

Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering

TGGS

KMUTNB

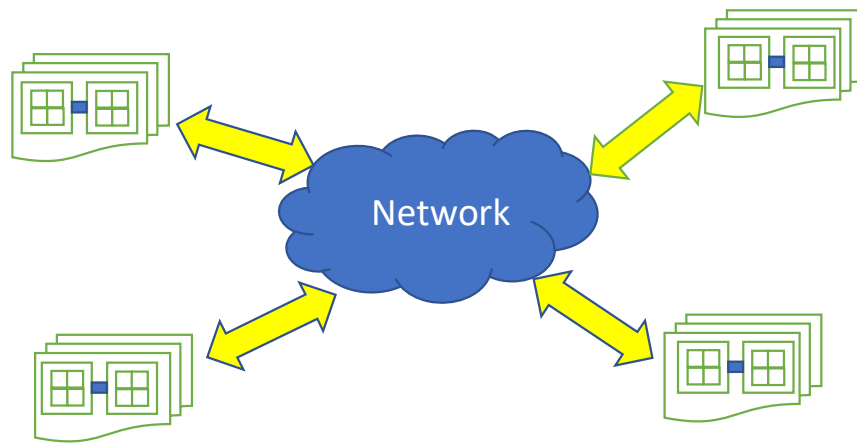
Lecture 10:

- ❑ Distributed-Memory Programming with MPI

Distributed-Memory Architectures

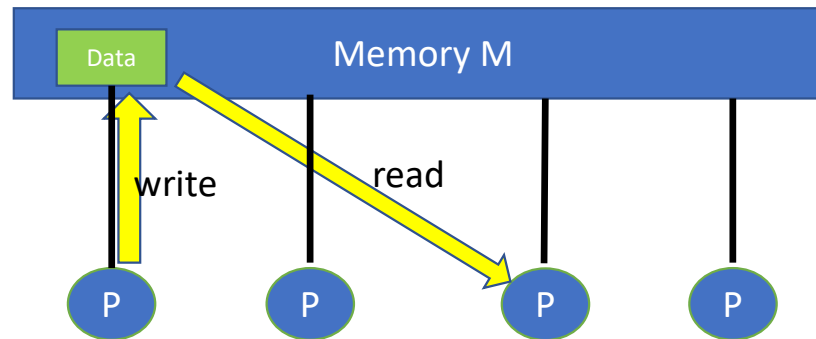
A **HPC cluster** consists of compute nodes

- each node has **no direct access to other nodes' memory**
- each node runs a separate copy of OS



Shared-Memory Programming Model

All processing elements **P** have direct access to memory **M**.

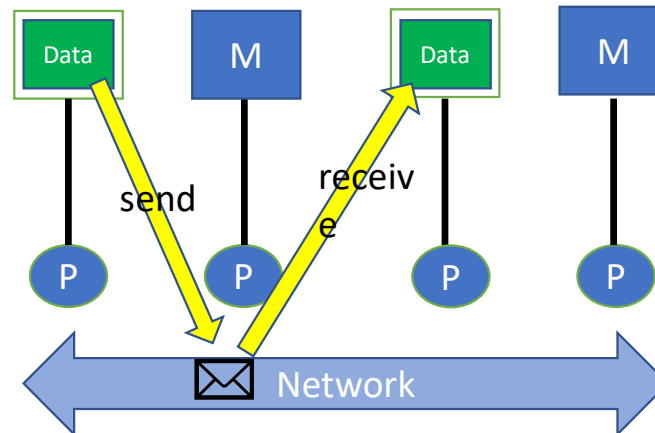


Data exchange is achieved through **read/write operations on shared variables** located in the **physical global address space**.

- Pthreads
- OpenMP

Distributed-Memory Programming Model

Each processing element **P** has its own memory module **M**.



Data exchange is achieved through **send/received operations** via a **message passing** mechanism over the network.

- Two distinct copies of the data are at the sender and the receiver because the two processing elements do not share the same physical address space.

Distributed-Memory Programming Model

Each processing element works on a separate memory module **M**.

Data exchange is achieved through **message passing**.

- Message Passing = Send/Receive operations.
- No shared variables are involved because the processing elements do not share a common virtual global address space.
- Synchronization is done implicitly via send/receive operations.

Single Program Multiple Data

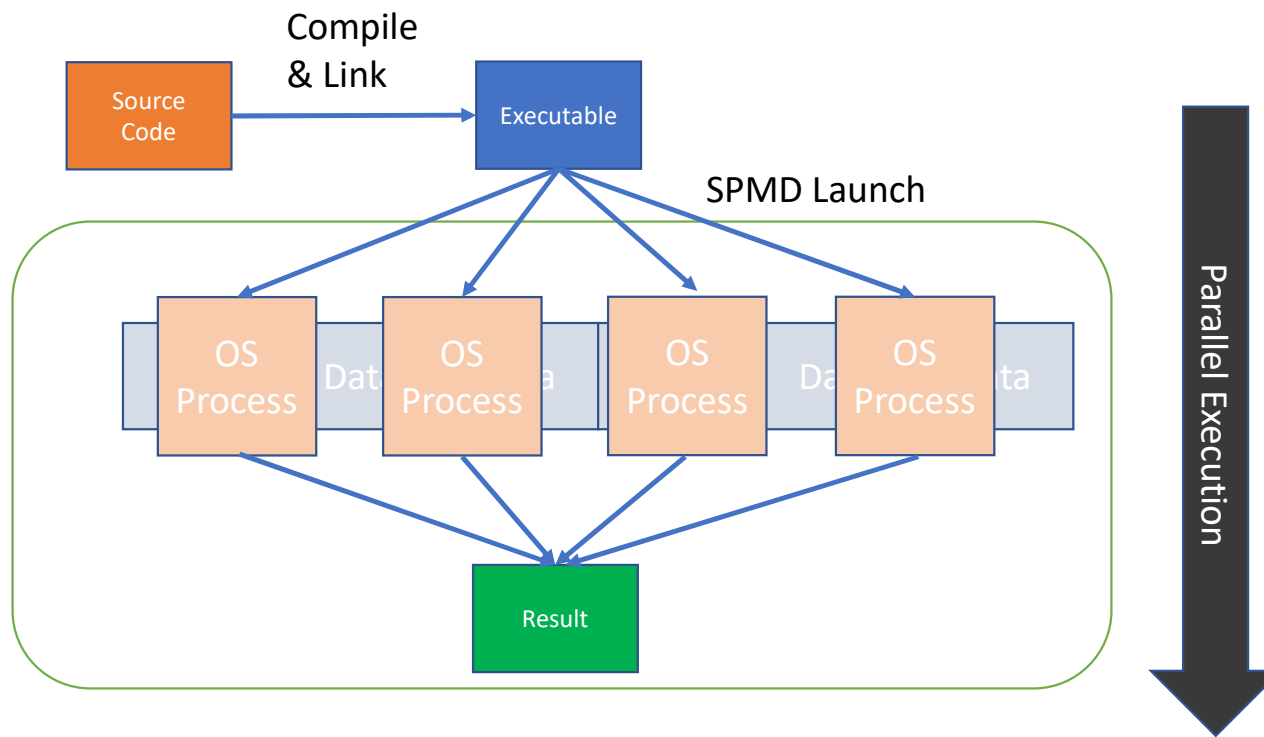
Single Program Multiple Data (SPMD)

- Single Binary Executable
- Multiple (different parts of) Data
- Each process identifies itself using a unique ID.

In an application that employs the distributed-memory programming model, each processing element has an OS process:

- Each runs the same binary executable.
- Each has its own virtual address space separated from those running on the other processing elements.
- Process use their unique IDs to steer control flow of the individual processes.

Single Program Multiple Data



Single Program Multiple Data

Requirements for SPMD environments

- Dynamic unique identification of processes
- Robust mechanisms for data exchange
 - Identity of sender and receiver
 - Amount of data
 - Type of data
 - Arrival of data
- Synchronization and communication mechanisms
 - Point-to-point communication
 - Collective communication
 - One-sided communication
- Process Launch and control mechanisms
- Portability across different platforms

Message Passing Interface (MPI)

Message Passing Interface

- The de-facto standard API for explicit message passing
- Maintained by the non-profit **Message Passing Interface Forum**
 - <https://www.mpi-forum.org>
- A number of implementations of the standard API by different vendors
 - MPICH, Intel MPI, Open MPI etc.
- Standard Bindings include
 - C and Fortran
 - Non-standard bindings for other languages also exist
 - C++, Python and Java

Message Passing Interface (MPI)

MPI is implemented as a runtime library, not a language/compiler extension.

Initialization Mechanism:

To use MPI in a program, the programmer needs to initialize the MPI runtime.

Classical MPI (pre-MPI 4.0)

- MPI_Init (with a **single thread**)
- MPI_Init_thread (with **multiple threads**)

New MPI (MPI 4.0)

- MPI_Session_init

MPI: Initialization & Finalization

Initialization and Finalization:

1. inclusion of the header file
2. Pre-initialization
 1. No MPI function calls are allowed with a few exceptions.
3. Initialization of the MPI environment
4. Parallel Computation and Communication
5. Finalization of the MPI environment
6. Post-finalization
 - No MPI function calls are allowed with a few exceptions.



A C code snippet illustrating MPI initialization and finalization. The code is enclosed in a light blue box with a red 'C' icon in the top right corner. On the left side of the code box, there are six numbered red circles (1-6) with blue brackets indicating the corresponding steps from the list on the left. The code itself is as follows:

```
1 #include <mpi.h>
2 {
3   int main(int argc, char **argv)
4   {
5     // ... some code ...
6     MPI_Init(&argc, &argv);

    // ... computation & communication ...

    MPI_Finalize();
    // ... wrap-up ...
    return 0;
  }
```

MPI: Initialization (Single-Threaded)

Initialization:

```
C:      ierr = MPI_Init(&argc, &argv);  
Fortran: CALL MPI_Init(ierr)
```

- **MPI_Init** initializes the **MPI runtime environment** and makes the calling process a member of MPI_COMM_WORLD.
 - Both arguments can be NULL.
 - An error code is returned (for the C binding)
- **MPI_Init** must be called once during the program execution.

MPI: Finalization

Finalization:

```
C:      ierr = MPI_Finalize();  
Fortran: CALL MPI_Finalize(ierr)
```

- **MPI_Finalize** cleans up and terminates the **MPI runtime environment**.
- **MPI_Finalize** must be called once before the process terminates.
 - It is not recommended to have other code after the call.

MPI: General Structure of an MPI Program

How many MPI processes are there?

- The calling process can find out how many processes there are in the MPI program with **MPI_Comm_size**.

Who am I?

- The calling process can find out its identity or **rank** with **MPI_Comm_rank**.
- Ranks are numbered **starting from zero**.

```
#include <mpi.h>

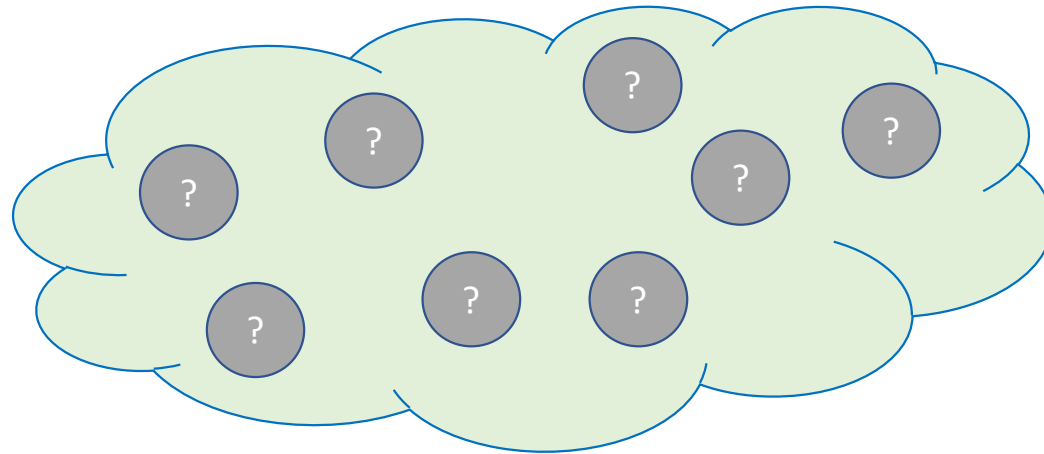
int main(int argc, char **argv)
{
    // ... some code ...
    int ierr = MPI_Init(&argc, &argv);
    int numberOfProcs, rank;
    // ... more code ...
    1 ierr = MPI_Comm_size(MPI_COMM_WORLD,
                          &numberOfProcs);
    2 ierr = MPI_Comm_rank(MPI_COMM_WORLD,
                          &rank);
    // ... computation & communication ...
    ierr = MPI_Finalize();
    // ... wrap-up ...
    return 0;
}
```

MPI: Rank

The processes in an MPI program are initially indistinguishable.

MPI assigns each process a unique identity (**rank**) in a communication context (**communicator**).

- The **initial communication context** is **MPI_COMM_WORLD**, which contains all the MPI processes within the MPI program.

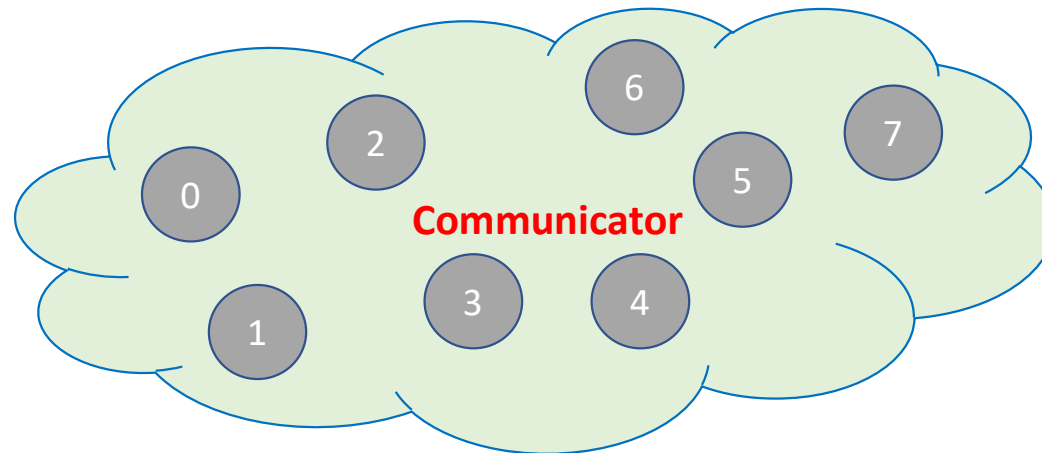


MPI: Rank

The processes in an MPI program are initially indistinguishable.

MPI assigns each process a unique identity (**rank**) in a communication context (**communicator**).

- The **initial communication context** is **MPI_COMM_WORLD**, which contains all the MPI processes within the MPI program.



MPI: Rank and Communicator

Rank:

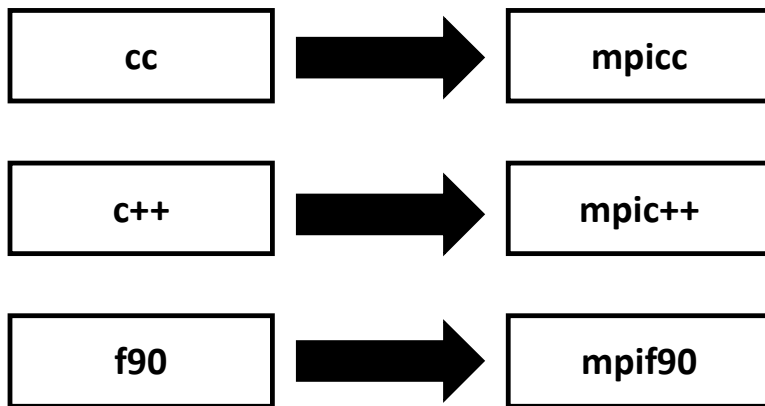
- Ranks range from 0 to n-1.
where n is the number of MPI processes in the communication context (communicator).
- MPI processes within an MPI program can have **different ranks in different communicators**.

Communicator:

- A communicator is **a logical context in which communication occurs**.
 - a group of processes together with additional information.
- The initial communicator **MPI_COMM_WORLD** is implicitly available in every MPI program.
 - a group of all processes the MPI runtime environment is initially launched with at startup.

MPI: Compiling an MPI Program

- MPI is a library with C header files.
- Most MPI vendors provide compiler wrappers that the programmer can conveniently use.

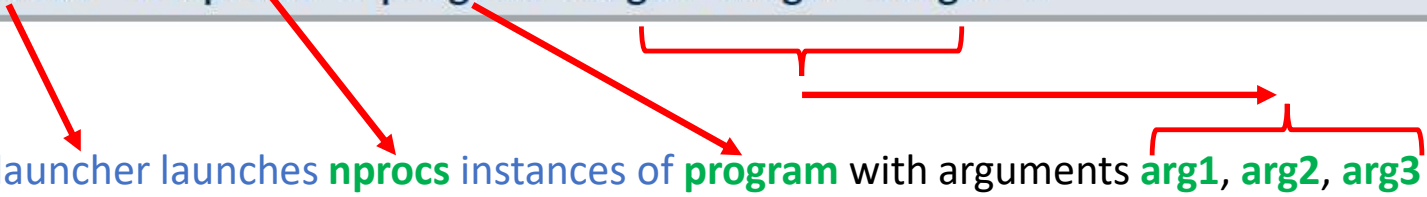


Note that names are not standardized.

MPI: Executing an MPI Program

Most implementations provide a **special launcher program mpiexec**.

```
mpiexec -n nprocs ... program <arg1> <arg2> <arg3> ...
```



- The launcher launches **nprocs** instances of **program** with arguments **arg1, arg2, arg3** etc.

The MPI standard specifies the **mpiexec** launcher program, but does not require it.

MPI: Executing an MPI Program

The launcher performs the following procedure at startup:

- help MPI processes find each other and establish the **MPI_COMM_WORLD** communicator
- redirect the standard output of all processes to the terminal
- redirect the terminal input to the standard input of rank 0
- forward received signals

MPI: Error Handling

Error codes indicate the success of MPI calls:

- Failure is indicated by an error code other than **MPI_SUCCESS**.

A code snippet in a light blue box with a dark blue 'C' icon in the top right corner. The code is in C and shows an if-statement checking if MPI_Init returns a value other than MPI_SUCCESS. If it does, there is an ellipsis indicating further code.

```
if (MPI_SUCCESS != MPI_Init(&argc, &argv))  
    ...
```

- An MPI error handler is called first before the call returns.
 - The default error handler for **non-IO calls** abort the entire MPI program.
- The values of error codes are **implementation-dependent**.
 - **MPI_Error_string** can be used decode error codes into human-readable information

MPI: Handles to MPI Opaque Objects

MPI objects such as communicators are referenced through **handles**.

- Process-local values cannot be passed from one process to another.
- Objects are referenced by handles are **opaques**.
 - Implementation-dependent
 - Blackbox to the user

Examples of handles in MPI are **MPI_Comm**, **MPI_Datatype**, **MPI_Request** etc.

MPI: Datatype Handle

MPI cannot automatically deduce the data types of supplied buffers at runtime.

- The program must provide additional information on the data types.

MPI datatype handles tell the MPI runtime environment how to:

- read binary values from the receive buffer
- write binary values into the send buffer
- correctly apply value alignments
- convert values between different machine representations in heterogeneous environments

MPI: Datatype Handle

MPI datatypes are **handles**.

- They cannot be used to declare variables of a specific language type.
- **sizeof(MPI_INT)** returns the size of the datatype handle, **not** the size of an int in the C programming language.

Basic datatypes corresponds to **native datatypes** in the programming language.

| MPI data type | C native data type |
|------------------|--------------------|
| MPI_CHAR | char |
| MPI_SHORT | short |
| MPI_INT | int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_UNSIGNED_INT | unsigned int |
| ... | ... |
| MPI_BYTE | |

No binary conversion is used for untyped data.

MPI: Local and Non-Local Operations

A **Local** operation requires

- no communication with another MPI process.
- Its completion depends only on the local operations.

A **Non-local** operation requires

- some specific, semantically MPI-related procedure to be called on another MPI process.
- Such an operation requires communication with another MPI process.

MPI: MPI Operations

MPI defines several operations that are **a sequence of steps** performed by the MPI runtime environment to establish and enable

- data transfer
- data synchronization

Four stages:

1. Initialization:
 - Resources (e.g. buffer address, arguments) are passed to the MPI runtime environment.
2. Starting:
 - The operation takes over control of the resources (e.g., buffer contents).
3. Completion:
 - The operation returns control of the resources (e.g., buffer contents).
4. Freeing:
 - The operation returns control of the remaining resources.

MPI: Blocking and Non-Blocking

A **blocking** procedure can return only when the associated operation completes locally.

- Upon completion, any supplied input arguments (e.g., supplied buffer) can be safely reused or deallocated.
 - For example, a blocking send operation **does not mean** the message must have been successfully delivered to the destination before the send operation can complete. (We will see more about this)

A **non-blocking** procedure may return before the associated operation completes locally.

- One or more additional MPI calls are required to complete the operation.
- The supplied input arguments are not allowed to be reused or deallocated until the operation completes.

MPI: Synchronous and Asynchronous

A **synchronous** procedure blocks the calling process until the operation completes.

- A synchronous procedure completes locally only with specific remote intervention.
 - In the case of a send-and-receive communication, both the sender and the receiver complete when the receiver has started to receive the message.

An **asynchronous** procedure may complete locally without remote intervention.

Note that synchronous/asynchronous implies blocking/non-blocking, but not vice versa.

References

- [1] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.
- [2] Marc-Andre Hermanns. 2021. *MPI in Small Bites*. PPCES 2021.