Lecture 13

Programming with Message Passing
Announcements

• Midterm return
Programming with Message Passing

• The primary model for implementing high performance 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”
  ‣ 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 message we need 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
  \texttt{nproc()} = \# processors
  \texttt{myRank()} = this process’s rank

• \textit{Point-to-point} communication
  • Simplest form of communication
  • Send a message to another process
    \texttt{Send(Object, Destination process ID)}
  • Receive a message from another process
    \texttt{Receive(Object)}
    \texttt{Receive(Source process, Object)}
Send andRecv

• 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
- Used to optimize performance
- **Split-phased**
  - Phase 1: initiate communication with the immediate ‘I’ variant of the point-to-point call
    \[
    \text{IRrecv( ), ISend( )}
    \]
  - Phase 2: synchronize
    \[
    \text{Wait( )}
    \]
  - Perform unrelated computations between the two phases

- Building a blocking call
  \[
  \text{Recv( )} = \text{IRrecv( )} + \text{Wait( )}
  \]
Restrictions on non-blocking communication

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

  `ISend(data,destination)`
  `Wait()` on `ISend()`
  Use the data

• Each pending `IRecv()` must have a distinct buffer
Communication Performance
Communication performance

• Communication performance can be a major factor in determining overall performance of an application

• Simplest cost model: $\alpha + \beta^{-1}\infty \cdot n$

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

• The bandwidth term dominates when the message is sufficiently long
  \[ \beta^{-1\infty} n \gg \alpha \]
Typical bandwidth curve
(SDSC Triton)

\[
\frac{1}{2} g B/sec 
\]

\[ N = 8 MB \]

\[ \alpha = 3.2 \mu sec \]
Half power point

• Let $T(n) =$ 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 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 $\alpha = \beta_{-1\infty} n_{1/2} \Rightarrow n_{1/2} = \alpha / \beta_{-1\infty}$
• Doesn’t generally predict actual value of $n_{1/2}$
• For SDSC’s Triton Cluster
  - $\alpha \approx 3.2 \mu s, \beta_{\infty} \approx 1.12 \text{ gbytes/sec} \Rightarrow n_{1/2} \approx 3.6\text{KB}$
  - The actual value of $n_{1/2} \approx 20\text{KB}$
Short message behavior
Intermediate length message behavior
Measuring communication performance with the Ring program

• Configure the processors logically in a ring
• Pass messages around the ring multiple times
• Assume there are \( p \) processors
• Neighbors of processor \( k \) are
  
  \begin{align*}
  \bullet & \quad (k + 1) \mod p \\
  \bullet & \quad (k + p - 1) \mod p
  \end{align*}
Measurement technique with Ring

for (int len = 1, l=0; len <= maxSize; len *= 2, l++)
    if (myid == 0) {
        // (WARM UP CODE)
        Start the timer
        for (int i = 0; i < trips; i++) {
            Irecv(buffer, (rank + p - 1)%p);
            Send(buffer, (rank + 1) % p);
            Wait for Irecv to complete
        }
        double delta = take time - start;
        Bandwidth = (long)((trips*len*nodes)/ delta /1000.0);
    } else { // myid != 0
        // (WARM UP CODE)
        for (int i = 0; i < trips; i++) {
           Recv(buffer, (rank + p - 1)%p);
            Send(buffer, (rank+1)%p);
        }
    }