Lecture 6

Programming with Message Passing
Announcements

• Project Proposals
  ♦ What are the goals of your project? Are they realistic?
  ♦ What are your hypotheses?
  ♦ What is your experimental method for proving or disproving your hypotheses?
  ♦ What experimental result(s) do you need to demonstrate?
  ♦ What would be the significance of those results?
  ♦ What code will you need to implement? What software packages or previously written software will use?
  ♦ A tentative division of labor among the team members.

• See me to discuss your project
Programming with Message Passing

• The primary model for implementing parallel applications
• Useful for understanding fundamental behavior in various types of parallel computers
• 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”
Message Passing

• 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”

Message buffers
Minimal message passing

• Query functions
  \[ \text{nproc}() = \# \text{processors} \]
  \[ \text{myRank}() = \text{this process’s rank} \]

• \textit{Point-to-point} communication
  – Simplest form of communication
  – Send a message to another process
    \[ \text{Send(} \text{Object, Destination process ID} \) \]
  – Receive a message from another process
    \[ \text{Receive(} \text{Object} \) \]
    \[ \text{Receive(} \text{Source process, Object} \) \]
Send and Recv

- When `Send()` returns, the message is “in transit”
  - A return doesn’t tell us if the message has been received
  - Somewhere in the system
  - Safe to overwrite the buffer
- `Receive()` blocks until the message has been received
  - Safe to use the data in the buffer
- Error if the source and destination object don’t have *identical* types

<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>
<tr>
<td>Print x, y</td>
<td>Print x, y</td>
</tr>
</tbody>
</table>
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
Non-blocking communication

- We’ve seen *blocking* calls that cause the program to wait for completion
- There is asynchronous, *non-blocking* communication
- These are needed to express certain algorithms
- Also used to improve performance
Non-blocking communication

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

• Building a blocking call
  \[ \text{Recv}( ) = \text{IRecv}( ) + \text{Wait}( ) \]
### Program behavior

<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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Send(y,1)</td>
<td>Send(y,0)</td>
</tr>
</tbody>
</table>

- Each pending IRecv() must have a distinct buffer
MPI

• We’ll program with a library called MPI “Message Passing Interface”
• We’ll use a variant of MPI called MPI-CH
• Callable from C, C++, Fortran, etc.
• All major vendors support MPI, but implementations differ in quality
Using MPI

• 125 routines in MPI-1
• 7 minimal routines needed by nearly every MPI program
  – start, end, and query MPI execution state (4)
  – non-blocking point-to-point message passing (3)
• Reference material: see http://www-cse.ucsd.edu/users/baden/Doc/mpi.html
Functionality we’ll will cover today

• Point-to-point communication
• Communicators
• Data types
• Tags
• Non-blocking communication
• Message Filtering
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

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();
}
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 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
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)
Send andRecv

```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);
```
\begin{verbatim}
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);
\end{verbatim}
MPI Datatypes

• MPI messages have a specified length
• The unit depends on the type of the data
• The length in bytes is \( \text{sizeof(type)} \times \# \text{ elements} \)
• Built-in MPI types (for C binding)
  • In C: `MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, MPI_LONG, MPI_UNSIGNED, MPI_BYTE,…`
• User defined types, e.g. structs
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 (like a subject: line in email)
  – `MPI_ANY_TAG` inhibits screening.
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_Recv( message, count, TYPE, MPI_ANY_SOURCE, MPI_ANY_TAG, COMMUNICATOR, &status);

MPI_Get_count( &status, TYPE, &recv_count );
```

```c
status.MPI_SOURCE    status.MPI_TAG
```
Non-blocking communication 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 in `MPI_Wait()`
  ```c
  MPI_Wait(&request, &status)
  ```
- Making above 3 calls in succession is equivalent to
  ```c
  MPI_Recv(buf, count, type, source, tag, comm, &status)
  ```
Buffering

• If there is not a pending receive, then an incoming message is placed in an anonymous system buffer
• When the receive gets posted, the message is moved into the user specified buffer
• Double copying reduces communication performance
Avoiding the overhead

• Non-blocking communication can help ameliorate this problem

• For more information see

  *MPI: The Complete Reference*, by Marc Snir et al.

  “Buffering and Safety”
Rendezvous

- When a long message is to be sent, can MPI just send the message?
- If so, then it sends the message. This is called a *rendezvous* implementation. What are the advantages and disadvantages?
Rendezvous and eager limits

- In an *eager* implementation, we just send the message
- The *eager limit* is the longest message that can be sent in eager mode
- Maximum value on IBM SP systems is 256K
- What about longer messages?
Send Modes

• MPI provides four different modes for sending a message
  – Standard: Send may or may not complete until matching receive is posted (whether or not the data is buffered is up to the implementation)
  – Synchronous: Send does not complete until matching receive is posted
  – Ready: Matching receive must already have been posted
  – Buffered: data is moved to a user-supplied buffer before sending

• See the handy reference at http://www-unix.mcs.anl.gov/mpi/sendmode.html
Sends that block

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

<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(x)</td>
</tr>
<tr>
<td>Send(y,1)</td>
<td>Send(y,0)</td>
</tr>
</tbody>
</table>
Communication performance

• Communication performance is a major factor in determining the overall performance of an application

• Let the message have a length \( n \)

• The simplest communication cost model is transfer time is 
  \[ \text{transfer time} = \alpha + \beta^{-1 \infty} n \]
  \( \alpha = \) message startup time
  \( \beta_{\infty} = \) peak bandwidth (bytes per second)
  \( n = \) message length
Startup and bandwidth

- The startup term dominates when the message is sufficiently short
  \[ \alpha \gg \beta^{-1}\infty n \]

- The bandwidth term dominates when the message is sufficiently long
  \[ \beta^{-1}\infty n \gg \alpha \]
Half power point

- Let $T(n) = \text{time to send a message of length } n$
- Let $\beta(n) = \text{the effective bandwidth}$
  
  $$\beta^{-1}(n) = n / T(n)$$

- We define the **half power point** $n_{1/2}$ as the message size required to achieve $\frac{1}{2} \beta_{\infty}$
  
  $$\frac{1}{2} \beta^{-1}_{\infty} = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_{\infty}$$

- In theory, this occurs when $\alpha = \beta^{-1}_{\infty} n_{1/2} \Rightarrow n_{1/2} = \alpha / \beta^{-1}_{\infty}$

- Doesn’t generally predict actual value of $n_{1/2}$

- For SDSC’s DataStar machine
  - $\alpha \approx 7.6 \mu s$, $\beta_{\infty} \approx 1580 \text{ Mbytes/sec} \Rightarrow n_{1/2} \approx 12\text{KB}$
  - The actual value of $n_{1/2} \approx 38\text{KB}$
  - In Assignment #2, you’ll explore this phenomenon
Typical bandwidth curve
(SDSC Blue Horizon)

\[ N_{1/2} \approx 100\text{KB} \]

\[ 390 \text{ MB/sec} \]

\[ N = 4\text{MB} \]
Typical bandwidth curve
(SDSC Triton)

\[ N^{1/2} \approx 60 \text{ KB} \]

\[ \alpha = 3.2 \text{ \mu sec} \]
Short message behavior
Intermediate length message behavior
More about modeling

• LogP model (Culler et al, 1993), is more precise, but the $\alpha$, $\beta$ model is often good enough

• All these models ignore important effects: switch and processor contention
Where does the time go?

• Under ideal conditions…
  – There is a pending receive waiting for an incoming message, which is transmitted directly to and from the users message buffer
  – There is no other communication traffic

• Assume a contiguous message