# PCSE Lecture 10

## MPI Point-to-Point Continued

Cyrus Proctor
Victor Eijkhout

April 7, 2015

# Wildcards

- Enables a programmer to avoid having to specify tag and/or source

## C Version

```
MPI_Status status;
int data[5];
int ierr;
ierr = MPI_Recv(&data[0], 5, MPI_INT,
                MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD,&status);
```

## Fortran Version

```
integer status(MPI_STATUS_SIZE)
integer mydata(5)
integer ierr
call MPI_RECV(mydata, 5, MPI_INTEGER, &
              MPI_ANY_SOURCE, MPI_ANY_TAG, &
              MPI_COMM_WORLD, status, ierr)
```

- MPI_ANY_SOURCE and MPI_ANY_TAG are wildcards
- status structure/array is used to get wildcard values

# Wilcards

- MPI_PROC_NULL
  - May be used in the "destination" or "source" fields for Send and Recv calls
  - The operation completes immediately
  - No communication actually takes place
- Particularly useful when handling edges/boundaries of partitioned data

# Persistent Communication

What is persistent communication?

- If you have point-to-point message-passing routine is called repeatedly with the same arguments, persistent communication can be used to avoid redundancy in setting up the message each time it is sent
- Persistent communication reduces the overhead of communication between the parallel task and the network adapter
- Good for data decomposition problems in which points are updated based on the values of neighboring points
- Pass all the Send/Recv arguments and perform the setup required for the communication only once

# Persistent Communication

Create requests:

- Start with special MPI_*_init commands to create the send and receive objects without actually sending any messages
- Very similar arguments to regular Send/Recv commands
- All persistent communication objects are nonblocking
- Calls populate an MPI_Request object for use later (like Isend/Irecv)

Send and receive messages:

- Use the MPI_Start command (non-blocking)
- Use MPI_Wait command to ensure data safety

Clean up objects:

- Use the MPI_Request_free routine to deallocate persistent objects

# Persistent Communication Example

## C Version

```
MPI_Request recv_obj;
MPI_Request send_obj;
MPI_Status status;

//Step 1) Initialize send/request objects
MPI_Recv_init (buf1, cnt, tp, src, tag, com, &recv_obj);
MPI_Send_init (buf2, cnt, tp, dst, tag, com, &send_obj);

for (i=1; i<BIGNUM; i++)
{
    //Step 2) Use start in place of recv and send
    //MPI_Irecv (buf1, cnt, tp, src, tag, com, &recv_obj);
    MPI_Start (&recv_obj);

    do_work(buf1,buf2);

    //MPI_Isend (buf2, cnt, tp, dst, tag, com, &send_obj);
    MPI_Start (&send_obj);

    //Wait for send to complete
    MPI_Wait (&send_obj, status);
    //Wait for receive to finish (no deadlock!)
    MPI_Wait(&recv_obj, status);
}

//Step 3) Clean up the requests
MPI_Request_free (&recv_obj);
MPI_Request_free (&send_obj);
```

# Persistent Communication Example

## Fortran Version

```
integer recv_obj ! request object
integer send_obj ! request object
integer stat(MPI_STATUS_SIZE)

! Step 1) Initialize send/request objects
call MPI_RECV_INIT(buf1, cnt, tp, src, tag, com, recv_obj, ierr)
call MPI_SEND_INIT(buf2, cnt, tp, dst, tag, com, send_obj, ierr)

do i=1, BIGNUM
    ! Step 2) Use start in place of recv and send
    ! call MPI_IRECV(buf1, cnt, tp, src, tag, com, recv_obj, ierr)
    call MPI_START(recv_obj, ierr)

    call do_work(buf1,buf2)

    ! call MPI_ISEND(buf2, cnt, tp, dst, tag, com, &send_obj, ierr)
    call MPI_START(send_obj, ierr)

    ! Wait for send to complete
    call MPI_WAIT(send_obj, stat)
    ! Wait for receive to finish (no deadlock!)
    call MPI_WAIT(recv_obj, stat)
end do

! Step 3) Clean up the requests
call MPI_REQUEST_FREE(recv_obj)
call MPI_REQUEST_FREE(send_obj)
```

# MPI_Wait and Friends

- MPI_Wait
- MPI_Waitany
- MPI_Waitall
- MPI_Waitsome

| | |
|---|---|
| C | MPI_Wait(&request, &status) |
| C | MPI_Waitany(count, &array_of_requests, &index, &status) |
| C | MPI_Waitall(count, &array_of_requests, &array_of_statuses) |
| C | MPI_Waitsome(incount, &array_of_requests, &outcount, &array_of_offsets, &array_of_statuses) |
| Fortran | MPI_WAIT(request, status, ierr) |
| Fortran | MPI_WAITANY(count, array_of_requests, index, status, ierr) |
| Fortran | MPI_WAITALL(count, array_of_requests, array_of_statuses, ierr) |
| Fortran | MPI_WAITSOME(incount, array_of_requests, outcount, array_of_offsets, array_of_statuses, ierr) |

MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

# MPI_Test and Friends

- MPI_Test
- MPI_Testany
- MPI_Testall
- MPI_Testsome

| | |
|---|---|
| C | MPI_Test(&request, &flag, &status) |
| C | MPI_Testany(count, &array_of_requests, &index, &flag, &status) |
| C | MPI_Testall(count, &array_of_requests, &flag, &array_of_statuses) |
| C | MPI_Testsome(incount, &array_of_requests, &outcount, &array_of_offsets, &array_of_statuses) |
| Fortran | MPI_TEST(request, flag, status, ierr) |
| Fortran | MPI_TESTANY(count, array_of_requests, index, flag, status, ierr) |
| Fortran | MPI_TESTALL(count, array_of_requests, flag, array_of_statuses, ierr) |
| Fortran | MPI_TESTSOME(incount, array_of_requests, outcount, array_of_offsets, array_of_statuses, ierr) |

MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.

# MPI_Probe

| | |
|---|---|
| C | MPI_Probe(source, tag, comm, &status) |
| Fortran | MPI_PROBE(source, tag, comm, status, ierr) |

- Performs a blocking test for a message (also has non-blocking cousin)
- The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag
- Allows you to take some sort of action before posting a receive for the message

# Probe, Test, Wait, Status

- Different ways of dealing with unpredictability
- Is there a message?
  - Wait: you have nothing else to do, so wait (blocking)
  - Test: non-blocking inspection of request, use if you have other things to do
- Once data is coming in:
  - Probe: inspect incoming data without receiving first
  - Status: get properties of received message

# References


- Victor Eijkhout, “Introduction to High Performance Scientific Computing”
- Victor Eijkhout, “Parallel Computing for Science and Engineering”
- Steve Lantz, “MPI One-Sided Communication”
- Mark Lubin, “Introduction into new features of MPI-3.0 Standard”
- “MPI: A Message-Passing Interface Standard Version-3.0”