Point-to-Point Messaging

- This is the fundamental operation in MPI
  - Send a message from one process to another
- Blocking I/O
  - Blocking provides built-in synchronization
  - Blocking can lead to deadlock
- Send and receive, let’s do an example

See nodeadlock.c
What’s in a message?

- First three arguments specify content

```c
int MPI_Send ( 
    void* sendbuf, 
    int count, 
    MPI_Datatype datatype, 
    int dest, 
    int tag, 
    MPI_Comm comm )
```
What’s in a message?

- First three arguments specify content
  
  ```c
  int MPI_Recv (  
      void* recvbuf,  
      int count,  
      MPI_Datatype datatype,  
      int source,  
      . . . )
  ```

- All MPI data are arrays
  - Where is it?
  - How many?
  - What type?
# MPI Datatypes

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>
Deadlock

- Conditions for deadlock
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

- Deadlocks are cycles in a resource dependency graph

Deadlock in MPI Messaging

- Synchronous: the caller waits on the message to be delivered prior to returning
  - So why didn’t our program deadlock?

See deadlock.c
Deadlock in MPI Messaging

- Synchronous: the caller waits on the message to be delivered prior to returning
  - So why didn’t our program deadlock?

- Blocking **standard** send may be implemented by the MPI runtime in a variety of ways
  - `MPI_Send( ..., MPI_COMM_WORLD )`
  - Buffered at sender or receiver
  - Depending upon message size, number of processes

- Converting to a mandatory synchronous send reveals the deadlock
  - `MPI_Ssend( ..., MPI_COMM_WORLD )`
  - But so could increasing the # of processors
Standard Mode

- MPI runtime chooses best behavior for messaging based on system/message parameters:
  - Amount of buffer space
  - Message size
  - Number of processors

- Preferred way to program??
  - Commonly used and realizes good performance
  - System take available optimizations

- Can lead to horrible errors
  - Because semantics/correctness changes based on job configuration. Dangerous!
Standard Mode Danger

- You develop program on small cluster
  - Has plenty of memory for small instances
  - Messages get buffered which hides unsafe (deadlock) messaging protocol

- You launch code on big cluster with big instance
  - Bigger messages and more memory consumption means that MPI can’t buffer messages
  - Standard mode falls back to synchronous sends
  - Your code breaks

- Best practice: test messaging protocols with synchronous sends, deploy code in standard mode
Avoiding Deadlock

- Conditions for deadlock
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

- Deadlocks are cycles in a resource dependency graph

- Avoiding deadlock in MPI
  - Create cycle-free messaging disciplines
  - Synchronize actions

See passitforward.c

Messaging Topology

- Pair sends and receives
  - No circular dependencies
  - Relies on/assumes even number of nodes!
Messing Topologies

- Order/pair sends and receives to avoid deadlocks

- For linear orderings and rings
  - Simplest and sufficient: \((n-1)\) send/receive, 1 receive/send
  - More parallel, alternate send/receive and receive/send

- For more complex communication topologies?

- Messaging topology dictates parallelism
  - Important part of parallel design