Lab#7: Send and Receive Operations in MPI

Blocking and Non-Blocking Send:

MPI Send ():MPI_Send does not complete until the buffer is empty (available for use)..

MPI Isend (): Non-blocking send. But not necessarily asynchronous. You CANNOT reuse the send buffer until either a successful wait/test or you certainly KNOW that the message has been received. An immediate send must return to the user without requiring a matching receive at the destination.

Blocking and Non-Blocking Recv:

MPI_recv: MPI_Recv does not complete until the buffer is full (available for use).

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_Irecv(): Creates a receive request and returns a receive request in an MPI_Request object. The caller is responsible for not changing the buffer until after waiting upon the resulting request object.

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

Waiting for non-blocking call completion

An MPI_Wait call waits for completion of the operation that created the request object passed to it. For a send, the semantics of the sending mode have been fulfilled (not necessarily that the message has been received). For a receive, the buffer is now valid for use, however the send has not necessarily completed (though obviously has been initiated).

int MPI_Wait(MPI_Request *request, MPI_Status *status)

Testing for non-blocking call completion An MPI_Test call returns immediately a flag value indicating whether a corresponding MPI_Wait would return immediately.

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

Example#01: Send Recv in MPI

```
#include <stdio.h>
#include <mpi.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 (size != 2) {
        if (rank == 0) {
            printf("This program is designed for exactly 2 processes.\n");
        }
        MPI_Finalize();
        return 1;
    }

    int data[1000];
    if (rank == 0) {
        // Initialize data in process 0
        for (int i = 0; i < 1000; i++) {
            data[i] = i;
        }

        // Send data to process 1
        MPI_Send(data, 1000, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        // Receive data in process 1
        MPI_Recv(data, 1000, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        // Print received data
        printf("Received data in process 1:\n");
        for (int i = 0; i < 1000; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Simultaneously Send and Recv: Allows simultaneous send and receive.

int MPI_Sendrecv (void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

Task#1: Convert the hello world program to print its messages in rank order. (Hint: Lab#6)

Task#2: Write two programs in MPI C where process with rank zero sends a message and a process with rank 1 receives a message with a tag=your student id. (Hint: Lab#6)

a) Use Blocking send receive and measure the time MPI_Wtime
b) Use Non-Blocking send receive and measure the time MPI_Wtime

Task#3: Write two programs in MPI C and measure their time using MPI_Wtime.

a) Use blocking send and receive an array of 10,000 elements
b) Use non-blocking send and receive an array of 10,000 elements
c) Simultaneously send receive an array of 10,000 elements using MPI_Sendrecv.

d) Measure the performance (execution time) of the part (a), part (b), part (c) for message (array size) 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000 and plot it on excel.

Example#2: Odd Even Sort

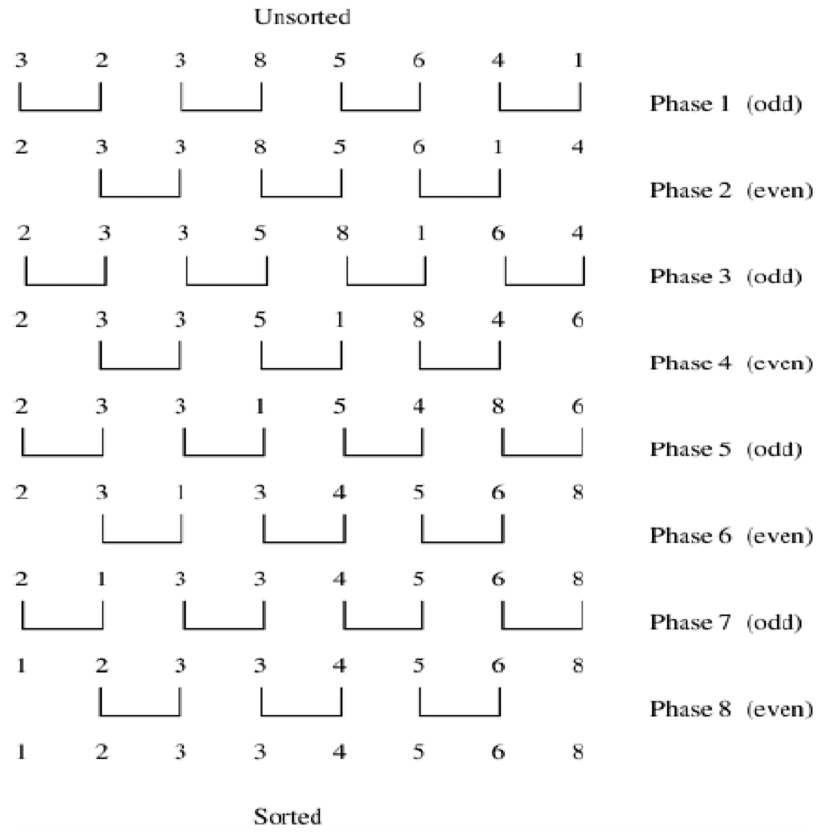
The ***odd-even transposition*** algorithm sorts *n* elements in *n* phases (*n* is even), each of which requires *n*/2 compare-exchange operations. This algorithm alternates between two phases, called the odd and even phases. Let <*a*1, *a*2, ..., *an*> be the sequence to be sorted. During the odd phase, elements with odd indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (*a*1, *a*2), (*a*3, *a*4), ..., (*an*-1, *an*) are compare-exchanged (assuming *n* is even). Similarly, during the even phase, elements with even indices are compared with their right neighbors, and if they are out of sequence they are exchanged; thus, the pairs (*a*2, *a*3), (*a*4, *a*5), ..., (*an*-2, *an*-1) are compare-exchanged. After *n* phases of odd-even exchanges, the sequence is sorted. Each phase of the algorithm (either odd or even) requires Q(*n*) comparisons, and there are a total of *n* phases; thus, the sequential complexity is Q(*n*2). The odd-even transposition sort is shown in Algorithm 9.3 and is illustrated in Figure 9.13.

Sorted

Algorithm 9.3 Sequential odd-even transposition sort algorithm.

```
procedure ODD-EVEN(n)
begin
   for i := 1 to n do
   begin
      if i is odd then
           for j := 0 to n/2 - 1 do
               compare-exchange($a_{2j+1}$, $a_{2j+2}$);
      if i is even then
           for j := 1 to n/2 - 1 do
               compare-exchange($a_{2j}$, $a_{2j+1}$);
   end for
end ODD-EVEN
```

Figure 9.13. Sorting *n* = 8 elements, using the odd-even transposition sort algorithm. During each phase, *n* = 8 elements are compared.



Example#2: Odd Even Sort

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

void oddEvenSort(int *arr, int n) {
    int sorted = 0;
    int phase, temp;

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) {
            for (int i = 1; i < n; i += 2) {
                if (arr[i - 1] > arr[i]) {
                    temp = arr[i];
                    arr[i] = arr[i - 1];
                    arr[i - 1] = temp;
                }
            }
        } else {
            for (int i = 1; i < n - 1; i += 2) {
                if (arr[i] > arr[i + 1]) {
                    temp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = temp;
                }
            }
        }
    }
}

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);

    int n = 10; // Change this to the desired array size
    int *arr = (int *)malloc(n * sizeof(int));

    if (rank == 0) {
        // Initialize the array with random values on the root process
        printf("Original Array: ");
        srand(123); // Seed for random number generation
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 100;
            printf("%d ", arr[i]);
        }
        printf("\n");
    }

    // Broadcast the array to all processes
    MPI_Bcast(arr, n, MPI_INT, 0, MPI_COMM_WORLD);

    // Perform parallel odd-even sort
    oddEvenSort(arr, n);

    // Gather the sorted portions back to the root process
    int *sortedArr = NULL;
    if (rank == 0) {
        sortedArr = (int *)malloc(n * sizeof(int));
    }
    MPI_Gather(arr, n, MPI_INT, sortedArr, n, MPI_INT, 0, MPI_COMM_WORLD);

    // Print the sorted array on the root process
    if (rank == 0) {
        printf("Sorted Array: ");
        for (int i = 0; i < n; i++) {
            printf("%d ", sortedArr[i]);
        }
        printf("\n");
        free(sortedArr);
    }

    free(arr);

    MPI_Finalize();
    return 0;
}
```

Task#4: Measure the communication and computation time of Example#2. Change the array size as n=100, 200,300,400, and 500. Is it fine grained or coarse grained? Clearly mention your observation.

Task#5: Write a program in MPI C to generate factorial up to 50 and 100 terms. Measure the performance (execution time) of the code for 2, 4, and 6 MPI processes and plot it on excel.