MPI 4

CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

● Learn about point-to-point non-blocking communication.
Blocking vs. Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

  - **Non-blocking**:
    - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
    - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user cannot predict when that will happen.
    - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
    - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.
MPI Blocking P2P

The threshold value (also called the “eager limit”) differs between systems.
MPI Non-Blocking P2P

MPI_ISEND (nonblocking standard send)

size \leq\ threshold

MPI_WAIT

no delay even though message is not yet in user's buffer on receiving node

transfer to buffer on receiving node can be avoided if MPI_Irecv posted early enough

MPI_Irecv

MPI_WAIT

no delay if MPI_WAIT is late enough
Non-blocking Communication

● Advantages:
  ▪ allows the separation between the initialization of the communication and the completion.
  ▪ can avoid deadlock
  ▪ can reduce latency by posting receive calls early

● Disadvantages:
  ▪ complex to develop, maintain and debug code
Non-blocking Send/Recv Details

- Non-blocking operation requires a minimum of two function calls: a call to start the operation and a call to complete the operation.
- The “request” is used to query the status of the communicator or to wait for its completion.
- The user must NOT overwrite the send buffer until the send (data transfer) is complete.
- The user can not use the receiving buffer before the receive is complete.
- MPI_Wait() returns when the operation is complete, and the status is updated for a receive.
MPI_Wait()  

**Name**  

MPI_Wait - Waits for an MPI send or receive to complete.

**Syntax**

**C Syntax**

```c
#include <mpi.h>
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

**Fortran Syntax**

```fortran
INCLUDE 'mpi.f'
MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER    REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

**C++ Syntax**

```c++
#include <mpi.h>
void Request::Wait(Status& status)
void Request::Wait()
```
MPI Non-Blocking Send &Recv

- C++ Syntax
  
  MPI::COMM_WORLD.Isend( data, count, datatype, dest, tag)
  MPI::COMM_WORLD.Irecv( data, count, datatype, dest, tag)
  Request::Wait(Status& status)
  Request::Wait()

- Example:

  MPI::Request req;
  req = MPI::COMM_WORLD.Isend(&val, count, MPI::INT, 1, tag)
  req.Wait();
```cpp
#include <iostream>
#include <cstdlib>
#include "mpi.h"

using namespace std;

int main(int argc, char **argv) {
    int nprocs, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI::Request request[4]; // required variable for non-blocking calls
    MPI::Status status[4];   // required variable for Waitall routine

    MPI::Init(argc,argv);
    nprocs = MPI::COMM_WORLD.Get_size();
    rank = MPI::COMM_WORLD.Get_rank();

    // determine left and right neighbors
    prev = rank-1;
    next = rank+1;
    if (rank == 0) prev = nprocs - 1;
    if (rank == (nprocs - 1)) next = 0;

    // post non-blocking receives and sends for neighbors
    request[0] = MPI::COMM_WORLD.Irecv(&buf[0], 1, MPI::INT, prev, tag1);
    request[1] = MPI::COMM_WORLD.Irecv(&buf[1], 1, MPI::INT, next, tag2);
    request[2] = MPI::COMM_WORLD.Isend(&rank, 1, MPI::INT, prev, tag2);
    request[3] = MPI::COMM_WORLD.Isend(&rank, 1, MPI::INT, next, tag1);

    // do some work while sends/receives progress in background

    // wait for all non-blocking operations to complete
    request[4].Wait(status[4]);

    // continue - do more work
    MPI::Finalize();
    return EXIT_SUCCESS;
}
```