Lecture 23: More on Point-to-Point Communication

William Gropp
www.cs.illinois.edu/~wgropp
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data cooperative.
- Data is explicitly sent by one process and received by another.
- An advantage is that any change in the receiving process’s memory is made with the receiver’s explicit participation.
- Communication and synchronization are combined.
One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes.
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled.
- One-sided operations are part of MPI.
Buffers

• When you send data, where does it go? One possibility is:

Process 0

User data

Local buffer

the network

Process 1

Local buffer

User data
Avoiding Buffering

• It is better to avoid copies:

This requires that \texttt{MPI\_Send} wait on delivery, or that \texttt{MPI\_Recv} return before transfer is complete, and we wait later.
Blocking and Non-blocking Communication

• So far we have been using blocking communication:
  ♦ MPI_Recv does not complete until the buffer is full (available for use).
  ♦ MPI_Send does not complete until the buffer is empty (available for use).

• Completion depends on size of message and amount of system buffering.
Sources of Deadlocks

• Send a large message from process 0 to process 1
  ♦ If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

• What happens with this code?

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

• This is called “unsafe” because it depends on the availability of system buffers
Solutions to the “safety” Problem

- Order the operations more carefully
- Supply receive buffer at same time as send (**MPI_Sendrecv**)
- Supply own buffer space (**MPI_Bsend**)
- Use non-blocking operations
  - Safe, but
  - not necessarily asynchronous
  - not necessarily concurrent
  - not necessarily faster
MPI’s Non-blocking Operations

- Non-blocking operations return (immediately) “request handles” that can be tested and waited on.

```c
MPI_Request request;
MPI_Isend(start, count, datatype, dest, tag, comm, &request);
MPI_Irecv(start, count, datatype, dest, tag, comm, &request);
MPI_Wait(&request, &status);
```

- One can also test without waiting:

```c
MPI_Test(&request, &flag, &status);
```
Multiple Completions

- It is sometimes desirable to wait on multiple requests:
  
  ```c
  MPI_Waitall(count, array_of_requests, array_of_statuses);
  MPI_Waitany(count, array_of_requests, &index, &status);
  MPI_Waitsome(incount, array_of_requests, &outcount, array_of_indices, array_of_statuses);
  ```

- There are corresponding versions of test for each of these.
Communication Modes

• MPI provides multiple modes for sending messages:
  ♦ Synchronous mode (MPI_Ssend): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  ♦ Buffered mode (MPI_Bsend): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  ♦ Ready mode (MPI_Rsend): user guarantees that a matching receive has been posted.
    • Allows access to fast protocols
    • undefined behavior if matching receive not posted
• Non-blocking versions (MPI_Issend, etc.)
• MPI_Recv receives messages sent in any mode.
Buffered Mode

- When MPI_Isend is awkward to use (e.g. lots of small messages), the user can provide a buffer for the system to store messages that cannot immediately be sent.
  
  ```c
  int bufsize;
  char *buf = malloc( bufsize );
  MPI_Buffer_attach( buf, bufsize );
  ...
  MPI_Bsend( ... same as MPI_Send ... )
  ...
  MPI_Buffer_detach( &buf, &bufsize );
  ```

- **MPI_Buffer_detach** waits for completion.

- Performance depends on MPI implementation and size of message.
Buffered Mode

- When MPI_Isend is awkward to use (e.g. lots of small messages), the user can provide a buffer for the system to store messages that cannot immediately be sent.

  integer bufsize, buf(10000)

  call MPI_Buffer_attach( buf, bufsize, ierr )
  ...
  call MPI_Bsend( ... same as MPI_Send ... )
  ...
  call MPI_Buffer_detach( buf, bufsize, ierr )

- MPI_Buffer_detach waits for completion.
- Performance depends on MPI implementation and size of message.
Computing the Buffersize

- For each message, you need to provide a buffer big enough for the data in the message and MPI_BSEND_OVERHEAD bytes.
- Data size for contiguous buffers is what you expect (e.g., in C, an array of n floats has size n * sizeof(float))
Test Your Understanding of Buffered Sends

• What is wrong with this code?

Call MPI_Buffer_attach( buf, &
    bufsize+MPI_BSEND_OVERHEAD, ierr )

Do i=1,n

    ...
    Call MPI_Bsend( bufsize bytes ... )
    ...
    Enough MPI_Revs( )
enddo

call MPI_Buffer_detach( buf, bufsize, &
    ierr )
Buffering is limited

- Processor 0
  i=1
  MPI_Bsend
  MPI_Recv
  i=2
  MPI_Bsend

- i=2 Bsend fails because first Bsend has not been able to deliver the data

- Processor 1
  i=1
  MPI_Bsend
  ... delay due to computing, process scheduling,...
  MPI_Recv
Correct Use of MPI_Bsend

• Fix: Attach and detach buffer in loop
  • Do i=1,n
    call MPI_Buffer_attach( buf, &
    bufsize+MPI_BSEND_OVERHEAD,ierr )
    ...
    call MPI_Bsend( bufsize bytes )
    ...
    Enough MPI_Recvs( )
    call MPI_Buffer_detach( buf, bufsize, ierr )
  enddo

Buffer detach will wait until messages have been delivered
Other Point-to Point Features

- MPI_Sendrecv
- MPI_Sendrecv_replace
- MPI_Cancel
  - Useful for multibuffering
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance
MPI_Sendrecv

• Allows simultaneous send and receive
• Everything else is general.
  ♦ Send and receive datatypes (even type signatures) may be different
  ♦ Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, ...)
  ♦ More general than “send left”

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendRecv(1)</td>
<td>SendRecv(0)</td>
</tr>
</tbody>
</table>
Using PMPI routines

- PMPI allows selective replacement of MPI routines at link time (no need to recompile)
- Some libraries already make use of PMPI
- Some MPI implementations have PMPI bugs
  - `PMPI_Wtime()` returns 0
  - PMPI in a separate library that some installations have not installed
Profiling Interface

Call MPI_Send

Call MPI_Bcast

User Program

MPI_Send

Profiling Library

PMPI_Send

MPI_Bcast

MPI_Send

MPI Library
Using the Profiling Interface From C

```c
static int nsend = 0;

int MPI_Send(const void *start, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
{
    nsend++;
    return PMPI_Send(start, count, datatype,
                     dest, tag, comm);
}
```
Using the Profiling Interface from Fortran

Block data
common /mycounters/ nsend
data nsend/0/
end

subroutine MPI_Send(start, count, datatype, dest,&
tag, comm, ierr)
integer start(*), count, datatype, dest, tag, comm
common /mycounters/ nsend
save /mycounters/
nsend = nsend + 1
call PMPI_Send(start, count, datatype, &
dest, tag, comm, ierr)
end
Test Yourself: Find Unsafe Uses of MPI_Send

• Assume that you have a debugger that will tell you where a program is stopped (most will). How can you find unsafe uses of MPI_Send (calls that assume that data will be buffered) by running the program without making assumptions about the amount of buffering.

♦ Hint: Use MPI_Ssend
Finding Unsafe uses of MPI_Send

subroutine MPI_Send( start, count, datatype, dest, tag, comm, ierr )
integer start(*), count, datatype, dest, tag, comm
call PMPI_Ssend(start, count, datatype, dest, tag, comm, ierr )
end

• MPI_Ssend will not complete until the matching receive starts
• MPI_Send can be implemented as MPI_Ssend
• At some value of count, MPI_Send will act like MPI_Ssend (or fail)
Finding Unsafe Uses of MPI_Send II

- Have the application generate a message about unsafe uses of MPI_Send
  - Hint: use MPI_Issend
subroutine MPI_Send(start, count, datatype, dest, tag, comm,&
   ierr)

use mpi
integer start(*), count, datatype, dest, tag, comm
integer request, status(MPI_STATUS_SIZE)
double precision tend, delay
parameter (delay=10.0d0)
logical flag

call PMPI_Isend(start, count, datatype, dest, tag, comm, &
   request, ierr)
flag = .false.
tend   = MPI_Wtime()+ delay
Do while (.not. flag .and. t1 .gt. MPI_Wtime())
   call PMPI_Test(request, flag, status, ierr)
Enddo
if (.not. flag) then
   print *, 'MPI_Send appears to be hanging'
call MPI_Abort(MPI_COMM_WORLD, 1, ierr)
endif
end
Discussion

• Write a C version of MPI_Send that checks for unsafe buffering. Modify it to permit messages smaller than sizeThreshold bytes.

• This version busy waits for completion. Discuss some strategies for reducing the overhead. How do those depend on the system (OS, hardware, etc.)?