Lecture 11

Message Passing

Introduction to MPI
Announcements

- Mac Mini lab (APM 2402)
  - Tuesday at 4pm to 6pm
- Sign on to bang and run the Basic example interactively and with batch
Projects

- Counts for 60% of your grade
- Complete in 3 weeks
- See the (growing) list of projects at cseweb.ucsd.edu/classes/fa12/cse260-b/Projects/ProjectList.html
- CUDA, MPI or CUDA + MPI
- Select your project from the list by 11/9
  - A limited number of self-proposed projects, requires a proposal
- Progress report: 11/21 (Weds)
- Final Report: 12/7 (Friday)
Project Proposals

• Due 11/9
  ♦ 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
  ♦ A preliminary list of milestones—with completion dates
Projects!

- Stencil method in 3 dimensions
- Multigrid
- Communication avoiding matrix multiplication (MPI)
- Algorithm based fault tolerance (MPI)
- 3D Fast Fourier Transform (MPI or CUDA)
- Particle simulation (MPI)
- Groups of 3 will do a more ambitious project
  - MPI projects can add communication overlap
  - MPI + CUDA
- Propose your own
- Make your choice by 11/9

www-cse.ucsd.edu/classes/fa12/cse260-b/Projects/ProjectList.html
Today’s lecture

- Message passing
- MPI
Programming with Message Passing

- The primary model for implementing parallel applications
- 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 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
  - may or may not be assigned a different physical processor
- 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 one, 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

• Requires a sender and an explicit recipient that must be aware of one another

• Message passing performs two events
  ♦ Memory to memory block copy
  ♦ Synchronization signal on receiving end: “Data arrived”
A minimal interface

- Query functions
  \( \text{nproc()} = \# \text{ processors} \)
  \( \text{myRank()} = \text{this process's rank} \)

- Point-to-point communication
  - Simplest form of communication
  - Send a message to another process
    \( \text{Send(Object, Destination process ID)} \)
  - Receive a message from another process
    \( \text{Receive(Object)} \)
    \( \text{Receive(Source process, Object)} \)
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
Message completion

• A \texttt{Send( ) may or may not} complete…

• … before a \texttt{Recv( ) has been posted}

• “May or may not” depends on the implementation

• Some programs may deadlock on certain message passing implementations

This program may hang

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Send (x,1)}</td>
<td>\texttt{Send(y,0)}</td>
</tr>
<tr>
<td>\texttt{Recv (y,1)}</td>
<td>\texttt{Recv(x,0)}</td>
</tr>
</tbody>
</table>

This program is “safe”

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Send (x,1)}</td>
<td>\texttt{Recv(x,0)}</td>
</tr>
<tr>
<td>\texttt{Recv (y,1)}</td>
<td>\texttt{Send(y,0)}</td>
</tr>
</tbody>
</table>
Buffering

- Where does the data go when you send it?
- It might be buffered
- Preferable to avoid the extra copy
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 sources.
Asynchronous, non-blocking communication

- **Immediate return, does not wait for completion**
  - Required to express certain algorithms
  - Optimize performance: message flow problems
- **Split-phased**
  - Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call
    \[
    \text{IRecv( ), ISend( )}
    \]
  - Phase 2: synchronize
    \[
    \text{Wait( )}
    \]
  - Perform unrelated computations between the two phases
- **Building a blocking call**
  \[
  \text{Recv( )} = \text{IRecv( ) + Wait( )}
  \]
Restrictions on non-blocking communication

• The message buffer may not be accessed between an \texttt{IRecv()} (or \texttt{ISend()}) and its accompanying \texttt{Wait()}

\texttt{ISend(data,destination)}
\texttt{Wait()} on \texttt{ISend()}
Use the data

• Each pending \texttt{IRecv()} must have a distinct buffer
# Overlap

<table>
<thead>
<tr>
<th>Overlap</th>
<th>No Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x,req)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Send(..)</td>
<td>Send(…)</td>
</tr>
<tr>
<td>Compute(y)</td>
<td>Wait(x)</td>
</tr>
<tr>
<td>Wait(req)</td>
<td>Compute(x)</td>
</tr>
<tr>
<td>Compute(x)</td>
<td>Compute(y)</td>
</tr>
</tbody>
</table>

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

©2012 Scott B. Baden /CSE 260/ Fall 2012
MPI

• We’ll program with a library called MPI “Message Passing Interface”
  ◆ 125 routines in MPI-1
  ◆ 7 minimal routines needed by every MPI program
    • start, end, and query MPI execution state (4)
    • non-blocking point-to-point message passing (3)

• Callable from C, C++, Fortran, etc.

• All major vendors support MPI, but implementations differ in quality

• Reference material: see http://www-cse.ucsd.edu/users/baden/Doc/mpi.html
Functionality we’ll will cover today

• Point-to-point communication
• Non-blocking communication
• Message Filtering
• Communicators
A first MPI program: “hello world”

```
#include "mpi.h"

int 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("Hello, world! I am process %d of %d.\n", rank, size);
    MPI_Finalize();
    return(0);
}
```
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
• With *collective communication*, all the processors participate in communication
• In point-to-point message passing we can filter messages in various ways
• This allows us to organize message passing activity conveniently
Point-to-point messages

• 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
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);
Communicators

• 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)
• A mechanism for screening messages
MPI Tags

• Tags enable processes to organize or screen messages

• Each sent message is accompanied by a user-defined integer tag:
  - Receiving process can use this information to organize or filter messages
  - `MPI_ANY_TAG` inhibits screening.
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
  - We don’t specify the as the \# byte
• MPI specifies a set of built-in types for each of the primitive types of the language
• In C: **MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, MPI_LONG, MPI_UNSIGNED, MPI_BYTE,**…
• Also defined types, e.g. structs
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);
status.MPI_SOURCE  status.MPI_TAG
```
Immediate mode send and receive

- Asynchronous, non-blocking communication
  - Immediate return does not indicate completion
  - Must synchronize with a `Wait()` before reusing buffer (Send) or consuming data (Receive)
- An extra `request` argument, used to refer to a message we are synchronizing

```c
MPI_Request request;
MPI_Irecv(buf, count, type, src, tag, comm, &request)
MPI_Wait(&request, &status)
```

- `Irecv + Wait = Recv`
  ```c
  MPI_Recv(buf, count, type, src, tag, comm, &status)
  ```

- `Immediate Send`
  ```c
  MPI_Isend(buf, count, type, dest, tag, comm, &request)
  ```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Avoiding an unsafe program

- The program on the left may deadlock if there isn't enough storage to receive the message.
- In the program on the right, MPI has pre-allocated storage for the incoming message so there's no possibility of running out of storage.

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

1. Iteration 1: 1 → 2 & 0  0 → 1  (0 → 2)  2 → 0 & 1
2. 1 begins iteration 2: 1 → 2
3. 0 → 2 (but for iteration 1)
4. Problem: irecv in P2 receiving data from P1 in iteration 2 while it expects data from P0 in iteration 1

For $i = 1$ to $n$
MPI_Request req1, req2;
MPI_Status status;
MPI_Irecv(buff, len, CHAR, ANY_NODE, TYPE, WORLD,&req1);
MPI_Irecv(buff2,len, CHAR, ANY_NODE, TYPE, WORLD,&req2);
MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
MPI_Send(buff, len, CHAR, prevnode, TYPE, WORLD);
MPI_Wait(&req1, &status);
MPI_Wait(&req2, &status);
End for
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.
- Non-blocking communication can help avoid this problem.
- *MPI: The Complete Reference*, by Marc Snir et al.
  “Buffering and Safety”
- *Send modes* are also useful
  www-unix.mcs.anl.gov/mpi/sendmode.html
Rendezvous

• When a long message is to be sent, can MPI just send the message?
• For “short” message, it can. This is *eager mode*
• The *eager limit* is the longest message that can be sent in eager mode
• See M. Banikazemi et al., IEEE TPDS, 2001, “MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems”
• For long messages, MPI first sends a scout to get permission to send the message
• This is called *rendezvous mode*
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](http://www-unix.mcs.anl.gov/mpi/sendmode.html)
Where does the time go?

- Communication performance can be a major factor in determining application performance
- Under ideal conditions…
  - There is a pending receive waiting for an incoming message, which is transmitted directly to and from the user's message buffer
  - There is no other communication traffic
- Assume a contiguous message
- LogP model (Culler et al, 1993)
Communication performance

• The so-called $\alpha \beta$ model is often good enough

• Message passing time = $\alpha + \beta^{-1} \infty n$

  $\alpha = \text{message startup time}$

  $\beta_\infty = \text{peak bandwidth (bytes per second)}$

  $n = \text{message length}$

• “Short” messages: startup term dominates

  $\alpha >\!> \beta^{-1} \infty n$

• “Long” messages: bandwidth term dominates

  $\beta^{-1} \infty n >\!> \alpha$
Typical bandwidth curve (SDSC Triton)

\[ N_{1/2} \approx 20 \text{ KB} \]

\[ \alpha = 3.2 \mu\text{sec} \]

Long Messages: \( \beta^{-1} \infty n >> \alpha \)
Half power point

- Let $T(n) =$ time to send a message of length $n$
- Let $\beta(n) =$ 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 $\alpha = \beta^{-1}_\infty n_{1/2} \Rightarrow n_{1/2} = \alpha \beta_\infty$
- Doesn't generally predict actual value of $n_{1/2}$
- For SDSC’s Triton Cluster
  - $\alpha \approx 3.2 \mu$s, $\beta_\infty \approx 1.12 \text{ gbytes/sec} \Rightarrow n_{1/2} \approx 3.6$KB
  - The actual value of $n_{1/2} \approx 20$KB
Typical bandwidth curve
(SDSC Blue Horizon)

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

390 MB/sec

N = 4MB
Short and intermediate message lengths

![Graph showing message length versus time and gigabytes per second for Triton.](image-url)
The Ring program

- Configure the processors logically in a ring and pass messages around the ring multiple times
- Assume there are $p$ processors
- Neighbors of processor $k$ are
  - $(k + 1) \mod p$
  - $(k + p - 1) \mod p$
- See $PUB/Examples/MPI/Ring$

©2012 Scott B. Baden /CSE 260/ Fall 2012
for (int len = 1, l=0; len <= maxSize; len *= 2, l++)
if (myid == 0) {
   // (WARM UP CODE)
   const double start = MPI_Wtime( );
   for (int i = 0; i < trips; i++) {
      PROCESSOR 0 CODE
   }
   const double delta = MPI_Wtime( ) - start;
   Bandwidth = (long)((trips*len*nodes)/ delta /1000.0);
} else {   // myid != 0
   // (WARM UP CODE)
   for (int i = 0; i < trips; i++) {
      ALL OTHER PROCESSORS
   }
}
The Ring program

Processor 0:

```
MPI_Request req;
MPI_Irecv(buffer, len, MPI_CHAR, (rank + p - 1)%p,
    tag, MPI_COMM_WORLD, &req);
MPI_Send(buffer, len, MPI_CHAR, (rank + 1) % p,
    tag, MPI_COMM_WORLD);
MPI_Status status;
MPI_Wait(&req,&status);
```

All others:

```
MPI_Status status1;
MPI_Recv(buffer, len, MPI_CHAR, (rank + p - 1)%p,
    tag, MPI_COMM_WORLD, &status1);
MPI_Send(buffer, len, MPI_CHAR, (rank+1)%p,
    tag, MPI_COMM_WORLD);
```
Our second application:
the trapezoidal rule
The trapezoidal rule

• Use the trapezoidal rule to numerically approximate a definite integral, area under the curve
• Divide the interval \([a,b]\) into \(n\) segments of size \(h=1/n\)
• Area under the \(i^{th}\) trapezoid
  \[ \frac{1}{2} (f(a+i\times h)+f(a+(i+1)\times h)) \times h \]
• Area under the entire curve
  \(\approx\) sum of all the trapezoids

©2012 Scott B. Baden /CSE 260/ Fall 2012
Reference material

• For a discussion of the trapezoidal rule
  http://en.wikipedia.org/wiki/Trapezoidal_rule

• A applet to carry out integration

• Code on Triton (from Pacheco hard copy text)

  Serial Code
  $PUB/Pacheco/ppmpi_c/chap04/serial.c

  Parallel Code
  $PUB/Pacheco/ppmpi_c/chap04/trap.c
Serial code (Following Pacheco)

```c
main() {
    float f(float x) { return x*x; } // Function we're integrating

    float h = (b-a)/n; // h = trapezoid base width
                        // a and b: endpoints
                        // n = # of trapezoids

    float integral = (f(a) + f(b))/2.0;

    float x; int i;

    for (i = 1, x=a; i <= n-1; i++) {
        x += h;
        integral = integral + f(x);
    }
    integral = integral*h;
}
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Parallel Implementation of the Trapezoidal Rule

- Decompose the integration interval into sub-intervals, one per processor
- Each processor computes the integral on its local subdomain
- Processors combine their local integrals into a global one
First version of the parallel code

```c
local_n = n/p;             // # trapezoids; assume p divides n evenly
float local_a = a + my_rank*local_n*h,
    local_b = local_a + local_n*h,
    integral = Trap(local_a, local_b, local_n);

if (my_rank == ROOT) {    // Sum the integrals calculated by
    // all processes
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, MPI_ANY_SOURCE,
            tag, WORLD, &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1, MPI_FLOAT, ROOT, tag, WORLD);
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Playing the wild card

• We can take the sums in any order we wish
• The result does not depend on the order in which the sums are taken, except to within roundoff
• We use a linear time algorithm to accumulate contributions, but there are other orderings

```c
for (source = 1; source < p; source++)
    MPI_Recv(&integral, 1, MPI_FLOAT, MPI_ANY_SOURCE, tag, WORLD, &status);
    total += integral;
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Using collective communication

- We can take the sums in any order we wish
- The result does not depend on the order in which the sums are taken, except to within roundoff
- We can often improve performance by taking advantage of global knowledge about communication
- Instead of using point to point communication operations to accumulate the sum, use *collective* communication

```c
local_n = n/p;
float local_a = a + my_rank*local_n*h,
    local_b = local_a + local_n*h,
integral = Trap(local_a, local_b, local_n, h);
MPI_Reduce( &integral, &total, 1,
            MPI_FLOAT, MPI_SUM,
            ROOT,MPI_COMM_WORLD)
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Collective communication in MPI

- Collective operations are called by all processes within a communicator
- Broadcast: distribute data from a designated “root” process to all the others
  \[ \text{MPI\_Bcast}(\text{in}, \text{count}, \text{type}, \text{root}, \text{comm}) \]
- Reduce: combine data from all processes and return to a designated root process
  \[ \text{MPI\_Reduce}(\text{in}, \text{out}, \text{count}, \text{type}, \text{op}, \text{root}, \text{comm}) \]
- Allreduce: all processes get reduction: \text{Reduce} + \text{Bcast}
Final version

```c
int local_n = n/p;

float local_a = a + my_rank*local_n*h,
    local_b = local_a + local_n*h,
    integral = Trap(local_a, local_b, local_n, h);

MPI_Allreduce( &integral, &total, 1,
    MPI_FLOAT, MPI_SUM, WORLD)
```