Lecture 13

Writing parallel programs with MPI
Matrix Multiplication
Basic Collectives
Managing communicators
Announcements

• Extra lecture Friday 4p to 5.20p, room 2154
• A4 posted
  ♦ Cannon’s matrix multiplication on Trestles
• Extra office hours today: 2.30 to 5pm
Today’s lecture

• Writing message passing programs with MPI
• Applications
  ♦ Trapezoidal rule
  ♦ Cannon’s Matrix Multiplication Algorithm
• Managing communicators
• Basic collectives
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
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
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,
                  &status);
        total += integral;
    }
} else
    MPI_Send(&integral, 1, MPI_FLOAT, ROOT, tag, WORLD);
```

©2012 Scott B. Baden /CSE 260/
Winter 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)
```

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)

• Allreduce: all processes get reduction: \texttt{Reduce} + \texttt{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)
```
Collective Communication
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
  \[001 = 101 \otimes e_2, \quad e_2 = 100\]
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

\text{Allreduce}(\ )

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

- Can take advantage of duplex connections
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/~dемmель/cs267_Spr99/Lectures/Lect02.html
Simplest Serial Algorithm

“ijk”

for i := 0 to n-1
    for j := 0 to n-1
        for k := 0 to n-1
            C[i,j] += A[i,k] * B[k,j]

Parallel matrix multiplication

- Assume $p$ is a perfect square
- Each processor gets an $n/\sqrt{p} \times n/\sqrt{p}$ 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 ($\text{dgemm}, \text{sgemm}$)

\[
\begin{array}{ccc}
  p(0,0) & p(0,1) & p(0,2) \\
  p(1,0) & p(1,1) & p(1,2) \\
  p(2,0) & p(2,1) & p(2,2) \\
\end{array}
\]
**Canon’s algorithm**

- 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
- Consider iteration $i=1, j=2$:

Canon’s algorithm


- 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

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

- 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


- 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 \sqrt{p} -1
    CShift-left A[i; :] by i  // T= \alpha+\beta n^2/p
forall  j=0 to \sqrt{p} -1
    Cshift-up B[: , j] by j  // T= \alpha+\beta n^2/p
for  k=0 to \sqrt{p} -1
    forall  i=0 to \sqrt{p} -1 and j=0 to \sqrt{p} -1
        CShift-left A[i; :] by 1  // T= \alpha+\beta n^2/p
        Cshift-up B[: , j] by 1  // T= \alpha+\beta n^2/p
end forall
end for

T_p = 2n^3/p + 2(\alpha(1+\sqrt{p}) + \beta n^2/(1+\sqrt{p})/p)
E_p = T_1/(pT_p) = ( 1 + \alpha p^{3/2}/n^3 + \beta \sqrt{p}/n))^{-1}
    \approx ( 1 + O(\sqrt{p}/n))^{-1}
E_p \rightarrow 1 as (n/\sqrt{p}) grows [sqrt of data / processor]
Implementation
Communication domains

- Cannon’s algorithm shifts data along rows and columns of processors
- MPI provides communicators for grouping processors, reflect the communication structure of the algorithm
- An MPI communicator is a name space, a subset of processes that communicate
- Messages remain within their communicator
- A process may be a member of more than one communicator
Establishing row communicators

- Create a communicator for each row and column
- By Row

$$\text{key} = \text{myRank div } \sqrt{P}$$
Creating the communicators

MPI_Comm rowComm;
MPI_Comm_split( MPI_COMM_WORLD, myRank / √P, myRank, &rowComm);
MPI_Comm_rank(rowComm,&myRow);

- Each process obtains a new communicator
- Each process’ rank relative to the new communicator
- Rank applies only to the respective communicator
- Ordered according to myRank
More on `Comm_split`

```c
MPI_Comm_split(MPI_Comm comm, int splitKey,
    int rankKey, MPI_Comm* newComm)
```

- Ranks assigned arbitrarily among processes sharing the same `rankKey` value
- May exclude a process by passing the constant `MPI_UNDEFINED` as the `splitKey`
- Return a special `MPI_COMM_NULL` communicator
- If a process is a member of several communicators, it will have a rank within each one
Circular shift

• Communication with columns (and rows
Parallel print function
Parallel print function

- Debug 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
  
  ```
  PPF_Print( MPI_COMM_WORLD, "Hello world" );
  0–3: Hello world
  ```

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

- Lists of nodes
  
  ```
  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
  
  arch.intel.mpi.ppf  [arch.intel-mkl.mpi.ppf]

• Each module that uses the facility must
  
  #include “ptools_ppf.h”

• Look in $(PUB)/Examples/MPI/PPF for example programs
  ppfexample_cpp.C  and  test_print.c