PARALLEL COMPUTING

SUB CODE :-
BCS702

Lab Manual for
7th Sem

AMC
EDUCATION

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“Jnana Sangama”, Belagavi-590014, Karnataka, India

Lab Manual
On

“PARALLEL COMPUTING”
BCS702

VII Semester of
Bachelor of Engineering in Computer Science and Engineering of
Visvesvaraya Technological University, Belagavi

Prof. Snigdha Kesh

Assistant Professor
Dept. of Computer Science and Engineering
AMC Engineering College

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AMC ENGINEERING COLLEGE

18th K.M, Bannerghatta Main Road, Bangalore-560083

2024-2025

Vision of the Institute

"To be a leader in imparting value based Technical Education and Research for the benefit of society".

Mission of the Institute

M1	To provide state of the art Infrastructure facilities
M2	To implement modern pedagogical methods in delivering the academic programs with experienced and committed faculties
M3	To create a vibrant ambience that promotes Learning, Research, Invention and Innovations
M4	To undertake manpower and skill development programmers for Academic Institutions and Industries
M5	To enhance Institute Industry Interface through Collaborative Research and Consultancy
M6	To generate and disseminate knowledge through training program workshops, seminars, conferences, publications
M7	To be a more comprehensive college in terms of the number of programs offered
M8	To relentlessly pursue professional excellence with ethical and moral values

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

"To develop the department as a center of excellence in the area of Information Science and Engineering for the benefit of society".

Mission of the Department

MD1	To provide the State-of-the-Art Infrastructure and Technology in the field of Information Science and Engineering.
MD2	To deliver value based education through modern teaching pedagogy.
MD3	To impart theoretical, computational and practical knowledge in the area of Information Technology.
MD4	To collaborate with Institute, Industry and Research Organizations for the Cutting Edge Technologies.
MD5	To develop an Entrepreneurial, Ethical and Socially responsible professionals.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO 1	Graduates possess advanced knowledge of Computer Science & Engineering and excel in leadership roles to serve the society
PEO 2	Graduates of the program will apply Computer Engineering tools in core technologies for improving knowledge in the Interdisciplinary Research and/or Entrepreneurs
PEO 3	Graduates adapt Value-Based Proficiency in solving real time problems.

PROGRAMME SPECIFIC OUTCOMES

PSO 1	Professional Skills: Ability of using mathematical methodologies for analysis of computing concepts, data structure, computer hardware, layered technologies and suitable algorithm which in turn helps graduates to model, design and implement a system to meet specific requirement
PSO 2	Software Skills: Ability to build Software Engineering System for lifecycle development by using analytical knowledge in Computer Science & Engineering and applying modern methodologies

PROGRAM OUTCOMES (POs)

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, review 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 the 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 modeling 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 the 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's 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 LIST

Sl. NO.	**Program Description**
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 Outcomes: At the end of this course, students are able to:
CO1 - Explain the need for parallel programming
CO2 - Demonstrate parallelism in MIMD system
CO3 - Apply MPI library to parallelize the code to solve the given problem.
CO4 - Apply OpenMP pragma and directives to parallelize the code to solve the given problem
CO5 - Design a CUDA program for the given problem

COs and POs Mapping of lab Component

COURSE OUTCOMES	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	-	-	-	-	-	-	-	-	-	2	2	-	-
CO2	3	3	2	2	2	-	-	-	-	-	-	1	3	2	-
CO3	3	3	3	-	3	-	-	-	1	-	2	2	3	3	2
CO4	3	3	3	-	3	-	-	-	1	-	2	1	3	3	2
CO5	3	3	3	-	3	-	-	-	1	-		2	3	2	3

CO-PO Mapping Justifications:

CO1 - Understanding the fundamentals of parallel programming equips students to analyze the limitations of sequential processing. It also promotes lifelong learning as they explore the evolution and relevance of parallel computing models.

CO2 - Students gain insight into multi-core architecture by analyzing and implementing MIMD-based parallelism. This helps them apply theoretical knowledge to real-world hardware configurations and improve performance understanding

CO3 - Using MPI, students learn to structure and implement distributed applications that communicate through message passing. It strengthens their ability to design scalable solutions in distributed-memory environments.

CO4 - OpenMP allows students to parallelize code for shared-memory systems using compiler directives, improving execution speed. This fosters skills in identifying parallel sections and managing synchronization efficiently.

CO5 - Students apply CUDA to solve data-parallel problems on GPUs, learning to optimize memory and thread usage. This enhances their understanding of heterogeneous computing and modern accelerator-based systems.

LAB EVALUATION PROCESS

Component	**Details**	**Max. Marks**	**Scaling**	**Final CIE Marks**
Experiment Conduction & Laboratory Record	Continuous evaluation of each experiment report (including viva-voce). Each report = 10 marks. All reports added and scaled down.	Varies (per experiment)	Scaled down to **15**	15
Laboratory Test	Final lab test after completion of all experiments (Duration: 2–3 hours). Conducted for **50 marks**.	50	Scaled down to **10**	10

Rubric for Experiment Conduction & Laboratory Record (15 Marks)

Component	Excellent (Full Marks)	Good	Satisfactory	Needs Improvement
Write-up (3 Marks)	Well-structured, neat, complete with aim, procedure, circuit/algorithm, results.	Minor omissions, mostly neat and complete.	Incomplete, lacks clarity in some parts.	Missing/very poor write-up.
Execution (5 Marks)	Executes experiment correctly, minimal guidance needed, accurate results.	Minor errors, completes with little help.	Frequent guidance required, partial results.	Unable to perform independently, major errors.
Observation (4 Marks)	Accurate, systematic, neatly tabulated with proper units.	Mostly correct, minor inconsistencies.	Incomplete or some errors in data recording.	Observations missing/incorrect.
Viva (3 Marks)	Answers confidently with clear understanding of concepts.	Good understanding, few gaps.	Limited understanding, needs prompting.	Unable to answer, lacks conceptual clarity.

Rubric for Laboratory Test (10 Marks) (Scaled from 50)

Component	**Excellent** (Full Marks)	**Good**	**Satisfactory**	**Needs Improvement**	**Marks**
Write-up (2 Marks)	Clear, complete with aim, procedure/algorithm, neat presentation.	Mostly complete, minor omissions.	Incomplete, lacks clarity.	Very poor/absent.	0–2
Execution (4 Marks)	Implements experiment/program correctly, accurate results, minimal guidance.	Minor errors, mostly correct.	Partial results, frequent guidance needed.	Incorrect implementation, unable to execute.	0–4
Observation (2 Marks)	Accurate, systematic, neatly recorded with units/tables.	Mostly correct with minor mistakes.	Incomplete/partially correct.	Incorrect or missing.	0–2
Viva (2 Marks)	Answers confidently, demonstrates strong conceptual understanding.	Good understanding, few gaps.	Limited understanding, needs prompting.	Unable to answer.	0–2

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.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
// Function to merge two halves of the array
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 MergeSort
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 MergeSort using OpenMP sections
void mergeSortParallel(int* arr, int l, int r, int depth) {
        if (l < r) {
            int m = (l + r) / 2;
            if (depth <= 0) {
                // Fallback to sequential at depth limit
                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);
        }
}
// Helper to 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));

    // Seed for reproducibility
    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;
    // Validate and 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:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano openMp1.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ gcc -fopenmp openmp1.c -o mergesort
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ ./mergesort
Sequential sort time: 0.367992 seconds
Parallel sort time  : 0.196234 seconds
Speedup             : 1.88x
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

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

Code:

```
#include <stdio.h>
#include <omp.h>
int main() {
        int num_iterations;
        printf("Enter the number of iterations: ");
        scanf("%d", &num_iterations);
        // Optional: Set number of threads (or use OMP_NUM_THREADS)
        // omp_set_num_threads(2);
```

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

Output:

First Way

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano omp_static.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ gcc -fopenmp omp_static.c -o omp_static
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ ./omp_static
Enter the number of iterations: 5

Using schedule(static,2):

Thread 0 : Iteration 0
Thread 2 : Iteration 4
Thread 0 : Iteration 1
Thread 1 : Iteration 2
Thread 1 : Iteration 3
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

Second Way

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ export OMP_NUM_THREADS=2
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ gcc -fopenmp omp_static.c -o omp_static
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ ./omp_static
Enter the number of iterations: 5

Using schedule(static,2):

Thread 0 : Iteration 0
Thread 0 : Iteration 1
Thread 0 : Iteration 4
Thread 1 : Iteration 2
Thread 1 : Iteration 3
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

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

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
// Recursive Fibonacci using OpenMP tasks
int fib(int n) {
        int x, y;
        if (n <= 1) 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 the number of Fibonacci numbers to calculate: ");
        scanf("%d", &n);
        if (n <= 0) {
                printf("Please enter a positive integer.\n");
        return 0;
        }
        printf("First %d Fibonacci numbers using OpenMP tasks:\n", n);
        double start = omp_get_wtime();
        #pragma omp parallel
        {
                #pragma omp single
```

```
        {
            for (int i = 0; i < n; i++) {
                int result;
                #pragma omp task shared(result)
                {
                    result = fib(i);
                    #pragma omp critical
                    printf("Fib(%d) = %d\n", i, result);

                }
            }
        }
    }
    double end = omp_get_wtime();
    printf("Execution time: %.6f seconds\n", end - start);
    return 0;
}
```

Output:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano fibonacci.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ gcc -fopenmp fibonacci.c -o fibonacci
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ ./fibonacci
Enter the number of Fibonacci numbers to calculate: 12
First 12 Fibonacci numbers using OpenMP tasks:
Fib(0) = 0
Fib(1) = 1
Fib(3) = 2
Fib(4) = 3
Fib(5) = 5
Fib(6) = 8
Fib(7) = 13
Fib(2) = 1
Fib(8) = 21
Fib(9) = 34
Fib(10) = 55
Fib(11) = 89
Execution time: 0.000433 seconds
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

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.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
// Check if a number is prime
int is_prime(int num) {
        if (num <= 1) return 0;
        if (num == 2) return 1;
        if (num % 2 == 0) return 0;
        int limit = (int)sqrt(num);
        for (int i = 3; i <= limit; i += 2) {
                if (num % i == 0) return 0;
        }
        return 1;
}
int main() {
        int n;
        printf("Enter the upper limit (n): ");
        scanf("%d", &n);
        if (n < 2) {
                printf("There are no prime numbers <= %d\n", n);
        return 0;
        }
        // SERIAL VERSION
        double start_serial = omp_get_wtime();
        int* primes_serial = (int*)malloc((n + 1) * sizeof(int));
```

```
int count_serial = 0;
for (int i = 2; i <= n; i++) {
        if (is_prime(i)) {
                primes_serial[count_serial++] = i;
        }
}
double end_serial = omp_get_wtime();
double time_serial = end_serial - start_serial;
// PARALLEL VERSION
double start_parallel = omp_get_wtime();
int* primes_parallel = (int*)malloc((n + 1) * sizeof(int));
int count_parallel = 0;
#pragma omp parallel
{
int* local_primes = (int*)malloc((n / omp_get_num_threads() + 1) * sizeof(int));
int local_count = 0;
#pragma omp for nowait
for (int i = 2; i <= n; i++) {
                if (is_prime(i)) {
                        local_primes[local_count++] = i;
                }
}
 // Combine local results into global array (critical section)
  #pragma omp critical
{
        for (int i = 0; i < local_count; i++) {
                 primes_parallel[count_parallel++] = local_primes[i];
                }
}
```

```
        free(local_primes);
    }
    double end_parallel = omp_get_wtime();
    double time_parallel = end_parallel - start_parallel;
printf("\nNumber of primes found: %d\n", count_serial);
printf("Serial execution time  : %.6f seconds\n", time_serial);
printf("Parallel execution time: %.6f seconds\n", time_parallel);
printf("Speedup: %.2fx\n", time_serial / time_parallel);
free(primes_serial);
free(primes_parallel);
return 0;
}
```

Output:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano prime_number.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ gcc -fopenmp prime_number.c -o prime_number -lm
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ ./prime_number
Enter the upper limit (n): 10000

Number of primes found: 1229
Serial execution time  : 0.000339 seconds
Parallel execution time: 0.001263 seconds
Speedup: 0.27x
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ export OMP_NUM_THREADS=4
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ ./prime_number
Enter the upper limit (n): 100000

Number of primes found: 9592
Serial execution time  : 0.005297 seconds
Parallel execution time: 0.002355 seconds
Speedup: 2.25x
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ ./prime_number
Enter the upper limit (n): 1000000

Number of primes found: 78498
Serial execution time  : 0.120020 seconds
Parallel execution time: 0.041399 seconds
Speedup: 2.90x
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

Explanation:

- -fopenmp → enables OpenMP parallelization.
- -lm → links the math library (needed for sqrt()).
- -o prime_number → names the output executable.

Control number of threads: You can change how many threads OpenMP uses

- export OMP_NUM_THREADS=4
- . /prime_number

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

Before Executing Check MPI version is install or not

- **mpicc --version**

If not installed, install OpenMPI:

- **sudo apt update**
- **sudo apt install -y openmpi-bin openmpi-common libopenmpi-dev**

Code:

```
#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 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; // Example message
        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:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ sudo apt update
Hit:1 http://security.ubuntu.com/ubuntu bionic-security InRelease
Hit:2 http://archive.ubuntu.com/ubuntu bionic InRelease
Hit:3 http://archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:4 http://archive.ubuntu.com/ubuntu bionic-backports InRelease
Reading package lists... Done
Building dependency tree
Reading state information... Done
23 packages can be upgraded. Run 'apt list --upgradable' to see them.
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ sudo apt install -y openmpi-bin openmpi-common libopenmpi-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
libopenmpi-dev is already the newest version (2.1.1-8).
openmpi-bin is already the newest version (2.1.1-8).
openmpi-common is already the newest version (2.1.1-8).
The following packages were automatically installed and are no longer required:
  efibootmgr libefiboot1 libefivar1
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 23 not upgraded.
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpicc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano mpi.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpicc mpi.c -o mpi
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpirun -np 2 ./mpi
Process 0 sending number 100 to Process 1
Process 1 received number 100 from Process 0
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

Run with multiple processes

- **Use mpirun (or mpiexec): mpirun -np 2 ./mpi**
- `-np 2` → runs with **2 processes**.
- Each process runs the same program but executes different branches based on `rank`.

Note: If You Want to Run with more processes (more than 2) use: mpirun -np 4 ./mpi

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

Code:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int rank, size;
    int msg_send = 100, msg_recv;
    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("Run with at least 2 processes.\n");
        MPI_Finalize();
        return 0;
    }
    if (rank == 0) {
        printf("Process 0 sending to Process 1...\n");
        MPI_Send(&msg_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); // blocking send
        MPI_Recv(&msg_recv,1,MPI_INT,1,0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process 0 received from Process 1: %d\n", msg_recv);
    } else if (rank == 1) {
        printf("Process 1 sending to Process 0...\n");
        MPI_Send(&msg_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // blocking send
    MPI_Recv(&msg_recv,1,MPI_INT,0,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received from Process 0: %d\n", msg_recv);
    }
    MPI_Finalize();
```

```
    return 0;
}
```

Output:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano deadlock.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpicc deadlock.c -o
deadlock
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpirun -np 2 ./deadlock
Process 0 sending to Process 1...
Process 1 sending to Process 0...
Process 1 received from Process 0: 100
Process 0 received from Process 1: 100
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

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

Code:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int rank, size;
    int number;
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    // Get the rank and number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Process 0 gets the input
    if (rank == 0) {
        printf("Enter a number to broadcast: ");
        fflush(stdout); // Ensure prompt is printed before input
        scanf("%d", &number);
    }
```

```
    // Broadcast the number from process 0 to all other processes
    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // Each process prints the received number
    printf("Process %d received number: %d\n", rank, number);
    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Output:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano broadcast.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpicc broadcast.c -o broadcast
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpirun -np 4 ./broadcast
Enter a number to broadcast: 42
Process 1 received number: 42
Process 2 received number: 42
Process 3 received number: 42
Process 0 received number: 42
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

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

Code:

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

int main(int argc, char** argv) {
    int rank, size;
    int *send_data = NULL;  // only root will allocate
    int recv_data;

    MPI_Init(&argc, &argv);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Root process prepares data
if (rank == 0) {
    send_data = (int*)malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        send_data[i] = (i + 1) * 10;  // e.g., 10, 20, 30, ...
    }
}
// Scatter one element to each process
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);
// Modify received data
recv_data += 1;

// Gather results back at root
MPI_Gather(&recv_data, 1, MPI_INT, send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
// Root prints gathered results
if (rank == 0) {
    printf("Gathered data: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", send_data[i]);
    }
    printf("\n");
    free(send_data);
}
MPI_Finalize();
```

```
    return 0;
}
```

Output:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano scatter_gather.
c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpicc scatter_gather.c -o scatter_gather
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpirun -np 4 ./scatter_gather
Process 0 received: 10
Process 1 received: 20
Process 2 received: 30
Process 3 received: 40
Gathered data: 11 21 31 41
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

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

Code:

```
#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:

```
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ nano mpi_prg9.c
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpicc mpi_prg9.c -o mpi_prg9
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$ mpirun -np 4 ./mpi_prg9
Process 1 has value 2
Process 2 has value 3
Process 3 has value 4
Process 0 has value 1

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

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

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

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

[Process 0] MPI_Allreduce Results:
Sum=10
Prod=24
Max=4
Min=1
shhubham@DESKTOP-5OJQ75S:/mnt/c/Users/ASUS/Desktop/ns2$
```

Content Beyond Syllabus -BCS702

Program-1OpenMP Matrix Multiplication with Parallel Sections

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

#define N 3

int main() {
   int A[N][N] = {{1,2,3},{4,5,6},{7,8,9}};
   int B[N][N] = {{9,8,7},{6,5,4},{3,2,1}};
   int C[N][N];
   int i,j,k;

   double start = omp_get_wtime();

   #pragma omp parallel shared(A,B,C) private(i,j,k)
   {
      #pragma omp sections
      {
         #pragma omp section
         {
            for (i=0;i<N/2;i++) {
               for (j=0;j<N;j++) {
                  C[i][j] = 0;
                  for (k=0;k<N;k++)
                     C[i][j] += A[i][k]*B[k][j];
               }
            }
         }

         #pragma omp section
         {
            for (i=N/2;i<N;i++) {
               for (j=0;j<N;j++) {
                  C[i][j] = 0;
                  for (k=0;k<N;k++)
                     C[i][j] += A[i][k]*B[k][j];
               }
            }
         }
      }
   }
```

```
    double end = omp_get_wtime();

    printf("Resultant Matrix C = \n");
    for (i=0;i<N;i++) {
        for (j=0;j<N;j++)
            printf("%4d", C[i][j]);
        printf("\n");
    }

    printf("Parallel Execution Time: %f seconds\n", end-start);
    return 0;
}
```

OUTPUT

```
Resultant Matrix C =
  30  24  18
  84  69  54
 138 114  90
Parallel Execution Time: 0.0001 seconds
```

Program-2 MPI Ring Topology Communication

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

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

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

    send_data = rank;

    int next = (rank+1)%size;          // next process in ring
    int prev = (rank-1+size)%size;     // previous process in ring

    MPI_Sendrecv(&send_data,1,MPI_INT,next,0,
                 &recv_data,1,MPI_INT,prev,0,
                 MPI_COMM_WORLD,MPI_STATUS_IGNORE);

    printf("Process %d sent %d and received %d\n", rank, send_data, recv_data);

    MPI_Finalize();
    return 0;
}
```

OUTPUT

Process 0 sent 0 and received 3
Process 1 sent 1 and received 0
Process 2 sent 2 and received 1
Process 3 sent 3 and received 2

Viva Questions & Answers

1Q: What is the difference between OpenMP and MPI?
A: OpenMP is used for shared-memory parallelism (single machine with multiple cores), while MPI is used for distributed-memory parallelism (multiple machines/processes connected via network).

2Q: What is the purpose of the #pragma omp parallel directive?
A: It creates a team of threads that execute the enclosed block of code in parallel.

3Q: What does #pragma omp sections do?
A: It divides the program into independent sections, and each section is executed by a separate thread in parallel.

4Q: What is the use of OMP_SCHEDULE?
A: It controls how loop iterations are divided among threads (static, dynamic, guided). Example: static,2 assigns iterations in chunks of size 2 to threads.

5Q: What are OpenMP tasks and when are they used?
A: Tasks allow work units to be created and executed by any thread in the team. They are useful for irregular workloads like recursive Fibonacci calculation.

6Q: In MPI, what is the difference between MPI_Send/MPI_Recv and MPI_Bcast?
A: MPI_Send and MPI_Recv are point-to-point communication between two processes, while MPI_Bcast sends data from one root process to all other processes.

7Q: What is a deadlock in MPI, and how can it be avoided?
A: Deadlock occurs when processes wait indefinitely for each other (e.g., both calling MPI_Recv first). It can be avoided by changing the call sequence (send first, then receive) or using MPI_Sendrecv.

8Q: What is the significance of MPI_Reduce and MPI_Allreduce?
A: MPI_Reduce combines data from all processes to a root process (e.g., sum, min, max). MPI_Allreduce performs the same reduction but distributes the result to all processes.

9Q: What is ring topology communication in MPI?
A: It is a communication pattern where each process sends data to its next neighbor and receives from the previous neighbor, forming a logical ring.

10.Q: Why do we record both serial and parallel execution times?
A: To measure the speedup and efficiency of parallelization. It helps in comparing performance and understanding the benefits of parallel programming.