Lecture 3

Parallel Sorting (I)
Programming with MPI

Announcements

• Assignment #1 due in class on Thursday
• Still having problems with Valkyrie
• Verify login
• More news later
Today’s readings

- Text
  - Chap 6, §6.1-6.3 (pp 233-250)
  - Chap 9, §9.1 (pp 379-382), §9.3 (pp 394-398)
- *A User's Guide to MPI*, by Peter Pacheco (pp 1-10) at url ftp://math.usfca.edu/pub/MPI/mpi.guide.ps

Some simple programs

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

What is the outcome of this program?
Some simple programs

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

What is the outcome of this program?

Parallel Sorting

- Sorting is fundamental algorithm in data processing
  - Given an unordered set of keys $x_0$, $x_1$, ..., $x_{N-1}$
  - Return the keys in sorted order
- The keys may be character strings, floating point numbers, integers, or any object for which the relations $>$, $<$, and $==$ hold
- We’ll assume integers here
- Will talk about other algorithms later on
Parallel sorting

- We start with the well known bubble sort algorithm
- We can’t execute this formulation of the algorithm concurrently owing to the loop carried dependence in the inner loop
- The value of \( a[j] \) computed in iteration \( j \) depends on the \( a[i] \) computed in iterations \( 0, 1, \ldots, j-1 \)

\[
\text{for } i = N-1 \text{ to } 1 \text{ by } -1 \text{ do}
\]
\[
\text{done = TRUE;}
\]
\[
\text{for } j = 0 \text{ to } i-1 \text{ do}
\]
\[
\text{if } (a[i] < a[j]) \{ a[i] \leftrightarrow a[j];}
\]
\[
\text{done=FALSE; }
\]
\[
\text{end do}
\]
\[
\text{if (done) break;}
\]
\[
\text{end do}
\]

Odd/Even Transposition

- If we re-order the comparison operations we can parallelize the algorithm
- One way to re-order the computation is to …
  - number the points as even and odd
  - alternate between sorting over the odd and even points
- This algorithm parallelizes since there are no loop carried dependences
- All the odd (even) points are decoupled
- This is a classic technique
- There are two cases: \# processors \( p = n, p << n \)

\[
a_{i-1} \ a_i \ \ a_{i+1}
\]
Odd Even Transposition sort

- The fundamental operation is compare-exchange
- Compare-exchange(a[j], a[j+1])
  - swaps its arguments if they are in decreasing order
  - satisfies the post-condition that a[j] < a[j+1]
  - returns FALSE if a swap was made

The algorithm

done = false;

for i = 0 to n-1 do
  for j = 0 to n-1 by 2 do
    done &= Compare-exchange(a[j], a[j+1]);
  end do
  for j = 1 to n-1 by 2 do
    done &= Compare-exchange(a[j], a[j+1]);
  end do
  if (done) break;

end do
Odd Even sort in action

0  1  2  3  4  5
1  7  3  -1  5  6
Odd Even sort in action

0  1  2  3  4  5  
1  7  -1  3  5  6
Odd Even sort in action

0  1  2  3  4  5
1  -1  7  3  5  6

Odd Even sort in action

0  1  2  3  4  5
-1  1  7  3  5  6

Odd Even sort in action

0 1 2 3 4 5
-1 1 3 7 5 6
Odd Even sort in action

0 1 2 3 4 5
-1 1 3 5 7 6
Odd Even sort in action

When there are no swaps made in succeeding even and odd phases, we are done.
Odd Even sort in action

When there are no swaps made in succeeding even and odd phases, we are done

A simple parallel implementation

- We have 1 element per processor
- Running time is $\Theta(1)$ per sweep
- What is the total running time?
When P \ll N

• In practice, P\ll N
• We have N/P elements per processor
• The algorithm is a bit different
• A digression into implementation details

Partitioning the data

• Partitioning is the process of splitting up the data over address spaces
• We partition the data into intervals, assigning each to a unique processor
Data dependences

- Each interval needs two values found on neighboring processors
- These are located at the ends of the neighboring intervals

Ghost cells

- Use *ghost cells* to store the off processor values
- We can use the original serial loop to update the mesh without having to put communication inside
A faster algorithm

- Apply a block compare and swap operation
- Processors exchange chunks in odd-even pairs
- Each processor applies a local merge sort extract the smallest (largest) N/P values, discards the rest
- As a pre-processing step, each processor locally sorts its data using a fast serial algorithm like quicksort
- What is the running time?

Odd-even merge sort in action

- N values to be sorted
- Treat as four lists of M = N/4
- Sort each separately
- Compare and swap
- Final sorted list
Block compare and swap

-1 3 7 9 11  
2 4 8 12 14

Processor 0                      Processor 1

• Compare and swap
  – Each processor swaps data with its neighbor
    -1 3 7 9 11 2 4 8 12 14
  – Sorts the merged list
    -1 2 3 4 7 8 9 11 2 14
  – Processor 0 takes 5 smallest values: -1 2 3 4 7
  – Processor 1 takes 5 largest values: 8 9 11 12 14

Programming in MPI

• MPI = Message Passing Interface
• This is an implementation of the message passing model discussed in last lecture
• Idioms, “bells and whistles”
• There are 6 minimal routines needed by nearly every MPI program
  – 4 basic routines we use to start, end, and query MPI execution state
  – 2 basic message passing routines
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;
}
```

A second MPI program

```c
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
• Today we’ll talk about **point-to-point message passing**, in which a pair of processors communicate
• Next time we’ll discuss **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

Communicators

• One way of screening messages is through a communicator
• 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)
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
- Why don’t we use the # bytes as the length?
  - Support communication on heterogeneous machines with different storage representations without requiring compiler support
  - Avoid memory-to-memory copying

- 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,...`
- See `/usr/mpich/c/include/mpi.h`
- Later on we’ll discuss user defined types, e.g. structs
**MPI Tags**

- Each sent message is accompanied by a user-defined integer *tag*:
  - Receiving process can use this information to organize messages
  - May also filter messages (like a subject: line in email)
  - `MPI_ANY_TAG` inhibits screening.

```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);
```
Send and Recv

\begin{verbatim}
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);
\end{verbatim}
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);
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);
```

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_Status status;
MPI_Recv( message, count,
          TYPE, Destination,
          Tag, COMUNICATOR, &status);
MPI_Get_count(&status,TYPE, &recv_count );
```
Under the hood of MPI

• Note that 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
• The extra copying can be expensive
• Non-blocking can help ameliorate this problem
• For more information see
  
  *MPI: The Complete Reference*, by Marc Snir et al.  
  “Buffering and Safety”

Message passing implementations

• When a long message is to be sent, MPI first checks if the recipient has sufficient storage to receive the message
• If so, then it sends the message. This is called a *rendezvous* implementation. What are the advantages and disadvantages?
Eager limits

- In an *eager* implementation, we just send the message
- In practice, MPI implementations switch between the two modes
- The *eager limit* is the longest message that can be sent in eager mode

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
Non-blocking communication

- We’ve looked at blocking calls that cause the program to wait for completion
- MPI also provides asynchronous, non-blocking variants
- These are needed to express certain algorithms
- Their use can also help improve performance
- Asynchronous communication is split-phased
  - The first phase initiates communication activity
  - The second terminates the activity (synchronization)
  - We can take other actions between the two phases

Non-blocking communication

- Consider the following example of an “unsafe” program
- It may deadlock if there isn’t enough storage to receive the incoming message

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

- We initiate communication with an ‘I’ variant of the point-to-point call
- An extra request argument is required
  ```c
  MPI_Request request;
  MPI_Irecv(buf, count, type, dest, tag, comm, &request)
  ```
- We terminate communication with MPI_Wait()
  ```c
  MPI_Wait(&request, &status)
  ```
- The blocking call
  ```c
  MPI_Recv() = MPI_Irecv() + MPI_Wait()
  ```

An unsafe MPI program

- What can go wrong with this program?

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

- The system has pre-allocated storage for the incoming messages so there’s no possibility of running out of storage

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

Restrictions on non-blocking communication

- The message buffer may not be accessed between an IRecv( ) (or ISend( )) and its accompanying wait( )
- Why can’t we read the buffer during a pending ISend()?
- Each pending IRecv() must have a distinct buffer
An application – The Ring program

• Let’s configure the processors in a logical ring and pass messages around the ring
• Assume there are p processors
• What are the neighbors of processor k?
  ▪ \((k + 1) \mod p\)
  ▪ \((k + p - 1) \mod p\)

The Ring program

if (myrank == 0)
  for (int i = 0; i < trips; i++) {
    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);
  }
The Ring program - continued

define
else  // myrank != 0
for (int i = 0; i < trips; i++) {
    MPI_Status status;
    MPI_Recv(buffer, len, MPI_CHAR, (rank + p - 1)%p, tag, MPI_COMM_WORLD, &status);
    MPI_Send(buffer, len, MPI_CHAR, (rank+1)%p, tag, MPI_COMM_WORLD);
}