Lecture 13

MPI Applications
Basic Collectives
Asynchronous Communication
Profiling
Announcements

• No class on 2/17 → extra lecture will be held during section time slot on 2/25
Today’s lecture

• Applications
  ‣ Computing the area under a curve
  ‣ Stencil Methods

• MPI
  ‣ Collective communication
  ‣ Asynchronous communication

• Profiling
The trapezoidal rule

- Use the trapezoidal rule to numerically approximate the definite integral
How the trapezoidal rule works

- Divide the interval \([a,b]\) into \(n\) segments of size \(h=1/n\)
- Approximate the area under an interval using a trapezoid
- 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
Serial code (Following Pacheco)

main() { // $PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/serial.c

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

- General strategy is similar to the threads implementation
- Split 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

\[ \int_{a}^{b} f(x) \, dx \]
First version of the parallel code

// $PUB/Examples/MPI/Pacheco/ppmpi_c/chap04/trap.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, h);

// Sum integrals calculated by all processes

if (my_rank == ROOT) {
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
          MPI_COMM_WORLD, &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1, MPI_FLOAT, ROOT, tag,
      MPI_COMM_WORLD);
Playing the wild card

- We can take the sums in any order we wish
- Result does not depend on the ordering of the sums, except to within roundoff
- We use a linear time algorithm to accumulate contributions, but there are faster ones

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

• Instead of using point-to-point communication operations to accumulate the sum, use **collective** communication

• Often improves performance by taking advantage of global knowledge about 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
  MPI_Bcast(in, count, type, root, comm)

• Reduce: combine data from all processes and return to a designated root process
  MPI_Reduce(in, out, count, type, op, root, comm)
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

- Neighboring labels differ in exactly one bit position
  \[ 001 = 101 \otimes e_2, \quad e_2 = 100 \]

\[ e_2 = 100 \]
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 Allreduce()

• Equivalent to a Reduce + Bcast
• A clever algorithm performs an Allreduce in one phase rather than having perform separate reduce and broadcast phases
Allreduce

- What’s the clever algorithm?
Relevance to multicore computing

• Recall how we accumulated quantities with threads
• In the future, multicore processors may be NUMA
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)
Word problems
K-ary d-cubes

• **Definition**- A k-ary d-cube is an interconnection network with $k^d$ nodes
  ‣ There are $k$ nodes along all $d$ axis
  ‣ End around connections
  ‣ A generalization of a mesh and hypercube
  ‣ The hypercube is a special case with $k=2$
K-ary d-cubes

• Derive the diameter, number of links, & bisection bandwidth of a k-ary d-cube with p nodes

• Diameter:

• Bisection Bandwidth: $2k^{d-1}$
Hamiltonians

- A Hamiltonian cycle is a path in an undirected graph that visits each node exactly once.
- Any Hamiltonian circuit on a labeled hypercube defines a Gray code [Skeina].
- Map a 1D ring onto a hypercube, mesh, k-ary d-cube.

www.cs.sunysb.edu/~skiena/combinatorica/animations/ham.html
http://mathworld.wolfram.com/HamiltonianCycle.html
Stencil methods
Image smoothing

Original  15 iter  50 iter  100 iter  300 iter  1000 iter
Image smoothing algorithm

• Repeat as many times as needed

for (i,j) in 0:N-1 x 0:N-1
    
    u'[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1])/4

    u = u'

©2010 Scott B. Baden / CSE 160 / Winter 2010
Parallel Implementation

- Partition data into parts, assigning each to a unique processor
- Dependences involving values found on neighboring processes
- Buffer copies of neighboring values using “ghost” cells
Ghost cells in higher dimensions

- Ghost cells surround each local subproblem
- Non-contiguous data
- Inefficient to communicate individual values
- Use non-blocking communication
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
    
    \[ \text{lRecv( ), lSend( )} \]
  ‣ Phase 2: synchronize
    
    \[ \text{Wait( )} \]
  ‣ Perform unrelated computations between the two phases

• Building a blocking call
  
  \[ \text{Recv( )} = \text{lRecv( )} + \text{Wait( )} \]
Restrictions

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

ISend(data,destination)
Wait() on ISend()
Use the data

• Each pending IRecv() must have a distinct buffer
Immediate mode send and receive

• Immediate return does not indicate completion
• Must synchronization with a Wait()
• An extra request argument
  
  MPI_Request request;
  MPI_Irecv(buf, count, type, src, tag, comm, &request)
  MPI_Wait(&request, &status)

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

• Immediate Send
  
  MPI_Isend(buf, count, type, dest, tag, comm, &request)
### Overlap behavior

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

- A message buffer may not be accessed between an IRecv( ) (or ISend( )) and its accompanying wait( )
- Each pending IRecv() must have a distinct buffer
Managing ghost cells

- Post Ireceives( ) for all neighbors
- Send data to neighbors
- Wait for completion
Correctness and fairness

- When there are multiple outstanding iRecvs, MPI doesn’t say how incoming messages are matched…
- Or even if the process is fair

1. Iteration 1: $1 \rightarrow 2 \& 0 \quad 0 \rightarrow 1 \quad (0 \rightarrow 2) \quad 2 \rightarrow 0 \& 1$
2. $1$ begins iteration 2: $1 \rightarrow 2$
3. $0 \rightarrow 2$ (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
```
Summary

• Applications
  ‣ Computing the area under a curve
  ‣ Stencil Methods

• MPI
  ‣ Collective communication
  ‣ Asynchronous communication
Sensitivity to cache interference

- 3D Jacobi’s method
Summary

• Studied 2 applications motivating different MPI primitives
  ‣ Area under the curve: collective communication (more collectives coming)
  ‣ Stencil methods (image smoother): immediate mode communication (asynchronous, non-blocking)

• Profiling to find program hot spots

• Next time
  ‣ Under the hood of MPI