MPI Send/Receive
Blocked/Unblocked

Message Passing Interface

Tom Murphy

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

in focusing on Send and Receive

- Blocking
  - Send Communication Modes
- Non Blocking
  - Send/Receive and their friends
From where’d we come?

what were those 7 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_Get_processor_name (char *name, int *resultlen)
- 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 Send Modes

- Send is the focus
  - MPI_RECV works with all Sends
- Four modes to answer the questions …
  - Buffer copy or synchronization?
  - When can Sends/Receives Start/Finish?
- No change to parameters passed to send
- What changes is the name of the function
  - MPI_Ssend, MPI_Bsend, MPI_Rsend, and MPI_Send
4 Blocking Send modes

- Synchronous – Stoplight Intersection
  - No buffer
- Buffered – The roundabout You construct
  - Explicit user buffer
- Ready – Fire truck Stoplight Override
  - No buffer, no handshake
- Standard – The Roundabout
  - Internal buffer?
Exploring Blocking Send/Receive

BCCD list-package tool

- Commands to execute
  - su -
  - list-packages
    Select mpiBasics2 and press OK
  - exit
  - cd /usr/local/mpiBasics2
  - 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)
Synchronous

- Send can initiate
  - Before Receive starts
- Receive must start
  - Before Send actually does anything
- Safest and most portable
  - Doesn’t care about order of Send/Receive
  - Doesn’t care about size of internal buffer
- May have high synchronization overhead
Buffered

explicit user defined buffer

- Send can complete with write to buffer
  - Before Receive even starts
- Explicit buffer allocation
  - MPI_Buffer_attach
- Error, if buffer overflow
- Eliminates synchronization overhead
  - At cost of extra copy of data
Ready

no buffer - no synchronization

- Receive must initiate
  - Before Send start
- Minimize time Sender is idle
- Lowest sender overhead
  - No Sender/Receiver handshake
    As with Synchronous
  - No extra copy to buffer
    As with Buffered and Standard
Standard

- MPI decides
  - Whether to buffer or not
- Could be implemented as Synchronous
  - But users expect buffering
- Buffer Overdraft protection
- Potential for
  - Unexpected behavior
4 Blocking Send modes

- No change to arguments
  - Synchronous: MPI_Ssend
  - Buffered: MPI_Bsend
  - Ready: MPI_Rsend
  - Standard: MPI_Send

- Subtleties needed to avoid deadlock and other evils
Non-Blocking Send/Receive and friends

- Returns immediately
  - Allows for overlapping other work
- Don’t know if …
  - Data to be sent is out of the send buffer
  - Data to be received has arrived
- Can use MPI_Wait to block
- MPI_Test is non-blocking Status Check
  - So can keep doing other work
- MPI_PROBE(blocking) and MPI_IPROBE(non-blocking)
  - Check for existence of data to receive
### Non-Blocking Call Sequence

**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

- 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 completion of send or receive

- 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)
  - Till at least one completes, and says who
    - 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)`
You can probe

- 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

sometimes causing using deadlock

- “deadlock” facilitates test of 4 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)
- BCCD mpirun command line
  - mpirun -np 2 deadlock [SRA] mesg_len [SBRV]
Lab exercise using “deadlock” code

exploration by using and 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?