RV Educational Institutions®

RV Institute of Technology and Management

(AffiliatedtoVTU,Belagavi)

JP Nagar 8th Phase, Bengaluru-560076

Department of Information Science and Engineering

Course Name: Parallel Computing Lab

CourseCode: BCS702

VII Semester

2022 Scheme

Prepared By:

Prof. Abhayakumar S Inchal Assistant Professor, Dept of ISE, RVITM

Program 1: Write a Open MP program to sort an array on n elements using both sequential and parallel merge sort (using Section). Record the difference in execution time.

```
#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)
{
        inti, 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
                {
                         // Switch to serial to avoid too many threads
                        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++)
        {
                intval = rand() % 100000;
                arr_serial[i] = val;
                arr_parallel[i] = val;
         }
            // ---------- SERIAL MERGE SORT ----------
         clock_tstart_serial = clock();
         serialMergeSort(arr_serial, 0, SIZE - 1);
        clock_tend_serial = clock();
        doubletime_serial = (double)(end_serial - start_serial) / CLOCKS_PER_SEC;
        // ---------- PARALLEL MERGE SORT ----------
```

```
        clock_tstart_parallel = clock();
        parallelMergeSort(arr_parallel, 0, SIZE - 1, 0);
        clock_tend_parallel = clock();
        doubletime_parallel = (double)(end_parallel - start_parallel) / CLOCKS_PER_SEC;

          // ---------- OUTPUT ----------
        printf("Serial Merge Sort Time   : %.6f seconds\n", time_serial);
        printf("Parallel Merge Sort Time : %.6f seconds\n", time_parallel);
        // Optional: Verify correctness
        /*
        for (inti = 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;
}
```

OUTPUT:

Case:1

The number of elements in the array ar 100000

Serial Merge Sort Time : 0.010000 seconds

Parallel Merge Sort Time : 0.002000 seconds

Process returned 0 (0x0) execution time : 0.161 s

Case:2

The number of elements in the array ar 1000000

Serial Merge Sort Time : 0.176000 seconds

Parallel Merge Sort Time : 0.100000 seconds

Process returned 0 (0x0) execution time : 0.472 s

Case:3

The number of elements in the array ar 10000000

Serial Merge Sort Time : 1.907000 seconds

Parallel Merge Sort Time : 1.131000 seconds

Process returned 0 (0x0) execution time : 3.318 s

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

```
#include<stdio.h>
#include<omp.h>
int main()
{
        int n = 16,thread;
        printf("\n Enter the number of tasks");
        scanf("%d",&n);
        printf("\n Enter the number of threads");
        scanf("%d",&thread);
        omp_set_num_threads(thread);
        printf("\n --------------------------------------\n");
        #pragma omp parallel for schedule(static, 2)
        for (inti = 0; i< n; i++)
        {
                printf("Thread %d executes iteration %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

OUTPUT:

```
Enter the number of tasks24
Enter the number of threads12
 -------------------------------------
Thread 1 executes iteration 2
Thread 1 executes iteration 3
Thread 4 executes iteration 8
Thread 4 executes iteration 9
Thread 6 executes iteration 12
Thread 6 executes iteration 13
Thread 9 executes iteration 18
Thread 9 executes iteration 19
Thread 7 executes iteration 14
Thread 7 executes iteration 15
Thread 10 executes iteration 20
Thread 10 executes iteration 21
Thread 0 executes iteration 0
Thread 0 executes iteration 1
Thread 2 executes iteration 4
Thread 2 executes iteration 5
Thread 3 executes iteration 6
Thread 3 executes iteration 7
Thread 5 executes iteration 10
Thread 5 executes iteration 11
Thread 11 executes iteration 22
Thread 11 executes iteration 23
Thread 8 executes iteration 16
Thread 8 executes iteration 17
```

Program 3: **Write a OpenMP program to calculate n Fibonacci numbers using tasks**.

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

// Serial Fibonacci calculation function
intser_fib (long int n)
 {
        if (n < 2) return n;
        longint x, y;
        x = fib(n - 1);
        y = fib(n - 2);
        return x + y;
}

// parallel Fibonacci calculation function
int fib(long int n)
{
        if (n < 2) return n;
                longint x, y;
        #pragma omp task shared(x)
                x = fib(n - 1);
        #pragma omp task shared(y)
                y = fib(n - 2);
        #pragma omptaskwait
                return x + y;
}
int main()
{
        long int n = 10, result;
        clock_t start, end;
        double cpu_time;
        printf("\n enter the value of n");
        scanf("%ld",&n);
        start=clock();
          #pragma omp parallel
          {
```

```
        #pragma omp single
            result = fib(n);
      }
    end=clock();
    cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Fibonacci(%d) = %d\n", n, result);
    printf("\n the time used to execute the program in parallel mode= %f",cpu_time);
    start=clock();
    result = ser_fib(n);
    end=clock();
    cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("\nFibonacci(%d) = %d\n", n, result);
    printf("\n the time used to execute the program in sequential mode= %f",cpu_time);
    return 0;
}
```

OUTPUT:-

```
enter the value of n32
Fibonacci(32) = 2178309
 The time used to execute the program in parallel mode= 3.370000
Fibonacci(32) = 2178309
the time used to execute the program in sequential mode= 0.157000
```

Why the Program is Inefficient for Parallelism

1. Exponential Recursion Tree

- The Fibonacci recursive algorithm has time complexity of **O(2^n)**.
- For example, to compute fib(50), it spawns **>10^{15} function calls**.
- Even with task parallelism, you're simply **parallelizing a bad algorithm**.

2. Too Many Tiny Tasks (Task Explosion)

- #pragma omp task creates a **new task** for each recursive call.
- Millions of small tasks are created, **overloading the thread pool**.
- Each task has an overhead (context, scheduling), which **kills performance**.

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

```
#include <stdio.h>

#include <omp.h>

#include<time.h>

int is_prime(int n)

{

        if (n < 2) return 0;

        for (inti = 2; i*i<= n; i++)

        if (n % i == 0) return 0;

        return 1;

}

int main()

 {

        long n = 10000000;

        time_tstart,end;

        double cpu_time;

        printf("\n the range of numbers is 1 to %d\n",n);

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

         // Serial Execution

        start=clock();

        for (inti = 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\n",cpu_time);

        // Parallel Execution

        start=clock();


        #pragma omp parallel for
```

```
        for (inti = 1; i<= n; i++)
         {
                is_prime(i);
         }
        end = clock();
        cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("Time to compute prime numbers Parallel: %f\n",cpu_time);
        return 0;
}
```

OUTPUT 1 :

the range of numbers is 1 to 10000000

Time to compute prime numbers serially: 2.403000

Time to compute prime numbers Parallel: 0.491000

OUTPUT 2:

the range of numbers is 1 to 1000000

Time to compute prime numbers serially: 0.095000

Time to compute prime numbers Parallel: 0.025000

OUTPUT 3:

the range of numbers is 1 to 100000

Time to compute prime numbers serially: 0.006000

Time to compute prime numbers Parallel: 0.000000

Program 5.Write a MPI Program to demonstration ofMPI_Send and MPI_Recv.

```
// Filename: mpi_send_recv_demo.c
#include <stdio.h>
#include <mpi.h>

int main(intargc, 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;
}
```

OUTPUT:

Process 0 is sending number 42 to Process 1

Process 1 received number 42 from Process 0

Program 6. Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence.

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

int main(intargc, 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;
}
```

OUTPUT:

Process 1 received message: 0

Process 0 received message: 1

Program 7. Write a MPI Program to demonstration of Broadcast operation.

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

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

        MPI_Init(&argc, &argv);                  // Initialize the MPI environment
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);        // Get the rank of the process

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

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

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

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

Output

Process 0 received data: 100
Process 1 received data: 100
Process 2 received data: 100
Process 3 received data: 100

Program 8. Write a MPI Program demonstration of MPI_Scatter and MPI_Gather.

```
#include <stdio.h>
#include <mpi.h>
int main(intargc, char** argv)
{
        int rank, size, send_data[4] = {10, 20, 30, 40}, recv_data;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        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 += 1;
        MPI_Gather(&recv_data, 1, MPI_INT, send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (rank == 0)
        {
                printf("Gathered data: ");
                for (inti = 0; i< size; i++)
                        printf("%d ", send_data[i]);
                printf("\n");
         }
        MPI_Finalize();
        return 0;
}
```

OUTPUT:

Process 0 received: 10

Process 1 received: 20

Process 2 received: 30

Process 3 received: 40

Gathered data: 11 21 31 41

Program 9. Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

```
#include <stdio.h>
#include <mpi.h>
int main(intargc, char** argv)
{
    int rank, value, sum, max;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    value = rank + 1;

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

    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;
}
```

Output:

Sum using Reduce: 10

Max using Allreduce (rank 0): 4

Max using Allreduce (rank 1): 4

Max using Allreduce (rank 2): 4

Max using Allreduce (rank 3): 4