Carnegie Mellon University Qatar

Distributed Systems

15-440 - Fall 2019

Recitation 7 Handount

1 Intended Learning Outcome (ILO)

The ILO of this recitation is:

- Apply parallel programs according to the MPI standard.

2 Objectives

The ILO of this recitation is:

- Provide students with a brief overview on the MPI programming model.

- Help students develop and run MPI parallel programs.

3 MPI Overview

- What is MPI?

  - MPI = Message Passing Interface.
  - It's a library of routines that can be used to create
  - Major goal = Portability

- Reasons for using MPI

  - Standardization
  - Portability
  - Performance
  - Functionality
  - Availability

- Main Features

  - Can run simultaneously on different architectures
  - Independent and concurrent execution of programs
  - Each processor has private memory and address space
  - Deadlocks can occur

- Programming Model

  - With MPI all parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
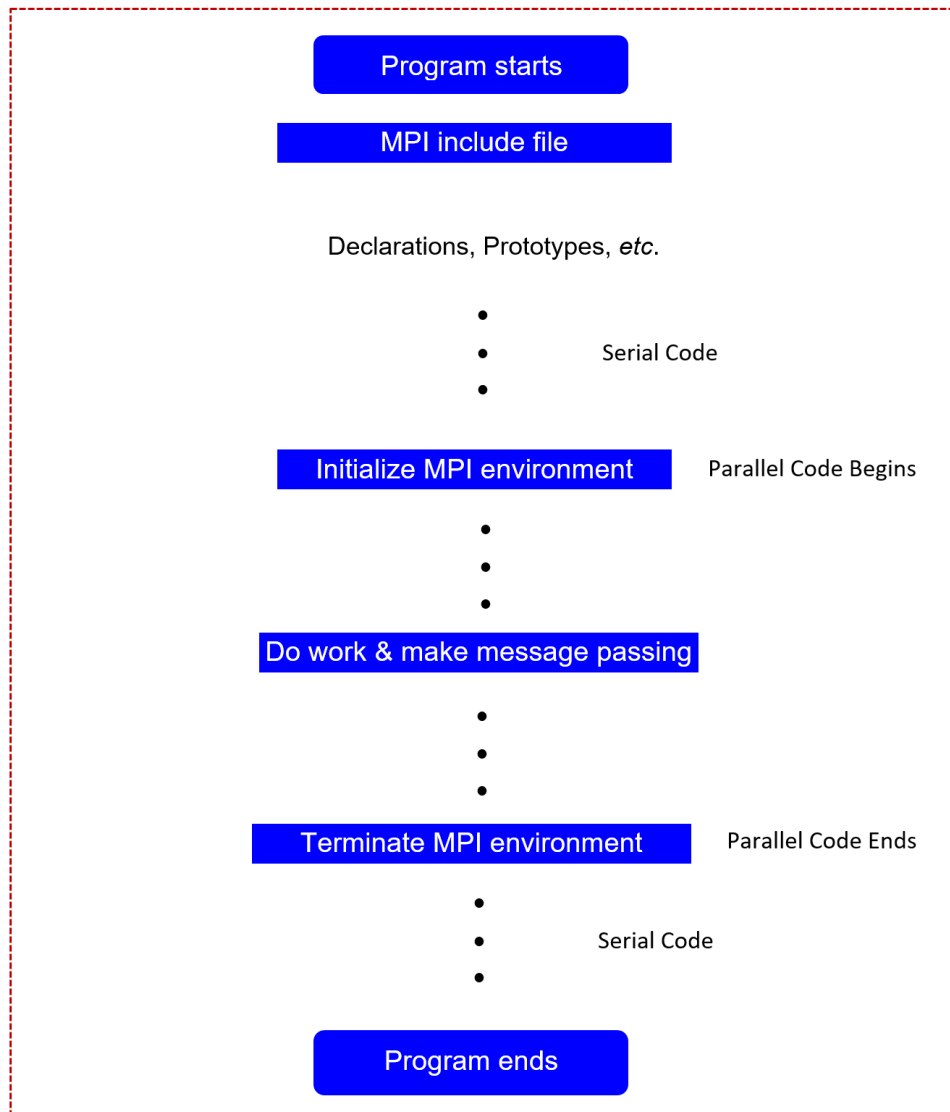
- Fundamentals
  - **Communicators and groups**
    - ∗ MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
    - ∗ Most MPI routines require you to specify a communicator as an argument.
    - ∗ Communicators and groups will be covered in more detail in the class lectures. For now, simply use `MPI_COMM_WORLD` whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.
  - **Rank**
    - ∗ Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.
    - ∗ Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if `rank=0` do this / if `rank=1` do that).
  - **General MPI Program Structure**

Program starts

MPI include file

Declarations, Prototypes, *etc*.

· · · Serial Code

Initialize MPI environment — Parallel Code Begins

· · ·

Do work & make message passing

· · ·

Terminate MPI environment — Parallel Code Ends

· · · Serial Code

Program ends

- Some Frequently Used MPI Routines

  1. `MPI_Init(int *argc, char ***argv)`

     This routine initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

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

     This routine determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application.

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

     This routine determines the rank of the calling process within the communicator.

  4. `MPI_Wtime()`

     This routine returns an elapsed wall clock time in seconds (double precision) on the calling processor.

  5. `MPI_Finalize()`

     This routine terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program no other MPI routines may be called after it.

  6. `MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

     This routine is a basic blocking send operation. It returns only after the application buffer in the sending task becomes free for reuse.

  7. `MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

     This routine receives a message and blocks until the requested data is available in the application buffer in the receiving task.

4   MPI Sample Exercise

Write an MPI program that will sum up elements of an array. The root process in your program acts as a master and the child processes act as slaves. The master allocates equal portions of the array to each slave. Each process then (including the master) calculates a partial sum of the portion of the array assigned to it. Afterwards, each slave sends its partial sum to the master, who collects all partial sums from all the slaves and calculates a grand total.