Lecture 8

MPI
Communicators
Collective communication
Matrix multiplication
Announcements

• Project proposal: in class next Thursday
• 1-2 page written proposal
• 5-7 minute in-class presentation
Today’s lecture

• MPI Applications
A first MPI Application
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.
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
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
Serial code (Following Pacheco)

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

float Trap(float a, float b, int n) {

    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;
    return integral;
}
```

©2010 Scott B. Baden / CSE 262 / Spring ’11
First version of the parallel code

\[
\text{local\_n} = \frac{n}{p}; \quad \text{// \# trapezoids; assume p divides n evenly}
\]

\[
\begin{align*}
\text{float local\_a} &= a + \text{my\_rank}\times\text{local\_n}\times h, \\
\text{local\_b} &= \text{local\_a} + \text{local\_n}\times h, \\
\text{integral} &= \text{Trap(local\_a, local\_b, local\_n)};
\end{align*}
\]

\[
\text{if (my\_rank == ROOT) \{ \quad \text{// Sum the integrals calculated by} \\
\text{\quad all processes}\}
\]

\[
\begin{align*}
\text{total} &= \text{integral}; \\
\text{for (source = 1; source < p; source++)} \{ \\
\quad \text{MPI\_Recv(&integral, 1, MPI\_FLOAT, MPI\_ANY\_SOURCE,} \\
\text{tag, WORLD, &status);} \\
\quad \text{total += integral}; \\
\}\}
\]

\[
\text{else} \\
\quad \text{MPI\_Send(&integral, 1, MPI\_FLOAT, ROOT, tag, WORLD)};
\]

©2010 Scott B. Baden / CSE 262 / Spring '11
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

\[
local_n = \frac{n}{p};
\]

```c
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)
```

©2010 Scott B. Baden / CSE 262 / Spring '11
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
  
  \texttt{MPI\_Bcast(in, count, type, root, comm)}

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

• Allreduce: all processes get reduction: \texttt{Reduce + Bcast}

• \texttt{www-cse.ucsd.edu/classes/fa10/cse260/Lectures/Lec14.pdf}
• \texttt{www-cse.ucsd.edu/classes/fa10/cse260/Lectures/Lec17.pdf}
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)
Communication performance

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

• Let the message have a length $n$

• Simplest cost model: $\alpha + \beta^{-1}_{\infty} \ n \ [\text{message length} = \ n]$

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

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

  $n = \text{message length}$

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

Short messages: $\alpha \gg \beta^{-1}\infty n$

$\alpha = 3.2 \mu\text{sec}$

Long Messages: $\beta^{-1}\infty n \gg \alpha$

$N_{1/2} \approx 20 \text{ KB}$

$N = 8\text{ MB}$

1.12 Gbps
Short and intermediate message lengths

Triton

![Graph showing message length vs. time in usec for Triton.]

![Graph showing message length vs. gigabytes/second for Triton.]

©2010 Scott B. Baden / CSE 262 / Spring '11
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
Parallel print function
Parallel print function

• Debugging output can be hard to sort out on the screen

• Many messages say the same thing
  
  Process 0 is alive!
  Process 1 is alive!
  ...
  Process 15 is alive!

• Compare with

  Processes[0–15] are alive!

• Parallel print facility
  
  http://www.llnl.gov/CASC/ppf
Summary of capabilities

- Compact format list sets of nodes with common output
  
  ```c
  PPF_Print( MPI_COMM_WORLD, "Hello world" );
  0–3: Hello world
  ```

- %N specifier generates process ID information
  
  ```c
  PPF_Print( MPI_COMM_WORLD, "Message from %N\n" );
  Message from 0–3
  ```

- Lists of nodes
  
  ```c
  PPF_Print(MPI_COMM_WORLD,
            (myrank % 2)
            ? "[%N] Hello from the odd numbered nodes!\n"
            : "[%N] Hello from the even numbered nodes!\n")
  [0,2] Hello from the even numbered nodes!
  [1,3] Hello from the odd numbered nodes!
  ```
Practical matters

• Installed in \$(PUB)/lib/PPF
• Use a special version of the arch file called
  \texttt{arch.intel.mpi.ppf}
• Each module that uses the facility must
  \texttt{#include "ptools_ppf.h"}
• Look in \$(PUB)/Examples/MPI/PPF for example programs
  \texttt{ppfexample_cpp.C} and \texttt{test_print.c}
Parallel Matrix Multiplication
Matrix Multiplication

• An important core operation in many numerical algorithms
• Given two conforming matrices $A$ and $B$, form the matrix product $A \times B$
  
  $A$ is $m \times n$
  $B$ is $n \times p$

• Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix
• See Demmel
  
  www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html
Simplest Serial Algorithm

“$i j k$”

\[
\begin{align*}
&\text{for } i := 0 \text{ to } n-1 \\
&\quad \text{for } j := 0 \text{ to } n-1 \\
&\quad \quad \text{for } k := 0 \text{ to } n-1 \\
&\quad \quad \quad C[i,j] += A[i,k] \times B[k,j]
\end{align*}
\]
Parallel matrix multiplication

• Assume \( p \) is a perfect square
• Each processor gets an \( \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \) chunk of data
• Organize processors into rows and columns
• Process rank is an ordered pair of integers
• Assume that we have an efficient serial matrix multiply

\[
\begin{array}{ccc}
p(0,0) & p(0,1) & p(0,2) \\
p(1,0) & p(1,1) & p(1,2) \\
p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\]
A simple parallel algorithm

• Apply the basic algorithm but treat each element $A[i,j]$ as a block rather than a single element.

• Thus, $A[i,k] \times B[k,j]$ is matrix multiply in $C[i, j] += A[i, k] \times B[k, j]$.
Cost

- Each processor performs $n^3/p$ multiply-adds
- But a significant amount of communication is needed to collect a row and a column of data onto each processor
- Each processor broadcasts a chunk of data of size $n^2/p$ within a row and a column of $\sqrt{p}$ processors
- Disruptive - distributes all the data in one big step
- High memory overhead
  - needs $2\sqrt{p}$ times the storage needed to hold A & B

©2010 Scott B. Baden / CSE 262 / Spring '11
Observation

• Each processor multiplies 2 skinny matrices: \( n/\sqrt{p} \times n \)
• But we can form the same product by computing \( \sqrt{p} \) separate matrix multiplies involving \( n/\sqrt{p} \times n/\sqrt{p} \) matrices and accumulating partial results

\[
\text{for } k := 0 \text{ to } n - 1 \text{ } \\
C[i, j] += A[i, k] \times B[k, j];
\]
A more efficient algorithm

- Take advantage of the organization of the processors into rows and columns
- Move data incrementally in $\sqrt{p}$ phases
- Circulate each chunk of data among processors within a row or column
- In effect we are using a ring broadcast algorithm
- Buffering requirements are $O(1)$
Canon’s algorithm

• Based on the above approach
• A slight reformulation to make things work out
• Consider iteration $i=1$, $j=2$:

Canon’s algorithm

\[ C_{1,2} = A_{1,0} \times B_{0,2} + A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \]

- We want \( A_{1,0} \) and \( B_{0,2} \) to reside on the same processor initially.
- Shift rows and columns so the next pair of values \( A_{1,1} \) and \( B_{1,2} \) line up.
- And so on with \( A_{1,2} \) and \( B_{2,2} \).
Skewing the matrices

\[ C_{1,2} = A_{1,0} \times B_{0,2} + A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \]

- We first skew the matrices so that everything lines up
- Shift each row \( i \) by \( i \) columns to the left using sends and receives
- Communication wraps around
- Do the same for each column
Shift and multiply

\[ C_{1,2} = A_{1,0} * B_{0,2} + A_{1,1} * B_{1,2} + A_{1,2} * B_{2,2} \]

- Takes \( \sqrt{p} \) steps
- Circularly shift
  - each row by 1 column to the left
  - each column by 1 row to the left
- Each processor forms the product of the two local matrices adding into the accumulated sum
Cost of Cannon’s Algorithm

forall i = 0 to √p -1
    CShift-left A[i, :] by i // T = α + β n^2/p
for all j = 0 to √p -1
    Cshift-up B[:, j] by j // T = α + β n^2/p
for k = 0 to √p -1
    forall i = 0 to √p -1 and j = 0 to √p -1
        CShift-leftA[i, :] by 1 // T = α + β n^2/p
        Cshift-up B[:, j] by 1 // T = α + β n^2/p
end forall
end for

T_P = 2n^3/p + 4√p (α + β n^2/p)
E_P = T_1 / T_P ≈ (1 + 2α p^{3/2} / n^3 + 2β√p/n))^{-1}
≈ (1 + O(√p/n))^{-1}

E_P → 1 as (n^2/√p) grows
Implementation
Communication domains

- A name space, subsets of processes that may communicate
- Define communicators that reflect the communication structure along rows and columns of the processor geometry
- Messages remain within their communicator
- A processor may be a member of more than one communicator
Establishing row communicators

```c
MPI_Comm rowComm;
MPI_Comm_split(MPI_COMM_WORLD, myRank / sqrt(P), myRank, &rowComm);
MPI_Comm_rank(rowComm,&myRow);
```

- Group processors by row (key)
- Each process computes a key based on its rank: `key = myid \div \sqrt{P}`
- Each process has a rank relative to the new communicator
- Rank applies only to the respective communicator
- Ordered according to `myRank`
MPI support

• Collective: `MPI_Comm_split(
    MPI_Comm comm,
    int splitKey,int rankKey,
    MPI_Comm* newComm);

• Each process obtains a new communicator, `newComm`, enabling it to communicate w/other processes having the same `splitKey` value

• Ranks are assigned arbitrarily among processes sharing the same `rankKey` value

• May exclude a process by passing the constant `MPI_UNDEFINED` as the `splitKey`
Circular shift

- Communication with columns (and rows)