Lecture 3

Introduction to message passing and SPMD programming

Programming with Message Passing

- In this course we’ll use the message passing programming model to handle communication
- Programs execute as a set of P processes
  - We specify P when we run the program
  - Assume each process is assigned a different physical processor
- Each physical process
  - is initialized with the same code but has private state
  - executes instructions at its own rate
  - has an associated rank, a unique integer in the range 0:P-1
- The sequence of instructions each process executes depends on its rank and the messages it sends and receives
What’s in a message?

• Messages are like email
• To send a message we specify
  – A destination
  – A message body (can be empty)
• To receive a message we need similar information, including a receptacle to hold the incoming data

Message Passing

• Recall that message based communication requires that sender and receiver be aware of one another
• There must be an explicit recipient of the message
• In message passing there are two events:
  – Memory to memory block copy
  – Synchronization signal on receiving end: “Data arrived”
The API

• Query functions
  \texttt{nproc()} = \# processors
  \texttt{myRank()} = my process ID

• Various message passing primitives

The message passing operations

• Simplest form of communication: \textit{point-to-point} messages
  – Send a message to another process
    \texttt{Send(Object, Destination process ID)}
  – Receive a message from another process
    \texttt{Receive(Object)}
Some semantic issues

- **Receive()** blocks until the message has been received
  - It is safe to use the data in the buffer
- When **Send()** returns, the message is “in transit”
  - It could be anywhere in the system, and might have been received
  - A return from **Send()** doesn’t tell us if the message has been received
  - It is safe to overwrite the data in the buffer
- An error if the source and destination object don’t have compatible types

Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent
- But if different processes send messages to the same destination, the order of receipt isn’t defined across processes
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
- But if different processes send messages to the same destination, the order of receipt isn’t defined across the processes.

A simple program

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x=1, y=0</td>
<td>int x=0, y=1</td>
</tr>
<tr>
<td>recv (x)</td>
<td>recv(y)</td>
</tr>
<tr>
<td>send(x,1)</td>
<td>send(y,0)</td>
</tr>
<tr>
<td>print x, y</td>
<td>print x, y</td>
</tr>
</tbody>
</table>

What is the outcome of this program?
Removing deadlock

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x=1, y=0</td>
<td>int x=0, y=1</td>
</tr>
<tr>
<td>Send(x,1)</td>
<td>Send(y,0)</td>
</tr>
<tr>
<td>Recv(x)</td>
<td>Recv(y)</td>
</tr>
<tr>
<td>Print x, y</td>
<td>Print x, y</td>
</tr>
</tbody>
</table>

What is the outcome of this program?

Non-blocking communication

- We’ve looked at blocking calls that cause the program to wait for completion
- There is asynchronous, non-blocking communication
- These are needed to express certain algorithms
- Their use can also help improve performance
Non-blocking communication

• Non-blocking communication is *split-phased*
  – In the first phase we initiate communication with and ‘I’ variant of the point-to-point call
    \[ \text{IRcv}( ), \text{ISend}( ) \]
  – In the second phase we synchronize
    \[ \text{Wait}( ) \]
  – We can carry out unrelated computations between the two phases

• Building a blocking call
  \[ \text{Recv}( ) = \text{IRcv}( ) + \text{Wait}( ) \]
Restrictions on non-blocking communication

- The message buffer may not be accessed between an IRecv() (or ISend()) and its accompanying wait()
- Why can’t we read the buffer during a pending ISend()?
- Each pending IRecv() must have a distinct buffer

Message passing in practice

- We’ll program with a library called MPI “Message Passing Interface”
- This is an implementation of the message passing model discussed in last lecture
- We’ll use a variant of MPI in this course called MPI-CH
- There are 6 minimal routines needed by nearly every MPI program
  - 4 basic routines we use to start, end, and query MPI execution state
  - 2 basic message passing routines
Using MPI

- 125 routines in MPI-1
- Point to point communication
- Communicators
- Message Filtering
- Measuring performance
- Reference material: see the MPI section of “Software available in the Course”
  [http://www.cse.ucsd.edu/classes/fa03/cse260/testbeds.html](http://www.cse.ucsd.edu/classes/fa03/cse260/testbeds.html)

Sending and receiving messages

- MPI provides a rich collection of routines to move data between address spaces
- Today we’ll talk about point-to-point message passing, in which a single pair of processors communicate
- Later on we’ll cover collective communication, when all the processors communicate together
- In point-to-point message passing we can filter messages in various ways
- This allows us to organize message passing activity conveniently
What’s in an MPI message?

• To send a message we need
  – A destination
  – A “type”
  – A message body (can be empty)
  – A context (called a “communicator” in MPI)

• To receive a message we need similar
  information, including a receptacle to hold
  the incoming data

A first MPI program: “hello world”

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv);
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```
A second MPI program

```c
main(int argc, char **argv ){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("I am process %d of %d.
        rank, size);
    MPI_Finalize();
}
```

Communicators

- One way of screening messages is through a communicator
- A communicator is a name-space (or a context) describing a set of processes that may communicate
- MPI defines a default communicator `MPI_COMM_WORLD` containing all processes
- MPI provides the means of generating uniquely named subsets (later on)
MPI Datatypes

- MPI messages have a specified length
- The unit depends on the type of the data
- The length in bytes is `sizeof(type) × # elements`
- Why don’t we use the # bytes as the length?
  - Support communication on heterogeneous machines with different storage representations without requiring compiler support
  - Avoid memory-to-memory copying

MPI Datatypes

- Because MPI is a library, we specify the type (and hence length) of an element
- To this end MPI specifies a set of built-in types, corresponding to the primitive types of the language from which MPI is called

  - In C: `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_CHAR`, `MPI_LONG`, `MPI_UNSIGNED`, `MPI_BYTE`,...

- See `/opt/mpich/myrinet/gnu/include/mpi.h`
- Later on we’ll discuss user defined types, e.g. structs
MPI Tags

- Each sent message is accompanied by a user-defined integer \textit{tag}:
  - Receiving process can use this information to organize messages
  - May also filter messages (like a subject: line in email)
  - \texttt{MPI\_ANY\_TAG} inhibits screening.

```c
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
              MPI_INT, 1,
              Tag, MPI_COMM_WORLD, &status);
} else MPI_Send(msg, 2,
               MPI_INT, 0,
               Tag, MPI_COMM_WORLD);
```

Send and Recv
Send and Recv

```c
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
             MPI_INT, 1,
             Tag, MPI_COMM_WORLD, &status);
}
else MPI_Send(msg, 2,
              MPI_INT, 0,
              Tag, MPI_COMM_WORLD);
```

Send and Recv

```c
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
             MPI_INT, 1,
             Tag, MPI_COMM_WORLD, &status);
}
else MPI_Send(msg, 2,
              MPI_INT, 0,
              Tag, MPI_COMM_WORLD);
```
Send and Recv

```
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
              MPI_INT, 1,
              Tag, MPI_COMM_WORLD, &status);
}
else MPI_Send(msg, 2,
              MPI_INT, 0,
              Tag, MPI_COMM_WORLD);
```
**Send and Recv**

```c
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2,
             MPI_INT, 1,
             Tag, MPI_COMM_WORLD, &status);
}
else  MPI_Send(msg, 2,
               MPI_INT, 0,
               Tag, MPI_COMM_WORLD);
```

**Message status**

- An MPI_Status variable is a struct that contains the sending processor and the message tag
- This information is useful when we haven’t filtered the messages
- We may also access the length of the received message (may be shorter than the message buffer)

```c
MPI_Status status;
MPI_Recv( message, count, 
TYPE, Destination, 
Tag, COMUNICATOR, &status);
MPI_Get_count(&status, TYPE, &recv_count );
```
Non-blocking communication

• MPI also provides asynchronous, *non-blocking* variants
• These are needed to express certain algorithms
• Their use can also help improve performance

Non-blocking communication

• Non-blocking communication is *split-phased*
  – In the first phase we initiate communication with an ‘I’ variant of the point-to-point call
    \( \text{IRcv}() \), \( \text{ISend}() \)
  – In the second phase we synchronize \( \text{Wait}() \)
  – We can carry out unrelated computations between the two phases
• Building a blocking call
  \( \text{Recv}() = \text{IRcv}() + \text{Wait}() \)
In MPI

- An extra request argument is required
  ```c
  MPI_Request request;
  MPI_Irecv(buf, count, type, source, tag, comm, &request)
  ```

- We use the request variable to specify which message we are synchronizing
  ```c
  MPI_Wait(&request, &status)
  ```