

1.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 void merge(int arr[], int l, int m, int r) {
6     int n1 = m - l + 1;
7     int n2 = r - m;
8     int *L = (int *)malloc(n1 * sizeof(int));
9     int *R = (int *)malloc(n2 * sizeof(int));
10
11    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
12    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
13
14    int i = 0, j = 0, k = l;
15    while (i < n1 && j < n2)
16        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
17
18    while (i < n1) arr[k++] = L[i++];
19    while (j < n2) arr[k++] = R[j++];
20
21    free(L);
22    free(R);
23 }
24
25 void mergeSortSequential(int arr[], int l, int r) {
26     if (l < r) {
27         int m = (l + r) / 2;
28         mergeSortSequential(arr, l, m);
29         mergeSortSequential(arr, m + 1, r);
30         merge(arr, l, m, r);
31     }
32 }
33
34 void mergeSortParallel(int arr[], int l, int r, int depth) {
35     if (l < r) {
36         int m = (l + r) / 2;
37
38         if (depth <= 0) {
39             mergeSortSequential(arr, l, m);
40             mergeSortSequential(arr, m + 1, r);
41         } else {
42             #pragma omp parallel sections
43             {
44                 #pragma omp section
45                 mergeSortParallel(arr, l, m, depth - 1);
46
47                 #pragma omp section
48                 mergeSortParallel(arr, m + 1, r, depth - 1);
49             }
50         }
51     }
52     merge(arr, l, m, r);
```

```
52     }
53 }
54
55 int main() {
56     int n = 100000;
57     int *arrSeq = (int *)malloc(n * sizeof(int));
58     int *arrPar = (int *)malloc(n * sizeof(int));
59
60     srand(0);
61     for (int i = 0; i < n; i++) {
62         arrSeq[i] = rand() % 100000;
63         arrPar[i] = arrSeq[i];
64     }
65
66     double start = omp_get_wtime();
67     mergeSortSequential(arrSeq, 0, n - 1);
68     double end = omp_get_wtime();
69     double seqTime = end - start;
70
71     start = omp_get_wtime();
72     mergeSortParallel(arrPar, 0, n - 1, 4);
73     end = omp_get_wtime();
74     double parTime = end - start;
75
76     printf("Sequential Sort Time: %f seconds\n", seqTime);
77     printf("Parallel Sort Time : %f seconds\n", parTime);
78     printf("Speedup = %.2fx\n", seqTime / parTime);
79
80     free(arrSeq);
81     free(arrPar);
82     return 0;
83 }
84 //gcc -fopenmp 1.c -o p1
85 //./p1
86
87
```

```
2.c

1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     int num_iterations;
6
7     printf("Enter the number of iterations: ");
8     scanf("%d", &num_iterations);
9
10    printf("\nUsing schedule(static,2):\n\n");
11
12    #pragma omp parallel
13    {
14        int tid = omp_get_thread_num();
15
16        #pragma omp for schedule(static, 2)
17        for (int i = 0; i < num_iterations; i++) {
18            printf("Thread %d : Iteration %d\n", tid, i);
19        }
20    }
21
22    return 0;
23 }
24 //gcc 2.c -o p2 -fopenmp
25 //./p2
26
27
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int fib(int n) {
6     int x, y;
7     if (n <= 1) return n;
8
9     #pragma omp task shared(x)
10    x = fib(n - 1);
11
12    #pragma omp task shared(y)
13    y = fib(n - 2);
14
15    #pragma omp taskwait
16    return x + y;
17}
18
19 int main() {
20     int n;
21     printf("Enter the number of Fibonacci numbers to calculate: ");
22     scanf("%d", &n);
23
24     if (n <= 0) {
25         printf("Please enter a positive integer.\n");
26         return 0;
27     }
28
29     printf("First %d Fibonacci numbers using OpenMP tasks:\n", n);
30
31     double start = omp_get_wtime();
32
33     #pragma omp parallel
34     {
35         #pragma omp single
36         {
37             for (int i = 0; i < n; i++) {
38                 int result;
39
40                 #pragma omp task shared(result)
41                 {
42                     result = fib(i);
43
44                     #pragma omp critical
45                     printf("Fib(%d) = %d\n", i, result);
46                 }
47             }
48         }
49     }
50
51     double end = omp_get_wtime();
```

```
52     printf("Execution time: %.6f seconds\n", end - start);  
53  
54     return 0;  
55 }  
56 //gcc 3.c -o p3 -fopenmp  
57 //./p3  
58
```

4.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <omp.h>
5
6 int is_prime(int num)
7 {
8     if (num <= 1) return 0;
9     if (num == 2) return 1;
10    if (num % 2 == 0) return 0;
11
12    int limit = (int) sqrt(num);
13    for (int i = 3; i <= limit; i += 2)
14    {
15        if (num % i == 0)
16            return 0;
17    }
18    return 1;
19 }
20
21 int main()
22 {
23     int n;
24     printf("Enter upper limit (n): ");
25     scanf("%d", &n);
26
27     if (n <= 2)
28     {
29         printf("There are no prime numbers < %d\n", n);
30         return 0;
31     }
32     double start_serial = omp_get_wtime();
33     int *primes_serial = (int *)malloc(n * sizeof(int));
34     int count_serial = 0;
35
36     for (int i = 2; i < n; i++)
37     {
38         if (is_prime(i))
39         {
40             primes_serial[count_serial++] = i;
41         }
42     }
43
44     double end_serial = omp_get_wtime();
45     double time_serial = end_serial - start_serial;
46
47     double start_parallel = omp_get_wtime();
48     int *primes_parallel = (int *)malloc(n * sizeof(int));
49     int count_parallel = 0;
50
51 #pragma omp parallel
```

```
52     {
53         int *local_primes = (int *)malloc(n * sizeof(int));
54         int local_count = 0;
55
56         #pragma omp for
57         for (int i = 2; i < n; i++)
58         {
59             if (is_prime(i))
60             {
61                 local_primes[local_count++] = i;
62             }
63         }
64
65         #pragma omp critical
66         {
67             for (int i = 0; i < local_count; i++)
68                 primes_parallel[count_parallel++] = local_primes[i];
69         }
70
71         free(local_primes);
72     }
73
74     double end_parallel = omp_get_wtime();
75     double time_parallel = end_parallel - start_parallel;
76
77     printf("\nNo. of primes found: %d\n", count_serial);
78     printf("Serial execution time: %f seconds\n", time_serial);
79     printf("Parallel execution time: %f seconds\n", time_parallel);
80     printf("Speedup: %.2fx\n", time_serial / time_parallel);
81
82     free(primes_serial);
83     free(primes_parallel);
84
85     return 0;
86 }
87 //gcc 4.c -o p4 -fopenmp -lm
88 //./p4
89
```

5.c

```
1 #include<mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[])
5 {
6     int rank, size;
7     int number;
8
9     MPI_Init(&argc, &argv);
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13    if (size < 2)
14    {
15        if (rank == 0)
16            printf("This program requires atleast 2 processes\n");
17
18        MPI_Finalize();
19        return 0;
20    }
21
22    if (rank == 0)
23    {
24        number = 100;
25        printf("Process 0 sending number %d to Process 1\n", number);
26
27        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
28    }
29    else if (rank == 1)
30    {
31        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
32
33        printf("Process 1 received number %d from Process 0\n", number);
34    }
35
36    MPI_Finalize();
37    return 0;
38 }
39 //mpicc 5.c -o p5
40 //mpirun -np 2 ./p5
41
```

6.c

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char** argv) {
5     int rank, size;
6     int msg_send = 100, msg_recv;
7     MPI_Init(&argc, &argv);
8
9     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10    MPI_Comm_size(MPI_COMM_WORLD, &size);
11
12    if (size < 2) {
13        if (rank == 0)
14            printf("Run with at least 2 processes.\n");
15        MPI_Finalize();
16        return 0;
17    }
18
19    if (rank == 0) {
20        printf("Process 0 sending to Process 1...\n");
21        MPI_Send(&msg_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); // blocking send
22        MPI_Recv(&msg_recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
23        printf("Process 0 received from Process 1: %d\n", msg_recv);
24
25    } else if (rank == 1) {
26        printf("Process 1 sending to Process 0...\n");
27        MPI_Send(&msg_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // blocking send
28        MPI_Recv(&msg_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
29        printf("Process 1 received from Process 0: %d\n", msg_recv);
30    }
31
32    MPI_Finalize();
33    return 0;
34 }
35 //mpicc 6.c -o p6
36 //mpirun -np 2 ./p6
37
```

7.c

```
1 #include<mpi.h>
2 #include<stdio.h>
3
4 int main(int argc, char *argv[])
5 {
6     int rank, size;
7     int number;
8
9     MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14    if (rank == 0)
15    {
16        number = 50;
17        printf("Process %d broadcasting number %d to all other processes.\n", rank, number);
18    }
19
20    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
21
22    printf("Process %d received number %d\n", rank, number);
23
24    MPI_Finalize();
25    return 0;
26 }
27 //mpicc 7.c -o p7
28 //mpirun -np 4 ./p7
29
```

```
8.c

1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[])
5 {
6     int rank, size;
7     int send_data[100], recv_data, gathered_data[100];
8
9     MPI_Init(&argc, &argv);
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13    if (rank == 0)
14    {
15        for (int i = 0; i < size; i++)
16        {
17            send_data[i] = i * 10;
18        }
19    }
20
21    MPI_Scatter(send_data, 1, MPI_INT, &recv_data,
22                1, MPI_INT, 0, MPI_COMM_WORLD);
23
24    recv_data = recv_data + rank;
25
26    MPI_Gather(&recv_data, 1, MPI_INT,
27                gathered_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
28
29    if (rank == 0)
30    {
31        printf("Gathered data in root process:\n");
32        for (int i = 0; i < size; i++)
33        {
34            printf("gathered_data[%d] = %d\n", i, gathered_data[i]);
35        }
36    }
37
38    MPI_Finalize();
39    return 0;
40}
41 //mpicc 8.c -o p8
42 //mpirun -np 4 ./p8
43
44
```

```
9.c

1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[])
5 {
6     int rank, size;
7     int value, sum, prod, max, min;
8     int all_sum, all_prod, all_max, all_min;
9
10    MPI_Init(&argc, &argv);
11    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &size);
13
14    value = rank + 1;
15
16    MPI_Reduce(&value, &sum, 1, MPI_INT,
17                MPI_SUM, 0, MPI_COMM_WORLD);
18
19    MPI_Reduce(&value, &prod, 1, MPI_INT,
20                MPI_PROD, 0, MPI_COMM_WORLD);
21
22    MPI_Reduce(&value, &max, 1, MPI_INT,
23                MPI_MAX, 0, MPI_COMM_WORLD);
24
25    MPI_Reduce(&value, &min, 1, MPI_INT,
26                MPI_MIN, 0, MPI_COMM_WORLD);
27
28    MPI_Allreduce(&value, &all_sum, 1, MPI_INT,
29                  MPI_SUM, MPI_COMM_WORLD);
30
31    MPI_Allreduce(&value, &all_prod, 1, MPI_INT,
32                  MPI_PROD, MPI_COMM_WORLD);
33
34    MPI_Allreduce(&value, &all_max, 1, MPI_INT,
35                  MPI_MAX, MPI_COMM_WORLD);
36
37    MPI_Allreduce(&value, &all_min, 1, MPI_INT,
38                  MPI_MIN, MPI_COMM_WORLD);
39
40    if (rank == 0)
41    {
42        printf("Sum = %d\n", sum);
43        printf("Product = %d\n", prod);
44        printf("Max = %d\n", max);
45        printf("Min = %d\n", min);
46    }
47
48    printf("Process %d has value %d\n", rank, value);
49
50    printf("Process %d sees (MPI_Allreduce) :\n"
51           "Sum = %d, Prod = %d,\n"
```

```
52     "Max = %d, Min = %d\n",
53     rank, all_sum, all_prod, all_max, all_min);
54
55     MPI_Finalize();
56     return 0;
57 }
58 //mpicc 9.c -o p9
59 //mpirun -np 4 ./p9
60
```