```
>mpicc src.c -o outputfile //(no xtn)
>mpirun -np n outputfile  //(n-number of processes to run)
```

1. Write an MPI C program to send numbers (Send & Recv)

```
#include <stdio.h>
#include "mpi.h“
intmain(intargc, char** argv)
{ /** send_recv_count.c**/
        int             my_rank, numbertoreceive[10], numbertosend[3]={73, 2, -16};
        int             recv_count, i;
        MPI_Status      status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        if (my_rank==0)
        {
                MPI_Recv( numbertoreceive, 3, MPI_INT, MPI_ANY_SOURCE,
                                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                printf("status.MPI_SOURCE= %d\n", status.MPI_SOURCE);
                printf("status.MPI_TAG= %d\n", status.MPI_TAG);
                printf("status.MPI_ERROR= %d\n", status.MPI_ERROR);
                MPI_Get_count(&status, MPI_INT, &recv_count);
                printf("Receive %d data\n", recv_count);
                for(i= 0; i< recv_count; i++)
                        printf("recv[%d] = %d\n", i, numbertoreceive[i]);
        }
        else
                MPI_Send( numbertosend, 3, MPI_INT, 0, 10, MPI_COMM_WORLD);
        MPI_Finalize();
        return 0;
}
```

2. Summation using parallelization

```
#include<iostream.h>
#include<mpi.h>
/**** add numbers from 1 to 1000 *****/
int main(intargc, char ** argv)
{
        int             mynode, totalnodes;
        int             sum,startval,endval,accum;
        MPI_Status      status;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
        MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
```

```
        sum = 0;
        startval= 1000*mynode/totalnodes+1;
        endval= 1000*(mynode+1)/totalnodes;
        for(inti=startval;i<=endval;i=i+1)
                sum = sum + i;
        if(mynode!=0)
                MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
        else
                for(intj=1;j<totalnodes;j=j+1)
                {
                        MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
                        sum = sum + accum;
                }
        if (mynode== 0)
                cout<< "The sum from 1 to 1000 is: " << sum << endl;
        MPI_Finalize();
}
```

3. **Write an MPI Program to demonstrate the "Deadlock" scenario i.e Two processes waiting to Receive from another**

```
#include <stdio.h>
#include "mpi.h“
/* process 0 receive a number from and send a number from process 1.
process 1 receive a number from and send a number to process 0 */
int main(int argc, char** argv)
{
        int             my_rank, numbertoreceive, numbertosend = -16;
        MPI_Status      status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        if (my_rank==0){
                MPI_Recv( &numbertoreceive, 1, MPI_INT, 1, 20, MPI_COMM_WORLD,
                                        &status);
                MPI_Send( &numbertosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
        }
        else if(my_rank == 1)
        {
```

```
                MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
                        &status);
                MPI_Send( &numbertosend, 1, MPI_INT, 0, 20, MPI_COMM_WORLD);
        }
        MPI_Finalize();
        return 0;
    }
```

4. MPI program using MPI_Send and MPI_Recv to pass a message around in a ring.

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

int main(int argc, char** argv)
{
    int world_rank;
    int world_size;
    int token;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Receive from the lower process and send to the higher process. Take care
    // of the special case when you are the first process to prevent deadlock.
    if (world_rank != 0) {
        MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Process %d received token %d from process %d\n", world_rank, token,
               world_rank - 1);
    }
```

```
    else {
            // Set the token's value if you are process 0
            token = -1;
    }
    MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size, 0,
                        MPI_COMM_WORLD);
    // Now process 0 can receive from the last process. This makes sure that at
    // least one MPI_Send is initialized before all MPI_Recvs (again, to prevent deadlock)
    if (world_rank == 0) {
            MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
            printf("Process %d received token %d from process %d\n", world_rank, token,
                        world_size - 1);
    }
    MPI_Finalize();
}
```

5. MPI Program involving Two processes ping pong a number back and forth, incrementing it until it reaches a given value.

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

int main(int argc, char** argv)
{
    const int PING_PONG_LIMIT = 10;

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Find out rank, size
```

```
 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// We are assuming 2 processes for this task p-2-p
if (world_size != 2) {
        fprintf(stderr, "World size must be two for %s\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
}

int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
    if (world_rank == ping_pong_count % 2) {
        // Increment the ping pong count before you send it
        ping_pong_count++;
        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
MPI_COMM_WORLD);
        printf("%d sent and incremented ping_pong_count %d to %d\n",
                        world_rank, ping_pong_count, partner_rank);
    }
    else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        printf("%d received ping_pong_count %d from %d\n",
                    world_rank, ping_pong_count, partner_rank);
    }
}
MPI_Finalize();
```

```
}
```

6. MPI Program to send and receive messages

```
//The process with rank 0 receives the messages,
//while all other processes transmit them:

 // MPI MPMD
 #include <stdio.h>
 #include <string.h>
 #include <mpi.h>

 #define MAX_DATA 1000
 #define MAX_MSG (MAX_DATA+20)

 int main(int argc, char** argv) {
     MPI_Status status;
     int rank, ip, np;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &np);
     if (rank == 0) {
         char msg[MAX_MSG + 1];
         for (ip = 1; ip < np; ip++) {
             MPI_Recv(msg, MAX_MSG, MPI_CHAR,
                 MPI_ANY_SOURCE, MPI_ANY_TAG,
                       MPI_COMM_WORLD, &status);
             int nchar;
             MPI_Get_count(&status, MPI_CHAR, &nchar);
             msg[nchar] = '\0';
             printf("%s\n", msg);
         }
         printf("You bet!!\n");
     }
     else {
         char msg[MAX_MSG + 1];
         char data[MAX_DATA + 1] = "Good Morning Sunshine!";
         sprintf_s(msg, MAX_MSG, "Process %d says %s", rank,
data);
         int len = strlen(msg);
         MPI_Send(msg, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
     }
     MPI_Finalize();
     return 0;
 }
```

7. MPI Program for parallel integration using non blocking calls

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
/* Prototype */
void other_work(int myid);
float integral(float ai, float h, int n);

int main(int argc, char* argv[])
{
        int             n, p, myid, tag, master, proc, ierr;
        float           h, integral_sum, a, b, ai, pi, my_int;
        MPI_Comm        comm;
        MPI_Request     request;
        MPI_Status      status;

        comm = MPI_COMM_WORLD;
        ierr = MPI_Init(&argc,&argv);         /* starts MPI */
        MPI_Comm_rank(comm, &myid);         /* get current process id */
        MPI_Comm_size(comm, &p);            /* get number of processes */

        master = 0;
        pi = acos(-1.0);  /* = 3.14159... */
        a = 0.;          * lower limit of integration */
        b = pi*1./2.;     /* upper limit of integration */
        n = 500;         /* number of increment within each process */
        tag = 123;       /* set the tag to identify this particular job */
        h = (b-a)/n/p;    /* length of increment */

        ai = a + myid*n*h;  /* lower limit of integration for partition myid */
        my_int = integral(ai, h, n);    /* 0<=myid<=p-1 */

        printf("Process %d has the partial result of %fn", myid, my_int);

        if(myid == master)
        {
                integral_sum = my_int;
                for (proc=1;proc<p;proc++) {
                        MPI_Recv(&my_int, 1, MPI_FLOAT,    /* triplet of buffer, size, data
                                                                type */
                                MPI_ANY_SOURCE,      /* message source */
                                MPI_ANY_TAG,         /* message tag */
                                comm, &status);      /* status identifies source, tag */
                        integral_sum += my_int;
        }
```

```
            printf("The Integral =%fn",integral_sum); /* sum of my_int */
        }
        else {
            MPI_Isend(      /* non-blocking send */
                            &my_int, 1, MPI_FLOAT,      /* triplet of buffer, size, data type */
                            master,tag,
                            comm, &request);      /* send my_int to master */
            other_work(myid);
            MPI_Wait(&request, &status);    /* block until Isend is done */
        }
        MPI_Finalize();                 /* let MPI finish up ... */
    }
    void other_work(int myid)
    {
        printf("more work on process %dn", myid);
    }

    float integral(float ai, float h, int n)
    {
        int j;
        float aij, integ;

        integ = 0.0;            /* initialize */
        for (j=0;j<j++) {       /* sum integrals */
         aij = ai + (j+0.5)*h;     /* mid-point */
         integ += cos(aij)*h;
        }
        return integ;
}
```

8. Given a matrix:

1 2 3 4 5 6

7 8 9 10 11 12

13 14 15 16 17 18

19 20 21 22 23 23

24 25 26 27 28 29

Create a submatrix that can look like:

1 3 5

13 15 17

24 26 28

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

```
int main(argc, argv)
int argc;
char **argv;
{
   int myrank, size ;
   int i, j, mym = 10, myn = 10;
   int a[10][10], b[5][5], c[5][5] ;
   MPI_Datatype subrow, submatrix ;
   MPI_Status status ;
   MPI_Aint sizeofint ;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   /* Initialize the local array */
   for (i = 0; i < mym; i++)
     for (j = 0; j < myn; j++)
           a[i][j] = i*myn + j + 1;

   /* Print the local matrix */
   for (i = 0; i < mym; i++){
     for (j = 0; j < myn; j++)
           printf("%d ", a[i][j]) ;
     printf("\n");
   }
   /* Create a submatrix datatype */
   /* Create datatype for the sub-row */
   MPI_Type_vector(5,1,2,MPI_INT,&subrow);
   /* Create datatype for the sub-matrix */
```

```
  MPI_Type_hvector(5,1,20*sizeof(int),subrow,&submatrix);
   /* Commit the datatype created */
   MPI_Type_commit(&submatrix);
   /* Send it to self and print it */

MPI_Sendrecv(a,1,submatrix,myrank,201,c,25,MPI_INT,myrank,201,MPI_COMM_WORLD,&
                                status);
   /* Print the submatrix */
   for (i = 0; i < 5; i++){
    for (j = 0; j < 5; j++)
          printf("%d ", c[i][j]) ;
    printf("\n");
   }
   MPI_Finalize();
}
```

9. MPI program to demonstrate the use of parallel processing for array elements addition

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

// size of array
#define n 10

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Temporary array for slave process
int a2[1000];

int main(int argc, char* argv[])
{

        int pid, np,
                elements_per_process,
                n_elements_recieved;
        // np -> no. of processes
        // pid -> process id
```

```
MPI_Status status;

// Creation of parallel processes
MPI_Init(&argc, &argv);

// find out process ID,
// and how many processes were started
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &np);

// master process
if (pid == 0) {
        int index, i;
        elements_per_process = n / np;

        // check if more than 1 processes are run
        if (np > 1) {
                // distributes the portion of array
                // to child processes to calculate
                // their partial sums
                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);
                }

                // last process adds remaining elements
                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];

		// collects partial sums from other processes
		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;
		}

		// prints the final sum of array
		printf("Sum of array is : %d\n", sum);
	}
	// slave processes
	else {
		MPI_Recv(&n_elements_recieved,
				1, MPI_INT, 0, 0,
				MPI_COMM_WORLD,
				&status);

		// stores the received array segment
		// in local array a2
		MPI_Recv(&a2, n_elements_recieved,
				MPI_INT, 0, 0,
				MPI_COMM_WORLD,
				&status);

		// calculates its partial sum
		int partial_sum = 0;
		for (int i = 0; i < n_elements_recieved; i++)
			partial_sum += a2[i];

		// sends the partial sum to the root process
		MPI_Send(&partial_sum, 1, MPI_INT,
				0, 0, MPI_COMM_WORLD);
	}

	// cleans up all MPI state before exit of process
	MPI_Finalize();

	return 0;
}
```

10. MPI program to demonstrate the use status objects in a wait for random amount of time

```
// Example of checking the MPI_Status object from an MPI_Recv call
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

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

        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
        if (world_size != 2) {
                fprintf(stderr, "Must use two processes for this example\n");
                MPI_Abort(MPI_COMM_WORLD, 1);
        }
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        const int MAX_NUMBERS = 100;
        int numbers[MAX_NUMBERS];
        int number_amount;
        if (world_rank == 0) {
                // Pick a random amount of integers to send to process one
                srand(time(NULL));
                number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;
                // Send the amount of integers to process one
                MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
                printf("0 sent %d numbers to 1\n", number_amount);
        }
       else if (world_rank == 1) {
                MPI_Status status;
                // Receive at most MAX_NUMBERS from process zero
                MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD,
                                &status);
                // After receiving the message, check the status to determine how many
                // numbers were actually received
                MPI_Get_count(&status, MPI_INT, &number_amount);
                // Print off the amount of numbers, and also print additional information
                // in the status object
                printf("1 received %d numbers from 0. Message source = %d, tag = %d\n",
                                number_amount, status.MPI_SOURCE, status.MPI_TAG);
        }
        MPI_Barrier(MPI_COMM_WORLD);
```

```
        MPI_Finalize();
}
```