Lecture 12

Message Passing
Announcements

• No class on 2/17 → extra lecture will be held during section time slot on 2/25
• Midterm return
Q2

( 1) #pragma omp parallel shared(err, u, unew, MaxIter), private(s)
( 2) for (s = 0; s < MaxIter; s++) {
  (3)   err = 0;
  (4)   for (i = 1; i < N-1; i++)
  (5)     unew[i] = (u[i-1] + u[i+1])/2.0;
  (6)   for (i = 1; i < N-1; i++) {
    (7)     double delta = fabs(u[i] - unew[i]);
    (8-9)    if (delta > err) err = delta;
  (10) }
(11-12) if ((s > 0) && ( err < epsilon )) break;
(13)   Swap u↔ unew /
(14) } /* End of iteration loop. */
Q2 – with corrections

( 1) #pragma omp parallel shared(err, u, unew, MaxIter), private(s)
( 2) for (s = 0; s < MaxIter; s++) {
  ( 3) err = 0;
  ( 4) for (i = 1; i < N-1; i++)
  ( 5)   unew[i] = (u[i-1] + u[i+1])/2.0;
  ( 6) for (i = 1; i < N-1; i++) {
    ( 7) double delta = fabs(u[i] - unew[i]);
    ( 8-9) if (delta > err) err = delta;
  (10) }
  (11) if ((s > 0) && (err < epsilon))
  (12)   break;
  (13) Swap u↔unew
  (14) } /* End of iteration loop. */
Q2 – Using a ‘for’ clause

( 1) #pragma omp parallel shared(err, u, unew, MaxIter), private(s)

( 2) for (s = 0; s < MaxIter; s++) {
    err = 0;

( 4)    for (i = 1; i < N-1; i++)

( 5)        unew[i] = (u[i-1] + u[i+1])/2.0;

( 6)    for (i = 1; i < N-1; i++) {

( 7)        double delta = fabs(u[i] - unew[i]);

( 8-9)       if (delta > err) err = delta;

(10) }

(11) if ((s > 0) && ( err < epsilon ))

(12)    break;

(13) Swap u← unew

(14) } /* End of iteration loop. */
Today’s lecture

- Message passing
- MPI
Programming with Message Passing

• **Primary model for implementing high performance applications**
• **Programs execute as a set of P processes**
  ‣ We specify P when we run the program
• **Each process**
  ‣ **Is initialized with the same code, but has private state**
    SPMD = “Same Program Multiple Data”
  ‣ Has an associated *rank*, a unique integer in the range 0:P-1
  ‣ Assume it’s assigned a different physical processor
• **The sequence of instructions a process executes depends on its rank and the messages it sends and receives**
• **Program execution is often called “bulk synchronous” or “loosely synchronous”**
Message Passing

• 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 on receiving end: “Data arrived”
A minimal interface

• Query functions
  \( \text{nproc}(\ ) = \# \text{ processors} \)
  \( \text{myRank}(\ ) = \text{this process’s rank} \)

• \textit{Point-to-point} communication
  ‣ Simplest form of communication
  ‣ Send a message to another process
    \( \text{Send(Object, Destination process ID)} \)
  ‣ Receive a message from another process
    \( \text{Receive(Object)} \)
    \( \text{Receive(Source process, Object)} \)
Send and Recv

- **When** `Send( )` returns, the message is “in transit”
  - A return doesn’t tell us if the message has been received
  - 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
An unsafe program

- A Send() may or may not complete…
- … before a Recv() has been posted
- “May or may not” depends on the implementation

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (x,1)</td>
<td>Send(y,0)</td>
</tr>
<tr>
<td>Recv (y,1)</td>
<td>Recv(x,0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (x,1)</td>
<td>Recv(x,0)</td>
</tr>
<tr>
<td>Recv (y,1)</td>
<td>Send(y,0)</td>
</tr>
</tbody>
</table>
Buffering

- Where does the data go when you send it?
- It might be buffered
- Preferable to avoid the extra copy
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 processes.
No guarantee with different senders

- If different processes send messages to the same destination
  - The order of receipt is defined from a single source
  - The order of receipt is not defined across multiple sources
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 http://www-cse.ucsd.edu/users/baden/Doc/mpi.html

• 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
- Communicators
- Data types
- Tags
- Non-blocking communication
- Message Filtering
A first MPI program: “hello world”

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv);
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

©2010 Scott B. Baden / CSE 160 / Winter 2010
Query functions

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("I am process %d of %d.\n", rank, size);
    MPI_Finalize();
}
Sending and receiving messages

• MPI provides a rich collection of routines to move data between address spaces

• A single pair of communicating processes use *point-to-point* communication

• Later on we’ll cover *collective communication*, when all the processors communicate together

• In point-to-point message passing we can filter messages in various ways

• This allows us to organize message passing activity conveniently
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 receptacle to hold the incoming data
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
Send and Recv

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);
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 filter messages
  - \texttt{MPI\_ANY\_TAG} inhibits screening.
MPI Datatypes

• MPI messages have a specified length
• The unit depends on the type of the data
  ‣ Length in bytes is sizeof(type) × # elements
  ‣ We don’t specify the as the # bytes
• 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
Message status

• An MPI_Status variable is a struct that contains the sending processor and the message tag
• This information is useful when we haven’t filtered the 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
```
What we covered today

• Message passing concepts
• A practical interface
• An application
• Next time
  ‣ Asynchronous communication
  ‣ Collective communication
  ‣ Applications
Fin