Prerana Educational and Social Trust (R)

PES INSTITUTE OF TECHNOLOGY & MANAGEMENT

NH-206, SAGAR ROAD, SHIVAMOGGA-577204

(Approved by AICTE, New Delhi | Affiliated to VTU, Belagavi | Recognized by Govt. of Karnataka
Accredited by NAAC **'A'** | NBA Accredited Programs: CSE, CV, ECE, ISE, EEE & ME)

Department of Computer Science & Design

LABORATORY MANUAL

Semester : VII

Subject : Parallel Computing Lab (PC Lab)

Subject Code : BCS702-L

Name: ______________________________

USN: ______________________________

Kavya S
Course Instructor

Dr. Pramod
Head Of Department

PARALLEL COMPUTING		Semester	VII
Course Code	**BCS702**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory/Practical		

Course objectives:
This course will enable to,

- Explore the need for parallel programming
- Explain how to parallelize on MIMD systems
- To demonstrate how to apply MPI library and parallelize the suitable programs
- To demonstrate how to apply OpenMP pragma and directives to parallelize the suitable programs
- To demonstrate how to design CUDA program

SL NO	Experiments
1	Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort (using Section). Record the difference in execution time.
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
3	Write a OpenMP program to calculate n Fibonacci numbers using tasks.
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.
5	Write a MPI Program to demonstration of MPI_Send and MPI_Recv.
6	Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence
7	Write a MPI Program to demonstration of Broadcast operation.
8	Write a MPI Program demonstration of MPI_Scatter and MPI_Gather
9	Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

Course outcomes (Course Skill Set):
At the end of the course, the student will be able to:

- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. Theminimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the practical component of the IPCC

- 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.
- The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

Suggested Learning Resources:
Textbook:

1. Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman.
2. Michael J Quinn – Parallel Programming in C with MPI and OpenMp, McGrawHill.

Reference Books:

1. Calvin Lin, Lawrence Snyder – Principles of Parallel Programming, Pearson
2. Barbara Chapman – Using OpenMP: Portable Shared Memory Parallel Programming, Scientific and Engineering Computation
3. William Gropp, Ewing Lusk – Using MPI:Portable Parallel Programing, Third edition, Scientific and Engineering Computation

Web links and Video Lectures(e-Resources):

1. Introduction to parallel programming: https://nptel.ac.in/courses/106102163

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

- Programming Assignment at higher bloom level (10 Marks)

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

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

// Merge function

void merge(int* arr, int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int* L = (int*)malloc(n1 * sizeof(int));
    int* R = (int*)malloc(n2 * sizeof(int));
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    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);
}

// Sequential Merge Sort

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);
    }
}

// Parallel Merge Sort

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);
    }
}

// Check if array is sorted

int isSorted(int* arr, int n) {
    for (int i = 1; i < n; i++) {
        if (arr[i - 1] > arr[i]) return 0;
    }
    return 1;
}
int main() {
    int n = 1000000;
    int* arrSeq = (int*)malloc(n * sizeof(int));
    int* arrPar = (int*)malloc(n * sizeof(int));
    srand(42);
    for (int i = 0; i < n; i++) {
        arrSeq[i] = rand() % 100000;
        arrPar[i] = arrSeq[i];
    }
    // Time sequential sort
    double start = omp_get_wtime();
    mergeSortSequential(arrSeq, 0, n - 1);
    double end = omp_get_wtime();
    double timeSeq = end - start;
    // Time parallel sort
    start = omp_get_wtime();
    mergeSortParallel(arrPar, 0, n - 1, 4); // You can tune depth
    end = omp_get_wtime();
    double timePar = end - start;
    // Output
    printf("Sequential sort time: %.6f seconds\n", timeSeq);
    printf("Parallel sort time  : %.6f seconds\n", timePar);
    printf("Speedup             : %.2fx\n", timeSeq / timePar);
    if (!isSorted(arrSeq, n)) printf("Sequential sort failed!\n");
    if (!isSorted(arrPar, n)) printf("Parallel sort failed!\n");
    free(arrSeq);
    free(arrPar);
    return 0;
}
```

Output:

```
PS E:\Program\PC> gcc -fopenmp 1r.c -o 1r
PS E:\Program\PC> .\1r.exe
Sequential sort time: 0.174000 seconds
Parallel sort time  : 0.100000 seconds
Speedup             : 1.74x
```

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

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

int main() {
    int num_iterations;

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

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

    return 0;
}
```

Output:

```
PS E:\Program\PC> gcc -fopenmp 2.c -o 2
PS E:\Program\PC> .\2.exe
Enter the number of iterations: 6
Thread 0: Iteration 0
Thread 0: Iteration 1
Thread 1: Iteration 2
Thread 1: Iteration 3
Thread 2: Iteration 4
Thread 2: Iteration 5
```

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

```
#include <stdio.h>
#include <omp.h>
int fib(int n) {
   int i, j;
   if (n < 2)
      return n;
   else {
      #pragma omp task shared(i) firstprivate(n)
      i = fib(n - 1);
      #pragma omp task shared(j) firstprivate(n)
      j = fib(n - 2);
      #pragma omp taskwait
      return i + j;
   }
}
int main() {
   int n = 10;
   omp_set_dynamic(0);
   omp_set_num_threads(4);
   #pragma omp parallel
   {
      #pragma omp single
      {
         printf("fib(%d) = %d\n", n, fib(n));
      }
   }
   return 0;
}
```

Output:

```
PS E:\Program\PC> gcc -fopenmp 3.c -o 3
PS E:\Program\PC> .\3.exe
fib(10) = 55
```

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>

int main() {
    int prime[1000], i, j, n;

    // Prompt user for input
    printf("In order to find prime numbers from 1 to n, enter the value of n: ");
    scanf("%d", &n);

    // Initialize all numbers as prime (set all to 1)
    for (i = 1; i <= n; i++) {
        prime[i] = 1;
    }

    // 1 is not a prime number
    prime[1] = 0;

    // Sieve of Eratosthenes with parallelization
    for (i = 2; i * i <= n; i++) {
        if (prime[i]) {
            #pragma omp parallel for
            for (j = i * i; j <= n; j += i) {
                prime[j] = 0;
            }
        }
    }

    // Print prime numbers
    printf("Prime numbers from 1 to %d are:\n", n);
    for (i = 2; i <= n; i++) {
        if (prime[i] == 1) {
            printf("%d\t", i);
        }
    }
    printf("\n");

    return 0;
}
```

Output:

```
PS E:\Program\PC> gcc -fopenmp 4.c -o 4
PS E:\Program\PC> .\4.exe
In order to find prime numbers from 1 to n, enter the value of n: 100
Prime numbers from 1 to 100 are:
2       3       5       7       11      13      17      19      23      29      31      37      41      43      47      53      59
 61     67      71      73      79      83      89      97
```

Program 5: Write a MPI Program to demonstration of MPI_Send and MPI_Recv.

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

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

    MPI_Init(&argc, &argv); // Initialize MPI environment
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get current process ID
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get total number of processes

    if (size < 2) {
        if (rank == 0) {
            printf("This program requires at least 2 processes.\n");
        }
        MPI_Finalize();
        return 0;
    }

    if (rank == 0) {
        number = 100; // Message to send
        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(); // Clean up the MPI environment
    return 0;
}
```

Output:

****Note** FROM HERE THE COMPILATION AND RUNNING CODE CHANGES**

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_send_recv.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_send_recv.c -o mpi_
send_recv
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 2 ./mpi_send_recv
Process 0 sending number 100 to Process 1
Process 1 received number 100 from Process 0
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

Use above compilation: mpicc filename.c -o filename

To run: mpirun -np (any number) ./filename

In windows:

```
PS E:\Program\PC> gcc 5r.c `-I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include" `-L"C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64" `-lmsmpi -o 5r.exe
PS E:\Program\PC> & "C:\Program Files (x86)\Microsoft SDKs\MPI\Bin\mpiexec.exe" -n 2 .\5r.exe
Process 1 received number 100 from Process 0
Process 0 sending number 100 to Process 1
```

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

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

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

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

    if (size < 2) {
        printf("This program requires at least 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (rank == 0) {
        // Rank 0 sends first, then receives
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent data to Process 1\n");

        MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received data from Process 1\n");
    } else if (rank == 1) {
        // Rank 1 receives first, then sends
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received data from Process 0\n");

        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Process 1 sent data to Process 0\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output:
Note
Use above compilation: mpicc filename.c -o filename
To run: mpirun -np (any number) ./filename
The Above instruction follows, use 5th program picture as reference

```
PS E:\Program\PC> gcc 6.c `-I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include" `-L"C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64" `-lmsmpi -o 6.exe
PS E:\Program\PC> & "C:\Program Files (x86)\Microsoft SDKs\MPI\Bin\mpiexec.exe" -n 2 .\6.exe
Process 1 received data from Process 0
Process 1 sent data to Process 0
Process 0 sent data to Process 1
Process 0 received data from Process 1
```

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

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

int main(int argc, char** argv) {
    int rank, size;
    int data; // The data to broadcast

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

    if (rank == 0) {
        data = 42; // Root process sets the data
        printf("Process %d is broadcasting data = %d\n", rank, data);
    }

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

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

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

Output:
Note
Use above compilation: mpicc filename.c -o filename
To run: mpirun -np (any number) ./filename
The Above instruction follows, use 5th program picture as reference

```
PS E:\Program\PC> gcc 7.c `-I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include" `-L"C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64" `-lmsmpi -o 7.exe
PS E:\Program\PC> & "C:\Program Files (x86)\Microsoft SDKs\MPI\Bin\mpiexec.exe" -n 2 .\7.exe
Process 1 received data = 42
Process 0 is broadcasting data = 42
Process 0 received data = 42
```

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

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

int main(int argc, char** argv) {
    int rank, size;
    int send_data[4] = {10, 20, 30, 40};  // Only root (rank 0) uses this fully
    int recv_data;

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

    // Scatter 1 int from root to all processes
    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);

    // Each process increments its value
    recv_data += 1;

    // Gather updated values at root
    MPI_Gather(&recv_data, 1, MPI_INT, send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Root prints the gathered result
    if (rank == 0) {
        printf("Gathered data: ");
        for (int i = 0; i < size; i++)
            printf("%d ", send_data[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output:

Note

Use above compilation: mpicc filename.c -o filename

To run: mpirun -np (any number) ./filename

The Above instruction follows, use 5th program picture as reference

```
PS E:\Program\PC> gcc 8r.c `-I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include" `-L"C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64" `-lmsmpi -o 8r.exe
PS E:\Program\PC> & "C:\Program Files (x86)\Microsoft SDKs\MPI\Bin\mpiexec.exe" -n 4 .\8r.exe
Process 3 received: 40
Process 0 received: 10
Gathered data: 11 21 31 41
Process 2 received: 30
Process 1 received: 20
```

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

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

int main(int argc, char** argv) {
    int rank, size;
    int value;
    int sum, prod, max, min;
    int sum_all, prod_all, max_all, min_all;

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

    // Each process sets its own value (e.g., rank + 1)
    value = rank + 1;
    printf("Process %d has value %d\n", rank, value);

    // ---------- MPI_Reduce ----------
    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);

    if (rank == 0) {
        printf("\n[Using MPI_Reduce at Root Process]\n");
        printf("Sum  = %d\n", sum);
        printf("Prod = %d\n", prod);
        printf("Max  = %d\n", max);
        printf("Min  = %d\n", min);
    }

    // ---------- MPI_Allreduce ----------
    MPI_Allreduce(&value, &sum_all, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&value, &prod_all, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
    MPI_Allreduce(&value, &max_all, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&value, &min_all, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    printf("\n[Process %d] MPI_Allreduce Results:\n", rank);
    printf(" Sum  = %d\n", sum_all);
    printf(" Prod = %d\n", prod_all);
    printf(" Max  = %d\n", max_all);
    printf(" Min  = %d\n", min_all);

    MPI_Finalize();
    return 0;
}
```

Output:

Note

Use above compilation: mpicc filename.c -o filename

To run: mpirun -np (any number) ./filename

The Above instruction follows, use 5th program picture as reference

```
PS E:\Program\PC> gcc 9r.c `-I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include" `-L"C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64" `-lmsmpi -o 9r.exe
PS E:\Program\PC> & "C:\Program Files (x86)\Microsoft SDKs\MPI\Bin\mpiexec.exe" -n 4 .\9r.exe
Process 3 has value 4

[Process 3] MPI_Allreduce Results:
 Sum  = 10
 Prod = 24
 Max  = 4
 Min  = 1
Process 2 has value 3

[Process 2] MPI_Allreduce Results:
 Sum  = 10
 Prod = 24
 Max  = 4
 Min  = 1
Process 1 has value 2

[Process 1] MPI_Allreduce Results:
 Sum  = 10
 Prod = 24
 Max  = 4
 Min  = 1
Process 0 has value 1

[Using MPI_Reduce at Root Process]
Sum  = 10
Prod = 24
Max  = 4
Min  = 1

[Process 0] MPI_Allreduce Results:
 Sum  = 10
 Prod = 24
 Max  = 4
 Min  = 1
```