Lecture 12

Programming with Message Passing
Announcements

• Triton/SDSC workshop and visit (Lecture 13)
  ♦ Weds Nov 3rd: 3.30 to 5:00 at SDSC (East Building)
  ♦ Directions
    • Head West up the road behind CSE
    • Right @ the intersection with Hopkins Parking Structure
    • SDSC is on your left, the auditorium is at street level

• Calendar
  ♦ Midterm: 11/9
  ♦ Midterm return & lecture
    11/12 (4p - 5:20p)
  ♦ No class on 11/16 & 11/18
CUDA picadillos
Measuring performance

• Two ways
  ◆ Use Cuda events/elapsed time (#ifdef CUDA TIMER)
  ◆ Use an ordinary timer, e.g. gettimeofday()

• See incrArray

• Kernel invocation is asynchronous

```c
cudaThreadSynchronize();
double t_device_compute = -getTime();
    COMPUTE
cudaThreadSynchronize();
t_device_compute += getTime();
```
CUDA Error Handling

- Cuda can silently fail, you can observe misleading performance
- E.g. if you specify an invalid grid / thread block dimensions

Beware that the last error can be cleared by successive kernel calls, so check frequently

```c
assert(cudaSuccess == cudaMemcpy(..);
printf("Cuda error: %s\n", cudaGetErrorString(cudaGetLastError()));
```

- Also see `checkCUDAError()` in `utils.cu` (incrArr)
- What about asynchronous calls?
Today’s lecture

- Message passing
- MPI
Programming with Message Passing

• **The** primary model for implementing parallel applications
• Programs execute as a set of $P$ processes
  - We specify $P$ when we run the program
  - Assume each process is assigned a different physical processor
• Each physical process
  - is initialized with the same code, but has private state
    - SPMD = “Same Program Multiple Data”
  - executes instructions at its own rate
  - has an associated *rank*, a unique integer in the range 0:$P$-1
• The sequence of instructions each 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
Buffering

- Where does the data go when you send it?
- It might be buffered
- Preferable to avoid the extra copy
An unsafe program

<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>

Reorder the calls

<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>

Use SendRecv

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendRecv(x,y,1)</td>
<td>SendRecv(x,y,0)</td>
</tr>
</tbody>
</table>
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.
Causality

- 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
Asynchronous, non-blocking communication

• Does not wait for completion
• Need to express certain algorithms
• Used to optimize performance
• Split-phased
  • Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call
    \[ I\text{Recv}( ), I\text{Send}( ) \]
  • Phase 2: synchronize
    \[ \text{Wait}( ) \]
  • Perform unrelated computations between the two phases
• Building a blocking call
  \[ \{\text{Send,Recv}\}( ) = I\{\text{Send,Recv}\}( ) + \text{Wait}( ) \]
Restrictions on non-blocking communication

- The message buffer may not be accessed between an `IRecv()` (or `ISend()`) and its accompanying `Wait()`

```
IRecv(data, source)
Crunch on data
Store into data
Wait() on ISend()
Use the data
```

- Each pending `IRecv()` must have a distinct buffer
Using IRecv to pre-post

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

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Send(y, 1)</td>
<td>Send(y, 0)</td>
</tr>
</tbody>
</table>
Another way to avoid an unsafe program

- The system has pre-allocated storage for the incoming messages so there’s no possibility of running out of storage.
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 260/ Fall '10
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();
}

©2010 Scott B. Baden /CSE 260/ Fall '10
Sending and receiving messages

• MPI provides a rich collection of routines to move data between address spaces
• A single pair of communicating processes use \textit{point-to-point} communication
• Later on we’ll cover \textit{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

©2010 Scott B. Baden /CSE 260/ Fall '10
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

```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);
```
MPI Tags

• Tags enable processes to organize or screen messages

• Each sent message is accompanied by a user-defined integer tag:
  ▶ Receiving process can use this information to organize or filter messages
  ▶ **MPI_ANY_TAG** inhibits screening
MPI Datatypes

- MPI messages have a specified length
- The unit depends on the type of the data
- The length in bytes is $\text{sizeof(type) \times \# elements}$
- We don’t use the # bytes as the length
  - Heterogeneous machines with different storage representations
  - Performance
MPI Datatypes

- Because MPI is a library, we specify the type (and hence length) of an element.
- To this end MPI specifies a set of built-in types, corresponding to the primitive types of the language from which MPI is called.

- In C: `MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, MPI_LONG, MPI_UNSIGNED, MPI_BYTE,…`

- Also defined types, e.g. structs.
- Why not pointer-based structures?
**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
```
Asynchronous, non-blocking communication

- An extra request argument is required

```c
MPI_Request request;
MPI_Irecv(buf, count, type, source, tag, comm, &request)
```

- We use the request variable to specify which message we are synchronizing in `MPI_Wait()

  ```c
  MPI_Wait(&request, &status)
  ```

- Making above 3 calls in succession is equivalent to

  ```c
  MPI_Recv(buf, count, type, source, tag, comm, &status)
  ```
Correctness and fairness

• When there are multiple outstanding iRecvs, MPI doesn’t say how incoming messages are matched…
• Or even if the process is fair

MPI_Request req1, req2;
MPI_Status status;
MPI_Irecv(buff, len, CHAR, ANY_NODE, TYPE, WORLD,&req1);
MPI_Irecv(buff2,len, CHAR, ANY_NODE, TYPE, WORLD,&req2);
MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
MPI_Send(buff, len, CHAR, prevnode, TYPE, WORLD);
MPI_Wait(&req1, &status);
MPI_Wait(&req2, &status);
Buffering

• If there is not a pending receive, then an incoming message is placed in an anonymous system buffer
• When the receive gets posted, the message is moved into the user specified buffer
• Double copying reduces communication performance
• Non-blocking communication can help avoid this problem
Avoiding the overhead

• Non-blocking communication can help ameliorate this problem

• For more information see

MPI: The Complete Reference, by Marc Snir et al.
“Buffering and Safety”
Rendezvous and eager limits

- When a message is to be sent, can MPI just send the message?
  - In an *eager* implementation, we just send the message
  - The *eager limit* is the longest message that can be sent in eager mode
  - Maximum value on IBM SP systems is 256K

- For longer messages, we have rendezvous mode communication, a sentinel is sent to obtain the permission to send the message
Send Modes

• MPI provides four different *modes* for sending a message
  ♦ Standard: Send *may or may not* complete until matching receive is posted (whether or not the data is buffered is up to the implementation)
  ♦ Synchronous: Send does not complete until matching receive is posted
  ♦ Ready: Matching receive must already have been posted
  ♦ Buffered: data is moved to a user-supplied buffer before sending

• See the handy reference at http://www-unix.mcs.anl.gov/mpi/sendmode.html
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: $\alpha + \beta^{-1}\infty n$

  [message length = $n$]

  $\alpha$ = message startup time

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

  $n$ = 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
Startup and bandwidth

- The startup term dominates when the message is sufficiently short

\[ \alpha >> \beta^{-1\infty} n \]

- The bandwidth term dominates when the message is sufficiently long

\[ \beta^{-1\infty} n >> \alpha \]
**Half power point**

- Let $T(n) =$ time to send a message of length $n$
- Let $\beta(n) =$ the effective bandwidth
  
  $$\beta^{-1}(n) = \frac{n}{T(n)}$$

- We define the **half power point** $n_{1/2}$ as the message size required to achieve $\frac{1}{2} \beta_\infty$

  $$\frac{1}{2} \beta^{-1}_\infty = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_\infty$$

- In theory, this occurs when $\alpha = \beta^{-1}_\infty n_{1/2} \Rightarrow n_{1/2} = \alpha \beta_\infty$

- Doesn’t generally predict actual value of $n_{1/2}$

- For SDSC’s Triton Cluster
  
  - $\alpha \approx 3.2 \mu s, \beta_\infty \approx 1.12$ gbytes/sec $\Rightarrow n_{1/2} \approx 3.6$KB
  - The actual value of $n_{1/2} \approx 20$KB
Typical bandwidth curve (SDSC Triton)

\[ N = 8\text{MB} \]

\[ N_{1/2} \approx 20\text{ KB} \]

\[ \alpha = 3.2\ \mu\text{sec} \]

1.12 GB/sec

©2010 Scott B. Baden /CSE 260/ Fall '10
Typical bandwidth curve
(SDSC Blue Horizon)

[Graph showing bandwidth curve with typical values and annotations]

©2010 Scott B. Baden /CSE 260/ Fall '10
Short message behavior

Triton

Length (bytes)

Usec

©2010 Scott B. Baden /CSE 260/ Fall '10
Intermediate length message behavior
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
  - \( (k + 1) \mod p \)
  - \( (k + p - 1) \mod p \)
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);
```