1.c

```c
#include <stdio.h>
#include <stdlib.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 = (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)
        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);
}

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

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

        if (depth <= 0) {
            mergeSortSequential(arr, l, m);
            mergeSortSequential(arr, m + 1, r);
        } else {
            #pragma omp parallel sections
            {
                #pragma omp section
                mergeSortParallel(arr, l, m, depth - 1);

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

int main() {
    int n = 100000;
    int *arrSeq = (int *)malloc(n * sizeof(int));
    int *arrPar = (int *)malloc(n * sizeof(int));

    srand(0);
    for (int i = 0; i < n; i++) {
        arrSeq[i] = rand() % 100000;
        arrPar[i] = arrSeq[i];
    }

    double start = omp_get_wtime();
    mergeSortSequential(arrSeq, 0, n - 1);
    double end = omp_get_wtime();
    double seqTime = end - start;

    start = omp_get_wtime();
    mergeSortParallel(arrPar, 0, n - 1, 4);
    end = omp_get_wtime();
    double parTime = end - start;

    printf("Sequential Sort Time: %f seconds\n", seqTime);
    printf("Parallel Sort Time  : %f seconds\n", parTime);
    printf("Speedup = %.2fx\n", seqTime / parTime);

    free(arrSeq);
    free(arrPar);
    return 0;
}
//gcc -fopenmp 1.c -o p1
//./p1


```

2.c

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

int main() {
    int num_iterations;

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

    printf("\nUsing schedule(static,2):\n\n");

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();

        #pragma omp for schedule(static, 2)
        for (int i = 0; i < num_iterations; i++) {
            printf("Thread %d : Iteration %d\n", tid, i);
        }
    }

    return 0;
}
//gcc 2.c -o p2 -fopenmp
//./p2
```

3.c

```c
#include <stdio.h>
#include <stdlib.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
    return x + y;
}

int main() {
    int n;
    printf("Enter the number of Fibonacci numbers to calculate: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 0;
    }

    printf("First %d Fibonacci numbers using OpenMP tasks:\n", n);

    double start = omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++) {
                int result;

                #pragma omp task shared(result)
                {
                    result = fib(i);

                    #pragma omp critical
                    printf("Fib(%d) = %d\n", i, result);
                }
            }
        }
    }

    double end = omp_get_wtime();
    printf("Execution time: %.6f seconds\n", end - start);

    return 0;
}
//gcc 3.c -o p3 -fopenmp
//./p3

```

4.c

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

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 upper limit (n): ");
    scanf("%d", &n);

    if (n <= 2)
    {
        printf("There are no prime numbers < %d\n", n);
        return 0;
    }
    double start_serial = omp_get_wtime();
    int *primes_serial = (int *)malloc(n * sizeof(int));
    int count_serial = 0;

    for (int i = 2; i < n; i++)
    {
        if (is_prime(i))
        {
            primes_serial[count_serial++] = i;
        }
    }

    double end_serial = omp_get_wtime();
    double time_serial = end_serial - start_serial;

    double start_parallel = omp_get_wtime();
    int *primes_parallel = (int *)malloc(n * sizeof(int));
    int count_parallel = 0;

    #pragma omp parallel
    {
        int *local_primes = (int *)malloc(n * sizeof(int));
        int local_count = 0;

        #pragma omp for
        for (int i = 2; i < n; i++)
        {
            if (is_prime(i))
            {
                local_primes[local_count++] = i;
            }
        }

        #pragma omp critical
        {
            for (int i = 0; i < local_count; i++)
                primes_parallel[count_parallel++] = local_primes[i];
        }

        free(local_primes);
    }

    double end_parallel = omp_get_wtime();
    double time_parallel = end_parallel - start_parallel;

    printf("\nNo. of primes found: %d\n", count_serial);
    printf("Serial execution time: %f seconds\n", time_serial);
    printf("Parallel execution time: %f seconds\n", time_parallel);
    printf("Speedup: %.2fx\n", time_serial / time_parallel);

    free(primes_serial);
    free(primes_parallel);

    return 0;
}
//gcc 4.c -o p4 -fopenmp -lm
//./p4

```

5.c

```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("This program requires atleast 2 processes\n");

        MPI_Finalize();
        return 0;
    }

    if (rank == 0)
    {
        number = 100;
        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();
    return 0;
}
//mpicc 5.c -o p5
//mpirun -np 2 ./p5

```

6.c

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

int main(int argc, char** argv) {
    int rank, size;
    int msg_send = 100, msg_recv;
    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) {
        printf("Process 0 sending to Process 1...\n");
        MPI_Send(&msg_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);   // blocking send
        MPI_Recv(&msg_recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received from Process 1: %d\n", msg_recv);

    } else if (rank == 1) {
        printf("Process 1 sending to Process 0...\n");
        MPI_Send(&msg_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);   // blocking send
        MPI_Recv(&msg_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received from Process 0: %d\n", msg_recv);
    }

    MPI_Finalize();
    return 0;
}
//mpicc 6.c -o p6
//mpirun -np 2 ./p6

```

7.c

```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;
}
//mpicc 7.c -o p7
//mpirun -np 4 ./p7

```

8.c

```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;
}
//mpicc 8.c -o p8
//mpirun -np 4 ./p8


```

9.c

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

    MPI_Finalize();
    return 0;
}
//mpicc 9.c -o p9
//mpirun -np 4 ./p9
```