Lecture 3

Introduction to message passing and SPMD programming
Announcements

• Run the “hello” program described in Assignment #2
• Read over the Getting Started with Valkyrie web page, especially runaway processes
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
    SPMD = “Same Program Multiple Data”
  – 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

• Program execution is often called “bulk synchronous” or “loosely synchronous”
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

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

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

• Various message passing primitives
The message passing operations

• Simplest form of communication: *point-to-point* messages
  – Send a message to another process
    Send(Object, Destination process)
  – Receive a message from another process
    Receive(Object)
    Receive(Source process, 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, may or may not 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 identical types**
Causality

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

• If different processes send messages to the same destination
  – The order of receipt is defined from a single source
  – The order of receipt is not defined across multiple sources
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(x)</td>
</tr>
<tr>
<td>Send(y,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?
Non-blocking communication

• We’ve looked at *blocking* calls that cause the program to wait for completion

• There is asynchronous, *non-blocking* communication

• Needed to express certain algorithms

• Used to improve performance
Non-blocking communication

• Non-blocking communication is *split-phased*
  – Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call
    \[ \text{IRrecv( ), ISend( )} \]
  – Phase 2: synchronize
    \[ \text{Wait( )} \]
  – May carry out unrelated computations between the two phases

• Building a blocking call
  \[ \text{Recv( )} = \text{IRrecv( )} + \text{Wait( )} \]
Fixing a deadlocked program

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recv(x)</td>
<td>Recv(x)</td>
</tr>
<tr>
<td>Send(y,1)</td>
<td>Send(y,0)</td>
</tr>
<tr>
<td>IRecv(x)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Wait(x)</td>
<td>Wait(x)</td>
</tr>
<tr>
<td>Send(y,1)</td>
<td>Send(y,0)</td>
</tr>
</tbody>
</table>
Fixing an unsafe program

- Consider the following example of an “unsafe” program
- It may deadlock if there isn’t enough storage to receive the incoming message

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(x,1)</td>
<td>Send(x,0)</td>
</tr>
<tr>
<td>Recv(y)</td>
<td>Recv(y)</td>
</tr>
</tbody>
</table>
Avoiding an unsafe program

- The system has pre-allocated storage for the incoming messages so there’s no possibility of running out of storage

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x)</td>
<td>IRecv(y)</td>
</tr>
<tr>
<td>Send(y,1)</td>
<td>Send(y,0)</td>
</tr>
</tbody>
</table>
Restrictions on non-blocking communication

• The message buffer may not be accessed between an IRecv( ) (or ISend( )) and its accompanying wait( )

• Each pending IRecv() must have a distinct buffer
Message passing in practice

• We’ll program with a library called MPI “Message Passing Interface”
• We’ll use a variant of MPI called MPI-CH
• Reference material: see the MPI section of “Software available in the Course”
  http://www.cse.ucsd.edu/classes/fa06/cse160/testbeds.html
• 125 routines in MPI-1
• 6 minimal routines
  – start, end, and query MPI execution state (4)
  – non-blocking point-to-point message passing (3)
Functionality we’ll will cover today

• Point-to-point communication
• Communicators
• Data types
• Tags
• Non-blocking communication
• Message Filtering
Sending and receiving messages

• MPI provides a rich collection of routines to move data between address spaces

• A single pair of communicating processes use *point-to-point* communication

• Later on we’ll discuss *collective communication*, in which all the processes 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
  – Destination
  – Message body (may be empty)
  – Tag
  – Name space (an MPI “communicator”)

• To receive a message we need similar information, including a receptacle to hold the incoming data
A first MPI program: “hello world”

#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

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. \n", 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

  \texttt{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) \times # elements`
- Why don’t we use the # bytes as the length?
  - Heterogeneous machines with different storage representations
  - Performance
MPI Datatypes

• Because MPI is a library, we specify the type (and hence length) of an element

• MPI provides 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`

• User defined types, e.g. structs, later on
MPI Tags

• Each sent message is accompanied by a user-defined integer *tag*:

• Receiving process can use this information to organize messages

• May also filter messages
  (subject: line in email)

  **MPI\_ANY\_TAG** inhibits screening.
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

```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 containing the sending processor and the message tag
- 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)

```
MPI_Recv( message, count,
    TYPE, MPI_ANY_SOURCE,
    MPI_ANY_TAG, COMMUNICATOR, &status);
MPI_Get_count(&status, TYPE, &recv_count);

status.MPI_SOURCE   status.MPI_TAG
```
Non-blocking communication

• MPI_Irecv specifies a request argument

```c
MPI_Request request;
MPI_Irecv(buf, count, type, source, tag, comm, &request)
```

• The request variable specifies which message we are synchronizing

```c
MPI_Wait(&request, &status)
```
Relationship to C* streams

• C* provides the stream abstraction
• FIFO of typed values from multiple sources to a single destination
• “Send” a value over a channel
  send(Channel, Value)
• Only one process may read the channel
  recv(Channel)