Parallel Programming
Lecture 9

Prof. Dr. Jonas Posner

jonas.posner@cs.hs-fulda.de

Hochschule Fulda
University of Applied Sciences

Winterterm 2025/2026
E-Learning: AI5085

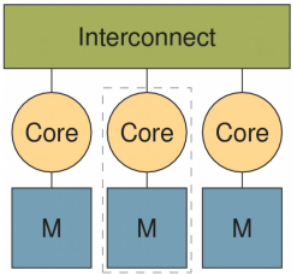
12th December 2025

Organization

- **No** lecture next week 😀
  - Exercises take place as usual
- Lectures will resume in the week beginning January 12
  - I.e, Lecture 10 on January 16, 2025
  - Exercises: questions regarding the OpenMP exam

Plan for Today

- Distributed Memory
- Literature
- What is MPI?
  - `Hello World` with MPI
  - Compile
  - Execute

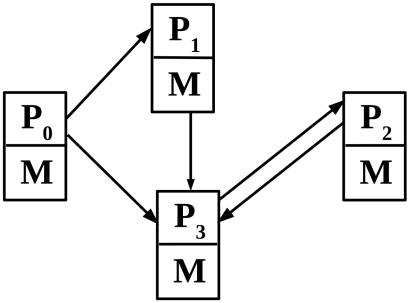


Literature—MPI

- MPI: A Message-Passing Interface Standard
  - `https://www.mpi-forum.org`
- *The same as for OpenMP:*
  - Rauber & Rünger: *Parallele Programmierung*, Springer 2012
  - Rauber & Rünger: *Parallel Programming for Multicore and Cluster Systems*, Springer, 2013
  - Hoffmann & Lienhart: *OpenMP: An Introduction to Parallel Programming with C/C++*, Springer 2008
  - Grama et al.: *Introduction to Parallel Computing*, Add-Wesley, 2004
  - Eijkhout: *The Art of HPC*, `https://theartofhpc.com`
- *And a lot more ...*

Model: Message Passing

- Also known as **two-sided communicatior**
- Multiple processes execute **in parallel**
- There is **no shared memory**
  - Thus, data distribution is required.
- Communication occurs **exclusively through sending and receiving messages**
- Sending and receiving are **explicit operations** performed by both processes involved
  - (in the classical model)



Message Passing Interface (MPI) (1/3)

- Designed primarily for **multicomputer systems**
  - But efficient implementations also exist for multiprocessors
- Current version: MPI 5.0 (June, 2025)
  - MPI 1.0 in 1994
  - MPI 4.0 in 2021
- MPI is a message-passing *standard*
  - For communication between processes
  - Defines syntax and semantic of library routines
  - Portable
  - *Not an implementation!*
- Developed by academia and industry
- `https://www.mpi-forum.org`

Message Passing Interface (MPI) (2/3)

- De-factor communication standard for parallel HPC programs
- Goals: high performance, scalability, and portability

- Data exchange must be performed by explicit message-passing
- One process is executed on one CPU / Core
- Each MPI process has its own private memory and has no direct access to the memory of the other processes

Message Passing Interface (MPI) (3/3)

- MPI only partially specifies:
  - Process startup
  - Process-to-processor mapping

- Language bindings exist for **C** (C++) and **Fortran**
- Typically, the number of processes is **fixed** at program start and remains constant during execution
- Common implementations: **Open MPI**, **MPICH**, **MVAPICH**, and others

Open MPI

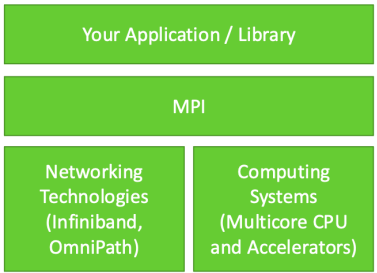
- *Open Source High Performance Computing*
- Implementation of the MPI standard
- Current Version: 5.0.7 (February 2025)
  - Full MPI-3.1; many elements from MPI-4.0
- Developed and maintained by a group of academic, research, and industry partners
- Language bindings: `C` (and `Fortran`)
- `https://www.open-mpi.org`

Other Models / Impls for Distributed Memory

- PGAS (Partitioned Global Address Space)
  - Parallel programming model
    - Assumes global memory, logically partitioned and a portion of it is local to each process
  - Library based: Global Arrays, OpenSHMEM
  - Compile based: Unified Parallel C (UPC), Co-Array Fortran (CAF)
  - Language-based: Chapel (Cray, HPE), X10 (IBM)

MPI: Getting Started

- Major MPI Features
  - Point to point Two sided Communication
  - Collective Communication
  - One-sided Communication
  - Parallel I/O
- Why?
  - Abstract message and file I/O exchange
    - Simplifies programming
  - Overlap computation and communication:
    - e.g. Non-blocking collectives and sending/receiving
  - Resiliency
    - Integrated failure detection



MPI—Hello World in C

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

int main(int argc, char** argv) {
  int world_size, world_rank;

  MPI_Init(&argc, &argv); // Start MPI
  MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get total processes
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get process ID

  printf("Hello from process %d of %d\n", world_rank, world_size);

  MPI_Finalize(); //Clean up MPI
  return 0;
}
```

MPI—Hello World in C

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

int main(int argc, char** argv) {
  int world_size, world_rank;

  MPI_Init(&argc, &argv); // Start MPI
  MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get total processes
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get process ID

  printf("Hello from process %d of %d\n", world_rank, world_size);

  MPI_Finalize(); //Clean up MPI
  return 0;
}
```

- Each process runs the same code
  - *Single Program Multiple Data (SPMP)*
- Processes are identified by ranks

MPI—Hello World in C

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

int main(int argc, char** argv) {
  int world_size, world_rank;

  MPI_Init(&argc, &argv); // Start MPI
  MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get total processes
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get process ID

  printf("Hello from process %d of %d\n", world_rank, world_size);

  MPI_Finalize(); //Clean up MPI
  return 0;
}
```

Compile: `mpicc -o hello hello.c`
Run: `mpirun -np 4 ./hello`

MPI—Hello World in C

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

int main(int argc, char** argv) {
  int world_size, world_rank;

  MPI_Init(&argc, &argv); // St
  MPI_Comm_size(MPI_COMM_WORLD,
  MPI_Comm_rank(MPI_COMM_WORLD,

  printf("Hello from process %d

  MPI_Finalize(); //Clean up MPI
  return 0;
}
```

Compile: `mpicc -o hello hello.c`
Run: `mpirun -np 4 ./hello`

Output is in non-deterministic order:

```
Hello from process 2 of 4
Hello from process 0 of 4
Hello from process 3 of 4
Hello from process 1 of 4
```

What Happens During Execution?

- Single Program, Multiple Data (SPMD)
- The program is started by an **MPI launcher**
  - E.g. `mpirun` or `srun` (old: `mpiexec`)
  - The launcher starts **multiple identical processes**
- Each process:
  - Calls `MPI_Init()` to join the global communicator
  - Receives a unique **rank ID** from 0 to $N-1$
  - Can query the total number of processes
  - Prints its own message independently
- Output order is **non-deterministic**
  - Processes run concurrently
- *No sending and receiving of messages*

MPI—Compilation and Execution

- Experimental environments:
  - Fulda HPC Cluster, LinuxLab machines, or your local setup
- **Compilation:**
  - `mpicc -O3 -Wall -o hello_mpi hello_mpi.c`
  - (for C++: use `mpiCC` or `mpic++`)
- **Interactive execution:**
  - `mpirun -n 4 ./hello_mpi`
  - Launches 4 processes that run in parallel and communicate via MPI
  - *Live Demonstration*

Basic MPI Functions

- Core MPI routines:
  - `MPI_Init`
  - `MPI_Finalize`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Send`
  - `MPI_Recv`
- All library functions, datatypes, and constants start with the prefix `MPI_` and are declared in `mpi.h`
- On success, functions return `MPI_SUCCESS`
  - Otherwise, the program will typically **abort by default**

MPI_Init and MPI_Finalize

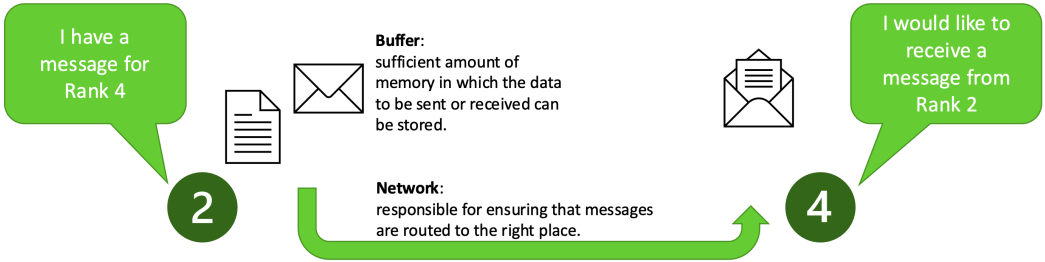
- `int MPI_Init(int *argc, char ***argv)`
  - Initializes the MPI runtime environment
  - **No** MPI functions may be called **before** `MPI_Init()`
  - The arguments correspond to the program's command-line parameters
  - `https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man3/MPI_Init.3.html`
- `int MPI_Finalize()`
  - Deregisters the calling process from the MPI runtime system
  - Does **not** terminate the program or process itself
  - After this call, **no further MPI functions** may be used
  - May block until all outstanding MPI operations are completed
  - `https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man3/MPI_Finalize.3.html`

Communicators

```
1 int MPI_Comm_rank(MPI_Comm comm, int *rank);
2 int MPI_Comm_size(MPI_Comm comm, int *size);
```

- A **communicator** (`comm`) is an internal, distributed MPI data structure that manages a set of processes
  - Each process holds a local part of this data structure, containing information about how to reach all other processes
- Within each communicator:
  - Processes are numbered starting from `0`.
  - `size` = total number of processes in the communicator.
  - `rank` = the process's own ID
- Currently, we use only: `comm = MPI_COMM_WORLD`
  - The communicator including **all processes**

Point-to-Point-Communication



Has to know:

- **Whom to send the data to?**
  Receiver's rank
- **What kind of data to send?**
  E.g., 100 integers
- **Message tag:**
  User-defined label (like an email subject)

Might have to know:

- **Who is sending?**
  `MPI_ANY_SOURCE` allowed
- **What is received?**
  E.g., up to 1000 int
- **Tag of the message?**
  `MPI_ANY_TAG` allowed

Sending Data (MPI_Send)

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

- buf = start address of the send buffer
  - Must be declared, allocated, and filled beforehand
- count = number of data elements to send
  - Measured in units of datatype)
- datatype = type of each message element
  - E.g. MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_UNSIGNED, MPI_BYTE
- dest = rank of the **destination process**
  - Must always be specified
- tag = identifier or label
  - Freely chosen by the programmer for application-specific use

Receiving Data (MPI_Recv) (1/2)

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

- buf = start address of the receive buffer
  - Must be declared and allocated before the call
- count = upper bound on the number of elements to receive
- datatype = datatype of the expected message elements
- source = rank of the sender, or MPI_ANY_SOURCE
- tag = specific tag value, or MPI_ANY_TAG

Receiving Data (MPI_Recv) (2/2)

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

- status = information about the received message:
  - status.MPI_SOURCE and status.MPI_TAG
- To retrieve the actual message length:
  - int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count);
- Alternatively: use MPI_Recv(..., MPI_STATUS_IGNORE) if message details are not needed

Matching Sending and Receiving

rank s calls:

```
MPI Send(send_buf, send_buf_size, send_type, dest, send_tag, send_comm);
```

rank q calls:

```
MPI Recv(recv_buf, recv_buf_size, recv_type, src,  recv_tag, recv_comm, &status);
```

- All 5 green parameters need to match to get message successfully through
  - All mandatory to be equal, except `recv_buf_size` $>=$ `send_buf_size`

MPI: Elementary Datatypes

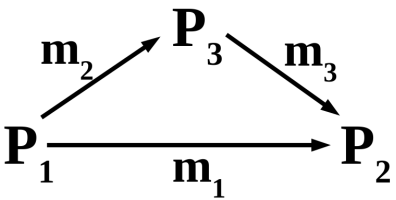
MPI_datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

- C types cannot be passed directly
  - Use **MPI datatypes** instead
- Advantage:
  - Interoperability with heterogeneous software and hardware

Conditions for Message Exchange

- Communicators must be **identical**, and all other parameters must be **compatible**
- Messages from the same sender to the same receiver are **not reordered**
- However, consider the following scenario:
  - P1: `Send m1`, then `Send m2`
  - P3: `Recv m2`, then `Send m3`
  - P2: `Recv from ANY_SOURCE`

  → **Either** `m1` **or** `m3` **may be received!**



- If matching `Send`/`Recv` operations are posted, MPI guarantees that **at least one** of them will complete (progress)
- **Fairness is not guaranteed!**

Standard Mode Communication (1/2)

- `MPI_Send` and `MPI_Recv` perform **blocking communication**:
  - After `MPI_Send` returns, the send buffer may be safely reused
  - After `MPI_Recv` returns, the receive buffer contains the rec. data
- In **standard mode**, returning from `MPI_Send` may indicate:
  - (1) The message has been delivered into the receiver's buffer, **or**
  - (2) The message has been **temporarily buffered** by the system
- Whether (1) or (2) occurs depends on:
  - The MPI implementation
  - The current system state (e.g., whether the receive is already posted, or if buffer space is available)

Standard Mode Communication (2/2)

- A program is called **safe** if its correctness does not depend on the existence or size of MPI system buffers
  - In particular, **no deadlocks** must occur regardless of buffering

Again: Hello World

```c
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int mes;
if (rank == 0) {
    mes = 42; // start value
    MPI_Send(&mes, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == size - 1) { // last rank does not forward
    MPI_Recv(&mes, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
    MPI_Recv(&mes, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    mes++; // modify payload
    MPI_Send(&mes, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
}
printf("Hello from rank %d/%d, message = %d\n", rank, size, mes);
MPI_Finalize();
return 0;
```

Rules for executing MPI programs on the cluster

- **Slurm should be used for any program execution!!!**
  - **No** local executions on the login node!
- Execute OpenMP Programs always on exact one node (64 Hardware-Threads)
- Execute MPI Programs with a maximum of 832 Processes
  - The cluster comprises 13 nodes (node-00—node-12), each with 64 physical cores (no hyperthreading!)
    $\Rightarrow$ Maximum number of cores: 832
    $\Rightarrow$ Start your MPI programs with a maximum of 832 processes
  - If you are using EduMPI, please do not exceed the maximum of 400 processes