PROGRAM 1

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.

AIM: 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.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include <math.h>
void merge(int arr[], int l, int m, int r) {
   int n1 = m - l + 1;
   int n2 = r - m;
   int *L = (int*)malloc(n1 * sizeof(int));
   int *R = (int*)malloc(n2 * sizeof(int));
   for (int i = 0; i < n1; i++) L[i] = arr[l + i];
   for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
   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++];
   free(L);
   free(R);
}
void mergeSort_seq(int arr[], int l, int r) {
   if (l < r) {
      int m = l + (r - l) / 2;
      mergeSort_seq(arr, l, m);
      mergeSort_seq(arr, m + 1, r);
```

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

                #pragma omp section
                mergeSort_par(arr, m + 1, r, depth + 1, max_depth);
            }
        } else {
            mergeSort_seq(arr, l, m);
            mergeSort_seq(arr, m + 1, r);
        }
        merge(arr, l, m, r);
    }
}
int isSorted(int arr[], int n) {
    for (int i = 1; i < n; i++)
        if (arr[i] < arr[i-1]) return 0;
    return 1;
}
int main() {
    int n = 1000000; // change size here
    int *arr1 = (int*)malloc(n * sizeof(int));
    int *arr2 = (int*)malloc(n * sizeof(int));
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr1[i] = rand() % 1000000;
```

```
    arr2[i] = arr1[i];
  }
  int max_threads = omp_get_max_threads();
  int max_depth = (int)(2 * log2(max_threads));
  double start, end, seq_time, par_time;
  start = omp_get_wtime();
  mergeSort_seq(arr1, 0, n - 1);
  end = omp_get_wtime();
  seq_time = end - start;
  printf("Sequential Time: %f seconds\n", seq_time);
  start = omp_get_wtime();
  mergeSort_par(arr2, 0, n - 1, 1, max_depth);
  end = omp_get_wtime();
  par_time = end - start;
  printf("Parallel Time:   %f seconds\n", par_time);
  printf("Sequential sorted: %s\n", isSorted(arr1, n) ? "Yes" : "No");
  printf("Parallel sorted:   %s\n", isSorted(arr2, n) ? "Yes" : "No");
  printf("Speedup (Seq/Par): %.2fx\n", seq_time / par_time);
  free(arr1);
  free(arr2);
  return 0;
}
```

Output:

```
Sequential Time: 0.375000 seconds
Parallel Time:   0.227000 seconds
Sequential sorted: Yes
Parallel sorted:   Yes
Speedup (Seq/Par): 1.65x
```

Result: the OpenMP program to sort an array on n elements using both sequential and parallel mergesort (using Section). Record the difference in execution time has been successfully executed

PROGRAM 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

AIM: 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

Program

```
#include <stdio.h>
#include <omp.h>
int main() {
   int n, i;
   printf("Enter number of iterations: ");
   scanf("%d", &n);
   for (i = 0; i < n; i += 2) {
      int tid = omp_get_thread_num();
      int end = (i + 1 < n) ? i + 1 : i;
      printf("Thread %d : Iterations %d -- %d\n", tid, i, end);
   }
   return 0;
}
```

Output:

```
Enter number of iterations: 4
Thread 0 : Iterations 0 -- 1
Thread 0 : Iterations 2 -- 3
```

Result: 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 has been successfully executed

PROGRAM 3

Write a OpenMP program to calculate n Fibonacci numbers using tasks

AIM: Write a OpenMP program to calculate n Fibonacci numbers using tasks

Program

```
#include <stdio.h>
#include <omp.h>
int fib(int n) {
   int x, y;
   if (n < 2) {
      return n;
   } else {
      x = fib(n - 1);
      y = fib(n - 2);
      return x + y;
   }
}
int main() {
   int n;
   printf("Enter number of Fibonacci terms: ");
   scanf("%d", &n);
   printf("Fibonacci series up to %d terms:\n", n);
   {
      {
         for (int i = 0; i < n; i++) {
            int result = fib(i);
            printf("%d ", result);
         }
      }
   }
   printf("\n");
   return 0;
}
```

Output:

```
Enter number of Fibonacci terms: 20
Fibonacci series up to 20 terms:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Result: a OpenMP program to calculate n Fibonacci numbers using tasks has been successfully executed

PROGRAM 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.

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

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
int is_prime(int num) {
   if (num < 2) 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 n: ");
   scanf("%d", &n);
   double start, end;
   start = omp_get_wtime();
   printf("\n[Serial] Prime numbers up to %d:\n", n);
   for (int i = 2; i <= n; i++) {
      if (is_prime(i)) {
         printf("%d ", i);
      }
   }
```

```
    end = omp_get_wtime();
    double serial_time = end - start;
    printf("\nSerial execution time: %f seconds\n", serial_time);
    start = omp_get_wtime();
    printf("\n[Parallel] Prime numbers up to %d:\n", n);
    for (int i = 2; i <= n; i++) {
        if (is_prime(i)) {
            {
                printf("%d ", i);
            }
        }
    }
    end = omp_get_wtime();
    double parallel_time = end - start;
    printf("\nParallel execution time: %f seconds\n", parallel_time);
    return 0;
}
```

Output:

```
Enter n: 150

[Serial] Prime numbers up to 150:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149
Serial execution time: 0.003000 seconds

[Parallel] Prime numbers up to 150:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149
Parallel execution time: 0.001000 seconds
```

Result: an OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times has been successfully executed

PROGRAM 5

Write a MPI Program to demonstration of MPI_Send and MPI_Recv

AIM: Write a MPI Program to demonstration of MPI_Send and MPI_Recv

Program

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
   int rank, size;
   char message[100];
   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 program needs at least 2 processes.\n");
      }
      MPI_Finalize();
      return 0;
   }
   if (rank == 0) {
      strcpy(message, "Hello from Process 0!");
      MPI_Send(message, strlen(message) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
      printf("Process 0 sent message: %s\n", message);
   }
   else if (rank == 1) {
      MPI_Recv(message, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process 1 received message: %s\n", message);
   }
   MPI_Finalize();
   return 0;
}
```

Output:

```
C:\Users\Abhishek N\Documents\Abhi\bin\Debug>mpiexec Abhi.exe
Process 1 received message: Hello from Process 0!
Process 0 sent message: Hello from Process 0!
```

Result: Write a MPI Program to demonstration of MPI_Send and MPI_Recv has been successfully executed

PROGRAM 6

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

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

Program: deadlock using point to point communication

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
   int rank, size;
   int data = 0;
   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("Run with at least 2 processes.\n");
      }
      MPI_Finalize();
      return 0;
   }
   if (rank == 0) {
      data = 100;
      printf("Process 0 trying to send...\n");
      MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      printf("Process 0 waiting to receive...\n");
      MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process 0 received: %d\n", data);
   }
   else if (rank == 1) {
      data = 200;
      printf("Process 1 trying to send...\n");
      MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

```
    printf("Process 1 waiting to receive...\n");
    MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received: %d\n", data);
  }
  MPI_Finalize();
  return 0;
}
```

Output:

```
C:\Users\Abhishek N>cd C:\Users\Abhishek N\Documents\Abhi\

C:\Users\Abhishek N\Documents\Abhi>cd bin

C:\Users\Abhishek N\Documents\Abhi\bin>cd debug

C:\Users\Abhishek N\Documents\Abhi\bin\Debug>mpiexec Abhi.exe
Process 0 trying to send...
Process 0 waiting to receive...
Process 0 received: 200
Process 1 trying to send...
Process 1 waiting to receive...
Process 1 received: 100
```

Program: avoidance of deadlock by altering the call sequence

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
  int rank, size;
  int data = 0;
  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("Run with at least 2 processes.\n");
    }
    MPI_Finalize();
    return 0;
```

```
    }
    if (rank == 0) {
        data = 100;
        printf("Process 0 sending data...\n");
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

        MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received: %d\n", data);
    }
    else if (rank == 1) {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received: %d\n", data);
        data = 200;
        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Process 1 sent data: %d\n", data);
    }
    MPI_Finalize();
    return 0;
}
```

Output:

```
C:\Users\Abhishek N\Documents\Abhi\bin\Debug>mpiexec Abhi.exe
Process 0 sending data...
Process 0 received: 200
Process 1 received: 100
Process 1 sent data: 200
```

Result: Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence has been successfully executed

PROGRAM 7

Write a MPI Program to demonstration of Broadcast operation

AIM: Write a MPI Program to demonstration of Broadcast operation

Program:

```
#include <mpi.h>
#include <stdio.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 process rank
   MPI_Comm_size(MPI_COMM_WORLD, &size);   // Get total processes
   if (rank == 0) {
      data = 42;  // Root initializes data
      printf("Process %d (root) broadcasting data = %d\n", rank, data);
   }
   MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
   printf("Process %d received data = %d\n", rank, data);
   MPI_Finalize();   // Finalize MPI
   return 0;
}
```

Output:

```
\Users\Abhishek N\Documents\Abhi\bin\Debug>mpiexec Abhi.exe
rocess 0 (root) broadcasting data = 42
rocess 0 received data = 42
rocess 1 received data = 42
rocess 2 received data = 42
rocess 3 received data = 42
```

Result: Write a MPI Program to demonstration of Broadcast operation has been successfully executed

PROGRAM 8

Write a MPI Program to demonstration of Broadcast operation

AIM: Write a MPI Program to demonstration of Broadcast operation

Program:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
   int rank, size;
   int send_data[100];   // array only used at root
   int recv_data;       // each process gets one element
   int gathered_data[100]; // array only used at root
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0) {
      printf("Root process initializing data...\n");
      for (int i = 0; i < size; i++) {
         send_data[i] = i * 10;   // Example: 0, 10, 20, ...
      }
   }
   MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
   printf("Process %d received %d\n", rank, recv_data);
   recv_data += 5;
   MPI_Gather(&recv_data, 1, MPI_INT, gathered_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (rank == 0) {
      printf("\nRoot process gathered results:\n");
      for (int i = 0; i < size; i++) {
         printf("gathered_data[%d] = %d\n", i, gathered_data[i]);
      }
   }
   MPI_Finalize();
   return 0;
}
```

Output:

```
C:\Users\Abhishek N\Documents\Abhi\bin\Debug>mpiexec Abhi.exe
Process 2 received 20
Process 1 received 10
Process 3 received 30
Root process initializing data...
Process 0 received 0

Root process gathered results:
gathered_data[0] = 5
gathered_data[1] = 15
gathered_data[2] = 25
gathered_data[3] = 35
```

Result: Write a MPI Program to demonstration of Broadcast operation has been successfully executed

PROGRAM 9

Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

AIM: Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

Program:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
   int rank, size;
   int value;
   int result;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   value = rank + 1;
   printf("Process %d has value %d\n", rank, value);
   if (rank == 0) printf("\n---- MPI_Reduce results at Root ----\n");
   MPI_Reduce(&value, &result, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
   if (rank == 0) printf("MPI_MAX = %d\n", result);
   MPI_Reduce(&value, &result, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
   if (rank == 0) printf("MPI_MIN = %d\n", result);
   MPI_Reduce(&value, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if (rank == 0) printf("MPI_SUM = %d\n", result);
   MPI_Reduce(&value, &result, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
   if (rank == 0) printf("MPI_PROD = %d\n", result);
   if (rank == 0) printf("\n---- MPI_Allreduce results at ALL processes ----\n");
   MPI_Allreduce(&value, &result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
   printf("Process %d: MPI_MAX = %d\n", rank, result);
   MPI_Allreduce(&value, &result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
   printf("Process %d: MPI_MIN = %d\n", rank, result);
   MPI_Allreduce(&value, &result, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
   printf("Process %d: MPI_SUM = %d\n", rank, result);
```

```
    MPI_Allreduce(&value, &result, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
    printf("Process %d: MPI_PROD = %d\n", rank, result);
    MPI_Finalize();
    return 0;
}
```

Output:

```
C:\Users\Abhishek N\Documents\Abhi\bin\Debug>mpiexec Abhi.exe
Process 2 has value 3
Process 2: MPI_MAX = 4
Process 2: MPI_MIN = 1
Process 2: MPI_SUM = 10
Process 2: MPI_PROD = 24
Process 0 has value 1

---- MPI_Reduce results at Root ----
MPI_MAX = 4
MPI_MIN = 1
MPI_SUM = 10
MPI_PROD = 24

---- MPI_Allreduce results at ALL processes ----
Process 0: MPI_MAX = 4
Process 0: MPI_MIN = 1
Process 0: MPI_SUM = 10
Process 0: MPI_PROD = 24
Process 3 has value 4
Process 3: MPI_MAX = 4
Process 3: MPI_MIN = 1
Process 3: MPI_SUM = 10
Process 3: MPI_PROD = 24
Process 1 has value 2
Process 1: MPI_MAX = 4
Process 1: MPI_MIN = 1
Process 1: MPI_SUM = 10
Process 1: MPI_PROD = 24
```

Result: Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD) has been successfully executed