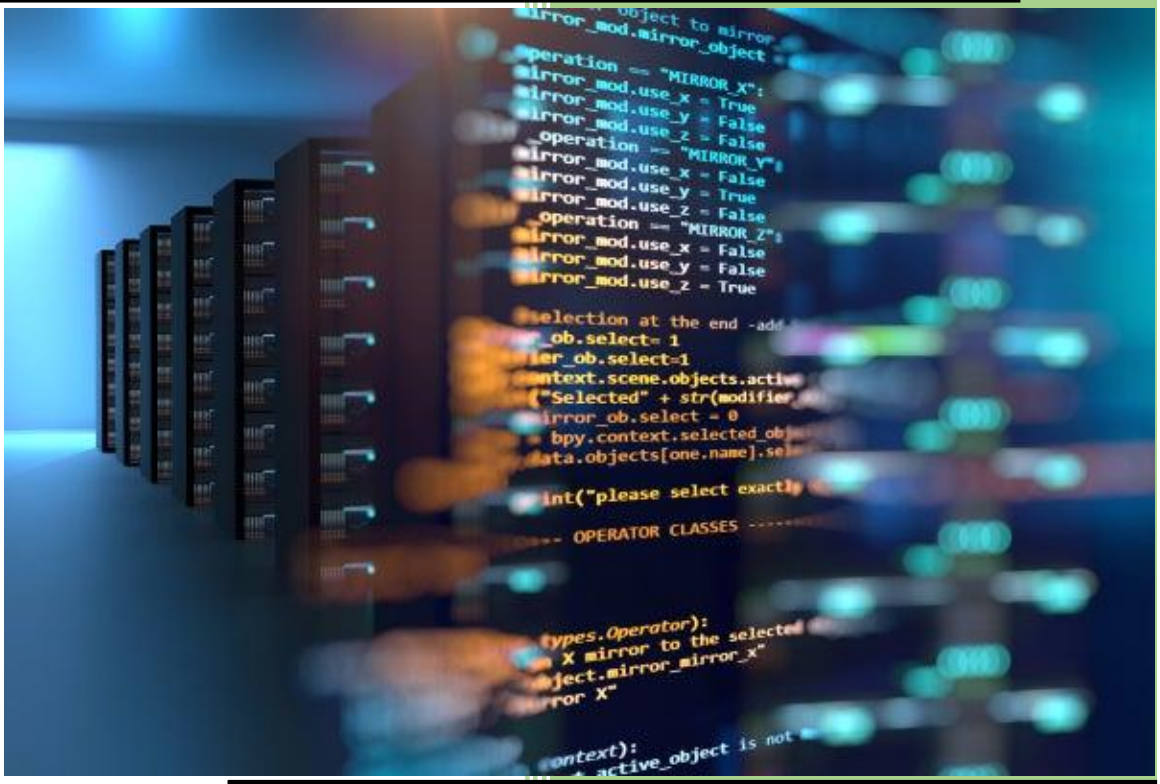
BLDEA's

Vachana Pitamaha Dr. P.G. Halakatti College of Engineering & Technology, Vijayapura

Lab Manual : PARALLEL COMPUTING (BCS702)



7th Sem Integrated Professional Core Course

Department of Computer Science & Engineering

2025-2026

PARALLEL COMPUTING (BCS702)

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

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.

Table of Contents

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.

Objective:
Sort an array using both sequential and parallel mergesort. Record the execution time of both.

Code:

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

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[n1], R[n2];

   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++];
}
```

```
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 parallel_mergesort(int arr[], int l, int r) {
   if(l < r) {
      int m = (l + r) / 2;
      #pragma omp parallel sections
      {
         #pragma omp section
         parallel_mergesort(arr, l, m);
         #pragma omp section
         parallel_mergesort(arr, m + 1, r);
      }
      merge(arr, l, m, r);
   }
}

int main() {
   int n = 100000;
   int *arr = malloc(n * sizeof(int));
   for(int i = 0; i < n; i++) arr[i] = rand() % 1000;
   int *copy = malloc(n * sizeof(int));
   memcpy(copy, arr, n * sizeof(int));
```

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

    memcpy(arr, copy, n * sizeof(int));
    start = omp_get_wtime();
    parallel_mergesort(arr, 0, n - 1);
    end = omp_get_wtime();
    printf("Parallel Time: %f\n", end - start);

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

Sample Output:

Sequential Time: 0.452341

Parallel Time: 0.278123

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

Objective:
Demonstrate static scheduling with chunk size = 2. Print which thread executes which iterations.

Code:

```
#include <stdio.h>
#include <omp.h>
int main() {
   int n;
   printf("Enter number of iterations: ");
   scanf("%d", &n);

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

Sample Output:

```
Thread 0 executes iteration 0
Thread 0 executes iteration 1
Thread 1 executes iteration 2
Thread 1 executes iteration 3
```

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

Objective:
Calculate first N Fibonacci numbers using tasks.

Code:

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

int fib(int n) {
   if (n < 2) return n;
   int x, y;

   #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 value of n: ");
   scanf("%d", &n);

   int result;
   #pragma omp parallel
   {
```

```
        #pragma omp single
        result = fib(n);
    }
    printf("Fibonacci(%d) = %d\n", n, result);
    return 0;
}
```

Sample Output:

Enter value of n: 10

Fibonacci(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.

Objective:
Find all prime numbers from 1 to N using parallel for.

Code:

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

int is_prime(int n) {
   if(n < 2) return 0;
   for(int i = 2; i <= sqrt(n); i++)
      if(n % i == 0) return 0;
   return 1;
}

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

   double start = omp_get_wtime();
   for(int i = 2; i <= n; i++)
      if(is_prime(i)) printf("%d ", i);
   double end = omp_get_wtime();
   printf("\nSerial Time: %f\n", end - start);

   start = omp_get_wtime();
```

```
    #pragma omp parallel for
    for(int i = 2; i <= n; i++) {
        if(is_prime(i))
            printf("%d ", i);
    }
    end = omp_get_wtime();
    printf("\nParallel Time: %f\n", end - start);

    return 0;
}
```

Sample Output:

Enter upper limit: 20

Serial Prime Numbers:

2 3 5 7 11 13 17 19

Serial Time: 0.000123 seconds

Parallel Prime Numbers:

2 3 5 7 11 13 17 19

Parallel Time: 0.000074 seconds

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

Objective:
Demonstrate sending and receiving messages using MPI.

Code:

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

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

   if(rank == 0) {
      number = 42;
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      printf("Process 0 sent number %d\n", number);
   } else if(rank == 1) {
      MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process 1 received number %d\n", number);
   }

   MPI_Finalize();
   return 0;
}
```

Program Description:

This MPI program involves at least two processes:

- Process 0 sends a message.

- Process 1 receives the message using MPI_Recv.

Assumptions:

- The message is an integer (42) sent from Process 0 to Process 1.

Sample Output (when run with 2 processes):

arduino

CopyEdit

Process 0 sending value 42 to Process 1

Process 1 received value 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

Objective:
Demonstrate deadlock and its avoidance using MPI.

Part A: Deadlock Example

Code (Deadlock-prone)

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

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

   if (rank == 0) {
      int msg = 100;
      MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   } else if (rank == 1) {
      int msg = 200;
      MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
   }

   MPI_Finalize();
   return 0;
}
```

Explanation:

Both processes wait for Recv first, which leads to a deadlock as neither can proceed to Send.

Sample Output (Deadlock)

$ mpirun -np 2 ./mpi_deadlock

 # Program hangs indefinitely — no output is produced

Part B: Deadlock-Free Version

Code (Avoiding Deadlock by Call Order)

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

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

   if (rank == 0) {
      int msg = 100;
      MPI_Send(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process 0 received %d from Process 1\n", data);
   } else if (rank == 1) {
      int msg = 200;
      MPI_Send(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process 1 received %d from Process 0\n", data);
   }
```

```
    MPI_Finalize();
    return 0;
}
```

Sample Output (Deadlock-Free)

```
$ mpirun -np 2 ./mpi_no_deadlock
Process 0 received 200 from Process 1
Process 1 received 100 from Process 0
```

Key Point:

Deadlock occurs when both processes block on MPI_Recv waiting for a message that hasn't been sent yet. To avoid this, either:

- Use MPI_Send first in at least one process.
- Or use MPI_Sendrecv() or MPI_Isend/MPI_Irecv.

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

Objective:
Broadcast a value from root process to all others.

Program Summary:

- **Objective: One process (typically rank 0) broadcasts a value to all other processes using MPI_Bcast.**
- **Assumption: Broadcasting an integer value = 42 from root process (rank 0) to all processes.**

Typical Code Snippet:

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

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

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

   if (rank == 0) {
      value = 42;
      printf("Process %d broadcasting value: %d\n", rank, value);
   }

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

   MPI_Finalize();
```

```
    return 0;
}
```

Sample Output (using 4 processes):

$ mpirun -np 4 ./mpi_broadcast

Process 0 broadcasting value: 42

Process 0 received value: 42

Process 1 received value: 42

Process 2 received value: 42

Process 3 received value: 42

Notes:

- MPI_Bcast ensures all processes have the same value.
- Execution order of printed output may vary across runs due to concurrent processes.
- Works for any datatype or structured data (with proper MPI_Datatype**).**

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

Objective: To demonstrate the use of MPI_Scatter to distribute chunks of an array from the root process to all processes, and MPI_Gather to collect modified results from all processes back to the root.

- **MPI_Scatter distributes different parts of an array to each process.**
- **Each process performs an operation (e.g., multiply received value by 2).**
- **MPI_Gather collects the updated data from all processes to the root process**

Code:

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

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

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

   if (rank == 0) {
      data[0] = 10; data[1] = 20; data[2] = 30; data[3] = 40;
      printf("Root process has array: ");
      for(int i = 0; i < 4; i++) printf("%d ", data[i]);
      printf("\n");
   }

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

```
    recv = recv * 2;

    MPI_Gather(&recv, 1, MPI_INT, data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Root process gathered array: ");
        for(int i = 0; i < 4; i++) printf("%d ", data[i]);
        printf("\n");
    }
    MPI_Finalize();
    return 0;
}
```

Sample Output (with 4 processes):

```
$ mpirun -np 4 ./mpi_scatter_gather
Root process has array: 10 20 30 40
Process 0 received 10
Process 1 received 20
Process 2 received 30
Process 3 received 40
Root process gathered array: 20 40 60 80
```

Notes:

- The root process splits its array of 4 elements and sends 1 element to each process.
- Each process doubles its value.
- The root process gathers the modified values into the result array.

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

Objective:

- Demonstrate how to use:
  - MPI_Reduce: Collective operation that sends the result to the root process only.
  - MPI_Allreduce: Performs the same reduction operation but shares the result with all processes.
- Operations used: MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD.

Code:

Each process has a local value (e.g., rank + 1) and applies the operations collectively.

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

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

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

   val = rank + 1; // Each process has value: 1, 2, ..., n
   printf("Process %d has value %d\n", rank, val);

   // MPI_Reduce: result only at root
   MPI_Reduce(&val, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   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, &prod, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);

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

    // MPI_Allreduce: result available to all processes
    MPI_Allreduce(&val, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&val, &max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&val, &min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    MPI_Allreduce(&val, &prod, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);

    printf("\nProcess %d - [Using MPI_Allreduce]: Sum=%d Max=%d Min=%d
Product=%d\n",
            rank, sum, max, min, prod);

    MPI_Finalize();
    return 0;
}
```

Sample Output (4 Processes):

```
$ mpirun -np 4 ./mpi_reduce_allreduce
Process 0 has value 1
Process 1 has value 2
Process 2 has value 3
Process 3 has value 4
```

[Using MPI_Reduce at Root Process]

Sum = 10

Max = 4

Min = 1

Product = 24

Process 0 - [Using MPI_Allreduce]: Sum=10 Max=4 Min=1 Product=24

Process 1 - [Using MPI_Allreduce]: Sum=10 Max=4 Min=1 Product=24

Process 2 - [Using MPI_Allreduce]: Sum=10 Max=4 Min=1 Product=24

Process 3 - [Using MPI_Allreduce]: Sum=10 Max=4 Min=1 Product=24

Notes:

- MPI_Reduce sends result only to root (rank 0).
- MPI_Allreduce makes the result available to all processes.
- Product of values 1×2×3×4 = 24.