MPI
Point to Point Communication
CDP
Message Passing Definitions

• Application buffer
  • Holds the data for send or receive
  • Handled by the user

• System buffer
  • Space for storing messages
  • Depending upon the type of send/receive operation
  • Allows communication to be asynchronous
  • Handled by the system

• For MPI_Send
  • The MPI standard allows the implementation to use a system buffer but does not require it
Communication modes

• Standard send
  • The basic send operation used to transmit data from one process to another

• Synchronous send
  • Blocks until the corresponding receive has actually occurred on the destination process

• Ready send
  • When a matching receive has already been posted
  • otherwise, an error occurs

• Buffered send
  • The programmer allocates a buffer for the data to be placed in until it can be sent

• Standard receive
  • The basic receive operation used to accept data sent from another process.
  • May be used to receive data from any of the possible send operations
Blocking communication

• A communication routine is blocking if the completion of the call is dependent on certain "events"

• Sends
  • The data must be successfully sent or safely copied to system buffer space, so that the application buffer that contained the data is available for reuse

• Receives
  • The data must be safely stored in the receive buffer, so that it is ready for use

• This is MPI’s way to avoid data race between the application and the MPI library
Blocking Synchronous Send

- Blocks until the corresponding receive has actually occurred on the destination process

Most of the waiting on the sending end can be eliminated if `MPI_Recv` will come before `MPI_Ssend`
Blocking Ready Send

- When a matching receive has already been posted
- otherwise, an error occurs

By default, the program exits if **MPI_Rsend** is called before notification arrives (no error code will be returned)
Blocking Buffered Send

- The programmer allocates a buffer for the data to be placed in until it can be sent.

Timing of `MPI_Recv` is irrelevant. `MPI_Bsend` returns as soon as data are copied from source to a buffer.
Blocking Standard Send

• The basic send operation used to transmit data from one process to another

• **Message size \( \leq \text{threshold} \)**
Blocking Standard Send

- The basic send operation used to transmit data from one process to another
- **Message size > threshold**
Deadlock

- Both processes send message to each other
- No one will be able to call a corresponding `MPI_Recv`
Avoiding deadlock

• Different ordering of calls between tasks
• Buffered mode
• Non-blocking calls
  • Later on...
• Other communication functions
  • For example: MPI_Sendrecv (not in this course)
Conclusions: Modes

• Synchronous mode
  • “Safest”
  • Most portable

• Ready mode
  • Lowest total overhead

• Buffered mode
  • Decouples sender from receiver
  • Allows user control

• Standard mode
  • Implementation-specific compromise
Non-blocking communication

• Method returns immediately
  • Unsafe to use buffers before actual completion
  • Check status of call via dedicated methods
    • later on...

• Allows overlapping of computation and communication
Non-blocking standard send
Non-blocking receive

- int MPI_Isend(..., MPI_Request *request)
- int MPI_Irecv(..., MPI_Request *request)
Non-blocking standard send
Non-blocking receive

• `int MPI_Isend(..., MPI_Request *request)`
• `int MPI_Irecv(..., MPI_Request *request)`
Conclusions: Non-blocking calls

• Gains
  • Avoid deadlock
  • Decrease synchronization overhead
• Post non-blocking sends and receives as early as possible
• Call wait as late as possible
• Must avoid writing to send buffer between `MPI_Isend` and `MPI_Wait`
• Must avoid reading and writing in receive buffer between `MPI_Irecv` and `MPI_Wait`
• Careful when using local variables (on the stack) that might go out of scope before they have been written to or read from!
Information About Non-Blocking MPI Call

• **MPI_Wait**
  • int MPI_Wait(MPI_Request *request, MPI_Status *status)
  • Blocking until communication is completed
  • Useful for both sender and receiver of non-blocking communications

• **MPI_Test**
  • int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
  • Non Blocking check for status of communication
  • Useful for both sender and receiver of non-blocking communications
Probing for Messages

• **MPI_Probe**
  • `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
  • Receiver is notified when messages arrive and are ready to be processed
  • From potentially *any* sender (**MPI_ANY_SOURCE**)
  • With potentially *any* tag (**MPI_ANY_TAG**)
  • Can check via the `status` struct the actual size of the message and allocate a buffer accordingly
  • This is a **blocking** call
    • Alternatively, we can use **MPI_Iprobe**
  • Can wait for any message then receive the message to a dedicated buffer
    • As oppose to receive the messages in a pre-defined order
Information About Messages

• **MPI_Status** contains information about received message
  • int MPI_Recv/Wait/Test/Probe/Iprobe(..., MPI_Status *status)
  • status.MPI_SOURCE is the actual source of the message
  • status.MPI_TAG is the actual tag of the message
  • status.MPI_ERROR is a return code (of type MPI_ERROR)
  • int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
    • Returns number of received elements in a message

• Circumstances in which it makes sense to check the status
  • Blocking receive or wait, when MPI_ANY_TAG or MPI_ANY_SOURCE has been used
  • **MPI_Probe** or **MPI_Iprobe** to obtain information about incoming messages
Null Process

• Each process sends a message to its neighbor
• It must check if it is in the edge
  • for (every partition)
    • if (right-neighbor is not edge) send right-neighbor; (or recv/test/wait)
    • if (left-neighbor is not edge) send left-neighbor;
• Instead, we can define its neighbor to be MPI_PROC_NULL
• MPI call with MPI_PROC_NULL as destination/source will return immediately without any side-effect
• The code is simplified and the checks is moved inside the MPI call
  • for (every partition)
    • send right-neighbor;
    • send left-neighbor;
Programming recommendations

• Start with non-blocking calls
  • Don’t start with blocking calls when you know you will switch later. The transition might not be trivial.

• Blocking calls
  • Use if you want tasks to synchronize
  • Use if wait immediately follows communication call
  • Start with synchronous, then switch to standard mode

• Evaluate performance and analyze code
  • If non-blocking receives are posted early, might consider ready mode
  • If there is too much synchronization overhead on sends, could switch to buffered mode

• Avoid deadlock by intelligent arrangement of sends/receives, or by posting non-blocking receives early

• Intelligent use of wildcards can greatly simplify logic and code

• Null processes move tests out of user code