CSE 160
Lecture 19
Programming with MPI
Today’s lecture

• MPI functionality
  ‣ Communicators and tags
  ‣ Collective communication
• Application: the trapezoidal rule
Programming with Message Passing

Process 0

Send(y,1)

Process 1

Recv(x)

Node 0

Node 1

Tan Nguyen

©2013 Scott B. Baden / CSE 160 / Winter 2013
Point to Point communication

```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 length**: Message length is determined by the number of elements in the `msg` array.
- **SOURCE Process ID**: The process ID of the sender, in this case, process 0.
- **Message Buffer**: The location where the received message is stored.
- **Message Tag**: The tag used to identify the message.
- **Communicator**: The communicator through which the communication takes place, in this case, `MPI_COMM_WORLD`.
- **Destination Process ID**: The process ID of the receiver, in this case, process 0.
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
```

©2013  Scott B. Baden / CSE 160 / Winter 2013
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) × # 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
Today’s lecture

• MPI functionality
  ‣ Communicators and tags
  ‣ Collective communication

• 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
  \[
  \approx \text{sum of all the trapezoids}
  \]

\[
\int_a^b f(x) \, dx
\]
Reference material

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

• A applet to carry out integration

• Code on Bang (from Pacheco hard copy text)
  Serial Code
  $PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/serial.c
  Parallel Code
  $PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/trap.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;
}
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

![Diagram of the trapezoidal rule with decomposition into sub-intervals]

©2013 Scott B. Baden / CSE 160 / Winter 2013
First version of the parallel code

int 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 (int 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);
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 (int source = 1; source < p; source++)
  MPI_Recv(&integral, 1, MPI_FLOAT,
           MPI_ANY_SOURCE, tag,
           WORLD, &status);
  total += integral;
}
Using collective communication

- 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)
```
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 + 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)
```
Today’s lecture

• MPI functionality
  ‣ Communicators and tags
  ‣ Collective communication

• Applications
  ‣ Trapezoidal rule
  ‣ Measuring communication performance
Broadcast

• The root process transmits of $m$ pieces of data to all the $p-1$ other processors
• With the linear ring algorithm this processor performs $p-1$ sends of length $m$
  ▸ Cost is $(p-1)(\alpha + \beta m)$
• Another approach is to use the hypercube algorithm, which has a logarithmic running time
Sidebar: what is a hypercube?

- A hypercube is a d-dimensional graph with $2^d$ nodes
- A 0-cube is a single node, 1-cube is a line connecting two points, 2-cube is a square, etc
- Each node has d neighbors
Properties of hypercubes

• A hypercube with p nodes has $\lg(p)$ dimensions
• *Inductive construction:* we may construct a d-cube from two (d-1) dimensional cubes
• **Diameter:** What is the maximum distance between any 2 nodes?
• **Bisection bandwidth:** How many cut edges (mincut)
Bookkeeping

• Label nodes with a binary reflected grey code
  http://www.nist.gov/dads/HTML/graycode.html

• Neighboring labels differ in exactly one bit position
  \[ 001 = 101 \oplus e_2, \quad e_2 = 100 \]
Hypercube broadcast algorithm with $p=4$

- Processor 0 is the root, sends its data to its hypercube “buddy” on processor 2 (10)
- Proc 0 & 2 send data to respective buddies
Reduction

• We may use the hypercube algorithm to perform reductions as well as broadcasts
• Another variant of reduction provides all processes with a copy of the reduced result
  \texttt{Allreduce()} 
• Equivalent to a \texttt{Reduce + Bcast}
• A clever algorithm performs an \texttt{Allreduce} in one phase rather than having perform separate reduce and broadcast phases
Allreduce

- Can take advantage of duplex connections
What we covered today

• MPI functionality
  ‣ Communicators and tags
  ‣ Collective communication
• Application: the trapezoidal rule
• Next time
  ‣ Asynchronous communication
  ‣ Measuring message passing performance