1.Demonstration of MPI_Send and MPI_Recv

```
#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 (size < 2) {
        if (rank == 0)
            printf("Run with at least 2 processes\n");
        MPI_Finalize();
        return 0;
    }

    if (rank == 0) {
        int x = 10;
        MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank 0 sent %d\n", x);
    } else if (rank == 1) {
        int y;
        MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank 1 received %d\n", y);
    }

    MPI_Finalize();
    return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~/mpi$ mpicc pg1.c -o pg1
chirudeep@DESKTOP-HTLOV56:~/mpi$ mpirun -np 2 ./pg1


Rank 0 sent 10
Rank 1 received 10
```

2. Demonstration of MPI_Scatter and MPI_Gather

```
#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);

    int send[100], recv, gather[100];

    if (rank == 0) {
        for (int i = 0; i < size; i++)
            send[i] = i + 1;   // simple numbers 1,2,3...
    }

    MPI_Scatter(send, 1, MPI_INT, &recv, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Rank %d received %d\n", rank, recv);

    recv = recv * 10; // small operation

    MPI_Gather(&recv, 1, MPI_INT, gather, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Gathered: ");
        for (int i = 0; i < size; i++)
            printf("%d ", gather[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~/mpi$ gedit pg2.c
chirudeep@DESKTOP-HTLOV56:~/mpi$ mpicc pg2.c -o sg
chirudeep@DESKTOP-HTLOV56:~/mpi$ mpirun -np 4 ./sg

Rank 1 received 2
Rank 2 received 3
Rank 3 received 4
Rank 0 received 1
Gathered: 10 20 30 40
```

3.Demonstration of MPI Broadcast operation

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

int main(int argc, char *argv[])
{
   int rank, size, data;

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

   if (rank == 0) {
      data = 100;  // value to broadcast
      printf("Root process sending: %d\n", data);
   }

   MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

   printf("Process %d received data = %d\n", rank, data);

   MPI_Finalize();
   return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~/mpi$ gedit pg3.c
chirudeep@DESKTOP-HTLOV56:~/mpi$ mpicc pg3.c -o broadcast
mpirun -np 4 ./broadcast

Process 2 received data = 100
Process 3 received data = 100
Root process sending: 100
Process 0 received data = 100
Process 1 received data = 100
```

4. Demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD).

REDUCE

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

int main(int argc, char *argv[])
{
   int rank, size, value, sum, max, min, prod;

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

   value = rank + 1; // simple value for each process

   MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   MPI_Reduce(&value, &max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
   MPI_Reduce(&value, &min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
   MPI_Reduce(&value, &prod,1, MPI_INT, MPI_PROD,0, MPI_COMM_WORLD);

   if (rank == 0) {
      printf("SUM  = %d\n", sum);
      printf("MAX  = %d\n", max);
      printf("MIN  = %d\n", min);
      printf("PROD = %d\n", prod);
   }

   MPI_Finalize();
   return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~/mpi$ mpicc pg4a.c -o reduce
mpirun -np 4 ./reduce

SUM  = 10
MAX  = 4
MIN  = 1
PROD = 24
```

ALLREDUCE

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

int main(int argc, char *argv[])
{
   int rank, size, value, sum;

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

   value = rank + 1;

   MPI_Allreduce(&value, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

   printf("Process %d got SUM = %d\n", rank, sum);

   MPI_Finalize();
   return 0;
}
```

```
mpicc pg4b.c -o allreduce
mpirun -np 4 ./allreduce

Process 2 got SUM = 10
Process 3 got SUM = 10
Process 0 got SUM = 10
Process 1 got SUM = 10
```

5. Write an OpenMP program that computes the sum of the first N integers using a parallel for loop. Use the #pragma omp for directive along with the private and reduction clauses.

```
#include <stdio.h>
#include <omp.h>

int main()
{
   int i, N = 100, sum = 0;

   #pragma omp parallel private(i) reduction(+:sum)
   {
      #pragma omp for
      for (i = 1; i <= N; i++)
         sum += i;
   }

   printf("Sum = %d\n", sum);
   return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~$ gcc pg5.c -o pg5 -fopenmp
chirudeep@DESKTOP-HTLOV56:~$ ./pg5

Sum = 5050
```

6.Write an OpenMP program to compute the Fibonacci sequence using task parallelism. The program should use a recursive function where each Fibonacci computation for fib(n-1) and fib(n-2) is created as an independent task.

```
#include <stdio.h>
#include <omp.h>

int fib(int n)
{
   int x, y;

   if (n < 2)
      return n;

   #pragma omp task shared(x)
   x = fib(n - 1);

   #pragma omp task shared(y)
   y = fib(n - 2);

   #pragma omp taskwait
   return x + y;
}

int main()
{
   int n = 10, result;

   #pragma omp parallel
   {
      #pragma omp single
      result = fib(n);
   }

   printf("Fibonacci(%d) = %d\n", n, result);
   return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~$ gcc pg6.c -o pg6 -fopenmp
chirudeep@DESKTOP-HTLOV56:~$ ./pg6

Fibonacci(10) = 55
```

7.Estimate the value of pi using:Parallelize the code by removing loop carried dependency.

```
#include <stdio.h>
#include <omp.h>

int main()
{
   long int N = 10000000;
   double pi = 0.0;

   #pragma omp parallel for reduction(+:pi)
   for (long int i = 0; i < N; i++)
   {
      double term = (i % 2 == 0 ? 1.0 : -1.0) / (2.0 * i + 1.0);
      pi += term;   // loop carried dependency removed using reduction
   }

   pi = 4.0 * pi;

   printf("Estimated PI = %lf\n", pi);
   return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~$ gcc pg7.c -o pg7 -fopenmp
chirudeep@DESKTOP-HTLOV56:~$ ./pg7

Estimated PI = 3.141593
```

8.Write a parallel C program using OpenMP in which each thread obtains its thread number and adds it to a global shared variable. Use the #pragma omp critical directive to avoid race conditions. Print the intermediate updates and the final sum

```
#include <stdio.h>
#include <omp.h>

int main()
{
   int sum = 0;

   #pragma omp parallel shared(sum)
   {
      int tid = omp_get_thread_num();

      #pragma omp critical
      {
         sum += tid;
         printf("Thread %d added its ID → Current sum = %d\n", tid, sum);
      }
   }

   printf("Final Sum = %d\n", sum);
   return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~$ gedit pg8.c
chirudeep@DESKTOP-HTLOV56:~$ gcc pg8.c -o pg8 -fopenmp
chirudeep@DESKTOP-HTLOV56:~$ ./pg8

Thread 10 added its ID → Current sum = 10
Thread 8 added its ID → Current sum = 18
Thread 7 added its ID → Current sum = 25
Thread 9 added its ID → Current sum = 34
Thread 6 added its ID → Current sum = 40
Thread 0 added its ID → Current sum = 40
Thread 11 added its ID → Current sum = 51
Thread 2 added its ID → Current sum = 53
Thread 4 added its ID → Current sum = 57
Thread 1 added its ID → Current sum = 58
Thread 3 added its ID → Current sum = 61
Thread 5 added its ID → Current sum = 66
Final Sum = 66
```

9.Write a program to sort an array of n elements using both sequential and parallel merge sort (using Section). Record the difference in execution time

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

void merge(int a[], int l, int m, int r)
{
   int i = l, j = m + 1, k = 0;
   int *temp = malloc((r - l + 1) * sizeof(int));

   while (i <= m && j <= r)
      temp[k++] = (a[i] < a[j]) ? a[i++] : a[j++];

   while (i <= m) temp[k++] = a[i++];
   while (j <= r) temp[k++] = a[j++];

   for (i = l, k = 0; i <= r; i++, k++)
      a[i] = temp[k];

   free(temp);
}

void mergeSort(int a[], int l, int r)
{
   if (l < r)
   {
      int m = (l + r) / 2;
      mergeSort(a, l, m);
      mergeSort(a, m + 1, r);
      merge(a, l, m, r);
   }
}

void parallelMergeSort(int a[], int l, int r)
{
   if (l < r)
   {
      int m = (l + r) / 2;
```

```
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSort(a, l, m);

            #pragma omp section
            mergeSort(a, m + 1, r);
        }

        merge(a, l, m, r);
    }
}

int main()
{
    int n = 100000;
    int *a1 = malloc(n * sizeof(int));
    int *a2 = malloc(n * sizeof(int));

    for (int i = 0; i < n; i++)
        a1[i] = a2[i] = rand() % 1000;

    double t1 = omp_get_wtime();
    mergeSort(a1, 0, n - 1);
    t1 = omp_get_wtime() - t1;

    double t2 = omp_get_wtime();
    parallelMergeSort(a2, 0, n - 1);
    t2 = omp_get_wtime() - t2;

    printf("Sequential Time = %lf sec\n", t1);
    printf("Parallel Time   = %lf sec\n", t2);

    free(a1);
    free(a2);

    return 0;
}
```

```
chirudeep@DESKTOP-HTLOV56:~$ gcc pg9.c -o pg9 -fopenmp
chirudeep@DESKTOP-HTLOV56:~$ ./pg9

Sequential Time = 0.024102 sec
Parallel Time   = 0.027413 sec
```