CSE 160
Lecture 15

Message Passing
Announcements
Today’s lecture

• Message passing
• The Message Passing Interface - MPI
• A first MPI Application – The Trapezoidal Rule
Multiprocessor organization

- Recall that with shared memory, the hardware automatically performs the global to local mapping using address translation mechanisms.
- 2 types, depends on uniformity of memory access times:
  - **UMA**: Uniform Memory Access time
    Also called a Symmetric Multiprocessor (SMP)
  - **NUMA**: Non-Uniform Memory Access time
Architectures without shared memory

- Each core has direct access to local memory only
- Send and receive *messages* to obtain copies of data from other nodes
- We call this a *shared nothing* architecture, or a *multicomputer*
- Similarity to NUMA
Programming with Message Passing

- Programs execute as a set of $P$ processes (user specifies $P$)
- Each process assumed to run on a different core
  - Usually initialized with the same code, but has private state
    $\text{SPMD} = \text{“Same Program Multiple Data”}$
  - Communicates with other processes by sending and receiving messages
  - Executes instructions at its own rate according to its $rank$ ($0:P-1$) and the messages it sends and receives
- Program execution is often called “bulk synchronous” or “loosely synchronous”
Bulk Synchronous Execution Model

- A process is either communicating or computing
- Generally, all processors are performing the same activity at the same time
- There can be instances when some are computing and some are communicating
- Pathological cases, when workloads aren’t well balanced
Message passing

• There are two kinds of communication patterns

• *Point-to-point* communication: a single pair of communicating processes copy data between address space

• *Collective communication*: all the processors participate, possibly exchanging information
Point-to-Point communication

- Messages are like email; to send one, we specify
  - A destination
  - A message body (can be empty)
- To receive a message we need similar information, including a receptacle to hold the incoming data
- Requires a sender and an explicit recipient that must be aware of one another
- Message passing performs two events
  - Memory to memory block copy
  - Synchronization signal at recipient: “Data has arrived”
Send and Recv

- Primitives that implement Pt to Pt communication
- When `Send()` returns, the message is “in transit”
  - A return doesn’t tell us if the message has been received
  - The data is somewhere in the system
  - Safe to overwrite the buffer
- `Receive()` blocks until the message has been received
  - Safe to use the data in the buffer

```
Send(y, 1)  Recv(x)
```

©2013 Scott B. Baden / CSE 160 / Fall 2013
Causality

• If a process sends multiple messages to the same destination, then the messages will be received in the order sent.

• If different processes send messages to the same destination, the order of receipt isn’t defined across sources.
Today’s lecture

• Message passing
• The Message Passing Interface - MPI
• A first MPI Application – The Trapezoidal Rule
MPI

• We’ll program with a library called **MPI**
  “Message Passing Interface”
  ‣ 125 routines in MPI-1
  ‣ 7 minimal routines needed by every MPI program
    • start, end, and query MPI execution state (4)
    • non-blocking point-to-point message passing (3)

• Reference material: see

• Callable from C, C++, Fortran, etc.

• All major vendors support MPI, but implementations differ in quality
Functionality we’ll will cover today

• Point-to-point communication
• Message Filtering
• Communicators and Tags
• Application: the trapezoidal rule
• Collective Communication
A first MPI program: “hello world”

#include "mpi.h"

int main(int argc, char **argv ){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("Hello, world! I am process %d of %d.\n", rank, size);
    MPI_Finalize();
    return(0);
}
MPI’s minimal interface

- Opening and closing MPI
  - MPI_Init and MPI_Finalize

- Query functions
  - `MPI_Comm_size()` = # processes
  - `MPI_Comm_rank()` = this process’ rank

- Point-to-point communication
  - Simplest form of communication
  - Send a message to another process
    - `MPI_Isend()` = Isend+Wait
    - `MPI_Send()` = Isend+Wait
  - Receive a message from another process
    - `MPI_Irecv()` = Irecv +Wait
    - `MPI_Recv()` = Irecv +Wait
  - Wait on an incoming message: `MPI_Wait()`
Point to Point Communication

Send(y,1)  
Recv(x)

Node 0
P0  P1  P2  P3 ...

Node 1
P4  P5  P6  P7 ...

Tan Nguyen
Point-to-point messages

• To send a message we need
  ‣ A destination
  ‣ A “type”
  ‣ A message body (can be empty)
  ‣ A context (called a “communicator” in MPI)

• To receive a message we need similar information, including a place to hold the incoming data

• We can filter messages, enabling us organize message passing activity
```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);
```
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 \textit{tag}:
  ‣ Receiving process can use this information to organize or \textit{filter} messages
  ‣ \texttt{MPI\_ANY\_TAG} inhibits tag filtering
Message status

- An MPI_Status variable is a struct that contains the sending processor and the message tag
- This information is useful when we aren’t filtering 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
```
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

• Message passing
• The Message Passing Interface - MPI
• A first MPI 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 
  \(\approx \) 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
First version of the parallel code

```c
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);
```

©2013  Scott B. Baden / CSE 160 / Fall 2013
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
  
  ```
  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: **Reduce + Bcast**
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)
What we covered today

• Message passing concepts
• A practical interface - MPI
• Next time
  ‣ Asynchronous communication
  ‣ More collective communication primitives
  ‣ New Applications