EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

AN AUTONOMOUS INSTITUTION, AFFILIATED TO VTU, APPROVED BY AICTE
ACCREDITED BY NATIONAL BOARD OF ACCREDITATION (NBA) - CSE, ECE & ISE

NAAC GRADE A

Department of Computer Science & Engineering

Course Code : **BCS702**

Course Name : **Parallel Computing Lab**

Course Faculty : **Prof. K Prasanth**

Semester : **VII Sem**

Sl N	Section	Course Faculty Name	Signature	Date
1	B	Mr K Prasanth		18-07-2025

Parallel Programming Lab Manual

1. OpenMP Program: Sequential and Parallel MergeSort

Sort an array of n elements using both sequential and parallel mergesort (using Section). Record execution time.

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

void merge(int arr[], int l, int m, int r) {
    int i = l, j = m + 1, k = 0;
    int* temp = (int*)malloc((r - l + 1) * sizeof(int));
    while (i <= m && j <= r)
        temp[k++] = (arr[i] <= arr[j]) ? arr[i++] : 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];
    free(temp);
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(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* arr = (int*)malloc(n * sizeof(int));
    int* arr2 = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) arr[i] = arr2[i] = rand();

    double start = omp_get_wtime();
    mergeSort(arr, 0, n - 1);
    double end = omp_get_wtime();
    printf("Sequential Time: %f seconds\n", end - start);

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

    free(arr); free(arr2);
    return 0;
}
```

Expected Output:

- Displays execution time for both sequential and parallel implementations.

- May optionally print the sorted array.

2. OpenMP Program: Static Scheduling with Chunk Size 2

Divides loop iterations into static chunks of 2. Displays thread responsible for each iteration.

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

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

    #pragma omp parallel for schedule(static,2)
    for (i = 0; i < n; i++) {
        printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

Output:

- Shows which thread executed which iteration.
- Iterations are assigned in chunks of 2.

3. OpenMP Program: Fibonacci using Tasks

Calculates first n Fibonacci numbers using OpenMP tasks.

```
#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 n: ");
    scanf("%d", &n);
    int result;
    #pragma omp parallel
    {
        #pragma omp single
        result = fib(n);
    }
    printf("Fibonacci(%d) = %d\n", n, result);
    return 0;
}
```

Expected Output:

- Outputs the nth Fibonacci number.
- Execution trace may show task creation depending on implementation.

4. OpenMP Program: Prime Number Finder

Finds prime numbers from 1 to n using parallel for directive. Records serial and parallel execution times.

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

bool is_prime(int n) {
    if (n < 2) return false;
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0) return false;
    return true;
}

int main() {
    int n;
    printf("Enter upper limit: ");
    scanf("%d", &n);

    double start = omp_get_wtime();
    for (int i = 1; i <= n; i++)
        if (is_prime(i)) {}
    double end = omp_get_wtime();
    printf("Serial Time: %f\n", end - start);

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

    return 0;
}
```

Expected Output:

- Execution time for both serial and parallel computation.

- Optional: List of prime numbers from 1 to n.

5. MPI Program: MPI_Send and MPI_Recv

Demonstrates basic point-to-point communication using MPI_Send and MPI_Recv.

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

int main(int argc, char *argv[]) {
    int rank, size, value = 100;
```

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

    if (rank == 0) {
        MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent value %d\n", value);
    } else if (rank == 1) {
        MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received value %d\n", value);
    }

    MPI_Finalize();
    return 0;
}
```

Expected Output:
- Process 0 sends a value.

- Process 1 receives and prints the same value.

6. MPI Program: Deadlock and Avoidance

Demonstrates deadlock and avoidance using MPI_Send and MPI_Recv.

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

int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int value;
    if (rank == 0) {
        // Deadlock example (if both send first)
        // MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        // MPI_Recv(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        // Avoiding deadlock
        MPI_Recv(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 completed communication\n");
    } else if (rank == 1) {
```

```
        MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 completed communication\n");
    }

    MPI_Finalize();
    return 0;
}
```

Expected Output:

- May hang in deadlock mode if both processes call Send first.

- Successful communication when sequence is altered.

7. MPI Program: Broadcast Operation

Demonstrates MPI_Bcast to send a value from root process to all other processes.

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

int main(int argc, char *argv[]) {
    int rank, value;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        value = 50;
    }

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

    MPI_Finalize();
    return 0;
}
```

Expected Output:

8. MPI Program: Scatter and Gather

Demonstrates MPI_Scatter to distribute and MPI_Gather to collect data.

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

int main(int argc, char *argv[]) {
    int rank, size, data[4], recv;
```

```
    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 < 4; i++)
            data[i] = i + 1;
    }

    MPI_Scatter(data, 1, MPI_INT, &recv, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received %d\n", rank, recv);

    recv *= 2;

    MPI_Gather(&recv, 1, MPI_INT, data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Gathered values: ");
        for (int i = 0; i < 4; i++)
            printf("%d ", data[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

9. MPI Program: Reduce and Allreduce

Demonstrates MPI_Reduce and MPI_Allreduce operations (MAX, MIN, SUM, PROD).

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

int main(int argc, char *argv[]) {
    int rank, val, max, min, sum, prod;

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

    val = rank + 1;

    MPI_Reduce(&val, &max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&val, &min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
    MPI_Reduce(&val, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&val, &prod, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
```

```
    if (rank == 0) {
        printf("Max: %d\nMin: %d\nSum: %d\nProduct: %d\n", max, min, sum, prod);
    }

    MPI_Finalize();
    return 0;
}
```