VISVESVARAYA TECHNOLOGICAL UNIVERSITY
BELAGAVI-590 018,
KARNATAKA

PARALLEL COMPUTING LABORATORY

Subject Code: BCS702

Scheme: 2022 **Semester: VII**

Prepared by

Dr. Anitha G
Professor

Prof. Hemashree H C
Assistant Professor

Prof. Kamala R
Assistant Professor

Department of Information Science & Engineering
Bapuji Institute of Engineering and Technology,
Davangere-04.

Vision and Mission of the Institute

Vision

"To be **center of excellence** recognized **nationally** and **internationally**, in distinctive areas of **engineering education** and **research**, based on a culture of **innovation** and **invention**."

Mission

"BIET contributes to the **growth** and **development** of its students by imparting a board based **engineering education** and **empowering** them to be successful in their chosen field by inculcating in them **positive approach**, **leadership qualities** and **ethical values**."

Vision and Mission of the Department

Vision

"To be the **center of excellence** by adopting technological **innovation** in **academics** and **research** to develop competent **man power** for emerging needs of **society** and i**ndustries."**

Mission

M1: To provide **quality education** to meet the challenges of technological changes to succeed in their **professional career** and **higher education.**

M2: To inculcate the culture of **research, innovation** and **entrepreneur skills** among the students**.**

M3: To groom our students with the quality of **team spirit, leadership skills** and **ethical values**, to share and apply their knowledge for the **benefit of the society.**

Program Outcomes (POs) defined by NBA:

PO1	**Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem Analysis:** Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of Solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety and the cultural, societal and environmental considerations.
PO4	**Conduct Investigations of Complex Problems:** Use research-based knowledge and research Methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern Tool Usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
PO6	**The Engineer and Society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7	**Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.
PO9	**Individual and Team Work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long Learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives (PEOs):

PEO1- The graduates of program will have **excellence** through **principles** and practices of **information technology** combined with **fundamentals of engineering.**

PEO2 - The graduates of program will be prepared in **diverse areas** of **information science** for their **successful careers**, **entrepreneurship** and **higher studies**.

PEO3 - The graduates of program will **work effectively** as an **individual** and in a **team,** exhibiting **leadership qualities**, **communication skills** to meet the **goals** of the **organization.**

PEO4 - The graduates of program will grove their **profession** with **ethics,** m**anagement principles** to carry **societal responsibilities.**

Program Specific outcomes (PSOs):

PSO1- Problem Solving Skills - Ability to apply **standard principles** and **practices** of Information Technology to propose **feasible ideas** and **solutions** to computational tasks using **appropriate tools** and **techniques.**

PSO2- Knowledge of Information Technology – analyze, design, develop and **test** the computer based software in the areas related to **networks, cloud computing, web technology, data science and IoT.**

PSO3- Profession and Research Ability – Inculcate the knowledge to **excel** in **IT profession**, **entrepreneurship** and **research** with **ethical standards**.

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		

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)

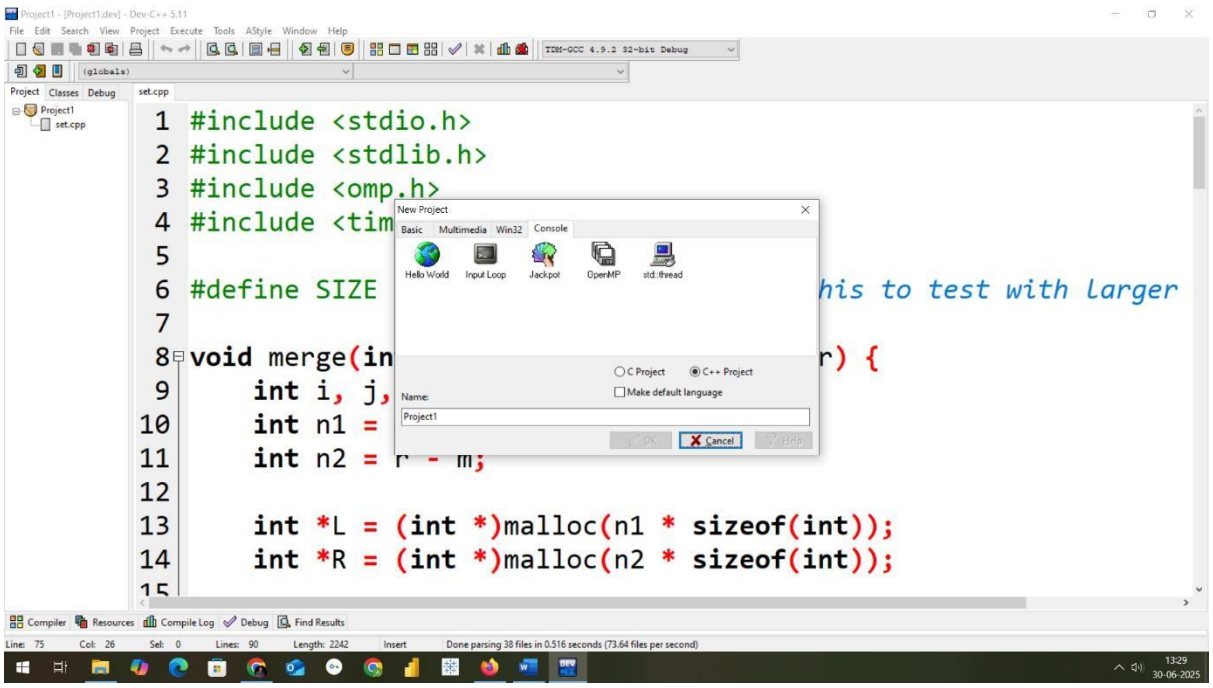
1. **Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort(using Section). Record the difference in execution time.**

STEP 1:

Open Dev C++

STEP 2:

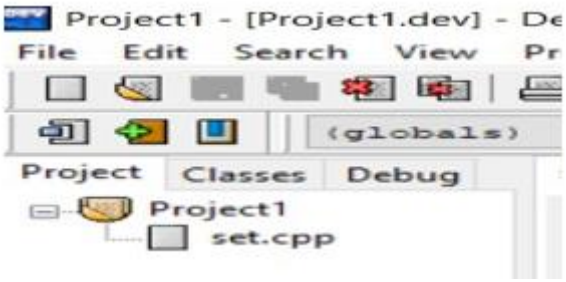
Click on File → Project → Select Console → OpenMP



Select Project as C++ and press OK

STEP 3:

After Project is created under project create set.cpp file and type program



Or use the below steps:

Create a new source file

- Go to **File → New → Source File**.
- Paste the OpenMP mergesort code

Save the file

- Save as mergesort_omp.c (use any file name of your convenient) (or .cpp, either works for C code in Dev-C++).

Enable OpenMP in compiler settings

By default, Dev-C++ uses the -std=c++14 (for C++) or -std=c11 (for C) options, but **OpenMP requires -fopenmp**.

1. Go to **Tools → Compiler Options…**
2. Tick **"Add the following commands when calling the compiler"**.
3. In the box, type: **-std=c99 -fopenmp**

Compile and Run

- Click **Execute → Compile & Run** (or press F11).
- If you saved as mergesort_omp.c, it will compile as C code.

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


void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    int i, j, k;

    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) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
```

```
            arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];


    free(L);
    free(R);
}


void sequential_mergesort(int arr[], int l, int r) {

    if (l < r) {
        int m = (l + r) / 2;
        sequential_mergesort(arr, l, m);
        sequential_mergesort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}


void parallel_mergesort(int arr[], int l, int r, int depth) {

    if (l < r) {
        int m = (l + r) / 2;

        if (depth <= 0) {
            sequential_mergesort(arr, l, m);
            sequential_mergesort(arr, m + 1, r);
        }
else {
            #pragma omp parallel sections
            {
                #pragma omp section
                parallel_mergesort(arr, l, m, depth - 1);

                #pragma omp section
                parallel_mergesort(arr, m + 1, r, depth - 1);
            }
        }


      merge(arr, l, m, r);
    }
}


int is_sorted(int arr[], int n) {
    int i;
    for (i = 1; i < n; i++) {
```

```
        if (arr[i - 1] > arr[i]) return 0;
    }
    return 1;
}

void print_array(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n, i;
    printf("Enter array size: ");
    scanf("%d", &n);

    int *arr1 = (int *)malloc(n * sizeof(int));
    int *arr2 = (int *)malloc(n * sizeof(int));

    srand(time(NULL)); // seed random numbers



    for (i = 0; i < n; i++) {
        arr1[i] = rand() % 100000; // random number 0–99999
        arr2[i] = arr1[i];
    }

    if (n <= 50) { // only print if small
        printf("\nOriginal array:\n");
        print_array(arr1, n);
    }

    double start, end;
    // Sequential mergesort
    start = omp_get_wtime();
    sequential_mergesort(arr1, 0, n - 1);
    end = omp_get_wtime();
    double seq_time = end - start;

    // Parallel mergesort
    start = omp_get_wtime();
```

```
    parallel_mergesort(arr2, 0, n - 1, 4);
    end = omp_get_wtime();
    double par_time = end - start;


    if (n <= 50) {
        printf("\nSorted array (Sequential):\n");
        print_array(arr1, n);
    }

    printf("\nArray size: %d\n", n);
    printf("Sequential mergesort time: %f seconds\n", seq_time);
    printf("Parallel mergesort time:   %f seconds\n", par_time);
    printf("Speedup: %.2fX\n", seq_time / par_time);


    if (is_sorted(arr1, n) && is_sorted(arr2, n)) {
        printf(" Both arrays are sorted correctly.\n");
    } else {
        printf("Sorting error occurred.\n");
    }

    free(arr1);
    free(arr2);
    return 0;
}
```

Step 4:

Goto Execute Menu and click on Compile and Run

Compile and Execution steps in DevC++

Step 1: Save and Compile the Program as **mergesort_omp.c**

Then compile it **using OpenMP support: gcc mergesort_omp.c -o mergesort_omp -fopenmp**

Step 2: Run the Program as **./mergesort_omp**

Then a prompt will get Enter array size:

OUTPUT 1:

Enter array size: 10

Original array:

72345 9821 67543 41234 56789 1023 29876 6543 87412 39874

Sorted array (Sequential):

1023 6543 9821 29876 39874 41234 56789 67543 72345 87412

Array size: 10

Sequential mergesort time: 0.000092 seconds

Parallel mergesort time: 0.000067 seconds

Speedup: 1.37X

Both arrays are sorted correctly.

OUTPUT 2:

Original array:

12345 876 4321 9876 3456 789 6543 2345 987 1234

Sorted array (Sequential):

789 876 987 1234 2345 3456 4321 6543 9876 12345

Array size: 10

Sequential mergesort time: 0.000081 seconds

Parallel mergesort time: 0.000057 seconds

Speedup: 1.42X

Both arrays are sorted correctly.

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

   int thread_start[100], thread_end[100];
   int i;

   for (i = 0; i < 100; i++) {
      thread_start[i] = -1;
      thread_end[i] = -1;
   }

   #pragma omp parallel for schedule(static,2)

   for (i = 0; i < n; i++) {
      int tid = omp_get_thread_num();

      if (thread_start[tid] == -1)
         thread_start[tid] = i;
      thread_end[tid] = i;
   }



   for (i = 0; i < 100; i++) {
      if (thread_start[i] != -1) {
         printf("Thread %d : Iterations %d -- %d\n", i, thread_start[i], thread_end[i]);

      }
   }

   return 0;
}
```

Step 1: Save and Compile the program as **omp_schedule_static2.c**

Then compile it **with OpenMP support: gcc omp_schedule_static2.c -o omp_schedule_static2 -fopenmp**

Step 2: Run the Program as ./omp_schedule_static2

Then a prompt will get Enter number of iterations: 8

Thread 0: Iterations 0 -- 1

Thread 1: Iterations 2 -- 3

Thread 2: Iterations 4 -- 5

Thread 3: Iterations 6 – 7

3. Write a OpenMP program to calculate n Fibonacci numbers using 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 number of Fibonacci terms: ");
   scanf("%d", &n);

   printf("Fibonacci Series:\n");

   for (int i = 0; i < n; i++) {
      int result;

      #pragma omp parallel
      {
         #pragma omp single
         {
            result = fib(i);
         }
      }

      printf("%d ", result);
   }

   printf("\n");
   return 0;
}
```

Step 1: Save and Compile the Program as **fib_task_omp.c**

Then compile it **with OpenMP support**: **gcc fib_task_omp.c -o fib_task_omp -fopenmp**

Step 2: Run the Program: ./fib_task_omp

OUTPUT1: Enter number of Fibonacci terms: 6

Fibonacci Series:

0 1 1 2 3 5

OUTPUT2: Enter number of Fibonacci terms: 10

Fibonacci Series:

0 1 1 2 3 5 8 13 21 34

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

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

int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);

    printf("\nPrime numbers from 1 to %d:\n", n);
    for (int i = 1; i <= n; i++) {
        if (is_prime(i))
            printf("%d ", i);
    }
    printf("\n");

    double start_time, end_time;

    start_time = omp_get_wtime();
    for (int i = 1; i <= n; i++) {
        is_prime(i);
    }
    end_time = omp_get_wtime();
    printf("Serial Time: %f seconds\n", end_time - start_time);

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

    return 0;
}
```

Step 1: Save and Compile the Program as **prime_parallel_omp.c**

Then compile it **with OpenMP support**:

gcc prime_parallel_omp.c -o prime_parallel_omp -fopenmp -lm

[-lm links the math library because of sqrt()]

Step 2: Run the Program as **./prime_parallel_omp**

OUTPUT1: Enter the value of n: 20

Prime numbers from 1 to 20:

2 3 5 7 11 13 17 19

Serial Time: 0.000020 seconds

Parallel Time: 0.000010 seconds

OUTPUT2: Enter the value of n: 100000

Prime numbers from 1 to 100000:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 ... (continues)

Serial Time: 0.482153 seconds

Parallel Time: 0.156987 seconds

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

Step 1: Open terminal

Type: nano mpi_send_recv.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);              // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get process rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get total number of processes
    if (size < 2) {
        printf("This program requires at least 2 processes.\n");
        MPI_Finalize();
        return 1;
    }
    if (rank == 0) {
        // Process 0 sends a message to process 1
        char msg[] = "Hello from Process 0";
        MPI_Send(msg, sizeof(msg), MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent message: %s\n", msg);
    }
    else if (rank == 1) {
        // Process 1 receives the message from process 0
        char recv_msg[100];
        MPI_Recv(recv_msg, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process 1 received message: %s\n", recv_msg);
    }
    MPI_Finalize(); // Finalize MPI
    return 0;
}
```

Step 1: Save the Program as **mpi_send_recv.c**

Step 2: Compile using **mpi_send_recv.c -o mpi_send_recv**

If it compiles successfully, you'll get an executable file named: mpi_send_recv

Step 3: Execute the Program with Multiple Processes

OUTPUT1: with 2 processes: mpirun –oversubscribe –np 2 ./mpi_send_recv

Process 0 sent message: Hello from Process 0

Process 1 received message: Hello from Process 0

OUTPUT2: with 4 processes: mpirun –oversubscribe –np 4 ./mpi_send_recv

Process 0 sent message: Hello from Process 0

Process 1 received message: Hello from Process 0

This program requires at least 2 processes.

This program requires at least 2 processes.

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

Step 1: Open terminal

Type nano deadlock.c

```
include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
int rank, size;
int send_data, recv_data;
MPI_Status status;
int avoid_deadlock = 0;  // Flag: 0 = deadlock demo, 1 = avoid deadlock

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size != 2) {
   if (rank == 0)
      printf("Please run this program with exactly 2 processes.\n");
   MPI_Finalize();
   return 0;
}
// Optional: set flag from command line argument
if (argc > 1) {
   avoid_deadlock = atoi(argv[1]);
}
send_data = rank;  // Just send rank as data
if (!avoid_deadlock) {
   // Demonstration of Deadlock: both send first, then receive
   if (rank == 0) {
      printf("[Deadlock] Process 0 sending to 1\n");
      MPI_Send(&send_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

```
            printf("[Deadlock] Process 0 waiting to receive from 1\n");
            MPI_Recv(&recv_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
            printf("[Deadlock] Process 0 received %d from process 1\n", recv_data);
        } else if (rank == 1) {
            printf("[Deadlock] Process 1 sending to 0\n");
            MPI_Send(&send_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
            printf("[Deadlock] Process 1 waiting to receive from 0\n");
            MPI_Recv(&recv_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            printf("[Deadlock] Process 1 received %d from process 0\n", recv_data);
        }
    } else {
        // Deadlock Avoidance: order of send/recv changed
        if (rank == 0) {
            printf("[Avoidance] Process 0 sending to 1\n");
            MPI_Send(&send_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
            printf("[Avoidance] Process 0 waiting to receive from 1\n");
            MPI_Recv(&recv_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
            printf("[Avoidance] Process 0 received %d from process 1\n", recv_data);
        } else if (rank == 1) {
            printf("[Avoidance] Process 1 waiting to receive from 0\n");
            MPI_Recv(&recv_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            printf("[Avoidance] Process 1 received %d from process 0\n", recv_data);
            printf("[Avoidance] Process 1 sending to 0\n");
            MPI_Send(&send_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    return 0;
}
```

1. Compile the program

Save the file as deadlock.c, then compile it using: **mpicc deadlock.c -o deadlock**

2. Run the program with 2 processes

Run using mpirun or mpiexec (depending on MPI installation):

(a) To **demonstrate deadlock: mpirun –oversubscribe --np 2 ./deadlock 0**
or simply: **mpirun –oversubscribe –np 2 ./deadlock**

Expected output is:

- Both processes (rank 0 and rank 1) will **try to send first** using MPI_Send.
- Since both are waiting for each other's Recv, the program will **hang (deadlock)**.
- Partial output will be like this:

  [Deadlock] Process 0 sending to 1

  [Deadlock] Process 1 sending to 0

(b) **To avoid deadlock**

Now, set the flag to 1 when running: **mpirun –oversubscribe –np 2 ./deadlock 1**

Expected output (no hang):

[Avoidance] Process 0 sending to 1

[Avoidance] Process 1 waiting to receive from 0

[Avoidance] Process 1 received 0 from process 0

[Avoidance] Process 1 sending to 0

[Avoidance] Process 0 waiting to receive from 1

[Avoidance] Process 0 received 1 from process 1

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

Type nano broadcast.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
   int rank, size;
   int data;

   // Initialize the MPI environment
   MPI_Init(&argc, &argv);

   // Get the rank (process ID) and size (total number of processes)
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   if (rank == 0) {
      // Process 0 (root) initializes the data
      data = 100;
      printf("Process %d is broadcasting data = %d\n", rank, data);
   }
   // Broadcast data from root (rank 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);

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

}
```

Step 1: Compile: mpicc broadcast.c -o broadcast

Step 2: Run with multiple processes: mpirun -oversubscribe --np 4 ./broadcast

Output:

Process 0 is broadcasting data = 100

Process 0 received data = 100

Process 1 received data = 100

Process 2 received data = 100

Process 3 received data = 100

8. Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

Type nano scgat

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
   int rank, size;
   int send_data[100];  // Array to hold data in root process
   int recv_data;       // Variable to receive scattered data
   int gathered_data[100]; // Array to gather results back to root
   // Initialize the MPI environment
   MPI_Init(&argc, &argv);

   // Get the rank and size
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   // Root process initializes the data
   if (rank == 0) {
      printf("Root process initializing data...\n");
      for (int i = 0; i < size; i++) {
         send_data[i] = i + 1; // Fill with values 1, 2, 3, ...
      }
   }

   // Scatter data from root to all processes
   MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

   // Each process prints the received value
   printf("Process %d received value %d from root.\n", rank, recv_data);
   // Each process performs some computation (for demo)
   recv_data = recv_data * 2;
```

```
    // Gather data from all processes back to root
    MPI_Gather(&recv_data, 1, MPI_INT, gathered_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Root process prints the gathered results
    if (rank == 0) {
        printf("\nRoot process gathered the following results:\n");
        for (int i = 0; i < size; i++) {
            printf("From process %d: %d\n", i, gathered_data[i]);
        }
    }
    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Step 1: Compile: mpicc scgat.c -o scgat

Step 2: Run with multiple processes: mpirun -oversubscribe --np 4 ./scgat

Output:

Root process initializing data...

Process 0 received value 1 from root.

Process 1 received value 2 from root.

Process 2 received value 3 from root.

Process 3 received value 4 from root.

Root process gathered the following results:

From process 0: 2

From process 1: 4

From process 2: 6

From process 3: 8

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

Type nano reallred

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

int main(int argc, char* argv[]) {
int rank, size;
int value;
int result;
// Initialize MPI
MPI_Init(&argc, &argv);
// Get rank and size
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
// Each process initializes its own value
value = rank + 1; // e.g., 1, 2, 3, 4, ...
printf("Process %d has value = %d\n", rank, value);

// 1. MPI_Reduce
   // Sum
MPI_Reduce(&value, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0)
   printf("\n[Using MPI_Reduce] SUM = %d\n", result);
// Product
MPI_Reduce(&value, &result, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
if (rank == 0)
   printf("[Using MPI_Reduce] PRODUCT = %d\n", result);
// Maximum
MPI_Reduce(&value, &result, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
if (rank == 0)
```

```
        printf("[Using MPI_Reduce] MAX = %d\n", result);

    // Minimum
    MPI_Reduce(&value, &result, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
    if (rank == 0)
        printf("[Using MPI_Reduce] MIN = %d\n", result);

    // 2. MPI_Allreduce
    int all_result;
    // Sum
    MPI_Allreduce(&value, &all_result, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    printf("\nProcess %d: [Using MPI_Allreduce] SUM = %d\n", rank, all_result);
    // Product
    MPI_Allreduce(&value, &all_result, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
    printf("Process %d: [Using MPI_Allreduce] PRODUCT = %d\n", rank, all_result);
    // Maximum
    MPI_Allreduce(&value, &all_result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    printf("Process %d: [Using MPI_Allreduce] MAX = %d\n", rank, all_result);
    // Minimum
    MPI_Allreduce(&value, &all_result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    printf("Process %d: [Using MPI_Allreduce] MIN = %d\n", rank, all_result);
    // Finalize MPI
    MPI_Finalize();
    return 0;
}
```

Step 1: Compile: mpicc reallred.c -o reallred

Step 2: Run with multiple processes: mpirun -oversubscribe --np 4 ./ reallred

Output

Process 0 has value = 1

Process 1 has value = 2

Process 2 has value = 3

Process 3 has value = 4

[Using MPI_Reduce] SUM = 10

[Using MPI_Reduce] PRODUCT = 24

[Using MPI_Reduce] MAX = 4

[Using MPI_Reduce] MIN = 1

Process 0: [Using MPI_Allreduce] SUM = 10

Process 1: [Using MPI_Allreduce] SUM = 10

Process 2: [Using MPI_Allreduce] SUM = 10

Process 3: [Using MPI_Allreduce] SUM = 10

Process 0: [Using MPI_Allreduce] PRODUCT = 24

Process 1: [Using MPI_Allreduce] PRODUCT = 24

Process 2: [Using MPI_Allreduce] PRODUCT = 24

Process 3: [Using MPI_Allreduce] PRODUCT = 24

Process 0: [Using MPI_Allreduce] MAX = 4

Process 1: [Using MPI_Allreduce] MAX = 4

Process 2: [Using MPI_Allreduce] MAX = 4

Process 3: [Using MPI_Allreduce] MAX = 4

Process 0: [Using MPI_Allreduce] MIN = 1

Process 1: [Using MPI_Allreduce] MIN = 1

Process 2: [Using MPI_Allreduce] MIN = 1

Process 3: [Using MPI_Allreduce] MIN = 1