Introduction to Parallel Programming with MPI

Lecture #1*: Introduction*

Andrea Mignone

Academic Year: 2022-2023

Dipartimento di Fisica
Turin University, Torino (TO)

Course Requisites

- In order to follow these lecture notes and the course material you will need to have some acquaintance with
  - Linux shell
  - C / C++ or Fortran compiler
  - Basic knowledge of numerical methods

- Further Reading & Links:
  - The latest reference of the MPI standard: https://www.mpi-forum.org/docs/
    Huge - but indispensible to understand what lies beyond the surface of the API
  - Online tutorials:
    - https://mpitutorial.com (by Wes Kendall)
    - https://www.codingame.com/playgrounds/349/introduction-to-mpi/introduction-to-distributed-computing
    - http://adl.stanford.edu/cme342/Lecture_Notes.html

The Need for Parallel Computing

- Memory- and CPU-intensive computations can be carried out using parallelism.
- Parallel programming methods on parallel computers provides access to increased memory and CPU resources not available on serial computers. This allows large problems to be solved with greater speed or not even feasible when compared to the typical execution time on a single processor.
- Serial application (codes) can be turned into parallel ones by fulfilling some requirements which are typically hardware-dependent.
- Parallel programming paradigms rely on the usage of message passing libraries. These libraries manage transfer of data between instances of a parallel program unit on multiple processors in a parallel computing architecture.

Parallel Programming Models

- Serial applications will not run automatically on parallel architectures (no such thing as automatic parallelism !).
- Any parallel programming model must specify how parallelism is achieved through a set of program abstractions for fitting parallel activities from the application to the underlying parallel hardware.
- It spans over different layers: applications, programming languages, compilers, libraries, network communication, and I/O systems.

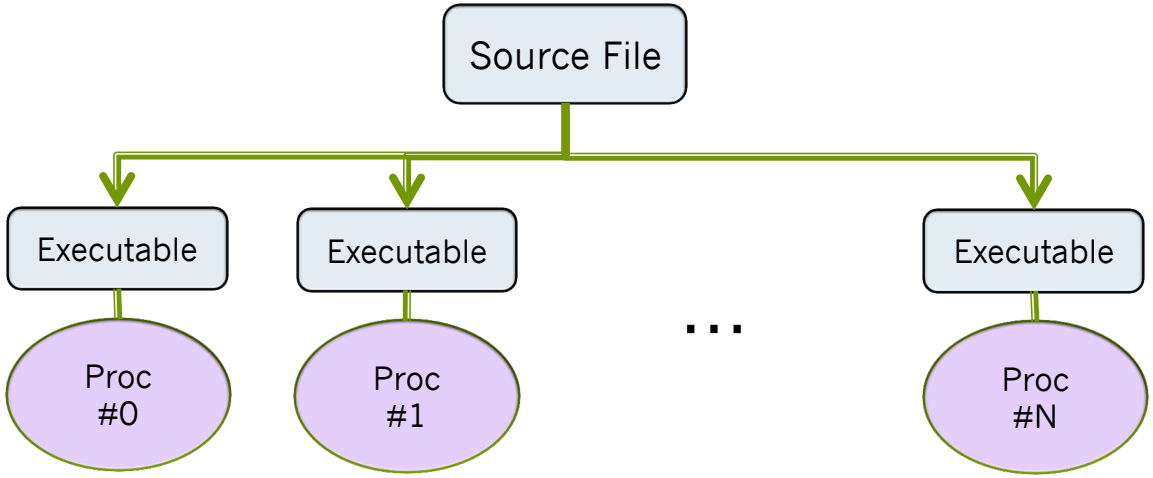
- **Flynn's taxonomy**: a classification (proposed by M.J.Flynn in 1966) based on number of simultaneous *instruction* and *data streams* seen by the processor during program execution.
- **Flynn's taxonomy** is employed as a tool in the design of modern processors and their functionality.

Flynn's Taxonomy

- SISD (Single Instruction, Single Data): a sequential computer which exploits no parallelism in either the instruction or data streams;
- SIMD (Single Instruction, Multiple Data): processes execute the same instruction (or operation) on a different data elements.
  - Example: an application where the same value is added (or subtracted) from a large number of data points (e.g. multimedia applications).
  - Advantage: processing multiple data elements at the same time with a single instruction can noticeably improve the performance.
  - Employed by vector computers.
- MISD (Multiple Instruction, Single Data): multiple instructions operate on one data stream (uncommon architecture);
- MIMD (Multiple Instruction, Multiple Data): at any time, different processes execute different instructions on different portions of data:
  - Single Program, Multiple Data (SPMD)
  - Multiple Programs, Multiple Data (MPMD)

SPMD Parallelism

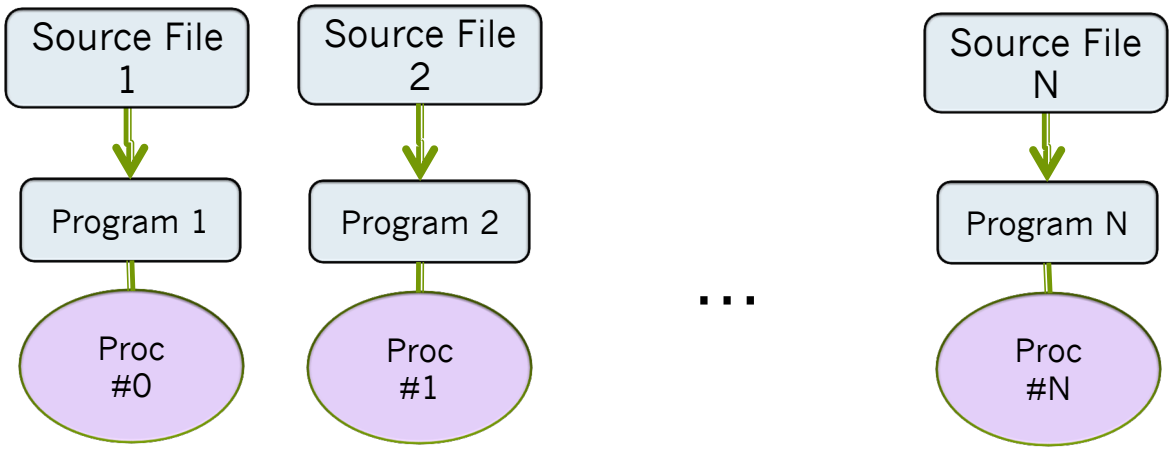
- The majority of MPI programs is based on the *Single-Program-Multiple-Data* (SPMD) paradigm;
- In this model, all processors run simultaneously a copy of the same program;



- However, each process works on a separate copy of the data;
- Processes can follow different control paths during the execution, depending of the process ID.

MPMD Parallelism

- In the Multiple programs, multiple data (MPMD) parallelism, each task can execute a different program:

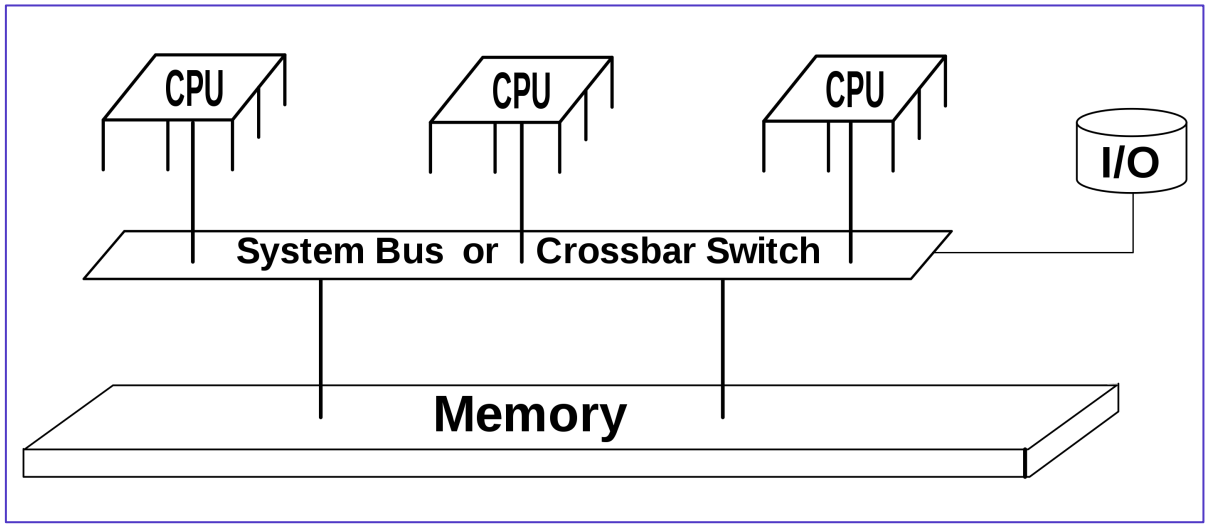


Parallel Programming Models

- By far, *SIMD* and *SPMD* are the most dominant parallel models.
- In the course we will be concerned with **SPMD** only.
- SPMD programs can exploit two different memory models:
  - Shared memory;
  - Distributed memory.
- The latest generation of parallel computers now uses a mixed shared/distributed memory architecture. Each node consists of a group of 2 to 16 (or more) processors connected via local, shared memory and the multiprocessor nodes are, in turn, connected via a high-speed communications fabric.

Parallel Architectures: Shared Memory

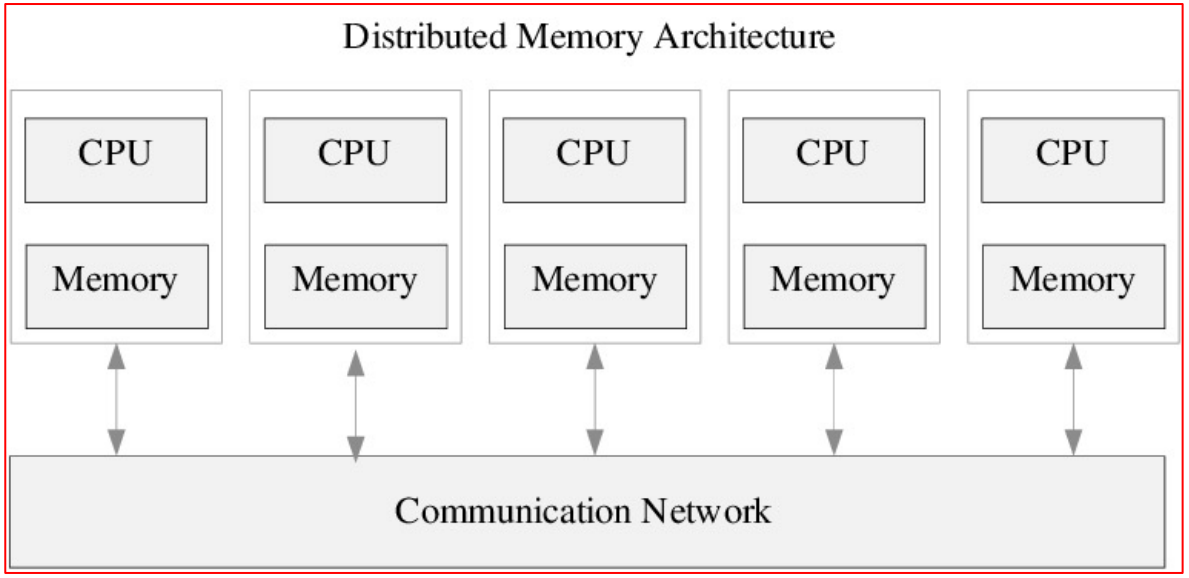
- In a shared memory computer, multiple processors share access to a global memory space via a high-speed memory bus.



- This global memory space allows the processors to efficiently exchange or share access to data.
- Typically, the number of processors used in shared memory architectures is limited to only a handful (2 - 16) of processors. This is because the amount of data that can be processed is limited by the bandwidth of the memory bus connecting the processors.

Parallel Architectures: Distributed Memory

- Distributed memory parallel computers are essentially a collection of serial computers (nodes) working together to solve a problem.



- Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network, usually a proprietary high-speed communications network.
- Data are exchanged between nodes as messages over the network.

The Message Passing Interface (MPI)

- The *Message Passing Interface* (MPI) is a standardized, vendor-independent and portable message-passing *library* defining syntax and semantic standards of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.

- MPI has over 40 participating organizations, including vendors, researchers, software library developers, and users.

- The goal of MPI is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs.

- MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.

What MPI is NOT

- MPI is *not* a programming language; but instead a realization of a computer model.

- It's not a new way of parallel programming (rather a realization of the old message passing paradigm that was around before as POSIX sockets)

- It's not automatically parallelizing code (rather, the programmer gets full manual control over all communications) ;

Downloading & Installing MPI

Two common implementations of MPI are:

- MPICH (http://www.mpich.org - recommended) is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. MPICH is distributed as source (with an open-source, freely available license). It has been tested on several platforms, including Linux (on IA32 and x86-64), Mac OS/X (PowerPC and Intel), Solaris (32- and 64-bit), and Windows.

- The Open MPI Project (https://www.open-mpi.org) is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners.

MPICH is supposed to be high-quality reference implementation of the latest MPI standard and the basis for derivative implementations to meet special purpose needs. Open-MPI targets the common case, both in terms of usage and network conduits.

Running MPI on a single CPU

- Modern laptop computer are equipped with more than one core (typically 2-6). In this case you can fully exploit the MPI library and achieve performance gain.

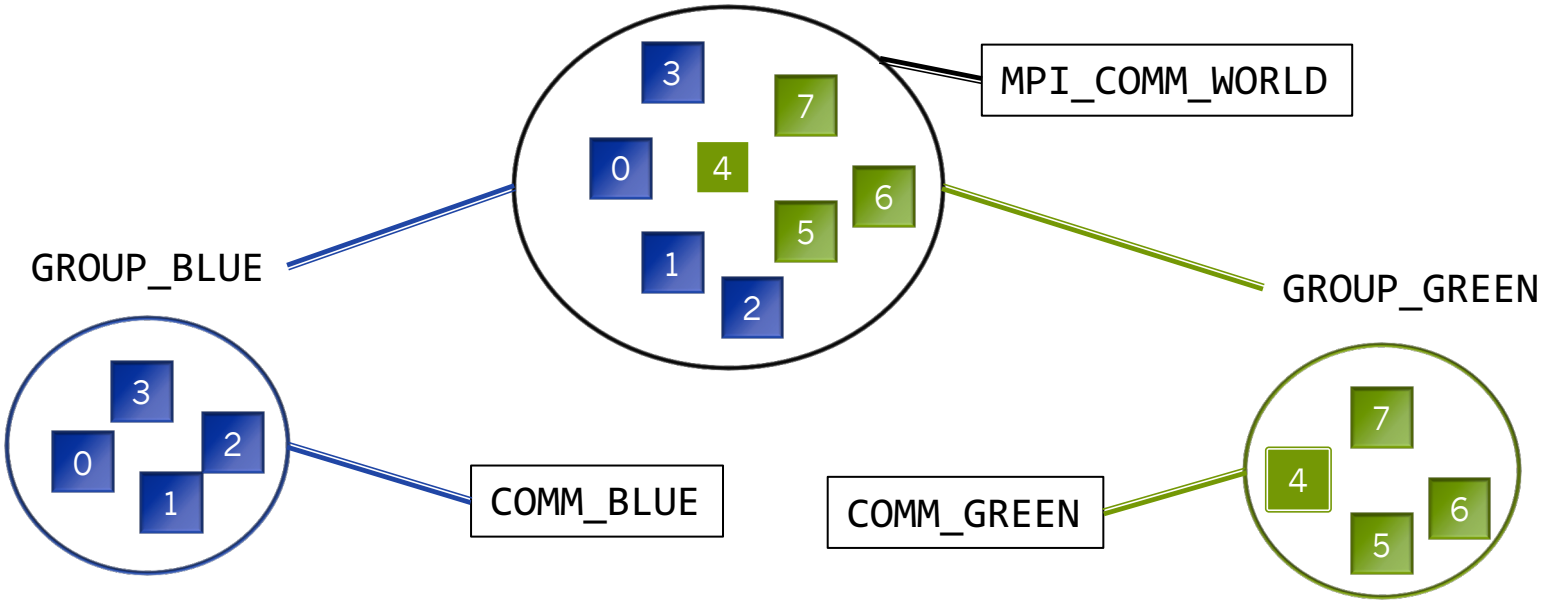
- If you have multiple cores, each process will run on a separate core.

- If you ask for more processes than the available core CPUs, everything will run, but with a lower efficiency. MPI creates virtual “processes” in this case. So, if you have a single CPU single-core machine, you can still use MPI but (yes, you can run multi-process jobs on a single-cpu single-core machine…)

Writing an MPI Program

MPI Group & Communicators

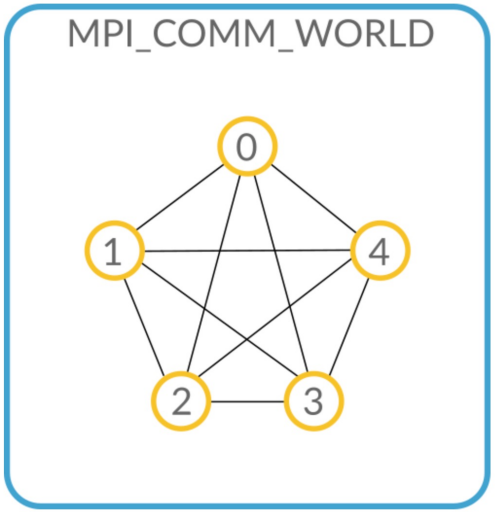
- An MPI *group* is a fixed, ordered set of unique MPI processes. In other words, an MPI group is a collection of processes, e.g.



- A *communicator* represents a process' membership in a larger process group. It holds a group of processes that can communicate with each other.
- You can see a communicator as a box grouping processes together, allowing them to communicate.
- First the group is created with the desired processes to be included within the group and then the group is used to create a communicator.

The Default Communicator

- The default communicator is called `MPI_COMM_WORLD`.
- It basically groups all the processes when the program starts.
- The diagram shows a program which runs with five processes. Every process is connected and can communicate inside `MPI_COMM_WORLD`.
- The **size** of a communicator does not change once it is created. Each process inside a communicator has a unique number to identify it: **rank** of the process.
- In the previous example, the **size** of `MPI_COMM_WORLD` is **5**. The **rank** of each process is the number inside each circle. The rank of a process always ranges from
- `MPI_COMM_WORLD` is not the only communicator in MPI. Custom communicators may also be created.



Example #1: our 1st MPI Program

- `[program1.c]` Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required

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

int main(int argc, char ** argv)
{
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("I am rank # %d of %d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

Include MPI library header file

Initialize the MPI execution environment

Obtain the rank and size of the communicator

Terminate MPI execution environment

Example #1: our 1st MPI Program

- Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required
- For a serial application we normally compile and run the code

```
> gcc my_program.c –o my_program    # Compile the code
> ./my_program                      # run on a single processor
```

- For a parallel application using MPI:

```
> mpicc my_program.c –o my_program    # Compile the code using MPI C compiler
> mpirun –np 4 ./my_program           # run on a 4 processors
```

- The result of the execution should be

```
I am rank # 0 of 4
I am rank # 1 of 4
I am rank # 3 of 4
I am rank # 2 of 4
```

Example #2: Multiplication Tables

- [mult_tables.c] Suppose we now want different processors to do the multiplication table of 1, 2, 3, and 4:

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

int main(int argc, char ** argv)
{
  int i, rank, size;

/* -- Initialize MPI environment -- */

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* -- Create multiplication table  -- */

  printf ("[Rank # %d]\n", rank);
  for (i = 1; i <= 10; i++){
    printf ("  %d\n", i*(rank+1));
  }

  MPI_Finalize();
  return 0;

}
```

- Note that each processor will create a different table, depending on its rank. So rank #0 will print 1,2,3,... while rank #1 will do 2, 4, 6, 8, ... and so on;

- The output may be chaotic and not predictable since all processor try to access the output stream (stdio) concurrently.

Example #3: solving an Initial Value ODE

- `[multi_ode.c]` We now wish to solve the pendulum 2nd order ordinary differential equation

$$\frac{d^2\theta}{dt^2} = -\sin\theta$$

for 0 < t < 10, using different initial conditions at t=0:

$$\theta_0 = \theta(0) = k\frac{\pi}{15} \quad (\text{with} \quad \mathrm{k} = 1, 2, 3, 4) \qquad \dot{\theta}_0 = 0$$

- Using 4 processes we can have each task solving the same equation using a different initial condition.
- A 2nd –order Runge Kutta scheme with $\Delta t = 0.1$ can be used. The code will be the same, however:
  - Make sure the initial condition is assigned based on the rank
  - The output data file should be different for each processes.

Example #3: solving an Initial Value ODE

- We cast the 2nd –order ODE as a system of two 1st –order ODE:

$$\begin{cases} \dfrac{d\theta}{dt} = \omega \\ \dfrac{d\omega}{dt} = -\sin\theta \end{cases}$$

- The RK2 method can be illustrated as follows

$$\begin{aligned} Y^{n+1/2} &= Y^n + \tfrac{\Delta t}{2} R(Y^n) \\ Y^{n+1} &= Y^n + \Delta t R(Y^{n+1/2}) \end{aligned}$$

- Here **Y** and **R** are 2-element arrays containing the unknowns and the right hand sides for the two 1st –order ODE:

$$Y = [\theta, \dot{\theta}] \equiv [\theta, \omega]$$

$$R = [\omega, -\sin\theta]$$

- The output file should be a multi-column text file containing

.	.	.
t^n	θ^n	ω^n
.	.	.

Visualizing Data

- We can now plot the 4 solutions of the ODE using, e.g., gnuplot.