Getting Started with MPI

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Topics

- This chapter will familiarize you with some basic concepts of MPI programming, including the basic structure of messages and the main modes of communication.
- The topics that will be discussed are
  - The basic message passing model
  - What is MPI?
  - The goals and scope of MPI
  - A first program: Hello World!
  - Point-to-point communications and messages
  - Blocking and nonblocking communications
  - Collective communications

The Message Passing Model

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

The Message Passing Model

- MPI is intended as a standard implementation of the "message passing" model of parallel computing.
  - A parallel computation consists of a number of **processes**, each working on some local data. Each process has purely local variables, and there is no mechanism for any process to **directly** access the memory of another.
  - Sharing of data between processes takes place by message passing, that is, by explicitly sending and receiving data between processes.
- Note that the model involves **processes**, which need not, in principle, be running on different **processors**. In this course, it is generally assumed that different processes are running on different processors and the terms "processes" and "processors" are used interchangeably

The Message Passing Model

- The usefulness of the model is that it:
  - can be implemented on a wide variety of platforms, from shared-memory multiprocessors to networks of workstations and even single-processor machines.
  - generally allows more control over data location and flow within a parallel application than in, for example, the shared memory model. Thus programs can often achieve higher performance using explicit message passing. Indeed, performance is a primary reason why message passing is unlikely to ever disappear from the parallel programming world.

What is MPI?

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

What is MPI?

- MPI stands for "Message Passing Interface". It is a library of functions (in C) or subroutines (in Fortran) that you insert into source code to perform data communication between processes.

MPI-1

- The MPI-1 standard was defined in Spring of 1994.
  - This standard specifies the names, calling sequences, and results of subroutines and functions to be called from Fortran 77 and C, respectively. All implementations of MPI must conform to these rules, thus ensuring portability. MPI programs should compile and run on any platform that supports the MPI standard.
  - The detailed implementation of the library is left to individual vendors, who are thus free to produce optimized versions for their machines.
  - Implementations of the MPI-1 standard are available for a wide variety of platforms.

MPI-2

- An MPI-2 standard has also been defined. It provides for additional features not present in MPI-1, including tools for parallel I/O, C++ and Fortran 90 bindings, and dynamic process management.

Goals of MPI

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Goals of MPI

- The primary goals addressed by MPI are to
  - Provide source code portability. MPI programs should compile and run as-is on any platform.
  - Allow efficient implementations across a range of architectures.
- MPI also offers
  - A great deal of functionality, including a number of different types of communication, special routines for common "collective" operations, and the ability to handle user-defined data types and topologies.
  - Support for heterogeneous parallel architectures.

Why (Not) Use MPI?

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Why Use MPI?

- You should use MPI when you need to
  - Write portable parallel code.
  - Achieve high performance in parallel programming, e.g. when writing parallel libraries.
  - Handle a problem that involves irregular or dynamic data relationships that do not fit well into the "data-parallel" model.

Why Not Use MPI?

- You should not use MPI when you
  - Can achieve sufficient performance and portability using a data-parallel (e.g., High-Performance Fortran) or shared-memory approach (e.g., OpenMP, or proprietary directive-based paradigms).
  - Can use a pre-existing library of parallel routines (which may themselves be written using MPI).
  - Don't need parallelism at all!

Basic Features of Message Passing Programs

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Basic Features of Message Passing Programs

- Message passing programs consist of multiple instances of a serial program that communicate by library calls. These calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and finally terminate communications.
2. Calls used to communicate between pairs of processors.
3. Calls that perform communications operations among groups of processors.
4. Calls used to create arbitrary data types.

A First Program: Hello World!

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

A First Program: Hello World!

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {
    int err;
    err = MPI_Init(&argc, &argv);
    printf("Hello world!\n");
    err = MPI_Finalize();
}
```

A First Program: Hello World!

- For the moment note from the example that
  - MPI functions/subroutines have names that begin with MPI_.
  - There is an MPI header file (mpi.h or mpif.h) containing definitions and function prototypes that is imported via an "include" statement.
  - MPI routines return an error code indicating whether or not the routine ran successfully.

A First Program: Hello World!

- Each process executes a copy of the entire code. Thus, when run on four processors, the output of this program is

```
Hello world!
Hello world!
Hello world!
Hello world!
```

- However, different processors can be made to do different things using program branches, e.g.

```
if (I am processor 1)
...do something...
if (I am processor 2)
...do something else...
```

Point-to-Point Communications and Messages

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Point-to-Point Communications

- direct communication between two processors, one of which **sends** and the other **receives**
- In a generic send or receive, a **message** consisting of some block of data is transferred between processors. A message consists of an **envelope**, indicating the source and destination processors, and a **body**, containing the actual data to be sent.

Point-to-Point Communications

- MPI uses three pieces of information to characterize the message body

1. **Buffer** - the starting location in memory where outgoing data is to be found (for a send) or incoming data is to be stored (for a receive).
   – In C, buffer is the actual address of the array element where the data transfer begins.
2. **Datatype** - the type of data to be sent.
   – In the simplest cases this is an elementary type such as float, int, etc. In more advanced applications this can be a user-defined type built from the basic types. These can be thought of as roughly analogous to C structures, and can contain data located anywhere, i.e., not necessarily in contiguous memory locations. This ability to make use of user-defined types allows complete flexibility in defining the message content.
3. **Count** - the number of items of type datatype to be sent.

Communication Modes and Completion Criteria

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Communication Modes and Completion Criteria

- There are a variety of **communication modes** that define the procedure used to transmit the message, as well as a set of criteria for determining when the communication event (i.e., a particular send or receive) is **complete**.
  - For example, a **synchronous send** is defined to be complete when receipt of the message at its destination has been acknowledged.
  - A **buffered send**, however, is complete when the outgoing data has been copied to a (local) buffer; nothing is implied about the arrival of the message at its destination.
  - In all cases, completion of a send implies that it is safe to overwrite the memory areas where the data were originally stored.
- There are four communication modes available for sends:
  - Standard
  - Synchronous
  - Buffered
  - Ready
- For receives there is only a single communication mode.

Blocking and Nonblocking Communication

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Blocking and Nonblocking Communication

- In addition to the communication mode used, a send or receive may be **blocking** or **nonblocking**.
- A **blocking** send or receive does not return from the subroutine call until the operation has actually completed. Thus it insures that the relevant completion criteria have been satisfied before the calling process is allowed to proceed.
  - With a blocking send, for example, you are sure that the variables sent can safely be overwritten on the sending processor. With a blocking receive, you are sure that the data has actually arrived and is ready for use.

Blocking and Nonblocking Communication

- A nonblocking send or receive returns immediately, with no information about whether the completion criteria have been satisfied. This has the advantage that the processor is free to do other things while the communication proceeds "in the background." You can test later to see whether the operation has actually completed.
  - For example, a nonblocking synchronous send returns immediately, although the send will not be complete until receipt of the message has been acknowledged. The sending processor can then do other useful work, testing later to see if the send is complete. Until it is complete, however, you can not assume that the message has been received or that the variables to be sent may be safely overwritten.

```
ERROR: stackunderflow
OFFENDING COMMAND: ~

STACK:
```