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;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Identify which process you are
    if(rank == 0) {
        long long number = 77770778589LL;       // Data to send (use long long)
        MPI_Send(&number, 1, MPI_LONG_LONG, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent number %lld to Process 1.\n", number);
    }
    else if(rank == 1) {
        long long received_number;
        MPI_Recv(&received_number, 1, MPI_LONG_LONG, 0, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %lld from Process 0.\n", received_number);
    }
    MPI_Finalize();
    return 0;
}
```

Output

Process 1 received number 77770778589 from Process 0.

Process 0 sent number 77770778589 to Process 1.

2. Demonstration of MPI_Scatter and MPI_Gather

```
#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);
int send_data[10];   // Only root will use full data
int recv_value;      // Each process receives one value
// Root initializes data
if (rank == 0) {
    for (int i = 0; i < size; i++) {
        send_data[i] = (i + 1) * 10;   // 10, 20, 30, 40,...
    }
    printf("Root data: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", send_data[i]);
    }
    printf("\n");
}

MPI_Scatter(send_data,      // send buffer (root)
            1, MPI_INT,     // 1 integer to each process
            &recv_value,    // each process receives one integer
            1, MPI_INT,
            0, MPI_COMM_WORLD);

printf("Rank %d received %d\n", rank, recv_value);
recv_value *= 2;
int gathered[10];   // Root collects results here
MPI_Gather(&recv_value, 1, MPI_INT,
           gathered,     1, MPI_INT,
           0, MPI_COMM_WORLD);
```

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

Output

```
Root data: 10 20 30 40
Rank 0 received 10
Rank 1 received 20
Rank 2 received 30
Rank 3 received 40
Gathered Result: 20 40 60 80
```

3. Demonstration of MPI Broadcast operation

```
#include<stdio.h>
#include<mpi.h>
int main(int argc,char ** argv){
MPI_Init(&argc,&argv);
int rank,data;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0){
data=100;
printf("root data is:%d\n",data);
}
MPI_Bcast(&data,1,MPI_INT,0,MPI_COMM_WORLD);
```

```
printf("processor %d received data:%d\n",rank,data);
MPI_Finalize();
return 0;
}
```

Output

```
root data is:100
processor 0 received data:100
processor 1 received data:100
processor 2 received data:100
processor 3 received data:100
processor 4 received data:100
processor 5 received data:100
processor 6 received data:100
processor 7 received data:100
```

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

```
#include<mpi.h>
#include<stdio.h>
int main(int argc,char **argv){
MPI_Init(&argc,&argv);
int  rank,size;
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
int sum,max,min,prod;
int asum,amax,amin,aprod;
int value=rank+1;
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",sum);
printf("prod:%d",prod);
printf("min:%d",min);
printf("max:%d",max);
}
printf("\n");
MPI_Allreduce(&value,&asum,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
MPI_Allreduce(&value,&amax,1,MPI_INT,MPI_MAX,MPI_COMM_WORLD);
MPI_Allreduce(&value,&amin,1,MPI_INT,MPI_MIN,MPI_COMM_WORLD);
MPI_Allreduce(&value,&aprod,1,MPI_INT,MPI_PROD,MPI_COMM_WORLD);

printf("rank=%d allsum=%d allmax=%d allmin=%d
allprod=%d\n",rank,asum,amax,amin,aprod);
MPI_Finalize();
return 0;
}
```

Output

```
sum: 36  prod: 40320  min: 1  max: 8
rank=0 allsum=36 allmax=8 allmin=1 allprod=40320
rank=1 allsum=36 allmax=8 allmin=1 allprod=40320
rank=2 allsum=36 allmax=8 allmin=1 allprod=40320
rank=3 allsum=36 allmax=8 allmin=1 allprod=40320
```

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 = 10;
    int a[10] = {1,2,3,4,5,6,7,8,9,10};
    int sum = 0;
    int i;   // required for private(i)
    #pragma omp parallel for private(i) reduction(+:sum)
    for (i = 0; i < n; i++) {
        sum += a[i];
    }
    printf("Parallel Sum = %d\n", sum);
    return 0;
}
```

Output

Parallel Sum = 55

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 <= 1)
        return n;
    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);
    #pragma omp taskwait        // wait for both tasks to finish
    return x + y;
}
```

```
int main() {
    int n = 10;
    int result;
    #pragma omp parallel
    {
        #pragma omp single      // only one thread starts the recursion
       {
            result = fib(n);
        }
    }
    printf("Fibonacci(%d) = %d\n", n, result);
    return 0;
}
```

Output

Fibonacci(10) = 55

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

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
int main(){
long long i,n=1000000;
long long count;
double x,y,pi;
#pragma omp parallel for private(i) reduction(+:count)
for(int i=0;i<n;i++){
x=(double)rand()/RAND_MAX;
y=(double)rand()/RAND_MAX;
if((x*x+y*y)<=1){
```

```
count++;
}
}
pi=4.0*(double)count/(double)(n);
printf("pi value is:%f",pi);
return 0;
}
```

Output

Pi value is : 3.140440

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
    {
        int tid = omp_get_thread_num();
        #pragma omp critical
        {
            sum += tid;
            printf("Thread %d updated sum to %d\n", tid, sum);
        }
    }
    printf("\nFinal Sum = %d\n", sum);
    return 0;
}
```

Output

Thread 0 updated sum to 0

Thread 1 updated sum to 1

Thread 3 updated sum to 4

Thread 2 updated sum to 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 <time.h>
#include <omp.h>
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) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    while (i < n1)
```

```
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}
void sequential_merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;

        sequential_merge_sort(arr, l, m);
        sequential_merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
void parallel_merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            parallel_merge_sort(arr, l, m);

            #pragma omp section
            parallel_merge_sort(arr, m + 1, r);
        }
        merge(arr, l, m, r);
    }
}
int main() {
    int A[] = {5, 3, 8, 9, 1, 2, 6, 4};
    int n = sizeof(A) / sizeof(A[0]);
```

```
    int seq[n];
    int par[n];
    for (int i = 0; i < n; i++) {
        seq[i] = A[i];
        par[i] = A[i];
    }
    // Sequential time
    double t1 = omp_get_wtime();
    sequential_merge_sort(seq, 0, n - 1);
    double t2 = omp_get_wtime();

    // Parallel time
    double t3 = omp_get_wtime();
    parallel_merge_sort(par, 0, n - 1);
    double t4 = omp_get_wtime();
    printf("\nSequential Time: %f seconds\n", t2 - t1);
    printf("Parallel Time:   %f seconds\n", t4 - t3);
    printf("\nSorted Array:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", par[i]);
    printf("\n");
    return 0;
}
```

Output

Sequential Time: 0.00001 seconds

Parallel Time: 0.00008 seconds

Sorted Array:

1 2 3 4 5 6 8 9