High Performance Computing

Lecture 05: Message Passing programming
Using MPI
Part A: 10-10-2023

“” **High Performance Computing**

2023-2024 Stefano Cozzini

Agenda

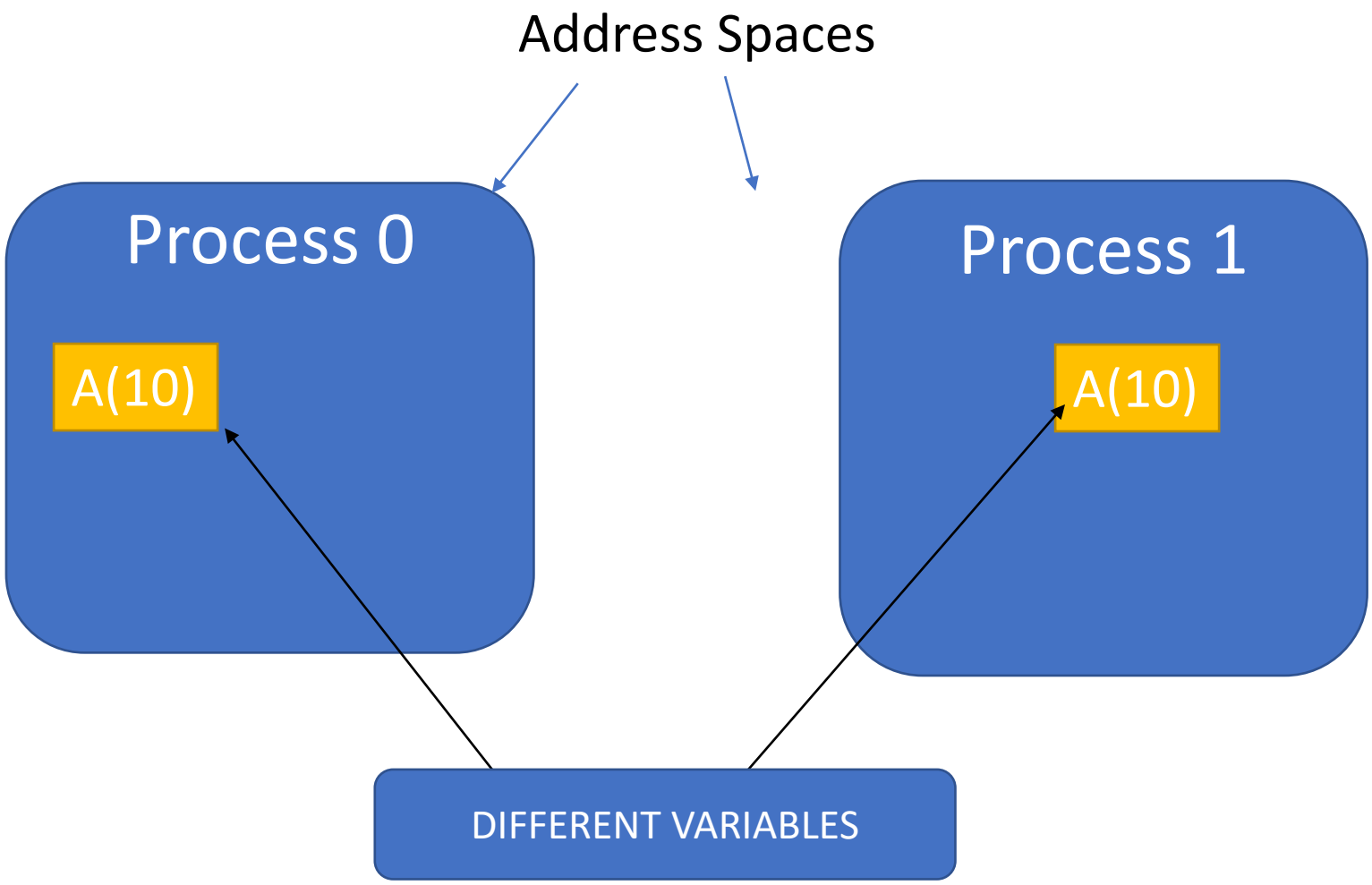
- Recap:
  - Message Passing Paradigm
- Basic on Message Passing Interface

- Point-to-Points operation
- Collective operations

Message Passing paradigm

- Parallel programs consist of separate processes, each with its own address space
- Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
- Programmer manages
  - Memory motion
  - Data distribution

Shared nothing approach



Message Passing Pro&Cons

- Pros
- Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately.
- Cons
- Data is scattered on separated address spaces
- The programmer is responsible for many of the details associated with data communication between processors.
- Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

Message Passing approach= MPI

- MPI is a “standard by consensus,” originally designed in an open forum that included hardware vendors, researchers, academics, software library developers, and users, representing over 40 organizations.
- This broad participation in its development ensured MPI’s rapid emergence as a widely used standard for writing message-passing programs.
- MPI is not a true standard; that is, it was not issued by a standards organization such as ANSI or ISO.

What is MPI ?

A STANDARD...

- The actual implementation of the standard is demanded to the software developers of the different systems
- Standard is discussed at the www.mpi-forum.org
- In all systems MPI has been implemented as a library of subroutines over the network drivers and primitives
- many different implementations
  - MPICH (the original one)
  - OpenMPI
  - IntelMPI

MPI evolution

- MPI “standard” was introduced by the MPI Forum in May, 1994 and updated in June, 1995.
- MPI-2.0 was completed in 1997,
- MPI-3.0 was finalized in 2012.
- MPI-3.1 was finalized in 2015, which provides small additions and fixes to MPI-3.0.
- MPI 3 is still the most widely used standard (implanted in both Intel and openMPI
- MPI-4.0 was released on June 9, 2021.
- Current efforts focus on MPI 4.1 and MPI 5.0.

Some reasons to use MPI

- INTERNATIONAL STANDARD
- MPI evolves: MPI 1.0 was first introduced in 1994, most current version is MPI 4.0 (June 2021)
- Available on almost all parallel systems (free MPICH, Open MPI used on many clusters), with interfaces for C/C++ and Fortran
- Supplies many communication variations and optimized functions for a wide range of needs
- Works both on distributed memory (DM) and shared memory (SM) hardware architectures
- Supports large program development and integration of multiple modules

How to program with MPI ?

- MPI is a library:
  - All operations are performed with subroutine calls
- Basic definitions are in
  - mpi.h for C/C++
  - mpif.h for Fortran 77 and 90
  - MPI module for Fortran 90 (optional)

How to compile MPI Programs

- NO STANDARD: left to the implementations:
- Generally:
  - You should specify the appropriate include directory (i.e. -I/mpidir/include)
  - You should specify the mpi library (i.e. -L/mpidir/lib -lmpi)
- Usually MPI compiler wrappers do this job for you. (i.e. mpicc)

How to run MPI programs ?

- The MPI Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- Many implementations provided `mpirun –np 4 a.out` to run an MPI program
- The specific commands to run a program on a parallel system are defined by the environment installed on the parallel computer

How to write an MPI program ?

- Modify your serial program to insert MPI routines which distribute data and loads to different processors.

WHICH MPI ROUTINES DO I NEED ?

Basic Features of MPI routines

- Calls may be roughly divided into four classes:
  - Calls used to initialize, manage, and terminate communications
  - Calls used to communicate between pairs of processors. (Pair communication)
  - Calls used to communicate among groups of processors. (Collective communication)
  - Calls to create data types.

Minimal set of MPI routines

- MPI_INIT: initialize MPI
- MPI_COMM_SIZE: how many Processors?
- MPI_COMM_RANK: identify the Processor
- MPI_SEND : send data
- MPI_RECV: receive data
- MPI_FINALIZE: close MPI

(Almost) All you need is to know this 6 calls

Our first program

Fortran

```
PROGRAM hello
 INCLUDE ‘mpif.h‘
 INTEGER err
 CALL MPI_INIT(err)
 call  MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
 call  MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
 print *, 'I am ', rank, ' of ', size
 CALL MPI_FINALIZE(err)
END
```

C

```
#include <stdio.h>
#include <mpi.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 %d of %d\n", rank, size );
  MPI_Finalize();
```

Some notes/observations

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI “job”
- Each statement executes independently in each process including the printf/print statements
- I/O not part of MPI-1 (MPI-IO part of MPI-2)
- print and write to standard output or error not part of MPI-1 or MPI-2 or MPI-3
- output order is undefined (may be interleaved by character, line, or blocks of characters),
- A consequence of the requirement that non-MPI statements execute independently

Type signatures

- All MPI routines have some similarities in their naming and in the parameters that they require. However, there are differences between C and Fortran, so let's look at these separately.
- C bindings
  - For C, the general type signature is

    rc = MPI_Xxxxx(parameter, ...)
  - Note that case is important here, as it is with all C code. For example, MPI must be capitalized, as must the first character after the underscore. Everything after that must be lower case. All C MPI functions return an integer return code, which can be used to determine if the call succeeded.
  - If rc == MPI_SUCCESS, then the call was successful.
- C programs should include the file "mpi.h". This contains definitions for MPI functions and constants like MPI_SUCCESS.

- Fortran bindings
  - For Fortran, the general type signature is

    Call MPI_XXXXX(parameter,..., ierror)
  - Note that case is not important here. So, an equivalent form would be

    call mpi_xxxxx(parameter,..., ierror)
- Instead of having a function that returns an error code, as in C, the Fortran versions of MPI calls are subroutines that usually have one additional parameter in the argument list, ierror, which is the return code. Upon success, ierror is set to MPI_SUCCESS.

Inizialing/Finalizing MPI program

- Initializing the MPI environment
  - C:

```
int MPI_Init(int *argc, char ***argv);
```

  - Fortran:

```
INTEGER IERR
CALL MPI_INIT(IERR)
```

- Finalizing MPI environment
  - C:

```
int MPI_Finalize()
```

  - Fortran:

```
INTEGER IERR
CALL MPI_FINALIZE(IERR)
```

This two subprograms should be called by all processes, and no other MPI calls are allowed before **`mpi_init`** and after **`mpi_finalize`**

MPI Communicator

- The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.
- It identifies the group of all processes.
- All MPI communication subroutines have a communicator argument. The Programmer could define many communicators at the same time

There is a default communicator (automatically defined):

MPI_COMM_WORLD

Communicator size and processor rank

How many processors are associated with a communicator?

C:

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

Fortran:

```
INTEGER COMM, SIZE, IERR
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
OUTPUT:  SIZE
```

What is the ID of a processor in a group?

C:

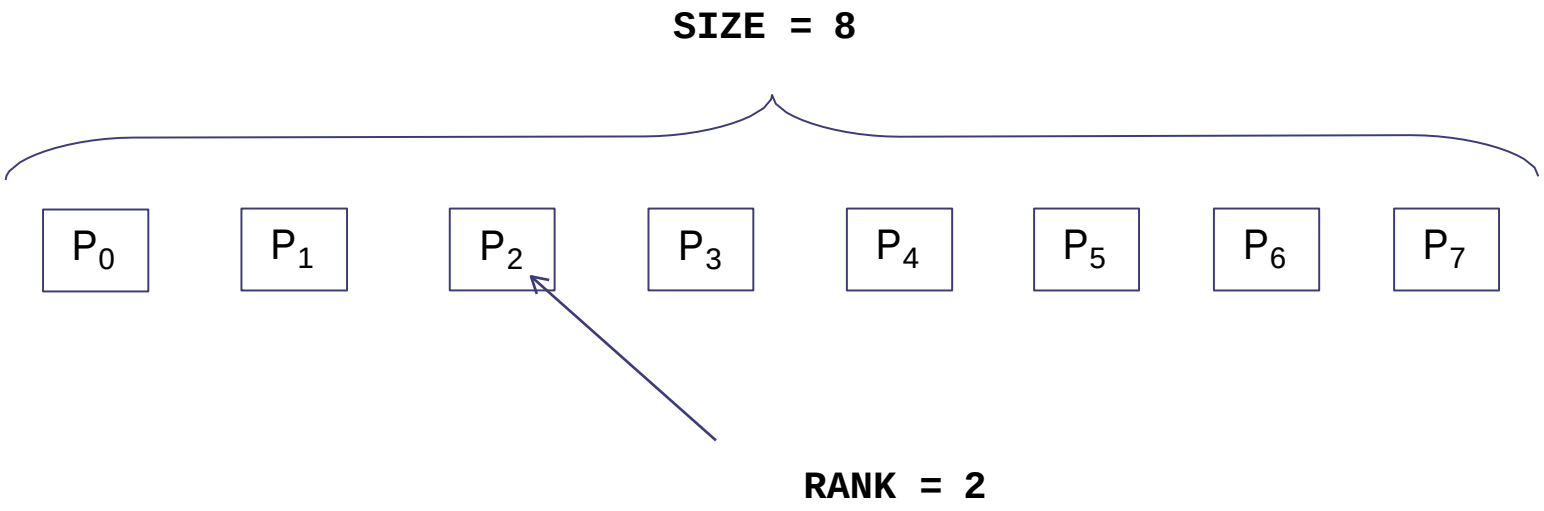
```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

```
INTEGER COMM, RANK, IERR
CALL MPI_COMM_RANK(COMM, RANK, IERR)

OUTPUT:  RANK
```

Communicator size and processor rank



Size is the number of processors associated to the communicator

rank is the index of the process within a group associated to a communicator (**rank** = 0,1,...,N-1). The rank is used to identify the source and destination process in a communication