CSE502: Foundations of Parallel Programming

Lecture 21: Point-to-Point Communications in MPI

Vivek Kumar

Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Class

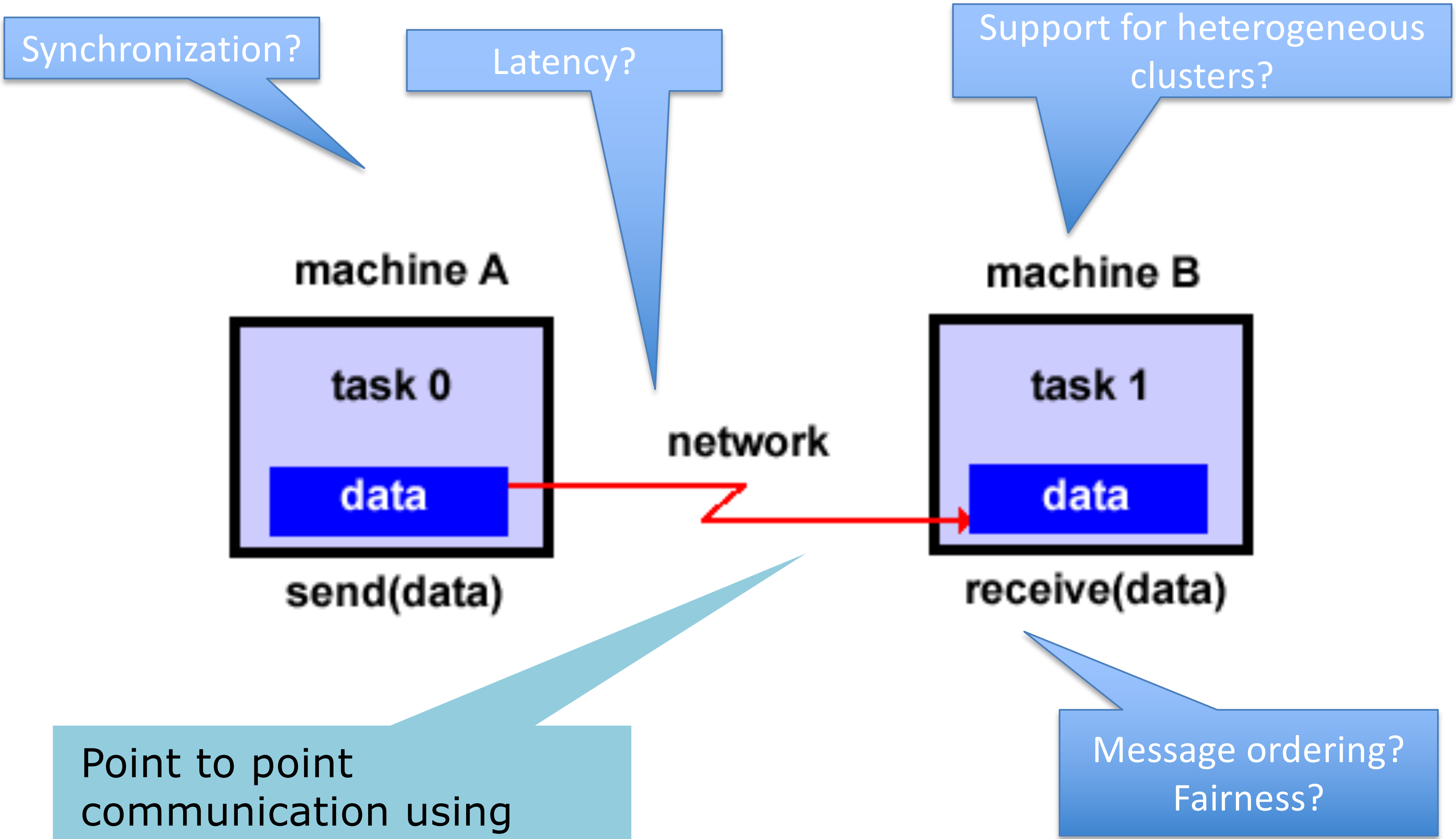
- Introduction to distributed memory parallel programming using the MPI

```
// the header file containing MPI APIs
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(argc, argv);
    int rank, nprocs;
    // Get the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    // Get the rank of this process in MPI_COMM_WORLD
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf(“My rank is %d in world of size %d\n”, rank, nprocs);
    // Terminate the MPI runtime
    MPI_Finalize();
    return 0;
}
```

Today's Class

- Point-to-point communication in MPI
  - Blocking
  - Non-blocking

Point-to-Point Communications in MPI



Point to point communication using message passing

How to Define the Type of Data Being Sent?

- MPI provides its own reference datatypes corresponding to the various elementary datatypes in C, C++, Fortran
  - Enables MPI to support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication)
  - **E.g., C: MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc.**
- User-defined datatypes are also supported
  - Requires usage of MPI APIs to define the datatype

Blocking APIs: MPI_Send and MPI_Recv

```
MPI_Send(void* buffer, int count,
         MPI_Datatype type, int destination, int tag,
         int communicator);

MPI_Recv(void* buffer, int count
         MPI_Datatype type, int source, int tag,
         int communicator, MPI_Status stats);
```

Parallel ArraySum Example

```
main(int argc, char **argv) {
  int rank=0, nproc=4;

  int array[SIZE * nproc]; // properly initialized
  // 2. calculate local sum
  int my_sum = 0, total_sum, tmp, tag=1, start = rank*SIZE;
  for (int i=start; i<SIZE+start; i++) my_sum += array[i];

}
```

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

Parallel ArraySum Example

```
main(int argc, char **argv) {
  int rank=0, nproc=4;
  MPI_Init(&argc, &argv);
  // 1. Get to know your world
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  int array[SIZE * nproc]; // properly initialized
  // 2. calculate local sum
  int my_sum = 0, total_sum, tmp, tag=1, start = rank*SIZE;
  for (int i=start; i<SIZE+start; i++) my_sum += array[i];

  MPI_Finalize();
}
```

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

Parallel ArraySum Example

```
main(int argc, char **argv) {
  int rank=0, nproc=4;
  MPI_Init(&argc, &argv);
  // 1. Get to know your world
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  int array[SIZE * nproc]; // properly initialized
  // 2. calculate local sum
  int my_sum = 0, total_sum, tmp, tag=1, start = rank*SIZE;
  for (int i=start; i<SIZE+start; i++) my_sum += array[i];
  // 3. All non-root processes send result to root processes (rank=0)
  if(rank > 0) {

  }
  else {






  }
  MPI_Finalize();
}
```

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

Parallel ArraySum Example

```
main(int argc, char **argv) {
  int rank=0, nproc=4;
  MPI_Init(&argc, &argv);
  // 1. Get to know your world
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  int array[SIZE * nproc]; // properly initialized
  // 2. calculate local sum
  int my_sum = 0, total_sum, tmp, tag=1, start = rank*SIZE;
  for (int i=start; i<SIZE+start; i++) my_sum += array[i];
  // 3. All non-root processes send result to root processes (rank=0)
  if(rank > 0) {
    MPI_Send(&my_sum, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
  else {



  }
  MPI_Finalize();
}
```

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

Parallel ArraySum Example

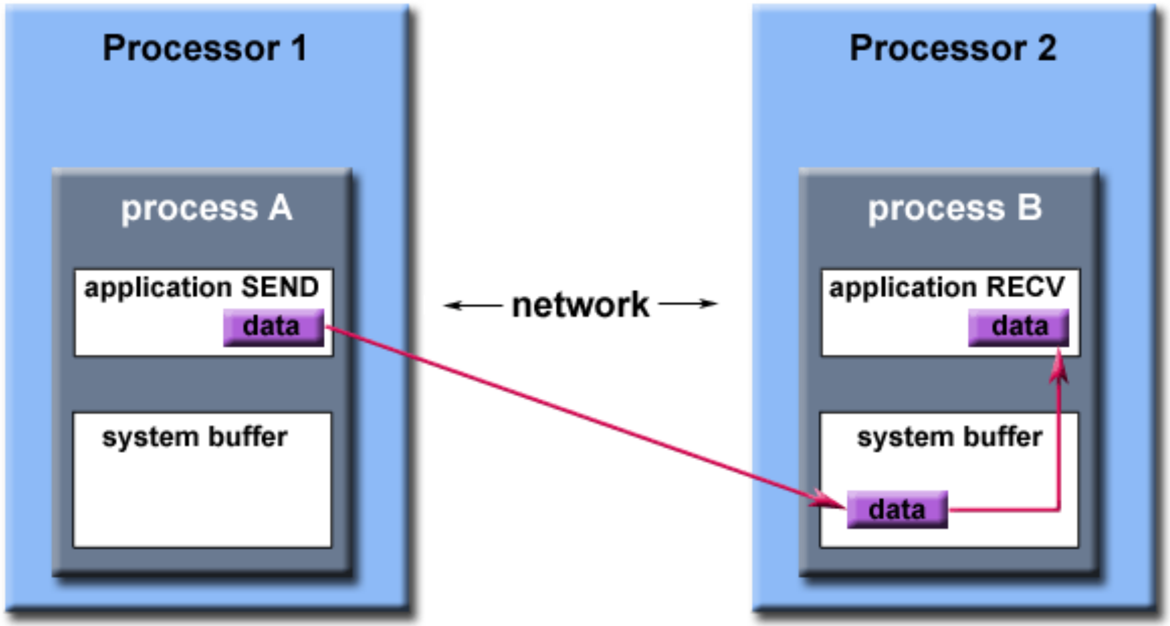
```
main(int argc, char **argv) {
  int rank=0, nproc=4;
  MPI_Init(&argc, &argv);
  // 1. Get to know your world
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  int array[SIZE * nproc]; // properly initialized
  // 2. calculate local sum
  int my_sum = 0, total_sum, tmp, tag=1, start = rank*SIZE;
  for (int i=start; i<SIZE+start; i++) my_sum += array[i];
  // 3. All non-root processes send result to root processes (rank=0)
  if(rank > 0) {
    MPI_Send(&my_sum, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
  else {
    total_sum = my_sum;
    for(int src=1; src<nproc; src++) {
      MPI_Recv(&tmp, 1, MPI_INT, src, tag, MPI_COMM_WORLD, &status);
      total_sum += tmp;
    }
  }
  MPI_Finalize();
}
```

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

When Does MPI_Send/MPI_Recv Returns?

- MPI_Send
  - This blocking API **only** “return” after it is safe for the sender to modify the send buffer
  - Doesn’t implies that the data is at destination
    - Might be sitting inside the system buffer
    - Implementation specific
- MPI_Recv
  - This blocking API **only** “return” after the data has arrived and is ready to be used by program

Message Buffering



Path of a message buffered at the receiving process

- Not possible to synchronize every MPI_Send with matching MPI_Recv
  - How to deal if a send arrives before a matching recv is posted?
  - How to deal with multiple sends arriving?
- "**MPI Implementations**" (not MPI standard!) typically reserves a system buffer to hold data in transit

Picture source: https://computing.llnl.gov/tutorials/mpi/

The “tag” in MPI_Send and MPI_Recv

- An integer value defined by the programmer
- Identifies the type of message
- MPI processes can use it to pair a specific type of send and recv operations
- If tags at send and recv doesn’t match then it will create a deadlock

Deadlock Due to Incorrect Tag

```
main(int argc, char **argv) {
  int rank, nproc;
  ............
  ............

  if(rank == 1) {
    int tag = 100;
    MPI_Send(&buffer, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
  else if(rank == 0) {
    int tag = 101;
    MPI_Recv(&buffer, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
  }

  ............
}
```

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

Message Ordering Guarantee

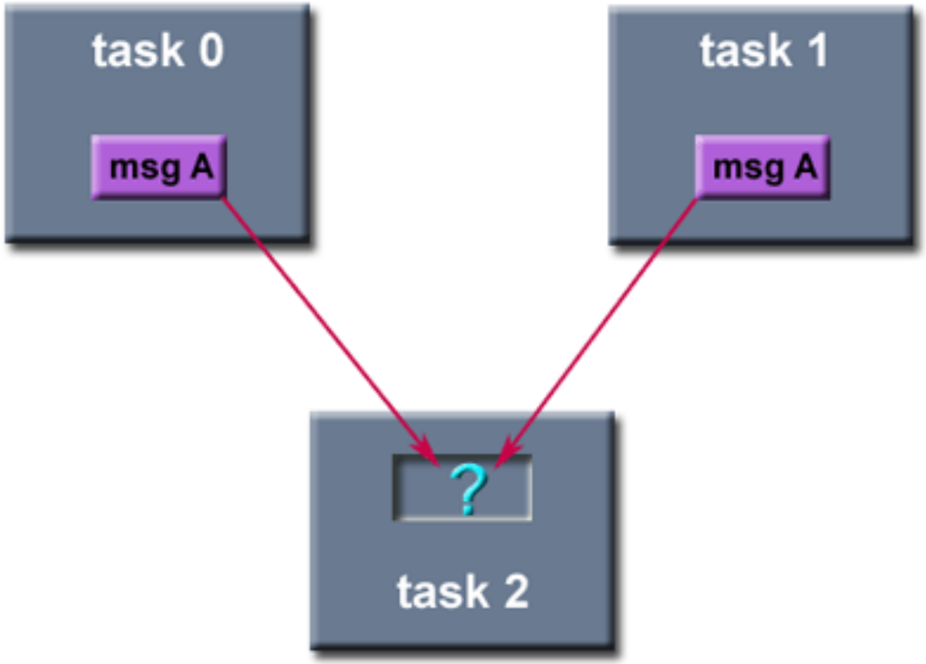
```
main(int argc, char **argv) {
  int rank, nproc;
  ............
  ............

  if(rank == 1) {
    for(int i=0; i<MAX; i++) {
      MPI_Send(&i, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }
  }
  else if(rank == 0) {
    int buffer[MAX];
    for(int i=0; i<MAX; i++) {
      MPI_Recv(&buffer[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
      assert(buffer[i] == i); // Never fails
    }
  }
  ............
}
```

- If a sender sends two messages (Msg_1 and Msg_2) in succession to same destination, and both match the same receive, the recv operation will always receive Msg_1 before Msg_2

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

No Guarantee for Fairness



- MPI does not guarantee fairness
- Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete

Picture source: https://computing.llnl.gov/tutorials/mpi/

Deadlock Due to Incorrect Pairing of Send and Recv

```
main(int argc, char **argv) {
  int rank, nproc;
  ............
  ............

  if(rank == 1) {
    MPI_Recv(&buffer1, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    MPI_Send(&buffer2, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
  else if(rank == 0) {
    MPI_Recv(&buffer1, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    MPI_Send(&buffer2, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
  }

  ............
}
```

- To remove deadlock swap the two MPI calls at **any one** of the ranks

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

Today's Class

- Point-to-point communication in MPI
  - Blocking
  - Non-blocking

Non-Blocking Point-to-Point Communications

- MPI_Isend
- MPI_Irecv

1. These APIs returns 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
2. Provide opportunities to overlap computations and communications – unlike their blocking counterparts

MPI_Isend and MPI_Irecv

- Exactly same syntax as their blocking counterpart, except a MPI_Request* parameter added
  - MPI_Request request is a handle to an internal MPI object. Everything about that non-blocking communication is through that handle.

```
MPI_Isend(void* buffer, int count,
          MPI_Datatype type, int destination, int tag,
          int  communicator, MPI_Request* req);
MPI_Irecv(void* buffer, int count
          MPI_Datatype type, int source, int tag,
          int communicator, MPI_Request* req);
// below API similar to finish in HClib
MPI_Wait(MPI_Request* req, MPI_Status *stats)
```

Allows Overlapping Computations and Communications

```
............
MPI_Request request;
MPI_Status status;

MPI_Isend(buffer, ......, &request); // or even MPI_Irecv

compute(); // Do some computation while the above MPI_Isend is in progress

MPI_Wait(&request, &status); // wait until MPI_Isend is complete
............
```

Using MPI_Isend and MPI_Irecv in Our Previous Example of Parallel ArraySum

```
main(int argc, char **argv) {
  ...........
  int partial_sum[nproc];
  MPI_Request req[nproc – 1]; // nproc = number of processes
  // 3. All non-root processes send result to root processes (rank=0)
  if(rank > 0) {
    MPI_Isend(&my_sum, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &req[rank-1]);
    MPI_Status stats;
    MPI_Wait(&req[rank-1], &stats);
  }
  else {
    partial_sum[0] = my_sum;
    for(int src=1; src<nproc; src++) {
     MPI_Irecv(&partial_sum[src], 1, MPI_INT, src, tag, MPI_COMM_WORLD, &req[src-1]);
    }
    MPI_Status stats[nproc-1];

    MPI_Waitall(nproc-1, req, stats);
  }
  ..........
}
```

This API waits for all given non-blocking communications to complete. In this case it’s the MPI_Irecv calls

Code available on github: https://github.com/vivkumar/cse502/tree/master/mpi

MPI Routines Covered Today

5. MPI_Send
6. MPI_Recv
7. MPI_Isend
8. MPI_Irecv
9. MPI_Wait
10. MPI_Waitall

Next Class

- Collective communications in MPI

Reading Material

- Tutorial on MPI by LLNL
  - https://computing.llnl.gov/tutorials/mpi/
- References on MPI routines with example
  - http://mpi.deino.net/mpi_functions