CSE 160
Lecture 16
Non-blocking Communication
Under the hood of MPI
Performance
Stencil Methods in MPI
Announcements
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 \texttt{IRcv()}, \texttt{ISend()}
  ‣ Phase 2: synchronize \texttt{Wait()}
  ‣ Perform unrelated computations between the two phases
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`
  ```c
  MPI_Recv(buf, count, type, src,  tag, comm, &status)
  ```
- `Immediate Send`
  ```c
  MPI_Isend(buf, count, type, dest, tag, comm, &request)
  ```
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)Compute(x)</td>
</tr>
<tr>
<td>Wait(req)</td>
<td>Compute(y)</td>
</tr>
<tr>
<td>Compute(x)</td>
<td></td>
</tr>
</tbody>
</table>

©2013 Scott B. Baden / CSE 160 / Fall 2013
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
• 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()`

  `ISend(data,destination) 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>Send (x, 1)</td>
<td>Send(y, 0)</td>
</tr>
<tr>
<td>Recv (y, 1)</td>
<td>Recv(x, 0)</td>
</tr>
</tbody>
</table>

This program is “safe”

<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>
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>IRrecv(x)</td>
<td>IRrecv(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*
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
  $\alpha >> \beta^{-1}_{\infty} n$

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

\[ \beta_\infty = 1.2 \text{ GB/sec} \]

@N = 8MB

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

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

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

©2013 Scott B. Baden / CSE 160 / Fall 2013
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
  - $\alpha \approx 3.2 \mu s$, $\beta_{\infty} \approx 1.2$ Gbytes/sec $\Rightarrow n_{1/2} \approx 3.6$KB
  - The actual value of $n_{1/2} \approx 20$KB

<table>
<thead>
<tr>
<th>Length (Bytes)</th>
<th>Bandwidth (GB/sec)</th>
<th>Time (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.31</td>
<td>3.247</td>
</tr>
<tr>
<td>2</td>
<td>0.62</td>
<td>3.219</td>
</tr>
<tr>
<td>4</td>
<td>1.24</td>
<td>3.216</td>
</tr>
<tr>
<td>8</td>
<td>2.47</td>
<td>3.244</td>
</tr>
<tr>
<td>16</td>
<td>4.91</td>
<td>3.258</td>
</tr>
<tr>
<td>32</td>
<td>8.3</td>
<td>3.855</td>
</tr>
<tr>
<td>64</td>
<td>15.81</td>
<td>4.047</td>
</tr>
<tr>
<td>128</td>
<td>25.28</td>
<td>5.062</td>
</tr>
<tr>
<td>256</td>
<td>48.25</td>
<td>5.305</td>
</tr>
<tr>
<td>512</td>
<td>86.25</td>
<td>5.936</td>
</tr>
<tr>
<td>1024</td>
<td>142.8</td>
<td>7.168</td>
</tr>
<tr>
<td>2048</td>
<td>209.3</td>
<td>9.786</td>
</tr>
<tr>
<td>4096</td>
<td>188.8</td>
<td>21.7</td>
</tr>
<tr>
<td>8192</td>
<td>334.7</td>
<td>24.48</td>
</tr>
<tr>
<td>16384</td>
<td>519.2</td>
<td>31.56</td>
</tr>
<tr>
<td>32768</td>
<td>718.6</td>
<td>45.6</td>
</tr>
<tr>
<td>65536</td>
<td>702.7</td>
<td>93.26</td>
</tr>
<tr>
<td>131072</td>
<td>897.1</td>
<td>146.1</td>
</tr>
<tr>
<td>262144</td>
<td>1039</td>
<td>252.4</td>
</tr>
<tr>
<td>524288</td>
<td>1124</td>
<td>466.4</td>
</tr>
<tr>
<td>1048576</td>
<td>1177</td>
<td>890.8</td>
</tr>
<tr>
<td>2097152</td>
<td>1201</td>
<td>1747</td>
</tr>
<tr>
<td>4194304</td>
<td>1216</td>
<td>3449</td>
</tr>
<tr>
<td>8388608</td>
<td>1223</td>
<td>6858</td>
</tr>
</tbody>
</table>
Short and intermediate message lengths
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 $\text{PUB/Examples/MPI/Ring}$
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:

```
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:

```
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)
Today’s lecture

• Communication performance of Stencil methods in MPI
• Parallel Print Function
• Matrix Multiplication with Cannon’s Algorithm
Stencil methods under message passing

- Recall the image smooth algorithm

```plaintext
for iter = 1 : nSmooth
    for (i,j) in 0:N-1 x 0:N-1
        Img\text{new}[i,j] = (Img[i-1,j]+Img[i+1,j]+Img[i,j-1]+Img[i, j+1])/4
    end
    Img = Img\text{new}
end
```

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
- Too expensive to communication individual array values one at a time

<table>
<thead>
<tr>
<th>1 \times 4</th>
<th>4 \times 1</th>
<th>2 \times 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

©2013 Scott B. Baden / CSE 160 / Fall 2013
**Ghost cells**

- Dependences on values found on neighboring processes
- Neighboring values according to the processor geometry
- Inefficient to communicate individual values
- “Overlap” or “ghost” cells hold a copies off-process values
- Ghost cells surround each local subproblem
- Non-contiguous data
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
  - 1 word = double precision floating point = 8 bytes

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

• Assumptions
  ‣ $\sqrt{P}$ divides $N$ evenly
  ‣ $N/\sqrt{P} > 2$
  ‣ 1 word = double precision floating pt. = 8 bytes

• Ignore the cost of packing message buffers

• $T_{\text{comm}} = 4(\alpha + 8\beta N/\sqrt{P})$
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 Triton System
  \( \alpha = 3.2 \text{ us}, \beta = 1/(1.2 \text{ GB/sec}) \)
  - \( N < 480(\sqrt{P}/(\sqrt{P} - 2)) \)
  - For \( P = 16 \), strips are preferable when \( N < 960 \)
- 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 \( N < 2340 \)
The Aliev-Panfilov Method

• Models signal propagation in cardiac tissue
  ‣ Demonstrates complex behavior of spiral waves that are known to cause life-threatening situations

• Reaction-diffusion system
  ‣ Reactions are the cellular exchanges of certain ions across the cell membrane during the cellular electrical impulse

• Our simulation has two state variables
  ‣ Transmembrane potential: e
  ‣ Recovery of the tissue: r
The Aliev-Panfilov Model

• Two parts
  ‣ 2 Ordinary Differential Equations
    • Kinetics of reactions occurring at every point in space
  ‣ Partial Differential Equation
    • Spatial diffusion of reactants

• First-order explicit numerical scheme

\[
\frac{\partial e}{\partial t} = \delta \nabla^2 e - ke(e-a)(e-1) - er, \quad \text{on } \Omega_T, \\
\frac{\partial r}{\partial t} = - \left[ \varepsilon + \frac{\mu_1 r}{\mu_2 + e} \right] [r + ke(e-b-1)], \quad \text{on } \Omega_T, \\
\bar{n} \cdot \delta \nabla e = 0 \text{ on } \partial \Omega, \quad \text{and} \quad (e, r)_{t=0} = (e(\cdot, 0), r(\cdot, 0)),
\]
Computational loop & data dependencies

- **ODE solver:**
  - No data dependency, trivially parallelizable
  - Requires a lot of registers to hold temporary variables

- **PDE solver:**
  - Jacobi update for the 5-point Laplacian operator.
  - Sweeps over a uniformly spaced mesh
  - Updates voltage to weighted contributions from the 4 nearest neighbors

```c
#pragma omp parallel for
for (j=1; j<=m+1; j++){
    double *RR = &R[j][1], *EE = &E[j][1];
    for (i=1; i<=n+1; i++, EE++, RR++) {
        // PDE SOLVER
        EE[0] = E_p[j][i]+α*(E_p[j][i+1]+E_p[j][i-1]-4*E_p[j][i]+E_p[j+1][i]+E_p[j-1][i]);
        // ODE SOLVER
        EE[0] += -dt*(kk*EE[0]*(EE[0]-a)*(EE[0]-1)+EE[0]*RR[0]);
        RR[0] += dt*(ε+M1* RR[0]/( EE[0]+M2))*(-RR[0]-kk*EE[0]*(EE[0]-b-1));
    }
}
```
Debugging tips

• Bugs?! Not in my code!
• MPI can be harder to debug than threads
• MPI is a library, not a language
• Command line debugging
• The seg fault went away when I added a print statement
• Garbled printouts
• 2D partitioning is much more involved than 1D
• Indices are swapped

\[
\text{for } (\text{int } j=1; j<=m+1; j++) \\
\quad \text{for } (\text{int } i=1; i<=n+1; i++) \\
\quad \quad E[j][i] = E\_prev[j][i] + \alpha (E\_prev[j][i+1] + E\_prev[j][i-1] - 4E\_prev[j][i] + E\_prev[j+1][i] + E\_prev[j-1][i]);
\]
Multidimensional arrays

- Remember that in C, a 2D array is really a 1D array of 1D arrays

```c
double **alloc2D(int m,int n){
    double **E;;
    E = (double**);
    malloc(sizeof(double*)*m + sizeof(double)*n*m);
    assert(E);
    for(int j=0;j<m;j++)
        E[j] = (double*)(E+m) + j*n;
    return(E);
}
```

\[
\begin{array}{|c|c|c|}
\hline
\text{RowPointers} & \text{A[0]} & \text{A[1]} & \text{A[2]} \\
\hline
\text{ArrayData} & \text{A[0][0]} & \text{A[0][1]} & \text{A[0][2]} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{N} & \text{A[0]} & 0,0 & 0,1 & 0,2 \\
\hline
\text{M} & \text{A[1]} & 1,0 & 1,1 & 1,2 \\
\hline
& \text{A[2]} & 2,0 & 2,1 & 2,2 \\
\hline
& \text{A[3]} & 3,0 & 3,1 & 3,2 \\
\hline
& \text{A[4]} & 4,0 & 1,1 & 4,2 \\
\hline
\end{array}
\]