Lab 1 – Introduction to MPI

Issues with shared memory programming

- Parallel tasks are run by threads.
- All threads live on the same node & share memory.
- Bottleneck: Resources on the single node.
- Overhead due to creation & deletion of threads.
- Can lead to race conditions.

Distributed Memory Programming

- Parallel tasks are processes.
- Each process has its own private memory.
- Processes need not be on the same node.
- Possibility of introducing new bugs like deadlocks.
- **MUST EXPLICITLY CODE IN THE COMMUNICATION BETWEEN PROCESSES: MPI (MESSAGE PASSING INTERFACE).**

What is MPI?

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs
- Uses either C or Fortran
- Defines the **syntax** and **semantics** of a core set of library routines
- Allows data to be passed between processes in a distributed memory environment
- Distributed memory
  - Each processor has local memory
  - Cannot directly access the memory of other processors

MPI Forum

- First message-passing interface standard
  - Successor to PVM (Parallel Virtual Machine)
- Sixty people from forty different organizations (academia and industry)
- International representation
- MPI 1.1 Standard developed from 92-94
- MPI 2.0 Standard developed from 95-97
- Standards documents
  - http://www.mcs.anl.gov/mpi/index.html
  - http://www.mpi-forum.org/docs/docs.html (postscript versions)

Goals and Scope

- MPI’s prime goals are:
  - To provide source-code portability
  - To allow efficient implementation
- It also offers:
  - A great deal of functionality
  - Support for heterogeneous parallel architectures

MPI Library

- Over 125 routines
- However, able to write fully-functional parallel programs using only 6 of them:
  - MPI_Init — Initializes MPI
  - MPI_Finalize — Terminates MPI
  - MPI_Comm_size — Determines no of processes
  - MPI_Comm_rank — Determines label of calling process
  - MPI_Send — Sends a message
  - MPI_Recv — Receives a message

Starting the MPI Library

- **MPI_Init**
  - Called only once prior to any calls to other MPI routines
  - Initializes the MPI environment
  - Must be called by all processes

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

  - argc and argv are the command line arguments of the C program
  - Upon successful execution, MPI_Init returns MPI_SUCCESS

Terminating the MPI Library

- **MPI_Finalize()**
  - Called at the end of the computation
  - Clean-up tasks to terminate the MPI environment
  - No MPI calls after MPI_Finalize
  - Must be called by all processes

`int MPI_Finalize()`

  - Upon successful execution, returns MPI_SUCCESS

Naming Practices & Argument Conventions

- All MPI routines, data types and constants are prefixed by **“MPI_”**
- MPI constants and data stuctures are defined for C in the file “mpi.h”.
- This header file must be included in each MPI program

Communicators

- Communication domain: set of processes that are allowed to communicate with each other
- Information about communication domains are stored in variables of type MPI_Comm, called **communicators**
- Used as arguments to all message transfer MPI routines
- Each process can belong to many different communication domains

MPI_COMM_WORLD

- In general, all processes may need to communicate with each other
- Default communicator, MPI_COMM_WORLD, which includes all processes involved in the parallel execution
- However, to perform communication only within a particular group of processes, use a communicator for each such group: no messages will interfere with messages destined to other groups (Using MPI_Group)

Find number of processes

- MPI_Comm_size

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

  - Returns in the variable size the number of processes that belong to the communicator comm

e.g.

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Rank identification

- MPI_Comm_rank

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

  - Every process is uniquely identified by its rank
  - Starts with zero and goes to (n-1) where n is the number of processes requested
  - On return, the variable rank stores the rank of the process

e.g.

```
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank)
```

Hello World

```
#include <mpi.h>
main(int argc, char *argv[])
{
int npes, myrank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
printf(“From process %d out of %d, Hello
   World!\n”, myrank, npes);
MPI_Finalize();
}
```