Parallel programming

MPI Interface

Getting Started with MPI Introduction

- MPI was designed to be a standard implementation of the message-passing model of parallel computing
- MPI itself is not a library
- There are many implementations of MPI:
  - MPICH,
  - MPICH2,
  - OpenMPI(we will use this),
  - LAM/MPI,
  - HP/MPI
  - BoostMPI (c++)

Introduction

- MPI program consists of two or more autonomous processes
- Processes communicate via calls to MPI communication routines
- Processes are identified according to their relative *rank* within a group (0, 1, . . . , groupsize-1)
- MPI does not allow for dynamic allocation of processes

MPI Header Files

- MPI header files contain the prototypes for:
  - functions/subroutines
  - definitions of macros
  - constants
  - datatypes used by MPI

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

MPI Naming Conventions

- The names of all MPI entities (routines, constants, types,…) begin with *MPI*_
- MPI_Xxxxx(parameter, ...)
- Example:
  - MPI_Init(&argc, &argv)
  - MPI_COMM_WORLD
  - MPI_REAL

Parallel programming

- Exit status of a call to an MPI function is returned as an "int"
- Example:

```
int err;
...
err = MPI_Init(&argc, &argv);
if (err == MPI_SUCCESS)
{
   ...
   routine ran correctly
   ...
}
 ...
```

MPI Datatypes

- MPI allows automatic translation between its own datatypes and corresponding datatypes in C
- As a general rule, the MPI datatype given in a receive must match the *MPI datatype* specified in the send

Basic MPI Datatypes

- MPI_CHAR
- MPI_INT
- MPI_DOUBLE

Special MPI Datatypes

- MPI_COMM
- MPI_STATUS

Initializing MPI

- The initialization routine *MPI_INIT* must be the first MPI routine called in any MPI program
- *MPI_INIT must* be called by all processes

```
int err;

...

err = MPI_Init(&argc, &argv);
```

Communicators

- *communicator* is a MPI handle that defines a group of processes
- processor can be a member of a number of different communicators
- This identifying number is known as the *rank* of the processor in that communicator
- If a processor belongs to more than one communicator, its rank in each can (and usually will) be different
- **MPI_COMM_WORLD**

Getting Communicator Information: Rank

- *MPI_COMM_RANK*
- Ranks are consecutive and start with 0
- A given processor may have different ranks in the various communicators to which it belongs
- `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
- MPI_COMM - a communicator

Getting Communicator Information: Size

- A processor can determine the size : *MPI_COMM_SIZE*
- `int MPI_Comm_size(MPI_Comm comm, int *size);`
- MPI_COMM, a communicator
- *size - *address* of the integer variable size

- If the communicator is MPI_COMM_WORLD, the number of processors returned from MPI_COMM_SIZE equals the number defined by:

  % mpirun -np 4 primer

Terminating MPI

- *MPI_FINALIZE* is the last MPI routine called in a program
- It terminates the program by cleaning up all MPI data structures, canceling operations that never completed
- MPI_FINALIZE *must* be called by all processes
- Once MPI_FINALIZE has been called, no other MPI routines (including MPI_INIT) may be called

```
err = MPI_Finalize();
```

Hello World!

```
#include "mpi.h"

#include <stdio.h>

int main(int argc, char *argv[])

{

  int id;

  int p;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  printf("Hello, world, from process %d\n", id);

  MPI_Finalize(); return 0;

}
```

Point-to-Point Communication

- *Point-to-point communication* is the fundamental communication facility provided by the MPI library
- Conceptually it is simple:
  - one process sends a message
  - another process receives it
- It is **not** that simple
- Crucial issue is what message to receive?
- Another issue is whether send and receive routines initiate communication operations and return immediately (nonblocking), or wait for the initiated communication operation to complete before returning (blocking).

Source and Destination

- One process (the source) sends
- Another process (the destination) receives
- In general, the source and destination processes operate asynchronously
- The source process may complete sending a message long before the destination process gets around to receiving it
- The destination process may initiate receiving a message that has not yet been sent

Messages

- Messages consist of two parts: the *envelope* and the *message body*
- The *envelope* of an MPI message has four parts:
  - **Source** — the sending process
  - **Destination** — the receiving process
  - **Communicator** — specifies a group of processes to which both source and destination belong
  - **Tag** — used to classify messages
- The *message body* has three parts:
  - **Buffer** — the message data
  - **Datatype** — the type of the message data
  - **Count** — the number of items of type datatype in buffer

Sending and Receiving Messages

- The source (the identity of the sender) is determined implicitly
- Envelope and body is given explicitly by the sending process
- *pending messages*
- Pending messages are not maintained in a simple FIFO queue
- To receive a message, a process specifies a message envelope that MPI compares to the envelopes of pending messages
- The receiving process must be careful to provide enough storage for the entire message

Blocking Send and Receive

- MPI_SEND
- MPI_RECV
- Both routines block the calling process until the communication operation is completed

Sending a Message: MPI_SEND

- The message body contains the data to be sent: *count* items of type *datatype*

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
int dest, int tag, MPI_Comm comm);
```

- All arguments are input arguments
- An error code is returned by the function

Receiving a Message: MPI_RECV

- The arguments in the message envelope determine what messages can be received
- The *source*, *tag*, and *communicator* arguments must match
- If the received message has more data than the receiving process is prepared to accept, it is an error and the program will abort
- If the sender and receiver use incompatible message datatypes, the results are undefined

Receiving a Message: MPI_RECV

- The *status* argument returns information about the message that was received

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int
source, int tag, MPI_Comm comm, MPI_Status *status);
```

- *buf* and *status* are output arguments; the rest are inputs
- An error code is returned by the function
- The meaning of the second argument: the maximum number of elements that the array *b* could hold

Example 01

Runtime Behavior

- When a message is sent using MPI_SEND one of two things may happen:
  - The message may be copied into an MPI internal buffer and transferred to its destination later, in the background
  - The message may be left where it is, in the program's variables, until the destination process is ready to receive it. At that time, the message is transferred to its destination

Blocking and Completion

- Both MPI_SEND and MPI_RECV block the calling processes. Neither returns until the communication operation it invoked is completed
- Messages that are copied into MPI internal buffer will occupy buffer space until the destination process begins to receive the message

Deadlock

- When two (or more) processes are blocked and each is waiting for the other to make progress, *deadlock* occurs

Example 02 - Deadlock