product user-op
MPI_Op_create( myProd, True, &myOp );

MPI_Reduce( a, answer, 100, ctype, 
            myOp, root, comm );

// At this point, the answer, which 
// consists of 100 Complexes,
// resides on root

source code: 12-00.c
**MPI_Gather**

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

- Gathers together values from a group of processes of the communicator `comm` (without reduction operation).
- The elements are stored in order of the numbers of the processes involved.
MPI_Gather

- sendbuf  – pointer to the buffer in which the message to be sent is located,
- sendcount – number of elements of type datatype to be sent,
- sendtype – type of elements to be sent,
- recvbuf  – pointer to the buffer in which the messages are stored when received,
- recvcount – number of elements of type datatype to be received,
- recvtype – type of elements to be sent, all elements of a message must have the same type,
- root    – rank of the receiving process,
- comm    – communicator that describes the group of processes that can exchange messages.
MPI_Gatherv

```c
int MPI_Gatherv ( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf, const int recvcounts[],
                  const int displs[],
                  MPI_Datatype recvtype,
                  int root,
                  MPI_Comm comm);
```

- Gathers together values out of messages with different size from a group of processes of the communicator `comm`.
- The root process specifies the number (`recvcounts`) and position of the storage within the receive buffer (`displs`).
MPI_Scatter

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

- Distribution of individual data (same size) by messages to all the processes of the communicator `comm`.
- The data is distributed and sent according to the numbers of the target processes.
MPI_Scatter

- `sendbuf` – pointer to the buffer containing the data to be sent,
- `sendcount` – number of elements of type `datatype` to be sent,
- `sendtype` – type of elements to be sent,
- `recvbuf` – pointer to the buffer in which the message is stored when received,
- `recvcount` – number of elements of type `datatype` to be received,
- `recvtype` – type of elements to be sent, all elements of a message must have the same type,
- `root` – rank of the receiving process,
- `comm` – communicator that describes the group of processes that can exchange messages.
MPI_Scatterv

```c
int MPI_Scatterv ( void *sendbuf,
                  const int sendcounts[],
                  const int displs[],
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int root,
                  MPI_Comm comm);
```

- Distribution of individual data of different sizes by messages to all the processes of the communicator `comm`. 
**MPI_Allgather**

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

- Gathers together data from messages from all processes of the communicator `comm` and distribute it to all involved processes.
- The elements are stored in the order of the numbers of the processes involved.
MPI_Allgather

- `sendbuf` – pointer to the buffer containing the data to be sent,
- `sendcount` – number of elements of type `datatype` to be sent,
- `sendtype` – type of elements to be sent,
- `recvbuf` – pointer to the buffer in which the message is stored when received,
- `recvcount` – number of elements of type `datatype` to be received,
- `recvtype` – type of elements to be sent, all elements of a message must have the same type,
- `comm` – communicator that describes the group of processes that can exchange messages.
MPI_Allgatherv

```c
int MPI_Allgatherv (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  const int recvcounts[],
                  const int displs[],
                  MPI_Datatype recvtype,
                  MPI_Comm comm);
```

- Gathering of individual messages from all processes of the communicator `comm` (without reduction operation) and distribution to all involved processes.
MPI_Allreduce

```c
int MPI_Allreduce ( void *sendbuf,
                    void *recvbuf,
                    int count,
                    MPI_Datatype datatype,
                    MPI_Op op,
                    MPI_Comm comm);
```

- Merging the content of different messages in a global reduction operation (accumulation operation) \( op \) and distributes the result back to all processes of the communicator \( comm \).
MPI_Allreduce

- sendbuf  – pointer to the buffer containing the data to be sent,
- recvbuf  – pointer to the buffer in which the message is stored when received,
- count    – number of elements of type datatype to be received,
- datatype – type of elements to be sent, all elements of a message must have the same type,
- op       – reduction operation,
- comm     – communicator that describes the group of processes that can exchange messages.
// part of simple MPI program with MPI_Allreduce
// to multiply a matrix and a vector

int m, local_m, n, p;
float a[MAX_N][MAX_LOC_M], local_b[MAX_LOC_M];
float c[MAX_N], sum[MAX_N];

local_m = m / p;
for (i = 0; i < n; i++) {
    sum[i] = 0;
    for (j = 0; j < local_m; j++)
        sum[i] = sum[i] + a[i][j] * local_b[j];
}
MPI_Allreduce (sum, c, n, MPI_FLOAT, MPI_SUM, comm);
MPI_Alltoall

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

- Total exchange of (individual) messages of equal size of all processes of the communicator comm (without reduction operation).
MPI_Alltoall

- `sendbuf` – pointer to the buffer containing the data to be sent,
- `sendcount` – number of elements of type `datatype` to be sent,
- `sendtype` – type of elements to be sent,
- `recvbuf` – pointer to the buffer in which the message is stored when received,
- `recvcount` – number of elements of type `datatype` to be received,
- `recvtype` – type of elements to be sent, all elements of a message must have the same type,
- `comm` – communicator that describes the group of processes that can exchange messages.
MPI_Alltoallv

int MPI_Alltoallv ( void *sendbuf,
                    const int sendcounts[],
                    const int sdispls[],
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    const int recvcounts[],
                    const int rdispls[],
                    MPI_Datatype recvtype,
                    MPI_Comm comm);

- Total exchange of (individual) messages with different size of all processes of
  the communicator comm (without reduction operation).
### MPI_Comm_group

```c
int MPI_Comm_group (MPI_Comm comm,
                    MPI_Group *group);
```

- Returns all processes assigned to the communicator `comm` in the group data structure `group`. 
MPI_Group_union

```c
int MPI_Group_union ( MPI_Group group1,
                     MPI_Group group2,
                     MPI_Group *new_group);
```

- Merges the processes of the groups `group1` and `group2` into a group `new_group`.
MPI_Group_intersection

int MPI_Group_intersection ( MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group);

- Produces a new group new_group as the intersection of the processes of the groups group1 and group2.
MPI_Group_difference

```c
int MPI_Group_difference ( MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group);
```

- Makes a new group `new_group` from the difference of the groups `group1` and `group2`. 
MPI_Group_incl

```c
int MPI_Group_incl (MPI_Group group,
                     int p,
                     const int ranks[],
                     MPI_Group *new_group);
```

- Creates a new group `new_group` from the processes of an existing group `group` by taking only the `p` processes listed in `ranks`. 
MPI_Group_excl

int MPI_Group_excl (MPI_Group group,
        int p,
        const int ranks[],
        MPI_Group *new_group);

- Creates a new group new_group from the processes of an existing group group by not adopting the p processes listed in ranks.
MPI_Group_compare

```c
int MPI_Group_compare ( MPI_Group group1,
                         MPI_Group group2,
                         int *res);
```

- Compares two groups and stores the result in `res`.
- For groups with the same processes in the same order `MPI_IDENT` is returned, for groups with the same processes in different orders `MPI_SIMILAR` and for different groups `MPI_UNEQUAL`. 
MPI_Group_free

```c
int MPI_Group_free (MPI_Group *group);
```

- Releases the data structure holding the group `group`. 
MPI_Comm_create

```c
int MPI_Comm_create ( MPI_Comm comm,
                     MPI_Group group,
                     MPI_Comm new_comm);
```

- Creates a new communicator `new_comm`, which addresses the processes of the group `group`, from the existing communicator `comm`. 
MPI_Comm_compare

```c
int MPI_Comm_compare ( MPI_Comm comm1,
                      MPI_Comm comm2,
                      int *res);
```

- Compares the two communicators `comm1` and `comm2` and stores the result in `res`.
- If `comm1` and `comm2` point to the same data structure, `MPI_IDENT` is returned. If there are different communicators with the same processes in the same order `MPI_CONGRUENT` is returned, if there are communicators with the same processes in different orders `MPI_SIMILAR` and if there are different communicators `MPI_UNEQUAL` is returned.
MPI_Comm_dup

```
int MPI_Comm_dup ( MPI_Comm comm,
                  MPI_Comm new_comm);
```

- Creates a new communicator `new_comm` with the same processes in the same order as the communicator `comm`.
MPI_Comm_split

int MPI_Comm_split (MPI_Comm comm,
                   int color,
                   int key,
                   MPI_Comm *new_comm);

- Split the processes of the communicator `comm` according to the values `color` in the order `key` and return the communicator in which the corresponding process is found.
- If a process has not set the value `color`, it will not be found in any of the created communicators.
MPI_Comm_free

int MPI_Comm_free ( MPI_Comm *comm);

- Releases the communicator comm after all message transmissions performed with this communicator have been completed.
MPI_Wtime

double MPI_Wtime ( void);

- Returns the time in seconds after a certain time. The elapsed processing time can be determined from the difference between second calls.
Concepts of Non-sequential and Distributed Programming

MPI-2
MPI-2

- MPI-2 is an extension of the MPI standard.
- It introduces new functions to support
  - memory transfers,
  - dynamic process management,
  - input/output operations.

- In particular, applications with very high resource requirements will be programmed with dynamic process generation and dynamic runtime behavior.
MPI_Comm_spawn

```c
int MPI_Comm_spawn (const char *command,
                    char *argv[],
                    int maxprocs,
                    MPI_Info info,
                    int root,
                    MPI_Comm comm,
                    MPI_Comm *intercomm,
                    int array_of_errcodes[]);
```

- Creates a number of `maxprocs` new MPI processes that execute the `command` program. The child processes still have to call `MPI_Init()` for execution.
MPI_Comm_spawn

- The parameters for MPI_Comm_spawn are:
  - `command` – program executed by the child processes,
  - `argv` – arguments passed to the child processes,
  - `maxprocs` – number of child processes to be created,
  - `info` – process information or `MPI_INFO_NULL`, to transfer the administration to the runtime system,
  - `root` – rank of the parent process,
  - `comm` – communicator of the parent process,
  - `intercomm` – communicator, the group of child processes,
  - `array_of_errcodes` – error code per child process.
MPI_Comm_get_parent

```c
int MPI_Comm_get_parent ( MPI_Comm *parent);
```

- Returns the communicator of the parent process in `parent`. 
MPI_Comm_spawn_multiple

```c
int MPI_Comm_spawn_multiple ( int count,
const char *commands[],
char **argv[],
int maxprocs[],
MPI_Info infos[],
int root,
MPI_Comm comm,
MPI_Comm *intercomm,
int array_of_errcodes[]);
```

- Starts `count` many command programs, each with `maxprocs` many processes.
- It creates one communicator for all child processes.
MPI_Comm_spawn_multiple

- **count** – number of programs to be started,
- **command** – program executed by the child processes,
- **argv** – arguments passed to the child processes,
- **maxprocs** – number of child processes to be created,
- **info** – process information or MPI_INFO_NULL, to transfer the administration to the runtime system,
- **root** – rank of the parent process,
- **comm** – communicator of the parent process,
- **intercomm** – communicator, the group of child processes,
- **array_of_errcodes** – error code per child process.
Concepts of Non-sequential and Distributed Programming

NEXT LECTURE
Design and Implementation of Parallel Applications

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2024)