The MPI Message-passing Standard Practical use and implementation (III)

SPD Course

26/02/2026

Massimo Coppola

POINT-TO-POINT COMMUNICATION MODES

Buffered Send

MPI_BSEND (buf, count, datatype, dest, tag, comm)

```
MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag,
   MPI_Comm comm)
```

- Same parameters as the standard send
- Explicitly relies on buffering
  - Can complete regardless of the matching receive = local completion
  - Triggers an error if no buffer space is available, unlike a standard Send
- Programmer has to allocate enough buffers for the process needs, and pass them to the MPI implementation

```
int MPI_Buffer_attach(void* buffer, int size)
int MPI_Buffer_detach(void* buffer_addr, int* size)
```

Synchronous Send

MPI_SSEND (buf, count, datatype, dest, tag, comm)

```
MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int
   tag, MPI_Comm comm)
```

- Same parameters as the standard send
- Enforces synchronous send operation
  - A program is ***safe*** if all its sends are Synchronous

Ready Send

UNIVERSITÀ DI PISA

MPI_RSEND (buf, count, datatype, dest, tag, comm)

- Again same parameters
- Optimizes implementation assuming a matching receive has been already posted
  - Used with ***permanent*** requests
  - When program semantics ensures the precondition
  - Together With SendRecv primitives
  - Note that:
    - Permanent requests and SendRecv are used solely as example cases
    - SendRecv is a single primitive to issue a send and a receive combined

Ready Send and SendRecv

```
// Process A
while (true) {
    recv ( … B …)
    do_compute()
    Rsend ( …B… )
}
```

```
//Process B
while (true) {
    do_compute()
    sendRecv( …A…)
}
```

BLOCKING AND NON-BLOCKING POINT-TO-POINT

Incomplete operations

- Separate communication **start** from its **completion**
- Available for **both** send and receive
- Primitive calls can return before completion
- Resources are NOT free
- Separate primitives for checking communication completion/status

- Useful if actual communication is offloaded to DMA, coprocessors etc.

Incomplete Send / Recv

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

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

int MPI_Irecv(void* buf, int count,
   MPI_Datatype datatype, int source,
   int tag, MPI_Comm comm, MPI_Request *request)
```

- MPI_ISEND Also combines with all modes
- MPI_I**B**SEND
- MPI_I**S**SEND
- MPI_I**R**SEND

Request objects

- Opaque objects
- Fully identify the communication operation
  - One to one match with communications
  - Requests are allocated by MPI when they become **active** (communication started, but not completed)
  - Requests are active until completion is not checked
- Can provide status and completion information
- The MPI_request type is the object handle
  - Uninitialized/**inactive** handle value: MPI_REQUEST_NULL
  - MPI does this whenever a request object is no longer needed (it becomes inactive) and it is freed

Waiting and Testing

MPI_WAIT(request, status)

- INOUT request request (handle)
- OUT status status ob ject (Status)

- Waits until the operation is complete
  - Returns the operations status
  - Clears the request handle

MPI_TEST(request, flag, status)

- Returns immediately
  - flag=true if the operation is complete
  - In this case, behaves as a completed WAIT
- Wait is a non-local operation, while Test is a local one
- MPI_REQUEST_NULL handles are silently ignored

Multiple Wait / Test

- MPI_WAITANY (count, array_of _requests, index, status)
  - Wait for one request from an array to complete (nondeterministic behaviour, no fairness)
  - index=MPI_UNDEFINED if no request is active
- MPI_WAITALL (count, array_of _requests, array_of _statuses)
  - Wait for all requests to complete
- MPI_WAITSOME(incount, array_of _requests,
  outcount, array_of _indices, array_of _statuses)
  - Wait for at least one request to complete, possibly several ones
  - You can implement your own preferred nondeterministic behaviour
  - outcount=MPI_UNDEFINED if no request is active

- MPI_TESTANY(count, array_of _requests, index, flag, status)
- MPI_TESTALL(count, array_of _requests, flag, array_of _statuses)
- MPI_TESTSOME(incount, array_of _requests,
  outcount, array_of _indices, array_of _statuses)

TEST/WAIT comments

- It is safe to call again and again the same primitive: eventually, all requests become inactive
- MPI_requests are handles
  - can be copied
  - it’s programmer’s responsibility not to use more than one copy
    - better invalidate them!
- Null handle is not the same as inactive
  - however, MPI_REQUEST_NULL is also inactive

~~MPI_Cancel~~

MPI_Cancel(request)

- MPI_cancel was deprecated in MPI 4.0
- Allows to cancel a nonblocking operation that is *still pending* == active request
  - can’t cancel operation after a successful WAIT or TEST
- Necessary to free up resources acquired by the active request
- Returns immediately (see MPI_Test_cancelled)
  - Intended as a low-overhead operation, MPI_Cancel has **local completion**, and may return before the operation is actually canceled
  - Doesn’t wait for any auxiliary communication/interrupt to complete
  - If successful, cancel makes the request inactive
    → TEST and WAIT calls on it become safe *local op.s*

~~MPI_Cancel~~

UNIVERSITÀ DI PISA

- However, cancel *may* fail
  - Example: an MPI_IBSend may have already copied the data to MPI-owned buffers → can't both cancel the operation and respect IBSend semantics
  - **either** the cancel succeeds (and frees all buffers) **or** the communication "completes" (may stall buffer!)
- Information about the cancel operation will be returned via the *status* of the nonblocking call
- It depends on program's semantics and code structure if MPI_cancel is needed at all
- MPI_cancel can cancel permanent comm. requests, but that's trickier

~~MPI_Test_cancelled~~

MPI_Test_cancelled(status, &flag)

- Allows to check (`flag==true`) whether a non-blocking operation was actually canceled
  - Reads the status from a TEST or WAIT
  - If an operation may be cancelled, it's mandatory to check for cancellation BEFORE using the status any other way
  - Depending on the send optimization, testing cancellation may require communications
  - Can be an expensive operation : contrary to MPI_Cancel, here we wait for any implementation-level communication to complete
  - Testing cancellation in general has **non local** completion

~~Cancel, Test_cancelled and~~ Finalize

UNIVERSITÀ DI PISA

- MPI_Finalize tells MPI that the program is about to end
  - all support can be shut down and implicitly allocated memory is freed
    - including most opaque objects
  - Does not free stuff *explicitly* allocated via MPI primitives
    - but process usually exits right away
- Processes must complete all communications they are involved with before calling Finalize
  - This may require canceling and testing cancellation of non-blocking calls
  - Canceling some operations (e.g. IBSend) may be impossible → the other party may need to complete them before finalizing

Reference Texts

- MPI standard (rev. 5.0) relevant parts for 3rd lesson
  - Check back on Chapter 2 sec. 2.2, 2.4,
  - Chapter 3:
    sec. 3.1 – 3.3 check back
    sec. 3.4, 3.5, 3.6 (3.6.1 can be skipped), 3.7, skip 3.8, 3.10
    persistent comm.s are in 3.9, and sendRecv in 3.7
  - Chapter 5:
    sec. 5.1 – to 5.1.2; 5.1.9 – 5.1.11