Collective Communication
Stencil methods with in MPI
Non-blocking Communication
Under the hood of MPI
Announcements

• Weds office hours changed for the remainder of quarter: 3:30 to 5:30
Today’s lecture

• Collective Communication
• Stencil methods with MPI
• Non-blocking Communication
• Under the hood of MPI
• Message Passing Performance
Recapping from last time

• We computed the area under the curve using point to point messaging, receiving messages in order of process ID

```c
if (my_rank == ROOT) {  // Sum the integrals
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, MPI_ANY_SOURCE, tag, WORLD, &status);
        total += integral;
    }
}
else
    MPI_Send(&integral, 1, MPI_FLOAT, ROOT, tag, MPI_COMM_WORLD);
```
Playing the wild card

• But the result does not depend on the order in which the sums are taken, except to within roundoff
• We can take the sums in any order we wish
• We use a linear time algorithm to accumulate contributions, but there are other orderings

```c
for (int source = 1; source < p; source++) {
    MPI_Recv(&integral, 1, MPI_FLOAT, 
              MPI_ANY_SOURCE, tag, 
              WORLD, &status);
    total += integral;
}
```
Using collective communication

• We can do ever better
• We can often improve performance by taking advantage of global knowledge about communication
• Instead of using point to point communication operations to accumulate the sum in linear time, use collective communication that runs in log time

```c
local_n = n/p;
float local_a = a + my_rank*local_n*h,
          local_b = local_a + local_n*h,
          integral = Trap(local_a, local_b, local_n, h);
MPI_Reduce( &integral, &total, 1,
             MPI_FLOAT, MPI_SUM,
             ROOT, MPI_COMM_WORLD)
```
Collective communication in MPI

• Collective operations are called by **all** processes within a communicator

• Here are 3 commonly used routines (more later)
  
  “root” process to all the others
  
  $\text{MPI\_Bcast}(\text{in}, \text{count}, \text{type}, \text{root}, \text{comm})$

• Reduce: combine data from all processes and return to a designated root process
  
  $\text{MPI\_Reduce}(\text{in}, \text{out}, \text{count}, \text{type}, \text{op}, \text{root}, \text{comm})$

• Allreduce: all processes get reduction: $\text{Reduce} + \text{Bcast}$
\begin{verbatim}
int local_n = n/p;

float local_a = a + my_rank*local_n*h,
    local_b = local_a + local_n*h,
    integral = Trap(local_a, local_b, local_n, h);

MPI_Allreduce( &integral, &total, 1, 
    MPI_FLOAT, MPI_SUM, WORLD)
\end{verbatim}
Today’s lecture

• Collective Communication
• Stencil methods with MPI
• Non-blocking Communication
• Under the hood of MPI
• Message Passing Performance
Recall the computational loop simulator

- **PDE solver:**
  - Updates voltage to weighted contributions from the 4 nearest neighbors updating the solution as a function of the values in the previous time step

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

\[
\begin{align*}
\text{for (j=1; j < m+1; j++)} & \{ \\
\text{for (i=1; i < n+1; i++)} & \{ \\
\mathbb{PDE SOLVER} & \\
E[j,i] &= E_p[j,i] + \alpha (E_p[j,i+1] + E_p[j,i-1] - 4E_p[j,i] + E_p[j+1,i] + E_p[j-1,i]) \\
\mathbb{ODE SOLVER} & \\
E[j,i] &+ dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]) \\
R[j,i] &+ dt*(\varepsilon + M1*R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1)) \\
\}\}
\end{align*}
\]
Data partitioning

- Partition computation and 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
- Communicate off-processor data
Global to local mapping

- In some applications, we need to compute a local to global mapping of array indices.
- In the Aliev-Panfilove method assignment, set constant values to in certain regions of the problem.
- We need to convert local (partition) coordinates to global coordinates, much like mapped coordinates.
Communication

- Expensive to mesh values individually
- Move data *en masse* into ghost regions, send the buffer to neighboring processors
- Non-contiguous on some boundaries, requires packing and unpacking
Managing ghost cells

- **Send** data to neighbors
- **Receive** from neighbors
- Can do this in stages: left/right, up/down
- Or all at once
What if we need to communicate the corners, too?

Is there an algorithm requiring only 4 communication steps?

Assume a square processor geometry
Performance is sensitive to processor geometry

- Aliev-Panfilov method running on Stampede
- 256 cores, n=6000

<table>
<thead>
<tr>
<th>Geometry</th>
<th>GFlops</th>
<th>Gflops w/o Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 256</td>
<td>616.03</td>
<td>753.79</td>
</tr>
<tr>
<td>2 128</td>
<td>625.09</td>
<td>866.4</td>
</tr>
<tr>
<td>4 64</td>
<td>626.91</td>
<td>899.05</td>
</tr>
<tr>
<td>8 32</td>
<td>637.84</td>
<td>892.17</td>
</tr>
<tr>
<td>16 16</td>
<td>633.7</td>
<td>890.45</td>
</tr>
<tr>
<td>32 8</td>
<td>613.18</td>
<td>896.13</td>
</tr>
<tr>
<td>64 4</td>
<td>600.48</td>
<td>846.87</td>
</tr>
<tr>
<td>128 2</td>
<td>582.38</td>
<td>813.44</td>
</tr>
<tr>
<td>256 1</td>
<td>547.21</td>
<td>691.83</td>
</tr>
</tbody>
</table>
Today’s lecture

- Collective Communication
- Stencil methods with MPI
- Non-blocking Communication
- Under the hood of MPI
- Message Passing Performance
Recapping Send and Recv

- When \textbf{Send( )} returns, the message is “in transit”
  - A return doesn’t tell us if the message has been received
  - The data is somewhere in the system
  - Safe to overwrite the buffer
- \textbf{Receive( )} blocks until the message has been received
  - Safe to use the data in the 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”
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>Recv( x,0)</td>
</tr>
<tr>
<td>Recv( y,1)</td>
<td>Send( y,0)</td>
</tr>
</tbody>
</table>
Asynchronous, non-blocking communication

• With Send or Receive, a return indicates the buffer may be reused, or that the data is ready
• The routines block until completion
• There is also an *non-blocking asynchronous* form, that does not wait for completion: “Immediate return”
  ◆ 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

- Must synchronize with a `Wait()` before reusing buffer (Send) or consuming data (Receive)
- A request argument enables us to refer to a message we are waiting on

```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)
```
A more effective way to manage ghost cells

• Use immediate mode communication
• Post `IReceive()` for all neighbors
• Send data to neighbors
• Wait for completion
Restrictions on non-blocking communication

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

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

Use the data

- Each pending `IRecv()` must have a distinct buffer
<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>

A message buffer may not be accessed between an IRecv( ) (or ISend( )) and its accompanying wait( )
Today’s lecture

• Collective Communication
• Stencil methods with MPI
• Non-blocking Communication
• Under the hood of MPI
• Message Passing Performance
Buffering

- Where does the message go when you send it?
- If there’s not a pending receive for an incoming message, it’s 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 avoid this problem

![Diagram showing the process of buffering and communication between processes](image-url)
Rendezvous

• When a long message is to be sent, can MPI just send the message?
• For “short” message, it can. This is **eager mode**
• **Eager limit**: 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

- Collective Communication
- Stencil methods with MPI
- Non-blocking Communication
- Under the hood of MPI
- Message Passing Performance
Message passing: 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 (Stampede)

\[ B_\infty = 5.1 \text{ GB/sec} \]

\[ @N = 4\text{MB} \]

\[ \alpha = 2.7 \mu \text{sec} \]

Long Messages:

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

\[ N_{1/2} \approx 44 \text{KB} \]
Half power point

- $T(n) = \text{time to send a message of length } n$
- Let $\beta(n) = \text{the effective bandwidth}$
  \[ \beta^{-1}(n) = \frac{n}{T(n)} \]
- 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 Stampede
  - $\alpha \approx 2.7 \ \mu s$, $\beta_{\infty} \approx 5.1 \ \text{Gbytes/sec}$
    \[ \Rightarrow n_{1/2} \approx 14\text{KB} \]
  - The actual value of $n_{1/2} \approx 44\text{KB}$
- Measurements from the Ring Program

<table>
<thead>
<tr>
<th>Length (Bytes)</th>
<th>Bandwidth (GB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>1.4377</td>
</tr>
<tr>
<td>16384</td>
<td>2.0730</td>
</tr>
<tr>
<td><strong>32768</strong></td>
<td><strong>2.2439</strong></td>
</tr>
<tr>
<td>65536</td>
<td>3.1130</td>
</tr>
<tr>
<td>131072</td>
<td>3.8838</td>
</tr>
<tr>
<td>262144</td>
<td>4.4210</td>
</tr>
<tr>
<td>524288</td>
<td>4.7619</td>
</tr>
<tr>
<td>1048576</td>
<td>4.9475</td>
</tr>
<tr>
<td>2097152</td>
<td>5.0454</td>
</tr>
<tr>
<td>4194304</td>
<td>5.0970</td>
</tr>
<tr>
<td>8388608</td>
<td>5.1220</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Length (Bytes)</th>
<th>Bandwidth (GB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>43776</td>
<td>2.517</td>
</tr>
<tr>
<td>44032</td>
<td>2.52</td>
</tr>
<tr>
<td><strong>44288</strong></td>
<td><strong>2.5528</strong></td>
</tr>
<tr>
<td>44544</td>
<td>2.5548</td>
</tr>
<tr>
<td>44800</td>
<td>2.5552</td>
</tr>
<tr>
<td>45056</td>
<td>2.5559</td>
</tr>
<tr>
<td><strong>45312</strong></td>
<td><strong>2.5562</strong></td>
</tr>
<tr>
<td>45568</td>
<td>2.5563</td>
</tr>
<tr>
<td>45824</td>
<td>2.5567</td>
</tr>
<tr>
<td>46080</td>
<td>2.5574</td>
</tr>
<tr>
<td>46336</td>
<td>2.5581</td>
</tr>
</tbody>
</table>
8K to 64KB

- 17152: 2.02 GB/s
- 17408: 1.36 GB/s
Debugging tips

• Bugs?! Not in my code!
• The seg fault went away when I added a print statement
• Garbled output
• 2D partitioning is much more involved than 1D
• MPI is a library, not a language
• Gdb not as useful when you have many processes
Parallel print function

• Problem: how to sort out all the output on the screen
• Many messages say the same thing

    Process 0 is alive!
    Process 1 is alive!
    ...
    Process 15 is alive!

• Compare with

    Processes[0–15] are alive!

• Parallel print facility

    http://www.llnl.gov/CASC/qpfp
Summary of capabilities

• Compact format list sets of nodes with common output
  ```c
  PPF_Print( MPI_COMM_WORLD, "Hello world" );
  0-3: Hello world
  ```

• %N specifier generates process ID information
  ```c
  PPF_Print( MPI_COMM_WORLD, "Message from %N\n" );
  Message from 0-3
  ```

• Lists of nodes
  ```c
  PPF_Print(MPI_COMM_WORLD, 
            (myrank % 2)
            ? "[%N] Hello from the odd numbered nodes!\n" : "[%N] Hello from the even numbered nodes!\n")
  [0,2] Hello from the even numbered nodes!
  [1,3] Hello from the odd numbered nodes!
  ```
Practical matters

• Installed in $(PUB)/lib/PPF
• Specify ppf=1 and mpi=1 on the “make” line or in the Makefile
  ♦ Defined in arch.gnu-4.7_c++11.generic
  ♦ Each module that uses the facility must
    
    #include “ptools_ppf_cpp.h”

• Look in $(PUB)/Examples/MPI/PPF for example programs ppfexample_cpp.C and test_print.c

• Uses a collective to gather all nodes’ output to a distinguished node [MPI_Gather( )]
• Installed on Bang and Stampede