Introduction to Parallel & Distributed Programming

Lec 19— MPI

Subodh Sharma | March 23, 2026

(RECAP) P-to-P Communication: Send & Recv

- **MPI_Send**:
  - Address of data
  - Count of elements
  - Datatype of elements
  - Destination rank
  - Message tag
  - Communicator

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        int x = 42;
        MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent value %d to process 1\n", x);
    }

    if (rank == 1) {
        int y;
        MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received value %d from process 0\n", y);
    }

    MPI_Finalize();
    return 0;
}
```

(RECAP) P-to-P Communication: Send & Recv

- **MPI_Recv**:
  - Source rank
    - **MPI_ANY_SOURCE**
  - Message tag
  - Communicator
  - Status object — information about the message received

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        int x = 42;
        MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent value %d to process 1\n", x);
    }

    if (rank == 1) {
        int y;
        MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received value %d from process 0\n", y);
    }

    MPI_Finalize();
    return 0;
}
```

(RECAP) Semantics of Send

- **MPI_Send**:
  - It is a blocking send call
  - Under Buffering:
    - Send can return once the buffering of the message is complete
  - Under no buffering:
    - Send acts as a rendezvous send
  - Completion semantics;
    - Complete when the payload is copied out of the sender’s address space

Semantics of Recv

- **MPI_Recv**:
  - It is a blocking call (with or without buffering)
  - Completion semantics;
    - Complete when the payload is copied out of the receiver’s address space
  - MPI_ANY_SOURCE: Can non-deterministically match with any one of the racing sends
    - Source of non-determinism
    - Can also be a source of non-deterministic deadlocks

Types of Send Calls

- **MPI_Send:** blocking send call (semantics dependent on buffer availability)
- **MPI_Bsend:** explicit user-supplied buffer is provided;
  - Check **MPI_Buffer_attach**
  - **Completion semantics?**
- **MPI_Ssend:** Synchronous send
  - Completion only when a matching receive is already posted
- **MPI_Rsend:** Ready send; expects the recv to be *already* posted. **How is it different from Ssend?**

Message Matching Semantics

- Two send calls from the same process to the same destination with the same tag must match in order
  - Check the matching rules with MPI_ANY_SOURCE and MPI_ANY_TAG
- Two recv calls from the same source with the same tag must match in order
- Fairness is not guaranteed
  - A send/recv may be starved because all matching calls are satisfied by others

Point to Point Nonblocking Calls

- **Isend**: Returns immediately (even before the buffer is copied out)
  - Buffer
  - Count, datatype, dest, tag, communicator
  - **Request:** object that stores the communication state of the call
  - **Why is it useful? —** Interleaves useful local work with communication
  - **How to test if the call has completed?**
    - Call a blocking MPI_Wait or MPI_Test calls (use request handle)

Point to Point Nonblocking Calls

- **Isend, Irecv**: Returns immediately (even before the buffer is copied out)
  - Buffer
  - Count, datatype, dest, tag, communicator
  - **Request:** object that stores the communication state of the call
  - **Why is it useful? —** Interleaves useful local work with communication
  - **How to test if the call has completed?**
    - Call a blocking MPI_Wait or MPI_Test calls (both use request handle)
      - MPI_Wait blocks until the machine nonblocking call completes
        - Check WaitAll, WaitAny, WaitSome variants
      - MPI_Test is just a single time check

MPI DataTypes

- MPI_Datatype objects are of type:
  - Char, Short, Int, Long, LongLong Int, Float, Double, Byte, WChar
  - Signed and unsigned variants
- Derived Datatypes: User defined types over primitive types
  - MPI doesn’t understand language’s layout (struct, union, etc.)
  - Typemap: (type0, disp0), .., (typeN, dispN)
    - ith entry is of type_i and starts at byte base + disp_i

Derived DataTypes: Example

```c
typedef struct {
    int id;
    double value;
} Item;
```

```c
/* There are 2 blocks: one int and one double */
int blocklengths[2] = {1, 1};
MPI_Aint displacements[2];
MPI_Datatype types[2] = {MPI_INT, MPI_DOUBLE};

/* Compute correct byte offsets */
MPI_Aint base;
MPI_Get_address(&x, &base);
MPI_Get_address(&x.id, &displacements[0]);
MPI_Get_address(&x.value, &displacements[1]);

displacements[0] -= base;
displacements[1] -= base;

/* Create and commit the derived datatype */
MPI_Type_create_struct(2, blocklengths, displacements, types, &item_type);
MPI_Type_commit(&item_type);
```