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

void merge(int arr[], int l, int m, int r) {
    int i = l, j = m + 1, k = 0;
    int temp[r - l + 1];

    while (i <= m && j <= r) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }
    while (i <= m) temp[k++] = arr[i++];
    while (j <= r) temp[k++] = arr[j++];

    for (i = l, k = 0; i <= r; i++, k++)
        arr[i] = temp[k];
}

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

void parallelMergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;

        #pragma omp parallel sections
        {
            #pragma omp section
            parallelMergeSort(arr, l, m);

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

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

```
int main() {
    int n = 100000;
    int arr1[n], arr2[n];

    for (int i = 0; i < n; i++) {
        arr1[i] = rand() % 1000;
        arr2[i] = arr1[i];
    }

    double start, end;

    start = omp_get_wtime();
    sequentialMergeSort(arr1, 0, n - 1);
    end = omp_get_wtime();
    printf("Sequential Merge Sort Time: %f seconds\n", end - start);

    start = omp_get_wtime();
    parallelMergeSort(arr2, 0, n - 1);
    end = omp_get_wtime();
    printf("Parallel Merge Sort Time: %f seconds\n", end - start);

    return 0;
}
```

- **Create**: gedit prg1.c
- **Compile**: gcc -fopenmp prg1.c -o prg1
- **Run**: export OMP_NUM_THREADS=4 && ./prg

PROGRAM 2

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

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

    int thread_start[100], thread_end[100];
    int i;

    for (i = 0; i < 100; i++) {
        thread_start[i] = -1;
        thread_end[i] = -1;
    }

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

        if (thread_start[tid] == -1)
            thread_start[tid] = i;
        thread_end[tid] = i;
    }

    for (i = 0; i < 100; i++) {
        if (thread_start[i] != -1) {
            printf("Thread %d : Iterations %d -- %d\n", i, thread_start[i], thread_end[i]);

        }
    }

    return 0;
}
```

- **Create**: gedit prg2.c
- **Compile**: gcc -fopenmp prg2.c -o prg2
- **Run**: export OMP_NUM_THREADS=4 && ./prg2

PROGRAM 3

```c
#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;

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

    printf("Fibonacci Series:\n");

    for (int i = 0; i < n; i++) {
        int result;

        #pragma omp parallel
        {
            #pragma omp single
            {
                result = fib(i);
            }
        }

        printf("%d ", result);
    }

    printf("\n");
    return 0;
}
```

- **Create**: gedit prg3.c

- **Compile**: gcc -fopenmp prg3.c -o prg3
- **Run**: export OMP_NUM_THREADS=4 && ./prg3

PROGRAM 4

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

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

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

    printf("\nPrime numbers from 1 to %d:\n", n);
    for (int i = 1; i <= n; i++) {
        if (is_prime(i))
            printf("%d ", i);
    }
    printf("\n");

    double start_time, end_time;

    start_time = omp_get_wtime();
    for (int i = 1; i <= n; i++) {
        is_prime(i);
    }
    end_time = omp_get_wtime();
    printf("Serial Time: %f seconds\n", end_time - start_time);

    start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 1; i <= n; i++) {
        is_prime(i);
    }
    end_time = omp_get_wtime();
    printf("Parallel Time: %f seconds\n", end_time - start_time);

    return 0;
}
```

- **Create**: gedit prg4.c
- **Compile**: gcc -fopenmp prg4.c -o prg4 -lm
- **Run**: export OMP_NUM_THREADS=4 && ./prg4

PROGRAM 5

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

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

    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("Please run with at least 2 processes.\n");
        MPI_Finalize();
        return 0;
    }

    if (rank == 0) {
        number = 100;
        printf("Process %d sending number %d to process 1\n", rank, 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 %d received number %d from process 0\n", rank, number);
    }

    MPI_Finalize();
    return 0;
}
```

- **Create**: gedit prg5.c
- **Compile**: mpicc prg5.c -o prg5
- **Run**: mpirun -np 2 ./prg5

PROGRAM 6

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

// Change this to 1 for deadlock, 2 for deadlock avoidance
#define DEADLOCK_PART 2

int main(int argc, char *argv[]) {
    int rank, size, num = 123;

    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 DEADLOCK_PART == 1
    // ----------- Part A: Deadlock -----------
    if (rank == 0) {
        printf("Process 0 waiting to receive from Process 1...\n");
        MPI_Recv(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        printf("Process 1 waiting to receive from Process 0...\n");
        MPI_Recv(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

#elif DEADLOCK_PART == 2
    // ----------- Part B: Deadlock Avoidance -----------
    if (rank == 0) {
        MPI_Send(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received back number: %d\n", num);
    } else if (rank == 1) {
        MPI_Recv(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Process 1 received and sent back number: %d\n", num);
    }
#endif

    MPI_Finalize();
    return 0;
}
```

- **Create**: gedit prg6.c
- **Compile**: mpicc prg6.c -o prg6
- **Run**: mpirun -np 2 ./prg6

PROGRAM 7

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

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

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

    if (rank == 0) {
        number = 50;
        printf("Process %d broadcasting number %d to all other processes.\n", rank, number);
    }

    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received number %d\n", rank, number);

    MPI_Finalize();
    return 0;
}
```

- **Create**: gedit prg7.c
- **Compile**: mpicc prg7.c -o prg7
- **Run**: mpirun -np 4 ./prg7

PROGRAM 8

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

int main(int argc, char *argv[]) {
    int rank, size;
    int send_data[100], recv_data, gathered_data[100];

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

    if (rank == 0) {
        for (int i = 0; i < size; i++) {
            send_data[i] = i * 10;
        }
    }

    MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    recv_data = recv_data + rank;
    MPI_Gather(&recv_data, 1, MPI_INT, gathered_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Gathered data in root process:\n");
        for (int i = 0; i < size; i++) {
            printf("gathered_data[%d] = %d\n", i, gathered_data[i]);
        }
    }

    MPI_Finalize();
    return 0;
}
```

- **Create**: gedit prg8.c
- **Compile**: mpicc prg8.c -o prg8
- **Run**: mpirun -np 4 ./prg8

PROGRAM 9

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

int main(int argc, char *argv[]) {
    int rank, size;
    int value, sum, prod, max, min;
    int all_sum, all_prod, all_max, all_min;

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

    value = rank + 1;

    MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &prod, 1, MPI_INT, MPI_PROD, 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_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);

    if (rank == 0) {
        printf("--- Results using MPI_Reduce (only root prints) ---\n");
        printf("Sum = %d\n", sum);
        printf("Product = %d\n", prod);
        printf("Max = %d\n", max);
        printf("Min = %d\n", min);
    }

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

    MPI_Finalize();
    return 0;
}
```

- **Create**: gedit prg9.c
- **Compile**: mpicc prg9.c -o prg9
- **Run**: mpirun -np 4 ./prg9