MPI for Scalable Computing
(continued from yesterday)

Bill Gropp, University of Illinois at Urbana-Champaign
Rusty Lusk, Argonne National Laboratory
Rajeev Thakur, Argonne National Laboratory
Costs of Unintended Synchronization
Unexpected Hot Spots

- Even simple operations can give surprising performance behavior.
- Examples arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable stalls
Mesh Exchange

- Exchange data on a mesh
Sample Code

- Do $i=1,n_{neighbors}$
  
  Call MPI_Send(edge(1,i), len, MPI_REAL,&
  
  nbr(i), tag, comm, ierr)

Enddo

Do $i=1,n_{neighbors}$

Call MPI_Recv(edge(1,i), len, MPI_REAL,&

nbr(i), tag, comm, status, ierr)

Enddo
Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of
  ```
  if (has down nbr) then
      Call MPI_Send( ... down ... )
  endif
  if (has up nbr) then
      Call MPI_Recv( ... up ... )
  endif
  ...
  ```
  sequentializes (all except the bottom process blocks)
### Sequentialization

<table>
<thead>
<tr>
<th>Start</th>
<th>Send</th>
<th>Start</th>
<th>Send</th>
<th>Start</th>
<th>Send</th>
<th>Start</th>
<th>Send</th>
<th>Start</th>
<th>Send</th>
<th>Send</th>
<th>Recv</th>
<th>Send</th>
<th>Recv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td></td>
<td>Send</td>
<td></td>
<td>Send</td>
<td></td>
<td>Send</td>
<td></td>
<td>Send</td>
<td></td>
<td>Send</td>
<td>Recv</td>
<td>Send</td>
<td>Recv</td>
</tr>
<tr>
<td>Recv</td>
<td></td>
<td>Recv</td>
<td></td>
<td>Recv</td>
<td></td>
<td>Recv</td>
<td></td>
<td>Recv</td>
<td></td>
<td>Recv</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fix 1: Use Irecv

- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(i), ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, ierr)
Enddo
Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice. Why?
Understanding the Behavior: Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)
Mesh Exchange - Step 1

- Exchange data on a mesh
Mesh Exchange - Step 2

- Exchange data on a mesh
Mesh Exchange - Step 3

- Exchange data on a mesh
Mesh Exchange - Step 4

- Exchange data on a mesh
Mesh Exchange - Step 5

- Exchange data on a mesh
Mesh Exchange - Step 6

- Exchange data on a mesh
Timeline from IBM SP

- Note that process 1 finishes last, as predicted
Distribution of Sends

'SEND' state length distribution

(in seconds)
68 states of 96 (70%)
Why Six Steps?

- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory
Fix 2: Use Irecv and Isend

- Do i=1,n_neighbors
  
  Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
  comm, requests(i),ierr)

Enddo

Do i=1,n_neighbors

  Call MPI_Isend(edge(1,i),len,MPI_REAL, nbr(i),tag,&
  comm, requests(n_neighbors+i), ierr)

Enddo

Call MPI_Waitall(2*n_neighbors, requests, statuses, ierr)
Mesh Exchange - Steps 1-4

- Four interleaved steps
Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors
Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization

- In many cases, there is more synchronization than required

- Use nonblocking operations and MPI_Waitall to defer synchronization