MPI Send/Receive Blocked/Unblocked

Tom Murphy
Director of Contra Costa College
High Performance Computing Center
Where are we headed?

in focusing on Send and Receive

- Blocking
  - Easiest, but might waste time
  - Send Communication Modes (same Receive)

- Non Blocking
  - Extra things that might go wrong
  - Might be able to overlap wait with other stuff
  - Send/Receive and their friends
From where ’d we come?

6 MPI commands

- MPI_Init (int *argc, char ***argv)
- MPI_Comm_rank (MPI_Comm comm, int *rank)
- MPI_Comm_size (MPI_Comm comm, int *size)
- MPI_Send (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- MPI_Recv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- MPI_Finalize ()
Four Blocking Send Modes

basically synchronous communication

- Send is the focus
  - MPI_RECV works with all Sends
- Four Send modes to answer the questions …
  - Do an extra copy to dodge synchronization delay?
  - How do Sends/Receives Start/Finish together?
- No change to parameters passed to send or receive
- What does change is the name of the function
  - MPI_Ssend, MPI_Bsend, MPI_Rsend, and MPI_Send
MPI Send/Receive Blocked/Unblocked

Oklahoma, July 29 - Aug 4 2012

4 Blocking Send modes

* Synchronous – *Stoplight Intersection*
  * No buffer, but both sides wait for other

* Buffered – *The roundabout You construct*
  * Explicit user buffer, alls well as long as within buffer

* Ready – *Fire truck Stoplight Override*
  * No buffer, no handshake, Send is the firetruck

* Standard – *The Roundabout*
  * Not so standard blend of Synchronous and Buffered
  * Internal buffer?
Synchronous

- MPI_Ssend
- Send can initiate, before Receive starts
- Receive must start, before Send sends anything
- Safest and most portable
  - Doesn’t care about order of Send/Receive
  - Doesn’t care about any hidden internal buffer
- May have high synchronization overhead
Buffered

explicit user defined buffer

- MPI_Bsend
- Send can complete, before Receive even starts
- Explicit buffer allocation, via MPI_Buffer_attach
- Error, if buffer overflow
- Eliminates synchronization overhead, at cost of extra copy
**Ready**

no buffer - no synchronization

- MPI_Rsend
- Receive must initiate, before Send starts
- Minimum idle Sender, at expense of Receiver
- Lowest sender overhead
  - No Sender/Receiver handshake
    As with Synchronous
  - No extra copy to buffer
    As with Buffered and Standard
MPI Send

Buffer may be on send side, receive side, or both

Could be Synchronous, but users expect Buffered

Goes Synchronous, if you exceed hidden buffer size

Potential for unexpected timing behavior
Non-Blocking Send/Receive

basically asynchronous communication

- Call returns immediately, which allows for overlapping other work
- User must worry about whether …
  - Data to be sent is out of the send buffer
  - Data to be received has finished arriving
- For sends and receives in flight
  - MPI_Wait – blocking - you go synchronous
  - MPI_Test – non-blocking - Status Check
  - Check for existence of data to receive
  - Blocking: MPI_Probe
    Non-blocking: MPI_Iprobe
Non-Blocking Call Sequence

Restricts other work you can do

**Sender**

MPI_Isend -> requestID

Don’t write to send buffer till send completes

requestID -> MPI_Wait

**Receiver**

MPI_Irecv -> requestID

Don’t use data till receive completes

requestID -> MPI_Wait
Non-blocking Send/Receive

request ID for status checks

- MPI_Isend(
  void *buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm,
  MPI_Request *request)

- MPI_Irecv(
  void *buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Request *request)
Return to blocking

waiting for send/receive to complete

- Waiting on a single send
  - MPI_Wait(MPI_Request *request, MPI_Status *status)

- Waiting on multiple sends (get status of all)
  - Till all complete, as a barrier
    - MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)
  - Till at least one completes
    - MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status)

- Helps manage progressive completions
  - int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses)
Tests don’t block
but give you same info as a wait

- **Flag true means completed**
  - `MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
  - `MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses)`
  - `int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)`

- **Like a non blocking MPI_Waitsome**
  - `MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses)`
Probe to Receive

you can know something's there

- **Probes yield incoming size**

- **Blocking Probe,**
  wait til match
  - `MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`

- **Non Blocking Probe,**
  flag true if ready
  - `MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)"
Non-Blocking Advantages

- Avoids Deadlock
- Decreases Synchronization Overhead
- Best to
  - Post non-blocking sends and receives as early as possible
  - Do waits as late as possible
  - Otherwise consider using blocking calls
Illustrative sample code

“deadlock” facilitates test of the four blocking send modes
Also serves as example code using these modes
How to use it:
- Two processors are each going to each do a send and receive
- First parameter specifies whether both send(S) first, or both receive first(R), or one first sends and the other first receives (A)
- Second parameter specifies how many bytes of data to send
- Third parameter specified which send mode to use: MPI_Ssend(S), MPI_Bsend (B), MPI_Rsend (R), or MPI_Send(S)

mpirun command line
- mpirun -np 2 deadlock [SRA] mesg_len [SBRV]
MPI Hello World

Let's explore some code

- Fire up a qsub interactive shell on AC
  - ssh <account>@ac.ncsa.uiuc.edu
  - cp ~/tra5/deadlock.c
  - qsub –I
  - mpdstartup
  - mpicc –o deadlock deadlock.c
  - mpirun -np 4 ./deadlock
Exploring Blocking Send/Receive

code: deadlock.c

- Commands to execute
  - mpicc -o deadlock deadlock.c
  - mpirun -np 2 deadlock order msgLen mode
    - order is R(receive first), S(send first), or A(alternate)
    - mode is B(Buffered), R(Ready), S(Synchronous), or V(Standard)
Lab exercise using “deadlock” code
explore by using/changing code

- Parameter study
  - Which parameters result is a successful run?
  - If a parameter set fails, why does it fail?
  - Is there a message length such that \( \frac{1}{2} \) the length and twice the length have two different behaviors?
  - For what modes does this happen?

- Code change questions
  - What happens if you make the code non-blocking?
  - What happens if you modify the code so sends block, but receives are non-blocking? Vice-versa?
  - What about MPI_Sendrecv?
**MPI_Sendrecv**

send/receive smooshed together

- MPI_Sendrecv(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  int dest, int sendtag,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int source, int recvtag,
  MPI_Comm comm, MPI_Status *status)