Message Passing Interface (MPI)

Part I

Compiling and Running

- Several implementations of MPI exist, we use OpenMPI
- mpicc is used as the compiler for C
- mpiCC is used for C++
- mpirun -np “number of cores” executable
- For example if your executable is called “myEx” and you want to run with 256 processors
- mpirun –np 256 myEx

Starting and Terminating MPI

- `int MPI_Init(int* argc, char ** argv[])`
  - Initializes the MPI library
  - Must be called before any other MPI functions
  - Call once and only once
  - Must be called by all processes
  - do not use argc/argv before passing them to MPI_Init(). There could be arguments in there that are not meaningful to you; MPI_Init() will remove these MPI-specific options.
- `int MPI_Finalize()`
  - Performs MPI library cleanup tasks
  - Any MPI calls after an MPI_Finalize() will fail
  - Must be called by all processes

Starting and Terminating MPI

- `int MPI_Init_thread(int* argc, char ** argv[], int tenv_required,int*tenv_provided)`
  - replacement for MPI_Init(); added in MPI-2
  - in addition, it initializes the threading environment
    - `MPI_THREAD_SINGLE`: Only one thread will execute.
    - `MPI_THREAD_FUNNELED`: Process may be multithreaded, but only the main thread will make MPI calls.
    - `MPI_THREAD_SERIALIZED`: Processes may be multithreaded, and multiple threads may make MPI calls, but thread-concurrent MPI calls are disallowed.
    - `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI with no restrictions.
  - the actual level of threading provided is returned in `tenv_provided`; you may not get what you want.
- Related MPI calls:
  - `MPI_Is_thread_main()`: see if calling thread is a main thread
  - `MPI_Query_thread()`: checks the threading level provided

Simple bookkeeping in MPI

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Returns the number of processes in the communicator in the variable `size`
- `int MPI_Comm_rank(MPI_Comm comm,int *rank)`
  - Returns a unique identifier (rank) of the calling process
  - Ranges from zero to n-1, where n is the number of processes in the communicator (both FORTRAN and C)

Communication domains

- communication domain: set of processes that are allowed to communicate with one another
- in MPI, the variable declared as MPI_Comm can represent a communicator
- use as argument to all message passing functions
- each process may belong to more than one communication domain
- `MPI_COMM_WORLD` is the default communicator, which includes all processes started via MPI via “mpirun” or similar
- the “rank” of a process is meaningful within a communication domain only

Passing a message: MPI_Send()

- `int MPI_Send(void *buffer, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm)`
  - Initiates a send of “buffer” and length “count” of type “datatype” to the process with rank “destination”.
  - “tag” can be used to distinguish messages from one another if there are multiple messages sent to the same rank
- when does MPI_Send() return?
  - MPI_Send is a blocking call; it will not return until it is safe to free or change the memory pointed to by **buffer**.
  - MPI standard does not specify; up to the library implementor
  - if the implementor provides an intermediate buffer [i.e., allows an eager send], it can return as soon as message has been copied
  - if not, it does not return until the matching receive call (on another process) has completed
  - **we have to assume the worst (that its return depends on the state of another process, i.e., it is synchronous)!**

Message delivery expectations

- typical MPI implementations follow the recommendations of the MPI standard (although they are not required to)
- short messages: use an “eager send”: message and envelope are sent immediately assuming destination can store it
  - a buffer typically exists on the receiving end for these “early arrival” messages
  - stored on the receiving end until the matching receive is posted
- long messages: use a synchronous send (or, “rendezvous”)
  - message is not sent until destination gives an OK: handshaking operation.
- limit between small and large messages: 16KB, typical. Also, the user can usually set the limit through environment variables

Passing a message: MPI_Recv() [1 of 2]

- `int MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,MPI_Status *status)`
  - Initiates a receive of `buffer` that has, at most, length `count` of type `datatype` from the process with rank `source`.
  - `tag` can be used to distinguish messages from one another if there are multiple messages sent to the same rank
  - `status` is a structure used to obtain information about the receive
  - MPI_Recv does not return until the message has been safely copied into the buffer (blocks..semantics are assured)

Passing a message: MPI_Recv() [2 of 2]

- MPI_Status: small structure which gives information on the completed receive operation
  - this structure contains the following members:
    - `MPI_SOURCE`
    - `MPI_TAG`
    - `MPI_ERROR`
- `MPI_Get_count(MPI_Status *status, MPI_Datatype dtype, int *count)`
  - can be used to determine the size of the received message, if needed

Wildcards for Receives

- To receive a message, we must specify its envelope exactly.
- To relax this requirement, we can use the following wildcards
  - `MPI_ANY_SOURCE`
  - `MPI_ANY_TAG`
- Or, if we want to post receive that does nothing,
  - `MPI_PROC_NULL`
  - the receive returns successfully immediately, with no change to the receive buffer)
  - `MPI_PROC_NULL` can be used for sends as well. No message is sent.

Notes for FORTRAN users

- all MPI routines are subroutines; invoked with the CALL statement
- all MPI routines are capitalized
- all MPI routines have additional argument “ierr” at the end of the argument list; this is the return value of the routine
  - exceptions: MPI_WTIME and MPI_WTICK
- all MPI objects (MPI_Datatype, MPI_Comm, etc.) are of type INTEGER

Notes for FORTRAN users

- examples:
  - `CALL MPI_SEND(a,10,MPI_DOUBLE_PRECISION,0,1, MPI_COMM_WORLD, ierr)`
  - `CALL MPI_RECV(a,10,MPI_DOUBLE_PRECISION,1,1, MPI_COMM_WORLD, status,ierr)`
    - source = status(MPI_SOURCE)
    - tag = status(MPI_TAG)

Communication Modes (1 of 4)

- MPI_Send() and MPI_Recv() are termed “standard” sends and “standard” receives
  - underlying implementation may or may not buffer MPI_Send()
  - we must assume that it is synchronous (i.e., not buffered; buffering has the effect of decoupling the sender and receiver)
- send modes (blocking) (section 3.4, MPI Standard 1.1)
  - standard: MPI_Send()
  - buffered: MPI_Bsend()
  - synchronous: MPI_Ssend()
  - ready: MPI_Rsend()

Communication Modes (2 of 4)

- standard: MPI_Send()
  - nonlocal assumed [remember that we assume the worst]
  - typical MPI implementation:
    - mapped to either (modified) MPI_Rsend() or MPI_Ssend() directly
    - "eager" send vs. "rendezvous" send, depending on message size
- buffered: MPI_Bsend()
  - local: its completion does not depend on another process.
  - user provides buffer
    - only one buffer for all buffered mode sends
    - `MPI_Buffer_attach(void*buffer,int size)`
    - `MPI_Buffer_detach(void *buffer_ptr,int *size)`
      - blocks until MPI has determined the the buffer is not used for any operations
    - remember MPI_BSEND_OVERHEAD (per message overhead) and MPI_Pack_size()
  - typical MPI implementation: copy message data to a buffer, then initiate a nonblocking send

Communication Modes (3 of 4)

- synchronous: MPI_Ssend()
  - nonlocal
  - can complete only when matching receive posted and data flow initiated
  - typical MPI implementation: sends a request-to-send message; receiver stores request. When matching receive posted, receiver sends a permission-to-send message, and the sender initiates data flow.
  - standard handshake protocol
- ready: MPI_Rsend()
  - local
  - should be started only if matching receive already posted; otherwise, the code is erroneous and the behavior is undefined.
  - removes the handshake synchronization overhead; improves performance; otherwise, same as standard mode send
  - typical MPI implementation: send message as soon as possible

Communication Modes (4 of 4)

- only one receive operation; can match any of the send modes:
  - MPI_Recv()
- various modes exist for nonblocking operations as well:
  - MPI_Isend (standard mode, nonblocking)
  - MPI_Ibsend (buffered mode, nonblocking)
  - MPI_Issend (synchronous mode, nonblocking)
  - MPI_Irsend(ready mode, nonblocking)

Nonblocking Communications

- all communication modes exist for nonblocking sends, but we will concentrate on MPI_Isend() in our examples
- MPI_Irecv(): nonblocking receive; can match any of the send calls, blocking or nonblocking

Nonblocking Communications

- `int MPI_Isend(void *buffer, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm, MPI_Request * req)`
  - arguments same as MPI_Send() with the addition of the MPI_Request parameter
  - request structure will allow us to test later for completion of the send operation
  - request structure is an “opaque object”

Nonblocking Communications

- `int MPI_Irecv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,MPI_Request *req)`
  - arguments same as MPI_Recv() with the addition of the MPI_Request parameter and removal of MPI_Status parameter
  - request structure will allow us to test later for completion of the send operation
  - request structure is an “opaque object”

Completion of Nonblocking Communications

- `MPI_Wait(MPI_Request *req, MPI_Status *status)`
  - used to complete nonblocking communication
  - completion means that sender free to reuse or free buffer; or, receiver is OK to use the buffer data
  - does not indicate that message has been received, unless the synchronous nonblocking send (MPI_Issend) was used (and even then, it only means that the matching receive has been initiated).
  - blocks until request identified by `req` is complete
  - one is allowed to call MPI_Wait with a null or inactive `req`, in which case it returns immediately with an empty status
- If we have an array of outstanding requests we want to wait on,
  - `MPI_Waitany(), MPI_Waitall(), MPI_Waitsome()`
  - waits for one, all, or at least one request to complete, respectively

Completion of Nonblocking Communications

- `MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)`
  - nonblocking version of MPI_Wait()
  - returns `flag =` nonzero if operation is complete, and sets the `status` object just as MPI_Wait() would
- If we have an array of outstanding requests we want to test for,
  - `MPI_Testany(), MPI_Testall(), MPI_Testsome()`
  - tests for one, all, or at least one request to complete, respectively

Pending Message Operations

- `MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status status)`
  - allows incoming messages to be checked for, and information about that message to be obtained
  - can use wildcards
  - blocks until matching message found
  - possible use: if the message is pending, the status object will contain the length of the message. Allocate enough memory, then do a real receive.
- `MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,MPI_Status status)`
  - Nonblocking version
- Opinion: using these functions excessively probably indicates a serious logic problem. Use only when truly appropriate.

Pending Message Operations

- `MPI_Cancel(MPI_Request *req)`
  - local; does not depend on another process's state
  - still necessary to complete the communication with an MPI_Wait() or similar
  - if canceling a send to which a matching receive has already been posted, we must still satisfy the matching receive with another send!
  - if canceling a receive, we must still satisfy the matching send with another receive!
  - not guaranteed to cancel; only marks a communication for cancellation
  - can be an expensive operation; use with caution
  - opinion: this function opens a "can of worms". Just don't open a communication if you aren't sure that it should actually happen.
- `MPI_Test_Cancelled(MPI_Status *status,int *flag)`
  - determines whether cancel was successful or not.