ಬಿ.ಎಂ.ಎಸ್. ತಾಂತ್ರಿಕ ಮತ್ತು ವ್ಯವಸ್ಥಾಪನಾ ಮಹಾವಿದ್ಯಾಲಯ
(ವಿ.ಟಿ.ಯು. ಅಡಿಯಲ್ಲಿನ ಸ್ವಾಯತ್ತ ಸಂಸ್ಥೆ)
BMS INSTITUTE OF TECHNOLOGY & MANAGEMENT
(Autonomous Under VTU)

PARALLEL COMPUTING LABORATORY MANUAL

Course Code: BCSL706
(Academic Year: 2025-26)

Department of Information Science & Engineering
BMS Institute of Technology and Management
Bengaluru - 560064

B.E. INFORMATION SCIENCE AND ENGINEERING

Choice Based Credit System (CBCS) for 2022 Scheme

SEMESTER -VII

PARALLEL COMPUTING LABORATORY(0:0:2:0)1

(Effective from the academic year 2025-26)

Course Code	BCSL706	**CIE Marks**	50
Teaching Hours/Week(L:T:P)	0:0:2	**SEE Marks**	50
Total Number of Contact Hours	26	**Exam Hours**	3

Course Objectives:

This course will enable students to:

1. Design and implement high performance versions of standard single threaded algorithms
2. Demonstrate the architectural features in the GPU and MIC hardware accelerators
3. Design programs to extract maximum performance in a multi core, shared memory execution environment processor
4. Develop programs using OPENMP, MPI and CUDA

Program List

1.	Given a nxn matrix A and a vector x of length n, their product y=A·x. Write a program to implement the multiplication using OpenMP PARALLEL directive.
2.	Consider a Scenario where a person visits a supermarket for shopping. He purchases various items in different sections such as clothing, gaming, grocery, stationary. Write an OpenMP program to process his bill parallelly in each section and display the final amount to be paid. (Sum of elements parallelly).
3.	Consider a Person named X on the earth; to find his accurate position on the globe we require the value of Pi. Write a program to compute the value of pi function by Numerical Integration using OpenMP PARALLEL section.
4	Using OpenMP, design and develop a multi-threaded program to generate and print Fibonacci Series. One thread must generate the numbers up to the specified limit and another thread must print them. Ensure proper synchronization
5.	A university awards gold medals to students who have achieved the highest CGPA. Write a program using OpenMP to find the student with the highest CGPA.
6.	Multiply two square matrices (1000,2000 or 3000dimensions). Compare the performance of a sequential and parallel algorithm using OpenMP.
7.	Assume you have n robots which pick mangoes in a farm. Write a program to calculate the total number of mangoes picked by n robots parallelly using MPI.
8.	Design a MPI program that uses blocking send/receive routines and non-blocking send/receive routines.
9.	Design a program that implements application of MPI Collective Communications.

PART B

CUDA is a parallel computing platform and an API model that was developed by Nvidia. Using CUDA one can utilize the power of Nvidia GPUs to perform general computing tasks, such as multiplying matrices and performing other linear algebra operations, instead of just doing graphical calculations. Students write programs in CUDA and understand the efficiency and power of parallelism.

Course Outcomes:
CO1:Design and implement high performance versions of standard single threaded algorithms
CO2: Demonstrate the architectural features in the GPU and MIC hardware accelerators
CO3 Design programs to extract maximum performance in a multicore, shared memory execution environment processor
CO4: Develop programs using OPENMP, MPI and CUDA

Text Books

1. Introduction to parallel computing Ananth Grama, Anshul Gupta, Vipin Kumar, George Karypis Pearson education publishers second edition, 2003
2. Programming Massively Parallel Processors on Approach David B Kirk, Wenmei W. Hwu Elsevier and nvidia publishers First edition 2010
3. Introduction to High Performance Computing for Scientists and Engineers Georg Hager, Gerhard Wellein Taylor and Francis Group, LLC, CRC Press 2011

Reference Books

1. Parallel Programming for Multicore and cluster systems Thomas Rauber and Gudula Runger Springer International Edition 2009
2. Parallel Programming in C with MPI and Open MP Michael J. Quin McGraw Hill 1st Edition

PC Lab Evaluation Scheme

1. Ten marks for every experiment (9X10 =90marks), Scaled down to ***30 marks***.
2. Ten marks for every experiment will be evaluated for write-up, program execution, the procedure followed while execution and viva voce after each exercise.
3. Internal practical test for 100 marks to be given and the marks scored will be scaled down to ***20 marks***.
4. A Minimum of ***20 marks*** is to be scored in CIE.
5. SEE examination for the Lab is to be conducted for 100 marks and reduced to ***50 marks***.
6. A Minimum of ***18marks*** is to be scored in SEE.

Note: Open Ended experiment will be done by the students in the Lab session. A total mark of 40 is to be scored by the student from both CIE and SEE together out of 100.

<u>EXPERIMENT 1 :</u>

Given a nxn matrix A and a vector x of length n, their product y=A·x. Write a program to implement the multiplication using OpenMP PARALLEL directive.

Program:

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

int main() {
    int n, i, j;

    // Input size of the matrix/vector
    printf("Enter the size of the matrix/vector (n): ");
    scanf("%d", &n);

    // Allocate memory for matrix A, vector x and result y
    double **A = (double **)malloc(n * sizeof(double *));
    double *x = (double *)malloc(n * sizeof(double));
    double *y = (double *)malloc(n * sizeof(double));

    for (i = 0; i < n; i++) {
        A[i] = (double *)malloc(n * sizeof(double));
    }

    // Input matrix A
    printf("Enter elements of matrix A (%d x %d):\n", n, n);
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%lf", &A[i][j]);

    // Input vector x
    printf("Enter elements of vector x (%d elements):\n", n);
    for (i = 0; i < n; i++)
        scanf("%lf", &x[i]);

    // Matrix-vector multiplication using OpenMP
    #pragma omp parallel for private(j)
    for (i = 0; i < n; i++) {
        y[i] = 0;
        for (j = 0; j < n; j++) {
            y[i] += A[i][j] * x[j];
        }
    }

    // Print the result vector y
    printf("Result vector y = A * x:\n");
    for (i = 0; i < n; i++) {
        printf("%lf ", y[i]);
    }
    printf("\n");

    // Free allocated memory
    for (i = 0; i < n; i++)
        free(A[i]);
    free(A);
    free(x);
    free(y);

    return 0;
}
```

Output:

```
bmsit@fedora:~$ gedit mm.c
bmsit@fedora:~$ cc mm.c
bmsit@fedora:~$ ./a.out
Enter the size of the matrix/vector (n): 3
Enter elements of matrix A (3 x 3):
1 2 3
4 5 6
7 8 9
Enter elements of vector x (3 elements):
1 1 1
Result vector y = A * x:
6.000000 15.000000 24.000000
```

<u>EXPERIMENT 2 :</u>

Consider a Scenario where a person visits a supermarket for shopping. He purchases various items in different sections such as clothing, gaming, grocery, stationary. Write an OpenMP program to process his bill parallelly in each section and display the final amount to be paid. (Sum of elements parallelly).

Program:

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

int main() {
    // Arrays representing item prices in each section
    double clothing[] = {1200.5, 799.0, 450.0};         // 3 items
    double gaming[] = {3000.0, 1500.0};                 // 2 items
    double grocery[] = {250.5, 130.75, 90.0, 60.25};    // 4 items
    double stationary[] = {45.5, 99.0};                 // 2 items

    // Sizes
    int n_clothing = sizeof(clothing)/sizeof(clothing[0]);
    int n_gaming = sizeof(gaming)/sizeof(gaming[0]);
    int n_grocery = sizeof(grocery)/sizeof(grocery[0]);
    int n_stationary = sizeof(stationary)/sizeof(stationary[0]);

    double total = 0.0;

    // Parallel computation using OpenMP reduction
    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < n_clothing; i++)
        total += clothing[i];

    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < n_gaming; i++)
        total += gaming[i];

    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < n_grocery; i++)
        total += grocery[i];

    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < n_stationary; i++)
        total += stationary[i];

    // Print the final bill
    printf("Final Bill Amount: ₹%.2lf\n", total);

    return 0;
}
```

OUTPUT:

```
bmsit@fedora:~$ gedit mm1.c
bmsit@fedora:~$ cc mm1.c
bmsit@fedora:~$  ./a.out
Final Bill Amount: ₹7625.50
```

EXPERIMENT 3 :

Consider a Person named X on the earth; to find his accurate position on the globe we require the value of Pi. Write a program to compute the value of pi function by Numerical Integration using OpenMP PARALLEL section.

Program:

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

int main() {
    long num_steps = 100000000; // Higher value gives better accuracy
    double step = 1.0 / (double)num_steps;
    double pi = 0.0;

    // Partial sums by sections
    double sum1 = 0.0, sum2 = 0.0;

    double start_time = omp_get_wtime();

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            for (long i = 0; i < num_steps/2; i++) {
                double x = (i + 0.5) * step;
                sum1 += 4.0 / (1.0 + x * x);
            }
        }

        #pragma omp section
        {
            for (long i = num_steps/2; i < num_steps; i++) {
                double x = (i + 0.5) * step;
                sum2 += 4.0 / (1.0 + x * x);
            }
        }
    }

    pi = step * (sum1 + sum2);
    double end_time = omp_get_wtime();

    printf("Computed Pi = %.15f\n", pi);
    printf("Execution Time = %f seconds\n", end_time - start_time);

    return 0;
}
```

OUTPUT:

```
bmsit@fedora:~$ gedit mm2.c
bmsit@fedora:~$ cc -fopenmp mm2.c
bmsit@fedora:~$ ./a.out
Computed Pi = 3.141592653589910
Execution Time = 0.144117 seconds
```

EXPERIMENT 4 :

Using OpenMP, design and develop a multi-threaded program to generate and print Fibonacci Series. One thread must generate the numbers up to the specified limit and another thread must print them. Ensure proper synchronization

Program:

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

#define MAX 100  // Maximum size for Fibonacci series

int main() {
    int n;
    int fib[MAX];  // Shared array to hold the Fibonacci series

    printf("Enter the number of terms in Fibonacci Series: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX) {
        printf("Invalid input. Please enter a number between 1 and %d.\n", MAX);
        return 1;
    }

    // Parallel region with two sections
    #pragma omp parallel sections shared(fib, n)
    {
        // Section 1: Generate Fibonacci series
        #pragma omp section
        {
            fib[0] = 0;
            if (n > 1)
                fib[1] = 1;

            for (int i = 2; i < n; i++) {
                fib[i] = fib[i - 1] + fib[i - 2];
            }

            // Ensure generation is complete before printing
            #pragma omp flush(fib)
        }

        // Section 2: Print Fibonacci series
        #pragma omp section
        {
            // Wait to ensure generation completes (simple sync)

            printf("Fibonacci Series:\n");
            for (int i = 0; i < n; i++) {
                printf("%d ", fib[i]);
            }
            printf("\n");
        }
    }
    return 0;
}
```

OUTPUT:

```
bmsit@fedora:~$ gedit mm3.c
bmsit@fedora:~$ cc -fopenmp mm3.c
bmsit@fedora:~$ ./a.out
Enter the number of terms in Fibonacci Series: 5
Fibonacci Series:
0 1 1 2 3
```

EXPERIMENT 5 :

A university awards gold medals to students who have achieved the highest CGPA. Write a program using OpenMP to find the student with the highest CGPA.

Program:

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

#define MAX_STUDENTS 100

typedef struct {
    char name[50];
    float cgpa;
} Student;

int main() {
    int n;
    Student students[MAX_STUDENTS];

    printf("Enter number of students: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX_STUDENTS) {
        printf("Invalid number of students.¥n");
        return 1;
    }

    // Input student data
    for (int i = 0; i < n; i++) {
        printf("Enter name and CGPA of student %d: ", i + 1);
        scanf("%s %f", students[i].name, &students[i].cgpa);
    }

    float maxCGPA = -1.0;
    int maxIndex = -1;

    // Parallel region to find student with highest CGPA
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        #pragma omp critical
        {
            if (students[i].cgpa > maxCGPA) {
                maxCGPA = students[i].cgpa;
                maxIndex = i;
            }
        }
    }

    if (maxIndex != -1) {
        printf("¥n Gold Medalist: %s with CGPA %.2f¥n", students[maxIndex].name,
students[maxIndex].cgpa);
    }
    return 0;
}
```

OUTPUT:

```
bmsit@fedora:~$ gedit mm4.c
bmsit@fedora:~$ cc -fopenmp mm4.c
bmsit@fedora:~$ ./a.out
Enter number of students: 3
Enter name and CGPA of student 1: RAHUL 7
Enter name and CGPA of student 2: RAKESH 8
Enter name and CGPA of student 3: ROHAN 9

 Gold Medalist: ROHAN with CGPA 9.00
```

EXPERIMENT 6 :

Multiply two square matrices (1000,2000 or 3000dimensions). Compare the performance of a sequential and parallel algorithm using OpenMP.

Program:

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

// Function to allocate memory for a 2D matrix
float** allocateMatrix(int size) {
    float **matrix = (float**) malloc(size * sizeof(float*));
    for (int i = 0; i < size; i++)
        matrix[i] = (float*) malloc(size * sizeof(float));
    return matrix;
}

// Fill a matrix with random values
void fillMatrix(float **mat, int size) {
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            mat[i][j] = rand() % 10;
}

// Sequential multiplication
void sequentialMultiply(float **A, float **B, float **C, int size) {
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) {
            C[i][j] = 0;
            for (int k = 0; k < size; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

// Parallel multiplication using OpenMP
void parallelMultiply(float **A, float **B, float **C, int size) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) {
            C[i][j] = 0;
            for (int k = 0; k < size; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

// Free dynamically allocated matrix
void freeMatrix(float **matrix, int size) {
    for (int i = 0; i < size; i++)
        free(matrix[i]);
    free(matrix);
}

int main() {
    int size;
    printf("Enter matrix size (1000, 2000, 3000): ");
    scanf("%d", &size);

    float **A = allocateMatrix(size);
    float **B = allocateMatrix(size);
    float **C_seq = allocateMatrix(size);
    float **C_par = allocateMatrix(size);
    fillMatrix(A, size);
    fillMatrix(B, size);

    double start, end;

    // Sequential multiplication
    start = omp_get_wtime();
    sequentialMultiply(A, B, C_seq, size);
    end = omp_get_wtime();
    printf("□ Sequential Time: %.4f seconds\n", end - start);

    // Parallel multiplication
    start = omp_get_wtime();
    parallelMultiply(A, B, C_par, size);
    end = omp_get_wtime();
    printf("⚡ Parallel Time (OpenMP): %.4f seconds¥n", end - start);

    // Cleanup
    freeMatrix(A, size);
    freeMatrix(B, size);
    freeMatrix(C_seq, size);
    freeMatrix(C_par, size);

    return 0;
}
```

OUTPUT:

```
bmsit@fedora:~$ gedit mm6.c
bmsit@fedora:~$ sudo dnf install openmpi openmpi-devel
bmsit@fedora:~$ export PATH=$PATH:/usr/lib64/openmpi/bin
bmsit@fedora:~$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64/openmpi/lib
bmsit@fedora:~$ mpicc mm6.c -o mangoes_mpi
bmsit@fedora:~$ mpirun -np 4 ./mangoes_mpi
```

```
Robot 0 picked 17 mangoes.
Robot 2 picked 11 mangoes.
Robot 1 picked 22 mangoes.
Robot 3 picked 94 mangoes.

Total mangoes picked by all robots: 144
```

<u>EXPERIMENT 7 :</u>

Assume you have n robots which pick mangoes in a farm. Write a program to calculate the total number of mangoes picked by n robots parallelly using MPI.

Program:

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

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

    MPI_Request request;
    MPI_Status status;
```

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

if (size < 2) {
    if (rank == 0)
        printf("Run the program with at least 2 processes.¥n");
    MPI_Finalize();
    return 0;
}

// -----------------------------
// ✅Blocking Communication
// -----------------------------
if (rank == 0) {
    printf("[Blocking] Process 0 sending data: %d¥n", data_send);
    MPI_Send(&data_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recv(&data_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("[Blocking] Process 1 received data: %d¥n", data_recv);
}

MPI_Barrier(MPI_COMM_WORLD); // Synchronize before next communication

// -----------------------------
// □ Non-Blocking Communication
// -----------------------------
if (rank == 0) {
    data_send = 200;
    printf("[Non-Blocking] Process 0 initiating send: %d¥n", data_send);
    MPI_Isend(&data_send, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status); // Ensure send completes
} else if (rank == 1) {
    MPI_Irecv(&data_recv, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status); // Ensure receive completes
```

```
        printf("[Non-Blocking] Process 1 received data: %d¥n", data_recv);
    }

    MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
bmsit@fedora:~$ gedit mm7.c
bmsit@fedora:~$ mpicc mm7.c -o mpi_com
bmsit@fedora:~$ mpirun -np 2 ./mpi_com
[Blocking] Process 0 sending data: 100
[Blocking] Process 1 received data: 100
[Non-Blocking] Process 1 received data: 200
[Non-Blocking] Process 0 initiating send: 200
```

<u>EXPERIMENT 8 :</u>

Design a MPI program that uses blocking send/receive routines and non-blocking send/receive routines.

Program:

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

int main(int argc, char *argv[]) {
    int rank, size;
    int numbers[4] = {10, 20, 30, 40};
    int recv_num, doubled, gathered[4], total_sum;
```

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

if (size != 4) {
    if (rank == 0) {
        printf("Please run this with exactly 4 processes.¥n");
    }
    MPI_Finalize();
    return 1;
}

// 1. Broadcast from root to all (not needed here, but for demonstration)
int broadcast_value = 100;
MPI_Bcast(&broadcast_value, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("Process %d received broadcast value: %d¥n", rank, broadcast_value);

// 2. Scatter - distribute one number to each process
MPI_Scatter(numbers, 1, MPI_INT, &recv_num, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("Process %d received number: %d¥n", rank, recv_num);

// Each process does some work (e.g., doubles the number)
doubled = recv_num * 2;

// 3. Gather - collect processed numbers at root
MPI_Gather(&doubled, 1, MPI_INT, gathered, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Root gathered doubled numbers: ");
    for (int i = 0; i < 4; i++) {
        printf("%d ", gathered[i]);
    }
    printf("¥n");
}

// 4. Reduce - compute sum of doubled numbers at root
MPI_Reduce(&doubled, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    if (rank == 0) {
        printf("Total sum of doubled numbers: %d¥n", total_sum);
    }

    // 5. Broadcast the total sum to all processes
    MPI_Bcast(&total_sum, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d sees total sum as: %d¥n", rank, total_sum);
    MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
bmsit@fedora:~$ gedit mm8.c
bmsit@fedora:~$ mpicc mm8.c -o mpi_collective
bmsit@fedora:~$ mpirun -np 4 ./mpi_collective
Process 0 received broadcast value: 100
Process 0 received number: 10
Root gathered doubled numbers: 20 40 60 80
Total sum of doubled numbers: 200
Process 0 sees total sum as: 200
Process 2 received broadcast value: 100
Process 2 received number: 30
Process 2 sees total sum as: 200
Process 3 received broadcast value: 100
Process 3 received number: 40
Process 3 sees total sum as: 200
Process 1 received broadcast value: 100
Process 1 received number: 20
Process 1 sees total sum as: 200
```

EXPERIMENT 9 :

Design a program that implements application of MPI Collective Communications.

Program:

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

int main(int argc, char* argv[]) {
    int rank, size, i;
    int n = 4; // total elements
    int data[4] = {1, 2, 3, 4};
    int recv;
    int squared;
    int gathered[4];
    int total_sum;

    MPI_Init(&argc, &argv); // Initialize MPI
    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 != n) {
        if (rank == 0) {
            printf("Run with exactly %d processes.\n", n);
        }
        MPI_Finalize();
        return 0;
    }

    // Broadcast n to all processes
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Scatter the array to all processes
    MPI_Scatter(data, 1, MPI_INT, &recv, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Each process squares its received value
    squared = recv * recv;

    // Gather the squared results to root process
    MPI_Gather(&squared, 1, MPI_INT, gathered, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Reduce to calculate sum of squares
    MPI_Reduce(&squared, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Print results at root
    if (rank == 0) {
        printf("Squared values: ");
        for (i = 0; i < n; i++) {
            printf("%d ", gathered[i]);
        }
        printf("\nSum of squares = %d\n", total_sum);
    }

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

OUTPUT:

```
bmsit@fedora:~$ gedit mm9.c
bmsit@fedora:~$ export PATH=$PATH:/usr/lib64/openmpi/bin
bmsit@fedora:~$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib64/openmpi/lib
bmsit@fedora:~$ mpicc -o mpi_collective mm9.c
bmsit@fedora:~$ mpirun -np 4 ./mpi_collective
Squared values: 1 4 9 16
Sum of squares = 30
```