Lecture 4

Writing parallel programs with MPI
Measuring performance
Announcements

• Wednesday’s office hour moved to 1.30
• A new version of Ring (Ring_new) that handles linear sequences of message lengths
• Project proposals due on Tuesday (10/10)
Microbenchmark to determine communication cost model parameters

- Communication cost model:
  \[ \text{transfer time} = \alpha + \beta n \]
  \[ \alpha = \text{message startup time} \]
  \[ \beta = \text{inverse peak bandwidth} \]
Communication Bandwidth (off node) on Blue Horizon

390 MB/sec
N = 4MB

n_{1/2} \sim 96KB
Communication times

Blue Horizon

usec

Bytes

$10^0$ $10^1$ $10^2$ $10^3$ $10^4$ $10^5$
The Ring program

- We configure the processors in a logical ring and pass messages around the ring multiple times
- Assume there are \( p \) processors
- Neighbors of processor \( k \) are
  - \((k + 1) \mod p\)
  - \((k + p -1) \mod p\)
Measurement technique with Ring

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:

```c
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:

```c
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);
```
Reporting and Displaying Performance

• Give the viewer sufficient information to…
  – Draw their own conclusions
  – Reproduce your results

• Tabulate and display the results fairly
  – Avoid misleading techniques
  – See the Bailey paper for examples of how not to display and report performance data
Challenges to measuring performance

• Reproducibility
  – Transient system operating conditions
  – Differing systems or program configuration

• Measurements are imprecise
  – “Heisenberg uncertainty principle:” measurement technique may affect performance
  – Overheads and inaccuracy

• Explain anomalous behavior, but ignore anomalies that are not significant
Complications

• Cost of measuring a full run is prohibitive
  – Ignore startup code if you plan to run for a much longer time in production

• Transient behavior
  – Repeat your measurements
  – “Warm up” the coded before collecting measurements
  – Ignore outliers unless their behavior is important to you
  – Average time, maximum time, minimum time?
Measurement collection

• Report the *best* timings
  ▶ Repeat results 3 to 5 times until at least 2 measures agree to within... 5%, 10%
  ▶ Report the minimum time
• Also report outliers
• A scatter plot or error bar can be useful
Timing collection

• Measures of time
  ▶ Elapsed, or “wall clock” time
  ▶ CPU time = system + user time
  ▶ Overhead, resolution, and quantization effects

• Measurement tools
  ▶ Unix time command does a reasonable job for long-running programs
  ▶ Hardware performance monitors
  ▶ System clocks
    • Often platform dependent, especially library routines
    • MPI provides MPI_Wtime( ), elapsed or “wall clock” time
Qualifying measurements

• Specify version and options
  – Compiler
  – Operating system
  – Numerical libraries

• Establish appropriate operating conditions
  – Program inputs
  – System environment variables
  – Dedicated system access

uname -a
Linux valkyrie.ucsd.edu 2.6.9-5.0.5.ELsmp #1 SMP Wed Apr 20 00:16:40 BST 2005 i686 i686 i386 GNU/Linux

g++ -v
Reading specs from /usr/lib/gcc/i386-redhat-linux/3.4.5/specs
Configured with: ....
host=i386-redhat-linux
Thread model: posix
gcc version 3.4.5 20051201 (Red Hat 3.4.5-2)
A first application

• Compute a numerical approximation to the definite integral

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

using the trapezoidal rule
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

\[a + i\times h \quad a + (i+1)\times h\]
Reference material

• For a discussion of the trapezoidal rule
  http://metric.ma.ic.ac.uk/integration/techniques/definite/numerical-methods/trapezoidal-rule

• A applet to carry out integration

• Code (from Pacheco hard copy text)
  PUB = /export/home/cs260x-public
  Serial Code
    PUB/Pacheco/ppmpi_c/chap04/serial.c
  Parallel Code
    PUB/Pacheco/ppmpi_c/chap04/trap.c
Serial code (Following Pacheco)

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;
}
The parallel algorithm

• 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; // Number of 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);

if (my_rank == 0) { // Sum the integrals calculated by all the processes
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag, WORLD, &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, WORLD);
```
Improvements

• The result does not depend on the order in which the sums are taken

• 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;
}
```
Collective communication

• 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
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
  \[ \text{MPI\_Bcast(in, count, type, root, comm)} \]

• Reduce: combine data from all processes and return to a designated root process
  \[ \text{MPI\_Reduce(in, out, count, type, op, root, comm)} \]
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
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

• Use variant of reduction

  \texttt{Allreduce()}  

• Everyone obtains a copy of the reduced result

• Equivalent to a \texttt{Reduce()} + \texttt{Bcast()}

• A clever algorithm performs an \texttt{Allreduce} in one phase rather than having perform separate reduce and broadcast phases
Improved parallel code

```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, count=1,
               MPI_FLOAT, MPI_SUM, WORLD)
```
Iterative methods for solving systems of equations
Mesh based methods

- Many physical problems are simulated on a uniform mesh in 1, 2 or 3 dimensions.
- Values are defined on a discrete set of points.
- A mapping from ordered pairs to physical observables like temperature and pressure.
- One application: differential equations.
Differential equations

• A **differential equation** is a set of equations involving derivatives of a function (or functions), and specifies a solution to be determined under certain constraints

• Constraints often specify **boundary conditions** or **initial values** that the solution must satisfy

• When the functions have multiple variables we have a **Partial Differential Equation (PDE)**

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{within a square box, } x,y \in [0,1] \\
u(x,y) = \sin(x)\sin(y) \quad \text{on } \partial \Omega, \text{ perimeter of the box}
\]

• When the functions have a single variable we have an **Ordinary Differential Equation (ODE)**

\[-u''(x) = f(x), x \in [0,1], u(0) = a, u(1) = b\]
Solving an ODE with a discrete approximation

• Solve the ODE
  \[-u''(x) = f(x), \ x \in [0, 1]\]

• Define \( u_{i,j} = u(i \times h, j \times h) \) at points
  \[ x = i \times h, \ y = j \times h, \ h = 1/(N-1) \]

• Approximate the derivatives
  \[ u'' \approx (u(x+h) - 2u(x) + u(x-h))/h^2 \]

• Obtain the system of equations
  \[ (u_{i-1} - 2u_i + u_{i+1})/h^2 = f_i \quad i \in 1..n-2 \]
Iterative solution

• Rewrite the system of equations
  
  \[-u_{i-1} + 2u_i - u_{i+1}]/h^2 = f_i , i \in 1..n-1

• It can be shown that the following \textit{Gauss-Seidel} algorithm will arrive at the solution …

• …. assuming an initial guess for the \( u_i \)

  \[
  \text{Repeat until the result is satisfactory} \\
  \text{for } i = 1 : N-1 \\
  u_i = (u_{i+1} + u_{i-1} + h^2 f_i)/2 \\
  \text{end for} \\
  \text{end Repeat}
  \]
Convergence

- Convergence is slow
- We reach the desired precision in $O(N^2)$ iterations
Estimating the error

• How do we know when the answer is “good enough?”
  • The computed solution has reached a reasonable approximation to the exact solution
  • We validate the computed solution in the field, i.e. wet lab experimentation
• But we often don’t know the exact solution, and must estimate the error
Using the residual to estimate the error

- Recall the equations
  \[ (-u_{i-1} + 2u_i - u_{i+1})/h^2 = f_i , \ i \in 1..n-1 \ [Au = f] \]

- Define the residual \( r_i \):
  \[ r_i = (-u_{i-1} + 2u_i - u_{i+1})/h^2 - f_i , \ i \in 1..n-1 \]

- Thus, our computed solution is correct when \( r_i = 0 \)

- We can obtain a good estimate of the error by finding the maximum \( r_i \) \( \forall i \)

- Another possibility is to take the root mean square (L2 norm)
  \[ \sqrt{\sum_i r_i^2} \]
Parallel implementation

- The first step is to partition the data
- We partition the data into intervals, assigning each to a unique processor
- Many to one mappings may also be useful
Data dependences

- Each interval depends on two endpoint values found on neighboring processes.
- We add “overlap” or “ghost” cells to hold a copy of the off-processor value.
Dependences

- Our attempt to parallelize the algorithm fails, since there are **loop carried dependences**
- The value of $u[i]$ computed in iteration $i$ depends on $u[i]$ computed in iteration $i-1$

```plaintext
for i = 1 : N-1
    u[i] = (u[i-1]+u[i+1] +h*h*f[i])/2
end for
```
Parallel implementation

• We avoid the dependences by renaming the LHS of the assignment
• Two arrays $u$ and $u_{\text{new}}$
• Update $u_{\text{new}}$
  
  for $i = 1: N-1$
  
  $u_{\text{new}}[i] = (u[i-1] + u[i+1] + h^2 f[i])/2$
  
  end for

• Then swap $u$ and $u_{\text{new}}$
Tradeoffs

• We can now parallelize the algorithm, since there are no longer any loop carried dependencies within the $i$ loop
• However, this change reduces the convergence rate by about a factor of two
• We have doubled the amount of work we need to perform in exchange for being able to realize parallel speedups
• This kind of tradeoff is common, but we can do better
Convergence check

• Each process computes the error for its assigned part of the problem
• We need a global error so that we compute a result that is consistent with the single processor implementation
• This requires collective communication: a reduction
• In MPI we use **Allreduce**, which leaves a copy the result in the memory of all processes