1. Program

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

#define SIZE 100000

// ---------------------- MERGE FUNCTION
void merge(int arr[], int left, int mid, int right)
{
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;

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

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

   i = 0;
   j = 0;
   k = left;

   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++];

   free(L);
   free(R);
}

// ---------------------- SERIAL MERGE SORT
void serialMergeSort(int arr[], int left, int right)
{
   if (left < right)
   {
      int mid = left + (right - left) / 2;
      serialMergeSort(arr, left, mid);
      serialMergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
   }
}

// ---------------------- PARALLEL MERGE SORT
void parallelMergeSort(int arr[], int left, int right, int depth)
{
   if (left < right)
```

```
    {
        int mid = left + (right - left) / 2;

        if (depth <= 4)
        {
#pragma omp parallel sections
            {
#pragma omp section
                parallelMergeSort(arr, left, mid, depth + 1);
#pragma omp section
                parallelMergeSort(arr, mid + 1, right, depth + 1);
            }
        }
        else
        {
            serialMergeSort(arr, left, mid);
            serialMergeSort(arr, mid + 1, right);
        }

        merge(arr, left, mid, right);
    }
}

// ---------------------- MAIN FUNCTION
int main()
{
    int *arr_serial = (int *)malloc(SIZE * sizeof(int));
    int *arr_parallel = (int *)malloc(SIZE * sizeof(int));

    // Initialize both arrays with the same random values
    for (int i = 0; i < SIZE; i++)
    {
        int val = rand() % 100000;
        arr_serial[i] = val;
        arr_parallel[i] = val;
    }

    // ---------- SERIAL MERGE SORT
    clock_t start_serial = clock();
    serialMergeSort(arr_serial, 0, SIZE - 1);
    clock_t end_serial = clock();
    double time_serial = (double)(end_serial - start_serial) / CLOCKS_PER_SEC;

    // ---------- PARALLEL MERGE SORT
    clock_t start_parallel = clock();
    parallelMergeSort(arr_parallel, 0, SIZE - 1, 0);
    clock_t end_parallel = clock();
    double time_parallel = (double)(end_parallel - start_parallel) / CLOCKS_PER_SEC;

    // ---------- OUTPUT ----------
    printf("Serial Merge Sort Time   : %.3f seconds\n", time_serial);
    printf("Parallel Merge Sort Time : %.6f seconds\n", time_parallel);

    // Optional: Verify correctness
    for (int i = 0; i < SIZE; i++)
    {
        if (arr_serial[i] != arr_parallel[i])
```

```
    {
      printf("Mismatch at index %d\n", i);
      break;
    }
  }

  free(arr_serial);
  free(arr_parallel);
  return 0;
}
```

This now compiles and runs fine with:

bash
CopyEdit
```
gcc -fopenmp filename.c -o mergesort
./mergesort
```

2. Program

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

int main()
{
  int n = 16, thread;

  printf("\nEnter the number of tasks: ");
  scanf("%d", &n);

  printf("\nEnter the number of threads: ");
  scanf("%d", &thread);

  omp_set_num_threads(thread);

  printf("\n---------------------------------------\n");

#pragma omp parallel for schedule(static, 2)
  for (int i = 0; i < n; i++)
  {
    printf("Thread %d executes iteration %d\n", omp_get_thread_num(), i);
  }

  printf("---------------------------------------\n");

  return 0;
}
```

3. Program

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

// Serial Fibonacci calculation function
```

```
int ser_fib(long int n)
{
   if (n < 2) return n;
   long int x, y;
   x = ser_fib(n - 1);
   y = ser_fib(n - 2);
   return x + y;
}

// Parallel Fibonacci calculation function
int fib(long int n)
{
   if (n < 2) return n;
   long int x, y;
#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()
{
   long int n = 10, result;
   clock_t start, end;
   double cpu_time;

   printf("\nEnter the value of n: ");
   scanf("%ld", &n);

   // Parallel Fibonacci
   start = clock();
#pragma omp parallel
   {
#pragma omp single
      result = fib(n);
   }
   end = clock();
   cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("Fibonacci(%ld) = %ld\n", n, result);
   printf("Time used in parallel mode = %f seconds\n", cpu_time);

   // Serial Fibonacci
   start = clock();
   result = ser_fib(n);
   end = clock();
   cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("Fibonacci(%ld) = %ld\n", n, result);
   printf("Time used in sequential mode = %f seconds\n", cpu_time);

   return 0;
}
```

Compile & Run

```
gcc -fopenmp fib.c -o fib.exe
fib.exe
```

4. Program

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

// Function to check if a number is prime
int is_prime(int n)
{
   if (n < 2) return 0;
   for (int i = 2; i * i <= n; i++)
   {
      if (n % i == 0) return 0;
   }
   return 1;
}

int main()
{
   long n = 10000000;
   clock_t start, end;
   double cpu_time;

   printf("\nThe range of numbers is 1 to %ld\n", n);
   printf("----------------------------------------------------------------\n");

   // Serial Execution
   start = clock();
   for (long i = 1; i <= n; i++)
   {
      is_prime(i);
   }
   end = clock();
   cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("Time to compute prime numbers serially: %f seconds\n", cpu_time);

   // Parallel Execution
   start = clock();
#pragma omp parallel for
   for (long i = 1; i <= n; i++)
   {
      is_prime(i);
   }
   end = clock();
   cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("Time to compute prime numbers in parallel: %f seconds\n", cpu_time);

   return 0;
}
```

Compile&Run

```
gcc -fopenmp prime.c -o prime.exe
prime.exe
```

5. Program

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

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

   // Initialize the MPI environment
   MPI_Init(&argc, &argv);

   // Get the number of processes
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   // Get the rank of the process
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

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

   if (rank == 0)
   {
      // Process 0 sends a number to Process 1
      number = 42;
      printf("Process 0 is sending number %d to Process 1\n", number);
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   }
   else if (rank == 1)
   {
      // Process 1 receives a number from Process 0
      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);
   }

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

Compile&Run

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

6. Program

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

```
int main(int argc, char *argv[])
{
   int rank, data_send, data_recv;

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

   data_send = rank;

   if (rank == 0)
   {
      MPI_Send(&data_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&data_recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }
   else if (rank == 1)
   {
      MPI_Send(&data_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&data_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   }

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

   MPI_Finalize();
   return 0;
}
```

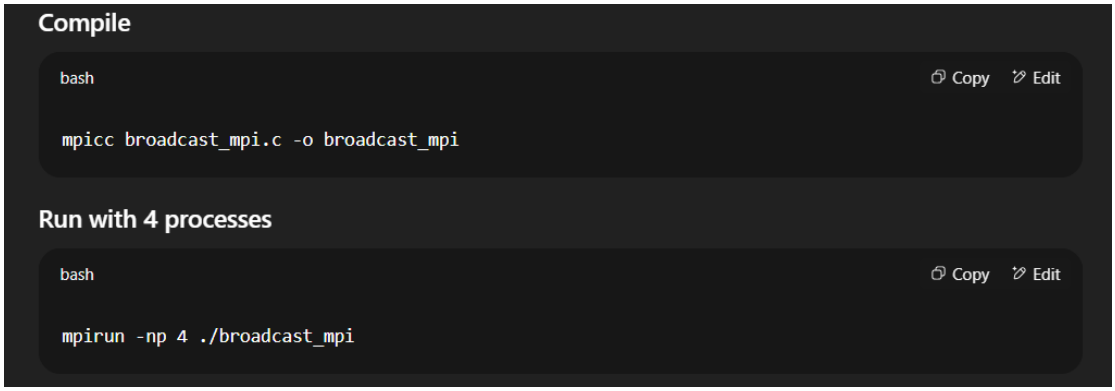
Compile&Run

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

7. Program

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

int main(int argc, char** argv)
{
   int rank, data = 0;

   // Initialize the MPI environment
   MPI_Init(&argc, &argv);

   // Get the rank of the process
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   // Root process sets the data
   if (rank == 0)
      data = 100;

   // Broadcast data from root to all processes
   MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

   // All processes print the data
   printf("Process %d received data: %d\n", rank, data);

   // Finalize the MPI environment
```

```
    MPI_Finalize();

    return 0;
}
```



8. Program

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

int main(int argc, char** argv)
{
    int rank, size;
    int send_data[4] = {10, 20, 30, 40};
    int recv_data;

    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get rank and size
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Scatter: send one element to each process
    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);

    // Each process increments the received value
    recv_data += 1;

    // Gather: collect updated values back to root process
    MPI_Gather(&recv_data, 1, MPI_INT, send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Root prints gathered data
    if (rank == 0)
    {
        printf("Gathered data: ");
        for (int i = 0; i < size; i++)
            printf("%d ", send_data[i]);
        printf("\n");
    }

    MPI_Finalize();
```

```
    return 0;
}
```

How to Compile

```
bash                                                    Copy  Edit

mpicc scatter_gather_mpi.c -o scatter_gather_mpi
```

How to Run with 4 Processes

```
bash                                                    Copy  Edit

mpirun -np 4 ./scatter_gather_mpi
```

9. Program

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

int main(int argc, char** argv)
{
    int rank, value, sum, max;

    MPI_Init(&argc, &argv); // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    value = rank + 1; // Each process gets a value

    // Reduce to find the sum at rank 0
    MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        printf("Sum using Reduce: %d\n", sum);

    // Allreduce to find the max across all processes
    MPI_Allreduce(&value, &max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    printf("Max using Allreduce (rank %d): %d\n", rank, max);

    MPI_Finalize();
    return 0;
}
```

How to Compile

```
bash                                                    Copy  Edit

mpicc reduce_allreduce_mpi.c -o reduce_allreduce_mpi
```

How to Run (Example with 4 processes)

```
bash                                                    Copy  Edit

mpirun -np 4 ./reduce_allreduce_mpi
```