CSE 160
Lecture 20

Under the hood of MPI
Asynchronous Communication
Performance
Stencil methods with message passing
Today’s lecture

• Asynchronous non-blocking, point to point communication
• Under the hood of MPI
• Communication Performance
• Stencil methods in MPI (A4)
Asynchronous, non-blocking communication

• With Send or Receive, a return indicates the buffer may be reused, or that the data is ready

• There is also an non-blocking asynchronous form, that does not wait for completion
  ‣ Required to express certain algorithms
  ‣ Optimize performance: message flow problems

• Split-phased
  ‣ Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call IRecv(), ISend()
  ‣ Phase 2: synchronize Wait()
  ‣ Perform unrelated computations between the two phases
Overlapping communication with computation

<table>
<thead>
<tr>
<th>Overlap</th>
<th>No Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x, req)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Send(..)</td>
<td>Send(…)</td>
</tr>
<tr>
<td>Compute(y)</td>
<td>Wait(x)</td>
</tr>
<tr>
<td>Wait(req)</td>
<td>Compute(x)</td>
</tr>
<tr>
<td>Compute(x)</td>
<td>Compute(y)</td>
</tr>
</tbody>
</table>

©2013 Scott B. Baden / CSE 160 / Winter 2013
Immediate mode send and receive

- Return does not indicate completion
- Must synchronize with a `Wait()` before reusing buffer (Send) or consuming data (Receive)
- An extra `request` argument is used to refer to a message we are synchronizing

```c
MPI_Request request;
MPI_Irecv(buf, count, type, src, tag, comm, &request)
MPI_Wait(&request, &status)
```

- `Irecv + Wait = Recv`
  - `MPI_Recv(buf, count, type, src, tag, comm, &status)`
- `Immediate Send`
  - `MPI_Isend(buf, count, type, dest, tag, comm, &request)`
Buffering

- Where does the data go when you send it?
- It may be stored in a system buffer
- Preferable to avoid the extra copy when possible
Anonymous 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.
- *MPI: The Complete Reference*, by Marc Snir et al.  
  “Buffering and Safety”
- *Send modes* are also useful
  www-unix.mcs.anl.gov/mpi/sendmode.html
Restrictions on non-blocking communication

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

\[ \text{ISend}(\text{data,destination}) \]
\[ \text{Wait()} \] on `ISend()`

Use the data

• Each pending `IRecv()` must have a distinct buffer
Message completion

• A `Send()` *may or may not* complete…
• … before a `Recv()` has been posted
• “May or may not” depends on the implementation
• Some programs may deadlock on certain message passing implementations

This program may deadlock

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

This program is “safe”

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

- The program on the left may deadlock if there isn't enough storage to receive the message.
- In the program on the right, MPI has pre-allocated storage for the incoming message 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>Send(x,1)</td>
<td>Send(x,0)</td>
</tr>
<tr>
<td>Recv(y)</td>
<td>Recv(y)</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>
Rendezvous

• When a long message is to be sent, can MPI just send the message?
• For “short” message, it can. This is *eager mode* (How do we implement this?)
• The *eager limit* is the longest message that can be sent in eager mode
• See M. Banikazemi et al., IEEE TPDS, 2001, “MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems”
• For long messages, MPI first sends a scout to get permission to send the message
• This is called *rendezvous mode*
Correctness and fairness

① Iteration 1: 1 → 2&0  0 → 1 (0 → 2)  2 → 0&1
② 1 begins iteration 2: 1 → 2
③ 0 → 2 (but for iteration 1)
④ Problem: irecv in P2 receiving data from P1 in iteration 2 while it expects data from P0 in iteration 1

For i = 1 to n
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);
End for

©2013 Scott B. Baden / CSE 160 / Winter 2013
Today’s lecture

• Asynchronous non-blocking, point to point communication
• Under the hood of MPI
• Communication Performance
• Stencil methods in MPI (A4)
Where does the time go?

- Communication performance can be a major factor in determining application performance
- Under ideal conditions…
  - There is a pending receive waiting for an incoming message, which is transmitted directly to and from the user’s message buffer
  - There is no other communication traffic
- Assume a contiguous message
- LogP model (Culler et al, 1993)
Communication performance

• The so-called $\alpha \beta$ model is often good enough

• Message passing time = $\alpha + \beta^{-1}\infty n$
  
  $\alpha$ = message startup time
  
  $\beta_{\infty}$ = peak bandwidth (bytes per second)
  
  $n$ = message length

• “Short” messages: startup term dominates
  
  $\alpha \gg \beta^{-1}\infty n$

• “Long” messages: bandwidth term dominates
  
  $\beta^{-1}\infty n \gg \alpha$
Typical bandwidth curve (SDSC Triton)

\[ \beta_\infty = 1.12 \text{ GB/sec} \]

@ \( N = 8 \text{ MB} \)

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

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

Long Messages: \( \beta^{-1}\infty n \gg \alpha \)
Half power point

- \( T(n) \) = time to send a message of length \( n \)
- Let \( \beta(n) \) = the effective bandwidth
  \( \beta^{-1}(n) = n / T(n) \)
- We define the **half power point** \( n_{1/2} \) as the message size need 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 \( \alpha = \beta^{-1}_\infty n_{1/2} \)
  \( \Rightarrow n_{1/2} = \alpha \beta_\infty \)
- Generally not a good predictor of \( n_{1/2} \)
- For SDSC’s Triton Cluster
  - \( \alpha \approx 3.2 \text{ } \mu\text{s}, \beta_\infty \approx 1.12 \text{ } \text{Gbytes/sec} \Rightarrow n_{1/2} \approx 3.6\text{ } \text{KB} \)
  - The actual value of \( n_{1/2} \approx 20\text{KB} \)

<table>
<thead>
<tr>
<th>Message Length</th>
<th>BW (GB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.29</td>
</tr>
<tr>
<td>2</td>
<td>0.61</td>
</tr>
<tr>
<td>4</td>
<td>1.16</td>
</tr>
<tr>
<td>8</td>
<td>2.46</td>
</tr>
<tr>
<td>16</td>
<td>4.99</td>
</tr>
<tr>
<td>32</td>
<td>8.31</td>
</tr>
<tr>
<td>64</td>
<td>15.98</td>
</tr>
<tr>
<td>128</td>
<td>24.86</td>
</tr>
<tr>
<td>256</td>
<td>47.83</td>
</tr>
<tr>
<td>512</td>
<td>85.39</td>
</tr>
<tr>
<td>1024</td>
<td>141.28</td>
</tr>
<tr>
<td>2048</td>
<td>211.05</td>
</tr>
<tr>
<td>4096</td>
<td>183.35</td>
</tr>
<tr>
<td>8192</td>
<td>330.86</td>
</tr>
<tr>
<td>16384</td>
<td>519.23</td>
</tr>
<tr>
<td>32768</td>
<td>718.64</td>
</tr>
<tr>
<td>65536</td>
<td>702.74</td>
</tr>
<tr>
<td>131072</td>
<td>897.12</td>
</tr>
<tr>
<td>262144</td>
<td>1038.6</td>
</tr>
<tr>
<td>524288</td>
<td>1124.1</td>
</tr>
<tr>
<td>1048576</td>
<td>1177.18</td>
</tr>
<tr>
<td>2097152</td>
<td>1200.74</td>
</tr>
<tr>
<td>4194304</td>
<td>1216.02</td>
</tr>
<tr>
<td>8388608</td>
<td>1223.27</td>
</tr>
</tbody>
</table>
Short and intermediate message lengths

©2013  Scott B. Baden / CSE 160 / Winter 2013
Measuring communication performance with 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
  - \((k + 1) \mod p\)
  - \((k + p - 1) \mod p\)
- See $PUB/Examples/MPI/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);
```
Today’s lecture

• Asynchronous non-blocking, point to point communication
• Under the hood of MPI
• Communication Performance
• Stencil methods in MPI (A4)
Stencil methods under message passing

- Recall the image smooth algorithm
- Repeat as many times as necessary

\[
\text{for } (i,j) \text{ in } 0: N-1 \times 0: N-1 \\
I^{\text{new}}[i,j] = \left( I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1] \right) / 4 \\
I = I^{\text{new}}
\]

Original 100 iter 1000 iter
Parallel Implementation

- Partition data, assigning each partition to a unique process
- Different partitionings according to the *processor geometry*
- Dependences on values found on neighboring processes
- “Overlap” or “ghost” cells hold a copies off-process values

![Diagram](https://via.placeholder.com/150)
Ghost cells in higher dimensions

- Ghost cells surround each local subproblem
- Non-contiguous data
- Inefficient to communicate individual values
Managing ghost cells

- Post IReceive () for all neighbors
- Send data to neighbors
- Wait for completion
Performance is sensitive to processor geometry

- Aliev- Panfilov method running on triton.sdsc.edu (Nehalem Cluster)
- 256 cores, n=2047, t=10 (8932 iterations)

<table>
<thead>
<tr>
<th>Geometry</th>
<th>GFlops</th>
<th>Gflops w/o Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 x 8</td>
<td>573</td>
<td>660</td>
</tr>
<tr>
<td>8 x 32</td>
<td>572</td>
<td>662</td>
</tr>
<tr>
<td>16 x 16</td>
<td>554</td>
<td>665</td>
</tr>
<tr>
<td>2 x 128</td>
<td>508</td>
<td>658</td>
</tr>
<tr>
<td>4 x 64</td>
<td>503</td>
<td>668</td>
</tr>
<tr>
<td>128 x 2</td>
<td>448</td>
<td>658</td>
</tr>
<tr>
<td>256 x 1</td>
<td>401</td>
<td>638</td>
</tr>
</tbody>
</table>
Communication costs for 1D geometries

• Assumptions
  ‣ P divides N evenly
  ‣ N/P > 2

• For horizontal strips, data are contiguous
  \[ T_{\text{comm}} = 2(\alpha + 8\beta N) \]
2D Processor geometry

• Assumptions
  ‣ $\sqrt{P}$ divides $N$ evenly
  ‣ $N/\sqrt{P} > 2$

• Ignore the cost of packing message buffers

• $T_{comm} = 4(\alpha+8\beta N/\sqrt{P})$
Summing up communication costs

• Substituting $T_{\gamma} \approx 16 \beta$

• 1-D decomposition

\[
(16N^2 \beta/P) + 2(\alpha + 8\beta N)
\]

• 2-D decomposition

\[
(16N^2 \beta/P) + 4(\alpha + 8\beta N/\sqrt{P})
\]
Comparative performance

• Strip decomposition will outperform box decomposition … resulting in lower communication times … when \[ 2(\alpha + 8\beta N) < 4(\alpha + 8\beta N/\sqrt{P}) \]

• Assuming \( P \geq 2 \):
  \[ N < (\sqrt{P}/(\sqrt{P} - 2))(\alpha/8\beta) \]

• On SDSC’s IBM SP3 system “Blue Horizon”
  \( \alpha = 24 \text{ us} \)
  \( \beta = 1/(390 \text{ MB/sec}) \)

• \( N < 1170 (\sqrt{P}/(\sqrt{P} - 2)) \)
• For \( P = 16 \), strips are preferable when \( N < 2340 \)