11.Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort(using Section). Record the difference in execution time.

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

#define MAX 1000000  // Max array size for testing

// Function to merge two subarrays
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

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

    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++];

    free(L);
    free(R);
}

// Sequential Merge Sort
void sequential_mergesort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        sequential_mergesort(arr, l, m);
        sequential_mergesort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Parallel Merge Sort using OpenMP sections
void parallel_mergesort(int arr[], int l, int r, int depth) {
    if (l < r) {
        int m = (l + r) / 2;
```

```c
        if (depth <= 0) {
            // Fall back to sequential if max depth reached
            sequential_mergesort(arr, l, r);
        } else {
            #pragma omp parallel sections
            {
                #pragma omp section
                parallel_mergesort(arr, l, m, depth - 1);

                #pragma omp section
                parallel_mergesort(arr, m + 1, r, depth - 1);
            }
            merge(arr, l, m, r);
        }
    }
}

void fill_array(int arr[], int n) {
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 10000;
}

void copy_array(int src[], int dest[], int n) {
    for (int i = 0; i < n; i++)
        dest[i] = src[i];
}

int main() {
    int n;
    printf("Enter number of elements (up to %d): ", MAX);
    scanf("%d", &n);

    if (n > MAX) {
        printf("Array size too large!\n");
        return 1;
    }

    int *arr_seq = (int *)malloc(n * sizeof(int));
    int *arr_par = (int *)malloc(n * sizeof(int));

    fill_array(arr_seq, n);
    copy_array(arr_seq, arr_par, n);

    // Sequential mergesort timing
    double start_seq = omp_get_wtime();
    sequential_mergesort(arr_seq, 0, n - 1);
    double end_seq = omp_get_wtime();

    // Parallel mergesort timing
    double start_par = omp_get_wtime();
    parallel_mergesort(arr_par, 0, n - 1, 4);  // depth = 4 gives 16 tasks max
    double end_par = omp_get_wtime();
```

```c
    printf("\nTime taken by Sequential MergeSort: %.6f seconds\n", end_seq - start_seq);
    printf("Time taken by Parallel MergeSort  : %.6f seconds\n", end_par - start_par);

    free(arr_seq);
    free(arr_par);
    return 0;
}
```

steps to run

```
gcc -fopenmp parallel_mergesort.c -o mergesort
./mergesort
```

output:Enter number of elements (up to 1000000): 500000

Time taken by Sequential MergeSort: 0.492137 seconds
Time taken by Parallel MergeSort : 0.213864 seconds

2.Write an OpenMP program that divides the Iterations into chunks containing 2 iterations, respectively (OMP_SCHEDULE=static,2). Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be the following:
a. Thread 0 : Iterations 0 −− 1
b. Thread 1 : Iterations 2 −− 3

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

int main() {
    int n;

    printf("Enter number of iterations: ");
    scanf("%d", &n);

    // Set number of threads (optional, you can also set OMP_NUM_THREADS env variable)
    omp_set_num_threads(2);

    #pragma omp parallel for schedule(static, 2)
    for (int i = 0; i < n; i++) {
        int tid = omp_get_thread_num();

        // To print only once per chunk, print when i % 2 == 0
        if (i % 2 == 0) {
            int chunk_start = i;
            int chunk_end = (i + 1 < n) ? i + 1 : i;
            printf("Thread %d : Iterations %d -- %d\n", tid, chunk_start, chunk_end);
        }
    }

    return 0;
}
```

steps to run

```
gcc -fopenmp omp_static_chunks.c -o omp_static_chunks
./omp_static_chunks
```

output:
Enter number of iterations: 8
Thread 0 : Iterations 0 -- 1
Thread 1 : Iterations 2 -- 3
Thread 0 : Iterations 4 -- 5
Thread 1 : Iterations 6 – 7

3.Write a OpenMP program to calculate n Fibonacci numbers using tasks.

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

// Recursive Fibonacci using OpenMP tasks
long long fib_task(int n) {
    if (n < 2)
        return n;

    long long x, y;

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

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

    #pragma omp taskwait
    return x + y;
}

int main() {
    int n;

    printf("Enter number of Fibonacci numbers to compute: ");
    scanf("%d", &n);

    long long *fib_array = (long long *)malloc(n * sizeof(long long));

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++) {
                // Create a task for each Fibonacci number calculation
                #pragma omp task shared(fib_array)
                fib_array[i] = fib_task(i);
            }
        }
    }

    // Wait for all tasks to finish before printing
    #pragma omp taskwait

    printf("Fibonacci numbers:\n");
    for (int i = 0; i < n; i++) {
        printf("fib(%d) = %lld\n", i, fib_array[i]);
    }

    free(fib_array);
```

```
    return 0;
}
```

steps to run:

```
gcc -fopenmp fib_openmp_tasks.c -o fib_tasks
./fib_tasks
```

output

```
Enter number of Fibonacci numbers to compute: 10

Fibonacci numbers:
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
```

4.Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.

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

// Function to check if a number is prime
int is_prime(int num) {
    if (num <= 1) return 0;
    if (num == 2) return 1;
    if (num % 2 == 0) return 0;

    int limit = (int)sqrt(num);
    for (int i = 3; i <= limit; i += 2) {
        if (num % i == 0)
            return 0;
    }
    return 1;
}

int main() {
    int n;

    printf("Enter the value of n: ");
    scanf("%d", &n);

    int *prime_serial = malloc((n + 1) * sizeof(int));
    int *prime_parallel = malloc((n + 1) * sizeof(int));

    // Initialize arrays
    for (int i = 0; i <= n; i++) {
        prime_serial[i] = 0;
        prime_parallel[i] = 0;
    }

    // Serial execution
    double start_serial = omp_get_wtime();
    for (int i = 1; i <= n; i++) {
        prime_serial[i] = is_prime(i);
    }
    double end_serial = omp_get_wtime();

    // Parallel execution
    double start_parallel = omp_get_wtime();
    #pragma omp parallel for schedule(static)
    for (int i = 1; i <= n; i++) {
        prime_parallel[i] = is_prime(i);
    }
    double end_parallel = omp_get_wtime();
```

```c
    // Verify correctness
    int mismatch = 0;
    for (int i = 1; i <= n; i++) {
        if (prime_serial[i] != prime_parallel[i]) {
            mismatch = 1;
            printf("Mismatch found at %d\n", i);
            break;
        }
    }

    if (!mismatch) {
        printf("\nPrime numbers from 1 to %d are:\n", n);
        for (int i = 1; i <= n; i++) {
            if (prime_parallel[i]) {
                printf("%d ", i);
            }
        }
        printf("\n");
    } else {
        printf("Error: Serial and parallel results differ.\n");
    }

    printf("\nExecution Time:\n");
    printf("Serial   : %.6f seconds\n", end_serial - start_serial);
    printf("Parallel : %.6f seconds\n", end_parallel - start_parallel);

    free(prime_serial);
    free(prime_parallel);

    return 0;
}
```

steps to run

```
gcc -fopenmp prime_omp.c -o prime_omp -lm
./prime_omp
```

output:

```
Enter the value of n: 30

Prime numbers from 1 to 30 are:
2 3 5 7 11 13 17 19 23 29

Execution Time:
Serial   : 0.000120 seconds
Parallel : 0.000065 seconds
```

5.Write a MPI Program to demonstration of MPI_Send and MPI_Recv.

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

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

    MPI_Init(&argc, &argv);              // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get process rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get number of processes

    if (size < 2) {
        if (rank == 0)
            printf("This program requires at least 2 MPI processes.\n");
        MPI_Finalize();
        return 0;
    }

    if (rank == 0) {
        number = 42; // The number to send
        printf("Process 0 sending number %d to process 1\n", number);
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
    }

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

steps to run

```
# Compile
mpicc mpi_send_recv.c -o mpi_send_recv

# Run with 2 processes
mpirun -np 2 ./mpi_send_recv
```

output:

```
Process 0 sending number 42 to process 1
Process 1 received number 42 from process 0
```

6.Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence.

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

int main(int argc, char* argv[]) {
    int rank, size, x = 100, y;

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

    if (size != 2) {
        if (rank == 0)
            printf("This demo requires exactly 2 MPI processes.\n");
        MPI_Finalize();
        return 0;
    }

    if (rank == 0) {
        printf("Process 0 sending to Process 1...\n");
        MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

        printf("Process 0 waiting to receive from Process 1...\n");
        MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else if (rank == 1) {
        printf("Process 1 sending to Process 0...\n");
        MPI_Send(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

        printf("Process 1 waiting to receive from Process 0...\n");
        MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    printf("Process %d completed communication.\n", rank);
    MPI_Finalize();
    return 0;
}
```

steps to run :

```
mpicc deadlock_mpi.c -o deadlock_mpi
mpirun -np 2 ./deadlock_mpi
```

output:

```
Process 0 sending to Process 1...
Process 1 sending to Process 0...
```

7.Write a MPI Program to demonstration of Broadcast operation.

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

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

    MPI_Init(&argc, &argv);                        // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);          // Get current process rank
    MPI_Comm_size(MPI_COMM_WORLD, &size);          // Get total number of processes

    if (rank == 0) {
        // Root process initializes the data
        data = 99;
        printf("Process %d broadcasting data = %d\n", rank, data);
    }

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

    // Each process prints the received data
    printf("Process %d received data = %d\n", rank, data);

    MPI_Finalize(); // Finalize the MPI environment
    return 0;
}
```

steps to run :

```
mpicc mpi_broadcast.c -o mpi_broadcast
mpirun -np 4 ./mpi_broadcast
```

output:

```
Process 0 broadcasting data = 99
Process 0 received data = 99
Process 1 received data = 99
Process 2 received data = 99
Process 3 received data = 99.
```

8.Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

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

int main(int argc, char* argv[]) {
    int rank, size;
    const int elements_per_proc = 2; // Number of elements per process

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

    int total_elements = elements_per_proc * size;
    int *data = NULL;

    if (rank == 0) {
        // Root initializes an array of data
        data = (int *)malloc(sizeof(int) * total_elements);
        for (int i = 0; i < total_elements; i++) {
            data[i] = i + 1;
        }
        printf("Root process has data to scatter:\n");
        for (int i = 0; i < total_elements; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

    // Each process receives elements_per_proc integers
    int *sub_data = (int *)malloc(sizeof(int) * elements_per_proc);

    // Scatter the data from root to all processes
    MPI_Scatter(data, elements_per_proc, MPI_INT,
                sub_data, elements_per_proc, MPI_INT,
                0, MPI_COMM_WORLD);

    // Each process modifies the received data (e.g., multiply by 2)
    for (int i = 0; i < elements_per_proc; i++) {
        sub_data[i] *= 2;
    }

    // Gather the modified data back to root
    MPI_Gather(sub_data, elements_per_proc, MPI_INT,
               data, elements_per_proc, MPI_INT,
               0, MPI_COMM_WORLD);

    // Root prints the gathered result
    if (rank == 0) {
        printf("\nRoot process received modified data from all processes:\n");
        for (int i = 0; i < total_elements; i++) {
```

```c
            printf("%d ", data[i]);
        }
        printf("\n");

        free(data);
    }

    free(sub_data);
    MPI_Finalize();
    return 0;
}
```

steps to run :

```
mpicc mpi_scatter_gather.c -o scatter_gather
mpirun -np 4 ./scatter_gather
```

output:

```
Root process has data to scatter:
1 2 3 4 5 6 7 8

Root process received modified data from all processes:
2 4 6 8 10 12 14 16
```

9.Write a MPI Program to 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;
    int value;

    MPI_Init(&argc, &argv);                 // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);   // Get rank
    MPI_Comm_size(MPI_COMM_WORLD, &size);   // Get number of processes

    // Each process sets its value to (rank + 1)
    value = rank + 1;

    int sum_result, prod_result, max_result, min_result;
    int all_sum, all_prod, all_max, all_min;

    // ----------- MPI_Reduce (result only in root) ------------------
    MPI_Reduce(&value, &sum_result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &prod_result, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &max_result, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &min_result, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("=== MPI_Reduce results at root process ===\n");
        printf("Sum  : %d\n", sum_result);
        printf("Prod : %d\n", prod_result);
        printf("Max  : %d\n", max_result);
        printf("Min  : %d\n", min_result);
    }

    // ----------- MPI_Allreduce (result in all processes) ------------
    MPI_Allreduce(&value, &all_sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&value, &all_prod, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
    MPI_Allreduce(&value, &all_max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&value, &all_min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    printf("Process %d:\n", rank);
    printf("  Allreduce Sum  = %d\n", all_sum);
    printf("  Allreduce Prod = %d\n", all_prod);
    printf("  Allreduce Max  = %d\n", all_max);
    printf("  Allreduce Min  = %d\n", all_min);

    MPI_Finalize();
    return 0;
}
```

steps to run :
mpicc mpi_reduce_allreduce.c -o reduce_allreduce
mpirun -np 4 ./reduce_allreduce

output:

```
=== MPI_Reduce results at root process ===
Sum  : 10
Prod : 24
Max  : 4
Min  : 1

Process 0:
  Allreduce Sum  = 10
  Allreduce Prod = 24
  Allreduce Max  = 4
  Allreduce Min  = 1
Process 1:
  Allreduce Sum  = 10
  Allreduce Prod = 24
  Allreduce Max  = 4
  Allreduce Min  = 1
Process 2:
  Allreduce Sum  = 10
  Allreduce Prod = 24
  Allreduce Max  = 4
  Allreduce Min  = 1
Process 3:
  Allreduce Sum  = 10
  Allreduce Prod = 24
  Allreduce Max  = 4
  Allreduce Min  = 1
```