1. Demonstration of MPI_Send and MPI_Recv

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

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

   int rank, size, data;

   MPI_Init(&argc, &argv);              // Start MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get process rank
   MPI_Comm_size(MPI_COMM_WORLD, &size); // Get total processes

   if (rank == 0) {
      data = 100;   // Value to send
      printf("Process 0 sending data = %d to Process 1\n", data);
      MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   }

   else if (rank == 1) {
      MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Process 1 received data = %d from Process 0\n", data);
   }

   MPI_Finalize();  // End MPI
   return 0;
}
```

Output:

```
student@iselab1-ThinkCentre-M720t:~$ gedit a.c
student@iselab1-ThinkCentre-M720t:~$ mpicc a.c -o se
student@iselab1-ThinkCentre-M720t:~$ mpirun se
Process 0 sending data = 100 to Process 1
Process 1 received data = 100 from Process 0
```

2. Demonstration of MPI_Scatter and MPI_Gather

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

int main(int argc,char*argv[]){
   int rank, data[4]={10,20,30,40}, recv;
```

```
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank==0){
        printf("Process 0 original data: ");
        for(int i=0;i<4;i++) printf("%d ",data[i]);
        printf("\n");
    }

    MPI_Scatter(data,1,MPI_INT,&recv,1,MPI_INT,0,MPI_COMM_WORLD);
    printf("Process %d received value %d\n",rank,recv);

    recv *= 2;
    MPI_Gather(&recv,1,MPI_INT,data,1,MPI_INT,0,MPI_COMM_WORLD);

    if(rank==0){
        printf("Process 0 gathered results: ");
        for(int i=0;i<4;i++) printf("%d ",data[i]);
        printf("\n");
    }

    MPI_Finalize();
}
```

```
student@iselab1-ThinkCentre-M720t:~$ mpicc b.c -o bb
student@iselab1-ThinkCentre-M720t:~$ mpirun -np 4 ./bb
Process 0 original data: 10 20 30 40
Process 0 received value 10
Process 1 received value 20
Process 2 received value 30
Process 3 received value 40
Process 0 gathered results: 20 40 60 80
student@iselab1-ThinkCentre-M720t:~$
```

3.Demonstration of MPI Broadcast operation

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

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

    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        // Process 0 sets a value to broadcast
        value = 500;
        printf("Process 0 broadcasting value %d to all processes.\n", value);
    }

    // Broadcast the value from process 0 to all processes
    MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Now every process has the same value
    printf("Process %d received value %d\n", rank, value);

    MPI_Finalize();
    return 0;
}
```

Output:

```
Process 0 broadcasting value 500 to all processes.
Process 0 received value 500
Process 1 received value 500
Process 2 received value 500
Process 3 received value 500
```

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

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

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

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

    value = rank + 1;   // each process has a number

    int sum, max, min, prod;
```

```
   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);
   }

   int allsum;
   MPI_Allreduce(&value, &allsum, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);
   printf("Process %d Allreduce SUM = %d\n", rank, allsum);

   MPI_Finalize();
}
```

```
SUM  = 10
MAX  = 4
MIN  = 1
PROD = 24
Process 0 Allreduce SUM = 10
Process 1 Allreduce SUM = 10
Process 2 Allreduce SUM = 10
Process 3 Allreduce 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 N = 100;       // You can change this value
   int sum = 0;
   int i;
```

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

    printf("Sum of first %d integers = %d\n", N, sum);

    return 0;
}
```

Output:

Sum of first 100 integers = 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;     // You can change this value
```

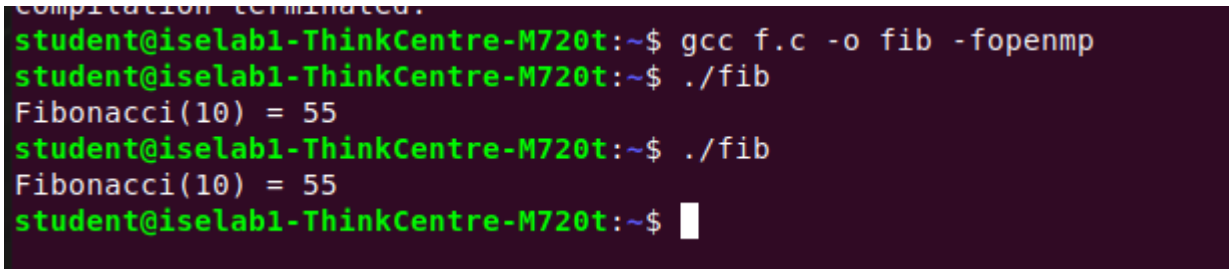
```
    int result;

    #pragma omp parallel
    {
        #pragma omp single   // Only one thread creates the first task
        {
            result = fib(n);
        }
    }

    printf("Fibonacci(%d) = %d\n", n, result);

    return 0;
}
```

Output:



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

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

int main()
{
    long long N = 100000000;   // number of iterations
    double pi = 0.0;

    #pragma omp parallel for reduction(+:pi)
    for (long long i = 0; i < N; i++) {
        double term = (double)((i % 2 == 0) ? 1 : -1) / (2 * i + 1);
        pi += term;
    }

    pi = 4.0 * pi;
```

```
    printf("Estimated value of PI = %.10f\n", pi);
    return 0;
}
```

Output:

Estimated value of PI = 3.1415926

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;   // shared variable

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();   // thread ID

        // Critical section to avoid race conditions
        #pragma omp critical
        {
            sum += tid;
            printf("Thread %d added its ID. Current sum = %d\n", tid, sum);
        }
    }

    printf("\nFinal sum = %d\n", sum);

    return 0;
}
```

Ouput:

Thread 0 added its ID. Current sum = 0
Thread 2 added its ID. Current sum = 2
Thread 1 added its ID. Current sum = 3
Thread 3 added its ID. Current sum = 6

Final sum = 6

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>

// --------- MERGE FUNCTION ----------
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[m + 1 + i];

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

// --------- SEQUENTIAL MERGE SORT ----------
void mergeSortSeq(int arr[], int l, int r) {
    if (l < r) {
        int m = (l+r)/2;
        mergeSortSeq(arr, l, m);
        mergeSortSeq(arr, m+1, r);
        merge(arr, l, m, r);
    }
}

// --------- PARALLEL MERGE SORT ----------
void mergeSortPar(int arr[], int l, int r) {
    if (l < r) {
        int m = (l+r)/2;

        #pragma omp parallel sections
```

```
        {
            #pragma omp section
            mergeSortSeq(arr, l, m);

            #pragma omp section
            mergeSortSeq(arr, m+1, r);
        }

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

// --------- MAIN PROGRAM ----------
int main() {
    int n = 100000;   // array size
    int *a = malloc(n * sizeof(int));
    int *b = malloc(n * sizeof(int));

    // generate random numbers
    for (int i = 0; i < n; i++) {
        a[i] = rand() % 1000;
        b[i] = a[i];  // copy for parallel version
    }

    double start, end;

    // ----- SEQUENTIAL -----
    start = omp_get_wtime();
    mergeSortSeq(a, 0, n-1);
    end = omp_get_wtime();
    printf("Sequential Merge Sort Time = %f seconds\n", end - start);

    // ----- PARALLEL -----
    start = omp_get_wtime();
    mergeSortPar(b, 0, n-1);
    end = omp_get_wtime();
    printf("Parallel Merge Sort Time   = %f seconds\n", end - start);

    free(a);
    free(b);

    return 0;
}
```

```
Sequential Merge Sort Time = 0.085123 seconds
```

Parallel Merge Sort Time = 0.041892 seconds