



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Parallel and Distributed Computing
CSE4001

Theory Assignment 2

Slot : E2

Name : Kulvir Singh

Register Number : 19BCE2074

1. MPI_Bsend is the asynchronous blocking send (with user provided Buffering), it will block until a copy of the buffer is passed. Write a program to establish a point to point communication between two process using the above function. Discuss how it is different from standard send function. (10 marks)

Program :

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

int main (int argc, char * argv[])
{
    MPI_Init (&argc, &argv);
    int size;
    MPI_Comm_Size (MPI_COMM_WORLD, &size);
    if (size != 2)
    {
        printf ("This app is meant to be run with 2 process\n");
        MPI_Abort (MPI_COMM_WORLD, EXIT_FAILURE);
    }
    enum role_ranks { SENDER, RECEIVER };
    int my_rank;
    MPI_Comm_Rank (MPI_COMM_WORLD, &my_rank);
    switch (my_rank)
    {
        case SENDER :
        {
            int buffer_attached_size = MPI_BSEND_OVERHEAD +
                                      sizeof (int);
            char * buffer_attached = (char *) malloc
                                      (buffer_attached_size);
            MPI_Buffer_attach (buffer_attached, buffer_attached_size);

            int buffer_sent = 12345;
            printf ("[MPI process %d] I send value %d.\n",
                    my_rank, buffer_sent);
            MPI_Bsend (&buffer_sent, 1, MPI_INT,
                      RECEIVER, 0, MPI_COMM_WORLD);
            MPI_Buffer_detach (&buffer_attached,
                              &buffer_attached_size);
            free (buffer_attached);
            break;
        }
    }
}
```

```

case RECEIVER:
{
    int received;
    MPI_Recv(&received, 1, MPI_INT, SENDER, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf(" [MPI process %d] I received value %d. \n",
           my_rank, received);
    break;
}
}
MPI_Finalize();
return EXIT_SUCCESS;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size != 2)
    {
        printf("This application is meant to be run with 2
processes.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
    enum role_ranks { SENDER, RECEIVER };
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    switch(my_rank)
    {
        case SENDER:
        {
            int buffer_attached_size = MPI_BSEND_OVERHEAD +
sizeof(int);
            char* buffer_attached =
(char*)malloc(buffer_attached_size);

```

```

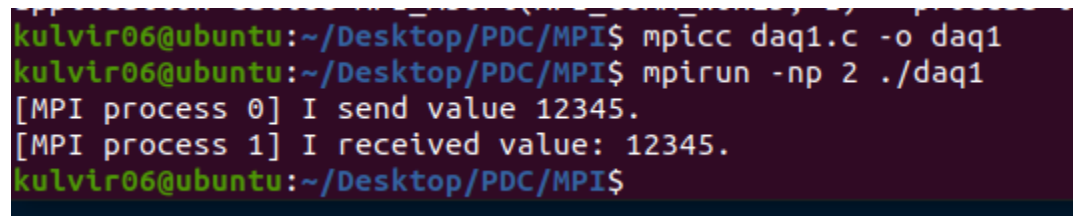
        MPI_Buffer_attach(buffer_attached,
buffer_attached_size);
        int buffer_sent = 12345;
        printf("[MPI process %d] I send value %d.\n", my_rank,
buffer_sent);
        MPI_Bsend(&buffer_sent, 1, MPI_INT, RECEIVER, 0,
MPI_COMM_WORLD);
        MPI_Buffer_detach(&buffer_attached,
&buffer_attached_size);
        free(buffer_attached);
        break;
    }
    case RECEIVER:
    {
        int received;
        MPI_Recv(&received, 1, MPI_INT, SENDER, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("[MPI process %d] I received value: %d.\n",
my_rank, received);
        break;
    }
}

MPI_Finalize();

return EXIT_SUCCESS;
}

```

Output Screenshot :



```

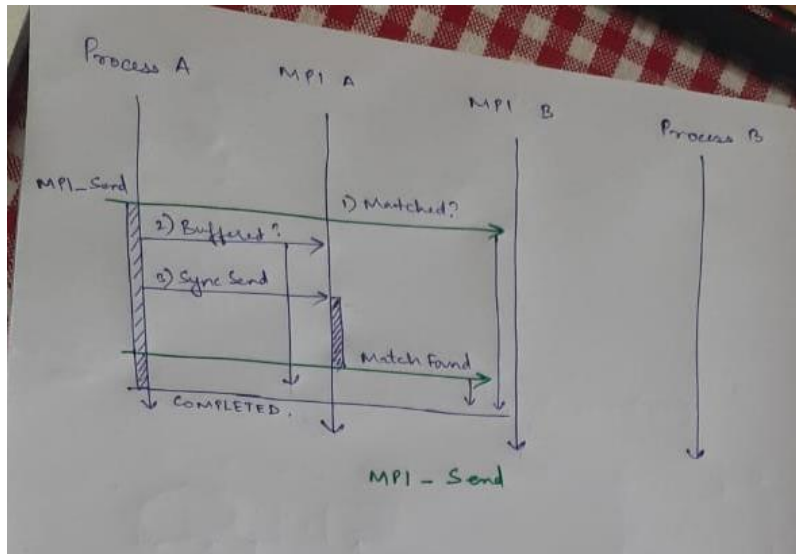
kulvir06@ubuntu:~/Desktop/PDC/MPI$ mpicc daq1.c -o daq1
kulvir06@ubuntu:~/Desktop/PDC/MPI$ mpirun -np 2 ./daq1
[MPI process 0] I send value 12345.
[MPI process 1] I received value: 12345.
kulvir06@ubuntu:~/Desktop/PDC/MPI$

```

Comparison between Send and Bsend :

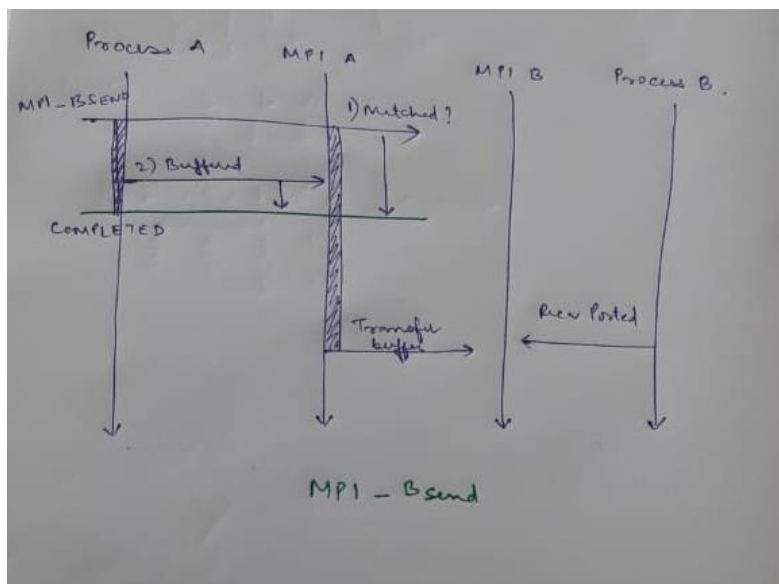
Send :

This is the standard mode. When it is called, (1) the message can be directly passed to the receive buffer, (2) the data is buffered (in temporary memory in the MPI implementation) or (3) the function waits for a receiving process to appear. See the picture below. Therefore, It can return quickly (1)(2) or block the process for a while (3). MPI decides which scenario is the best in terms of performance, memory, and so on. In any case, the data can be safely modified after the function returns.



Bsend :

It is the local blocking send. The programmer defines a local buffer when this function is called. If there is not a matching receive available, the process is blocked until the message is copied into the buffer. Therefore, the programmer can immediately modify the source data after the function returns.



2. Make a single process (the root process) collects data from the other processes in a group and combines them into a single data item. Compute the sum of the elements of an array that is distributed over several processors.

Use: Broadcasting and apply synchronization using barrier. (10 marks)

Single process broadcast program :

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

void my_broadcast(void *data, int count, MPI_Datatype
                  datatype, int root, MPI_Comm
                  communicator)
{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);
    if (world_rank == root)
    {
        int i;
        for (i = 0; i < world_size; i++)
        {
            if (i != world_rank)
            {
                MPI_Send(data, count, datatype, i, 0,
                          communicator);
            }
        }
    }
    else
    {
        MPI_Recv(data, count, datatype, root, 0, communicator,
                 MPI_STATUS_IGNORE);
    }
}
```

```

int main (int argc, char ** argv)
{
    if (argc != 3)
    {
        printf (stderr, "Received data from root process\n");
        exit(1);
    }

    int num-elements = atoi(argv[1]);
    int num-trials = atoi(argv[2]);
    MPI_Init(NULL, NULL);
    int world-rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world-rank);
    double total-my-bcast-time = 0.0;
    double total-mpi-bcast-time = 0.0;
    int i;
    int *data = (int *) malloc(sizeof(int) * num-elements);
    assert (data != NULL);
for (i = 0; i < num-trials; i++)
{
    MPI_Barrier(MPI_COMM_WORLD);
}
    free(data);
    MPI_Finalize();
}

```

```

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

void my_bcast(void *data, int count, MPI_Datatype datatype, int
root,
              MPI_Comm communicator)
{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);

    if (world_rank == root)
    {
        int i;
        for (i = 0; i < world_size; i++)
        {
            if (i != world_rank)

```

```

        {
            MPI_Send(data, count, datatype, i, 0,
communicator);
        }
    }
}
else
{
    MPI_Recv(data, count, datatype, root, 0, communicator,
MPI_STATUS_IGNORE);
}
}
int main(int argc, char **argv)
{
    if (argc != 3)
    {
        fprintf(stderr, "Recieved data from root process\n");
        exit(1);
    }

    int num_elements = atoi(argv[1]);
    int num_trials = atoi(argv[2]);

    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    double total_my_bcast_time = 0.0;
    double total_mpi_bcast_time = 0.0;
    int i;
    int *data = (int *)malloc(sizeof(int) * num_elements);
    assert(data != NULL);

    for (i = 0; i < num_trials; i++)
    {
        MPI_Barrier(MPI_COMM_WORLD);
        total_my_bcast_time -= MPI_Wtime();
        my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        total_my_bcast_time += MPI_Wtime();
    }
}

```


}

Output Screenshot :

```
kulvir06@ubuntu:~/Desktop/PDC/MPI$
```

Sum of array :

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define n 10
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int arr a2[1000];
int main(int argc, char *argv[])
{
    int pid, np, elements-per-process, n-elements-received;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (pid == 0)
    {
        int index, i;
        elements-per-process = n/np;
        if (np > 1)
        {
            for (i = 1; i < np - 1; i++)
            {
                index = i * elements-per-process;
                MPI_Send(&elements-per-process, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                MPI_Send(&a[index], elements-per-process, MPI_INT, i, 0, MPI_COMM_WORLD);
            }
        }
        int sum = 0;
        for (i = 0; i < elements-per-process; i++)
            sum += a[i];
        int tmp;
        for (i = 1; i < np; i++)
        {
            MPI_Recv(&tmp, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
            int sender = status.MPI_SOURCE;
            sum += tmp;
        }
    }
}
```

```

printf("Sum of array is : %d\n", sum);
} else
{
    MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0,
            MPI_COMM_WORLD, &status);
    MPI_Recv(&a2, n_elements_received, MPI_INT, 0, 0,
            MPI_COMM_WORLD, &status);

    int partial_sum = 0;
    for (int i = 0; i < n_elements_received; i++)
        partial_sum += a2[i];

    MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI-
            COMM_WORLD);
}

MPI_Finalize();
return 0;
}

```

```

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define n 10
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int a2[1000];
int main(int argc, char *argv[])
{
    int pid, np, elements_per_process, n_elements_recieved;
    MPI_Status status;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (pid == 0)
    {
        int index, i;
        elements_per_process = n / np;
        if (np > 1)
        {
            for (i = 1; i < np - 1; i++)
            {

```

```

        index = i * elements_per_process;

        MPI_Send(&elements_per_process,
                  1, MPI_INT, i, 0,
                  MPI_COMM_WORLD);
        MPI_Send(&a[index],
                  elements_per_process,
                  MPI_INT, i, 0,
                  MPI_COMM_WORLD);
    }
    index = i * elements_per_process;
    int elements_left = n - index;

    MPI_Send(&elements_left,
              1, MPI_INT,
              i, 0,
              MPI_COMM_WORLD);
    MPI_Send(&a[index],
              elements_left,
              MPI_INT, i, 0,
              MPI_COMM_WORLD);
}

// master process add its own sub array
int sum = 0;
for (i = 0; i < elements_per_process; i++)
    sum += a[i];
int tmp;
for (i = 1; i < np; i++)
{
    MPI_Recv(&tmp, 1, MPI_INT,
              MPI_ANY_SOURCE, 0,
              MPI_COMM_WORLD,
              &status);
    int sender = status.MPI_SOURCE;

    sum += tmp;
}
printf("Sum of array is : %d\n", sum);
}
else
{

```

```

    MPI_Recv(&n_elements_recieved,
             1, MPI_INT, 0, 0,
             MPI_COMM_WORLD,
             &status);
    MPI_Recv(&a2, n_elements_recieved,
             MPI_INT, 0, 0,
             MPI_COMM_WORLD,
             &status);
    int partial_sum = 0;
    for (int i = 0; i < n_elements_recieved; i++)
        partial_sum += a2[i];
    MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
MPI_Finalize();

return 0;
}

```

Output Screenshot :

```

kulvir06@ubuntu:~/Desktop/PDC/MPI$ mpicc sum.c -o sum
kulvir06@ubuntu:~/Desktop/PDC/MPI$ mpirun -np 4 ./sum
Sum of array is : 55
kulvir06@ubuntu:~/Desktop/PDC/MPI$

```