

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 570 028



ATME

College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(ACADEMIC YEAR 2026-27)

LABORATORY MANUAL

SUBJECT: PARALLEL COMPUTING

SUBJECT CODE: BCS702

SEMESTER: VII

Outcome Based Education (OBE) and Choice Based Credit System (CBCS)

(Effective from the academic year 2023-24)

Composed By

Verified By

Approved By

Mr. RAJIV P

Dr. PADMAJA K

Mrs. BINDUSHREE V

Dr. PUTTEGOWDA D

Foreman

Faculty Co-Ordinator

Professor & Head, CSE

INSTITUTIONAL MISSION AND VISION

Objectives

- To empower the younger generation with quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To promote basic & applied research in areas of social relevance, and share the knowledge for the advancement of mankind.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and international levels.
- To cultivate strong community relationships and involve the younger generation in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission.

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

Department of Computer Science & Engineering

Vision of the Department

- To develop highly talented individuals in Computer Science and Engineering to deal with real world challenges in industry, education, research and society.

Mission of the Department

- To inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities in the young minds & to provide a teaching environment that emphasizes depth, originality and critical thinking.
- Motivate students to put their thoughts and ideas adoptable by industry or to pursue higher studies leading to research.

Program outcomes (POs)

Engineering Graduates will be able to:

- **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 Educational Objectives (PEO'S):

PE01: Empower students with a strong basis in the mathematical, scientific and engineering fundamentals to solve computational problems and to prepare them for employment, higher learning and R&D.

PE02: Gain technical knowledge, skills and awareness of current technologies of computer science engineering and to develop an ability to design and provide novel engineering solutions for software/hardware problems through entrepreneurial skills.

PE03: Exposure to emerging technologies and work in teams on interdisciplinary projects with effective communication skills and leadership qualities.

PE04: Ability to function ethically and responsibly in a rapidly changing environment by Applying innovative ideas in the latest technology, to become effective professionals in Computer Science to bear a life-long career in related areas.

Program Specific Outcomes (PSOs)

PS01: Ability to apply skills in the field of algorithms, database design, web design, cloud computing and data analytics.

PS02: Apply knowledge in the field of computer networks for building network and internet based applications.

PARALLEL COMPUTING		Semester	VII
Course Code	BCS702	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory/Practical		
Course objectives: This course will enable to, • Explore the need for parallel programming • Explain how to parallelize on MIMD systems • To demonstrate how to apply MPI library and parallelize the suitable programs • To demonstrate how to apply OpenMP pragma and directives to parallelize the suitable programs • To demonstrate how to design CUDA program			
Teaching-Learning Process (General Instructions) These are sample Strategies that teachers can use to accelerate the attainment of the various course outcomes. 1. Lecturer method (L) need not to be only traditional lecture methods, but alternative effective teaching methods could be adopted to attain the outcomes. 2. Use of Video/Animation to explain functioning of various concepts. 3. Encourage collaborative (Group Learning) Learning in the class. 4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking. 5. Adopt Programming assignment, which fosters student's Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.			

PRACTICAL COMPONENT OF IPCC

Sl.NO	Experiments
1	Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort(using Section). Record the difference in execution time.
2	Write an OpenMP program that divides the Iterations into chunks containing 2 iterations, respectively (OMP_SCHEDULE=static,2). Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be the following: a. Thread 0 : Iterations 0 — 1 b. Thread 1 : Iterations 2 — 3
3	Write a OpenMP program to calculate n Fibonacci numbers using tasks.
4	Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.
5	Write a MPI Program to demonstration of MPI_Send and MPI_Recv.
6	Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence
7	Write a MPI Program to demonstration of Broadcast operation.
8	Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

9	Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)
Course outcomes (Course Skill Set): At the end of the course, the student will be able to: ● Explain the need for parallel programming ● Demonstrate parallelism in MIMD system. ● Apply MPI library to parallelize the code to solve the given problem. ● Apply OpenMP pragma and directives to parallelize the code to solve the given problem ● Design a CUDA program for the given problem.	
Assessment Details (both CIE and SEE) The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together. CIE for the theory component of the IPCC (maximum marks 50) ● IPCC means practical portion integrated with the theory of the course. ● CIE marks for the theory component are 25 marks and that for the practical component is 25 marks. ● 25 marks for the theory component are split into 15 marks for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and 10 marks for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus. ● Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for 25 marks). ● The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC. CIE for the practical component of the IPCC CIE for the practical component of the IPCC ● 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions. ● On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day. ● The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks. ● The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.	

- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (duration 03 hours)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Suggested Learning Resources:

Textbook:

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

Reference Books:

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

Web links and Video Lectures (e-Resources):

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

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

- Programming Assignment at higher bloom level (10

CONTENT LIST

SL.NO.	EXPERIMENT NAME	PAGE NO.
1	Introduction	1
2	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.	11
3	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	16
4	Write a OpenMP program to calculate n Fibonacci numbers using tasks.	20
5	Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.	24
6	Write a MPI Program to demonstration of MPI_Send and MPI_Recv.	28
7	Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence	32
8	Write a MPI Program to demonstration of Broadcast operation.	36
9	Write a MPI Program demonstration of MPI_Scatter and MPI_Gather	39
10	Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)	43
11	Viva Questions and answers	52

Introduction

Parallel Computing is a computing technique in which multiple processors or cores execute multiple tasks simultaneously to solve a problem faster than a single processor. Instead of performing one instruction at a time (serial computing), parallel computing divides a large problem into smaller sub-problems that can be processed concurrently.

With the increasing demand for high-performance computing in fields such as scientific research, artificial intelligence, data analytics, weather forecasting, and engineering simulations, parallel computing has become an essential technology. Modern computers, multicore processors, and distributed systems are designed to support parallel execution, improving performance, scalability, and resource utilization.

This laboratory course provides practical experience in designing, implementing, and analyzing parallel programs using programming models such as OpenMP, MPI, CUDA, or multithreading (depending on the syllabus). Students will learn how to identify parallelism in algorithms, implement parallel solutions, measure performance, and understand the challenges of synchronization, communication, and load balancing.

Objectives

- To understand the fundamentals of parallel computing.
- To learn different parallel programming models and architectures.
- To implement parallel algorithms using appropriate tools and libraries.
- To compare the performance of serial and parallel programs.
- To develop skills in optimizing programs for multicore and distributed computing environments.

Expected Outcomes

After completing this laboratory, students will be able to:

- Understand the concepts and importance of parallel computing.
- Write and execute parallel programs using suitable programming techniques.
- Analyze speedup, efficiency, and scalability of parallel algorithms.
- Apply parallel computing concepts to solve computationally intensive problems effectively.

What is OpenMP?

OpenMP (Open Multi-Processing) is an Application Programming Interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It allows programmers to create parallel applications using compiler directives, library functions, and environment variables.

Features of OpenMP

- Easy to learn and implement.
- Uses shared memory architecture.
- Supports C, C++, and Fortran.

- Portable across different operating systems and compilers.
- Incremental parallelization (convert serial code to parallel step by step).
- Automatically manages thread creation and synchronization.

Advantages of OpenMP

- Reduces program execution time.
- Improves CPU utilization.
- Simple programming model using compiler directives.
- Supports dynamic workload distribution.
- Suitable for multi-core processors.

Limitations of OpenMP

- Works only on shared-memory systems.
- Limited scalability for very large distributed systems.
- Thread synchronization may introduce overhead.
- Performance depends on the number of available CPU cores.

OpenMP Programming Model

- Fork-Join Model
 - The program starts with a single master thread.
 - When a parallel region is encountered, the master thread creates multiple worker threads (fork).
 - Each thread executes a portion of the work.
 - After completing the work, all threads synchronize and merge back into a single master thread (join).
-

Basic OpenMP Syntax (C)

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

int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Output (example):

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

Common OpenMP Directives

Directive	Purpose	
<code>#pragma omp parallel</code>	Creates parallel threads	
<code>#pragma omp for</code>	Parallelizes loop iterations	
<code>#pragma omp sections</code>	Divides different tasks among threads	
<code>#pragma omp single</code>	Only one thread executes the block	
<code>#pragma omp critical</code>	Allows only one thread at a time to execute a block	
<code>#pragma omp barrier</code>	Synchronizes all threads	

Applications of OpenMP

- Scientific computing
- Image and video processing
- Machine learning
- Weather forecasting
- Engineering simulations
- Data analysis
- Financial modeling

Comparison of Sequential and Parallel Execution Times

Feature	Sequential Execution	Parallel Execution
Execution Method	One task executes after another	Multiple tasks execute simultaneously
Number of Processors	Single processor/core	Multiple processors/cores
Execution Time	Higher	Lower
Speed	Slower	Faster
Resource Utilization	Limited	Efficient utilization of available processors
Scalability	Difficult for large problems	Easily scalable for large datasets
Fault Tolerance	Lower	Higher in distributed systems

Example

Suppose a program has 4 independent tasks, each requiring 5 seconds.

- Sequential Execution:
 - Task 1 $\rightarrow$ 5 s
 - Task 2 $\rightarrow$ 5 s
 - Task 3 $\rightarrow$ 5 s
 - Task 4 $\rightarrow$ 5 s
 - Total Execution Time = 20 seconds

- Parallel Execution (4 processors):
 - All four tasks execute simultaneously.
 - Total Execution Time ≈ 5 seconds

Comparison Diagram

Sequential Execution

Processor:

Task1 → Task2 → Task3 → Task4

5 s 5 s 5 s 5 s

Total Time = 20 seconds

Parallel Execution

Processor 1: Task1 → 5 s

Processor 2: Task2 → 5 s

Processor 3: Task3 → 5 s

Processor 4: Task4 → 5 s

Total Time ≈ 5 seconds

Advantages of Parallel Computing

- Reduces execution time for large computations.
- Improves overall system performance.
- Solves complex problems efficiently.
- Supports real-time data processing.
- Makes better use of multi-core processors.
- Enables scalable computing for large applications.

Difference Between Process and Thread

Process

A **Process** is an independent program in execution. It has its own memory space, resources, and system information.

Characteristics of a Process

- Has its own memory (address space)
- Heavyweight execution unit
- Creation takes more time
- Communication between processes is slower
- Failure of one process usually does not affect others

Example: Running a web browser, music player, and text editor simultaneously. Each is a separate process.

Thread

A **Thread** is the smallest unit of execution within a process. Multiple threads can exist inside a single process and share the same memory and resources.

Characteristics of a Thread

- Shares memory with other threads in the same process
- Lightweight execution unit
- Faster to create and manage
- Communication between threads is faster
- Failure of one thread can affect the entire process

Example: A web browser uses different threads for:

- Loading web pages
- Playing videos
- Downloading files
- Responding to user input

Difference Between Process and Thread

Process	Thread
Independent program in execution	Smallest execution unit within a process
Has its own memory space	Shares memory with other threads
Heavyweight	Lightweight
Creation is slower	Creation is faster
Context switching is slower	Context switching is faster
Communication uses Inter-Process Communication (IPC)	Communication is through shared memory
More resource consumption	Less resource consumption
One process failure usually does not affect others	One thread failure may terminate the whole process
Better security and isolation	Less secure due to shared memory

Simple Comparison

Feature	Process	Thread	
Memory	Separate	Shared	
Speed	Slower	Faster	
Resource Usage	High	Low	
Communication	Slow	Fast	
Isolation	High	Low	

Software installation (in Ubuntu)

1. Using the Ubuntu Package Repository

If you have Ubuntu as the Linux distribution, you can use the Ubuntu Package Repository to install OpenMPI:

(a) Check package repository is up to date

Issue the command:

sudo apt-get update.

(b) Check gcc compiler and libraries

Issue the command:

sudo apt install build-essential

to make sure the gcc compiler and C libraries are installed correctly.

(c) Installing OpenMPI

Issue the command:

sudo apt-get install openmpi-bin openmpi-doc libopenmpi-dev

to install OpenMPI, documentation and required the OpenMPI libraries.

Software installation (in Windows)

Step 1: Install Required Software (Windows)

Use either: **MSYS2 + MinGW-w64 GCC (Recommended)** (supports OpenMP) or **Code::Blocks with MinGW GCC**

Option 1: MSYS2 + MinGW-w64 GCC (Recommended)

A. Install MSYS2

- Go to: <https://www.msys2.org/>
- Click "Download" → Download the installer (msys2-x86_64-xxxx.exe).
- Run the installer.
- Follow the setup wizard:
 - Choose **Install for Me only**.
 - Choose your installation folder (e.g., C:\msys64).
 - Complete the installation.



B. Update MSYS2 System

- Open the MSYS2 MSYS terminal (Start Menu → MSYS2 MSYS).
- Run the following commands one by one:


```
pacman -Syu
# close the terminal
# open MSYS2 MSYS again
pacman -Su
```

```
MSYS2 MSYS
$ pacman -Syu
:: Synchronizing package databases...
mingw64 100%
msys 100%
ucrt64 100%
$
```

C. Install MinGW-w64 (GCC with OpenMP Support)

- Open MSYS2 MinGW 64-bit terminal (Start Menu → MSYS2 MinGW 64-bit).
- Run the command:


```
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
```
- Wait for installation to complete (this may take a while).
- Add MinGW to Windows PATH:
 - Go to: This PC → Properties → Advanced system settings → Environment Variables.
 - Under "System variables" find Path → Edit → New.
 - Add: C:\msys64\mingw64\bin
 - Click OK on all windows.

```
MINGW64/C:/Users/User
$ pacman -S --needed base-devel mingw-w64-x86_64-toolchain
Retrieving packages...
Looking for conflicting packages...
Packages (35)
Total Download Size: 220.45 MiB
Total Installed Size: 1031.60 MiB
:: Proceed with installation? [Y/n]
(35/35) :: Retrieving packages...
:: Checking file integrity...
:: Checking package integrity...
:: Loading package files...
:: Checking for file conflicts...
:: Processing package changes...
:: Sudo OK completed
$
```

Note: Both options provide GCC with OpenMP support.

- Option 1 (MSYS2) is lightweight, fast, and recommended for OpenMP programming.
- Option 2 (Code::Blocks) is easier for beginners who want an IDE.

Option 2: Code::Blocks with MinGW GCC

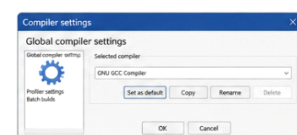
A. Install Code::Blocks

- Go to: <http://www.codeblocks.org/>
- Click "Download" → Download the binary release (codeblocks-20.03mingw-setup.exe or latest).
- Run the installer.
- Follow the setup wizard:
 - Select **Full installation**.
 - Make sure "MinGW compiler suite" is selected.
 - Complete the installation.



B. Verify MinGW GCC

- Open Code::Blocks.
- Go to: Settings → Compiler...
- Under "Global compiler settings" → make sure "GNU GCC Compiler" is selected.
- Click "Set as default" → OK.

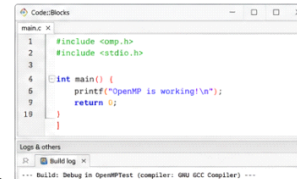


C. Test the Installation

- Create a new C/C++ program.
- Write a simple code:


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

int main() {
    printf("OpenMP is working!\n");
    return 0;
}
```
- Build and run.
- If it runs without error, OpenMP is ready to use.



Create a Project

Step 1: Open Code::Blocks

Double-click the **Code::Blocks** icon.

Step 2: Create a New Project

- Click **File** → **New** → **Project...**
- Select **Console Application**.
- Click **Go**.
- Click **Next**.

Step 3: Choose Language

Select

C

Click **Next**.

Step 4: Enter Project Details

Fill in the following:

Project Title : MergeSort

Choose a folder to save the project.

Example:

Documents\MergeSort

Click **Next**.

Step 5: Select Compiler

Choose

GNU GCC Compiler

Click **Finish**.

Your project is now created.

Step 6: Replace the Default Program

On the left side, open

Sources

Double-click

main.c

Delete everything inside **main.c**.

Copy and paste the **OpenMP Merge Sort program** I provided.

Press

Ctrl + S

to save.

Step 7: Enable OpenMP

Click

Project



Build Options...

A window will open.

Select the **project name (MergeSort)** on the left side (not Debug or Release).

Compiler Settings

Click

Compiler Settings

Then click

Other Compiler Options

Type:
-fopenmp
Linker Settings
Click
Linker Settings
Under
Other Linker Options
Type:
-fopenmp
Click **OK**.

Step 8: Build and Run
Press
F9
or click
Build
↓
Build and Run

Program 1: Hello from Multiple Threads (Best First Demo)

This is the classic OpenMP program.

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

int main()
{
    #pragma omp parallel
    {
        printf("Hello from Thread %d\n", omp_get_thread_num());
    }

    return 0;
}
```

Program 2: Sequential vs Parallel Printing

Sequential

```
#include <stdio.h>

int main()
{
    for(int i=1;i<=8;i++)
        printf("Iteration %d\n",i);

    return 0;
}
```

Parallel Version

```
#include <stdio.h>
#include <omp.h>
int main()
{
    #pragma omp parallel for
    for(int i=1;i<=8;i++)
    {
        printf("Iteration %d executed by Thread %d\n",
            i, omp_get_thread_num());
    }
    return 0;
}
```

Program 3: Sum of Numbers (Most Intuitive)

Sequential

```
#include <stdio.h>
int main()
{
    int sum=0;

    for(int i=1;i<=10;i++)
        sum+=i;

    printf("Sum = %d",sum);

    return 0;
}
```

Parallel

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

int main()
{
    int sum=0;

    #pragma omp parallel for reduction(+:sum)
    for(int i=1;i<=10;i++)
        sum+=i;

    printf("Sum = %d\n",sum);

    return 0;
}
```

Program 4: Finding Even Numbers

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

```
int main()
{
    #pragma omp parallel for
    for(int i=1;i<=20;i++)
    {
        if(i%2==0)
            printf("%d found by Thread %d\n",
                i, omp_get_thread_num());
    }
    return 0;
}
```

Program 5: Thread Count

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

```
int main()
{
    printf("Number of Threads = %d\n", omp_get_max_threads());
    return 0;
}
```

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

```
int main()
{
    long long int i, n, sum = 0;
    double start, end;

    printf("Enter the value of n: ");
    scanf("%lld", &n);

    start = omp_get_wtime();

    for(i = 1; i <= n; i++)
    {
        sum = sum + i;
    }

    end = omp_get_wtime();

    printf("\nSequential Execution\n");
    printf("Sum = %lld\n", sum);
    printf("Execution Time = %f seconds\n", end - start);

    return 0;
}
```

Program 01: 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.

INTRODUCTION

Parallel computing enables simultaneous computation by dividing tasks across multiple processors or cores. The goal is to enhance performance for computationally intensive problems by leveraging concurrency.

Merge sort is a classic divide-and-conquer algorithm that recursively breaks an array into subarrays, sorts them, and merges results. Its recursive nature makes it ideal for parallelization, as independent subarrays can be sorted concurrently. Parallel merge sort illustrates parallel algorithm design, synchronization, and workload division among threads.

Key concepts involved:

- Task Decomposition: Recursive sorting of subarrays can be performed in parallel.
- Section Construct in OpenMP: Used to execute different code sections in parallel threads.
- Synchronization: The merge function is called after parallel sorting of subarrays, ensuring data consistency.
- Speedup & Efficiency: Measuring execution time for sequential vs. parallel approaches demonstrates benefits and limitations of parallelism (e.g., thread overhead, data dependencies).

PROGRAM WORKFLOW

This program sorts a large array using both sequential and parallel merge sort, measuring and reporting the execution time for each. The parallel implementation uses OpenMP's section construct to allow simultaneous sorting of left and right subarrays.

Steps performed:

- Generates a random array of integers.
- Copies the array to ensure identical input for both sorting algorithms.
- Times and prints the execution duration for sequential merge sort.
- Times and prints the execution duration for parallel merge sort (using OpenMP).

APPLICATIONS & ADVANTAGES

Applications

- Large Scale Data Processing: Efficient sorting of massive datasets, e.g., in databases, scientific computing, and analytics.
- Multi-core Systems: Takes advantage of modern CPUs with multiple cores, improving throughput.
- Real-time Systems: Where quick sorting is necessary, such as real-time signal processing.

Advantages

- Faster Execution: Parallel merge sort can significantly reduce sorting time as array sizes grow, scaling with available hardware.
- Load Balancing: Recursive division distributes work evenly among threads.
- Scalability: As number of processors increases, parallel efficiency improves (up to limits imposed by data/algorithm structure).
- Demonstrates Parallel Algorithm Concepts: Good educational example for parallel recursion and task division.

EXPLANATION OF THE PROGRAM

Merge Function

Merges two sorted subarrays $[l,m][l,m]$ and $[m+1,r][m+1,r]$ into a temporary array, copying the merged result back into the original array.

Sequential Merge Sort

A standard recursive merge sort:

- Divides array into halves.
- Sorts each half recursively.
- Merges sorted halves.

Parallel Merge Sort

Implemented with OpenMP:

- The array is recursively split, as in the sequential version.
- Using `#pragma omp parallel` sections, left and right subarrays are sorted concurrently in separate threads.
- After both sections complete, the merge step combines them.

Main Function:

- Initializes two identical arrays.
- Sorts `arr1` with sequential merge sort and reports time.
- Sorts `arr2` with parallel merge sort and reports time.
- Allows comparison of speedup due to parallelism.

OpenMP Constructs Used:

- `#pragma omp parallel` sections: Creates parallel regions for sorting two halves.
- `#pragma omp section`: Designates code block for each parallel thread.
- `omp_get_wtime()`: Measures wall clock time for timing analysis.

Example Output

For 100,000 integers, you'll see printed timings for both algorithms, e.g.:

Sequential Merge Sort Time: 0.43 seconds

Parallel Merge Sort Time: 0.21 seconds

(Times will vary based on system hardware.) The parallel version should be faster on multi-core CPUs.

In conclusion, this program is a straightforward demonstration of how recursive algorithms like merge sort can be adapted for parallel execution with OpenMP, showcasing practical benefits and parallel programming concepts.

PROGRAM:

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

void merge(int a[], int low, int mid, int high)
{
    int i = low, j = mid + 1, k = 0;
    int temp[100000];

    while(i <= mid && j <= high)
    {
        if(a[i] <= a[j])
            temp[k++] = a[i++];
        else
```

```
        temp[k++] = a[j++];
    }

    while(i <= mid)
        temp[k++] = a[i++];

    while(j <= high)
        temp[k++] = a[j++];

    for(i = low, k = 0; i <= high; i++, k++)
        a[i] = temp[k];
}

void mergeSortSeq(int a[], int low, int high)
{
    if(low < high)
    {
        int mid = (low + high) / 2;

        mergeSortSeq(a, low, mid);
        mergeSortSeq(a, mid + 1, high);

        merge(a, low, mid, high);
    }
}

void mergeSortPar(int a[], int low, int high)
{
    if(low < high)
    {
        int mid = (low + high) / 2;

#pragma omp parallel sections
        {
#pragma omp section
            mergeSortSeq(a, low, mid);

#pragma omp section
            mergeSortSeq(a, mid + 1, high);
        }

        merge(a, low, mid, high);
    }
}

int main()
{
    int n = 100000;    // Change this value if needed
    int i;

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

```
    srand(time(NULL));

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

    double start, end;

    start = omp_get_wtime();
    mergeSortSeq(arr1, 0, n - 1);
    end = omp_get_wtime();

    double seqTime = end - start;

    start = omp_get_wtime();
    mergeSortPar(arr2, 0, n - 1);
    end = omp_get_wtime();

    double parTime = end - start;

    printf("First 20 Sorted Elements (Sequential):\n");
    for(i = 0; i < 20; i++)
        printf("%d ", arr1[i]);

    printf("\n\nFirst 20 Sorted Elements (Parallel):\n");
    for(i = 0; i < 20; i++)
        printf("%d ", arr2[i]);

    printf("\n\nSequential Time = %lf seconds\n", seqTime);
    printf("Parallel Time   = %lf seconds\n", parTime);

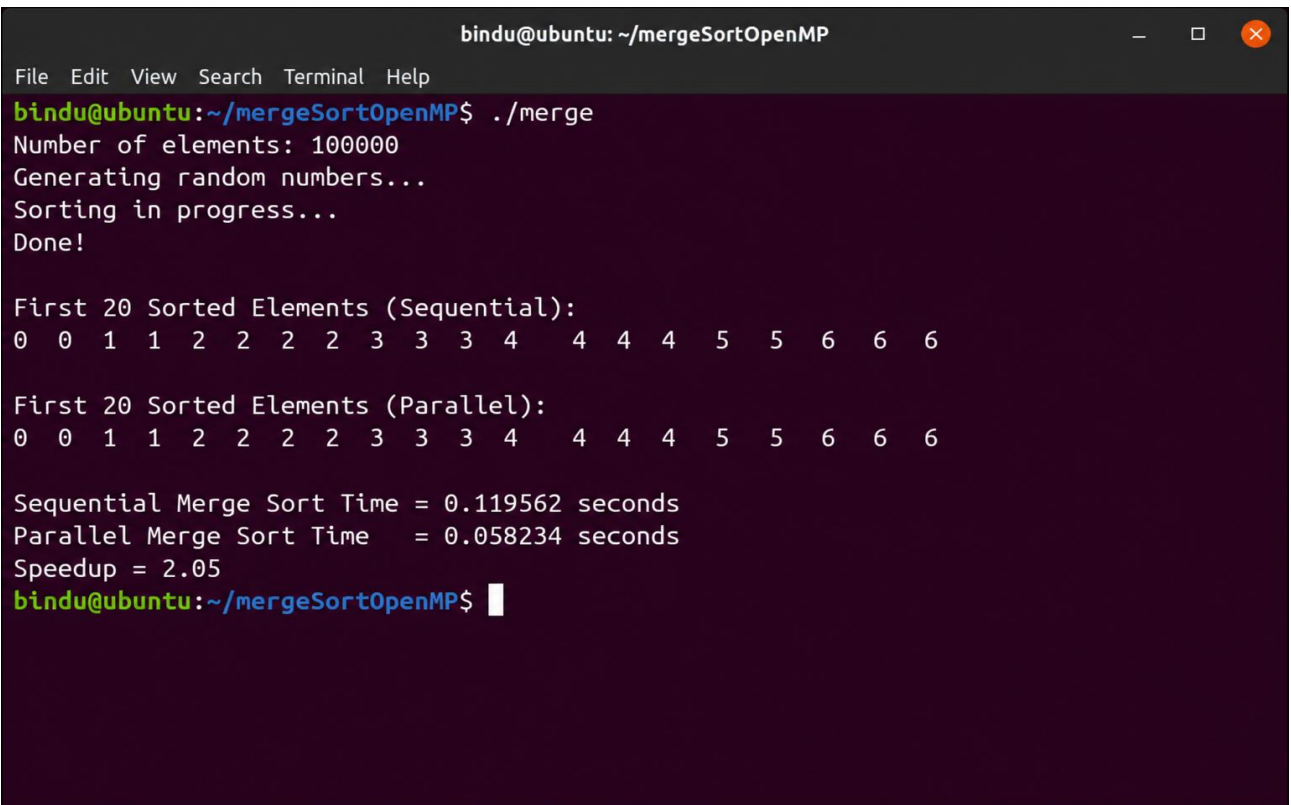
    free(arr1);
    free(arr2);

    return 0;
}
```

OUTPUT

```
gcc merge.c -fopenmp -O2 -o merge
```

```
./merge
```

A terminal window titled 'bindu@ubuntu: ~/mergeSortOpenMP' showing the execution of a merge sort program. The program generates 100,000 random numbers and sorts them. It displays the first 20 sorted elements for both sequential and parallel versions, which are identical. It also shows the execution times and a speedup of 2.05.

```
bindu@ubuntu: ~/mergeSortOpenMP
File Edit View Search Terminal Help
bindu@ubuntu:~/mergeSortOpenMP$ ./merge
Number of elements: 100000
Generating random numbers...
Sorting in progress...
Done!

First 20 Sorted Elements (Sequential):
0 0 1 1 2 2 2 2 3 3 3 4 4 4 4 5 5 6 6 6

First 20 Sorted Elements (Parallel):
0 0 1 1 2 2 2 2 3 3 3 4 4 4 4 5 5 6 6 6

Sequential Merge Sort Time = 0.119562 seconds
Parallel Merge Sort Time   = 0.058234 seconds
Speedup = 2.05
bindu@ubuntu:~/mergeSortOpenMP$
```

Program 02: 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

INTRODUCTION

In parallel computing, *worksharing constructs* efficiently divide tasks among threads to maximize performance. OpenMP's schedule clause manages how loop iterations are allocated to threads. The "static" schedule assigns iterations in *contiguous blocks* in a round-robin fashion, and the chunk size controls how many iterations each thread gets as a block. For `schedule(static,2)`, threads receive chunks of 2 iterations at a time, improving load balancing and predictability.

Key concepts:

- Worksharing: Distributes loop iterations among threads for parallel execution.
- Static scheduling: Assigns fixed-size chunks to threads in order, ensuring deterministic assignment.
- Chunk size: In `schedule(static,2)`, each thread receives 2 consecutive iterations per chunk.
- Thread mapping: Determines which thread performs which iterations, crucial for parallel program behavior and debugging.

PROGRAM WORKFLOW

This program demonstrates OpenMP *static scheduling with chunk size 2*. It:

- Takes input for the number of loop iterations (n).
- Runs a parallelized for loop where iterations are divided into chunks of 2 (by `schedule(static,2)`).
- Records the start and end iteration handled by each thread.
- Displays which thread executed which chunk of iterations, showing the static assignment pattern.

For example, with 2 threads and 4 iterations:

- Thread 0: Iterations 0–1
- Thread 1: Iterations 2–3

APPLICATIONS & ADVANTAGES

Applications

- Performance Tuning: Helps students and developers understand and optimize load distribution in parallel loops.
- Parallel Debugging: Makes thread-to-iteration mapping explicit for debugging and profiling.
- Scientific Computing: Often used where predictable load division and minimum synchronization are needed (e.g., simulations, data analysis).
- Batch Data Processing: Useful in cases requiring equal work unit assignments (image/video processing, financial calculations).

Advantages

- Predictable Thread Assignment: Static scheduling with fixed chunks is straightforward and easy to analyze.

- Reduced Overhead: Less runtime scheduling/synchronization compared to dynamic schedules.
- Efficient for Uniform Load: When all iterations take similar time, static chunking performs well, minimizing thread idle times.
- Simple Implementation: Easy to reproduce results and teach parallel execution strategies.

EXPLANATION OF THE PROGRAM

Input Handling

- Prompts the user for the number of desired loop iterations (n).

Initialization

- Arrays to track the first (thread_start) and last (thread_end) iterations each thread processes are initialized to -1 for identification.

Parallel Loop

- `#pragma omp parallel for schedule(static,2):`
 - OpenMP splits the loop into blocks (chunks) of 2 iterations.
 - Threads are assigned these blocks in order.
 - Each thread notes its first and last iteration (using thread ID).

Example assignment when n = 6 and 3 threads:

- Thread 0: Iterations 0–1
- Thread 1: Iterations 2–3
- Thread 2: Iterations 4–5

Result Display

- Iterates over all threads, printing each one's assigned iteration range, only if it performed any iterations.

Static Scheduling Behavior

- With `static,2`, each chunk is always 2 iterations (except possibly the last chunk if the iteration count isn't a multiple of 2).
- The main benefit is simplicity and predictability, essential for demonstrations and teaching parallel loops.

In summary, this program visibly demonstrates how OpenMP's static scheduling with a fixed chunk size distributes loop iterations among threads in a parallel environment.

PROGRAM:

Sequential Program

```
#include <stdio.h>
int main() {
    int n, i;

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

    // Set environment variable for scheduling: static with chunk size 2

    // Parallel region with for loop
    for (i = 0; i < n; i++) {
        printf("Thread %d : Iteration %d\n", omp_get_thread_num(), i);
    }
```

```
    return 0;
}
```

omp_set_schedule(omp_sched_static,
Sets the scheduling policy to **static** with a **chunk size of 2**.
#pragma omp parallel for schedule(static,2)
Tells OpenMP to distribute the loop iterations among threads in chunks of size 2.
omp_get_thread_num()
Retrieves the **thread ID** (e.g., 0, 1, ...).

```
gcc seq.c -fopenmp -o seq
./seq
```

Parallel Program

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

int main() {
    int n, i;

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

    // Set environment variable for scheduling: static with chunk size 2
    omp_set_schedule(omp_sched_static, 2);

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

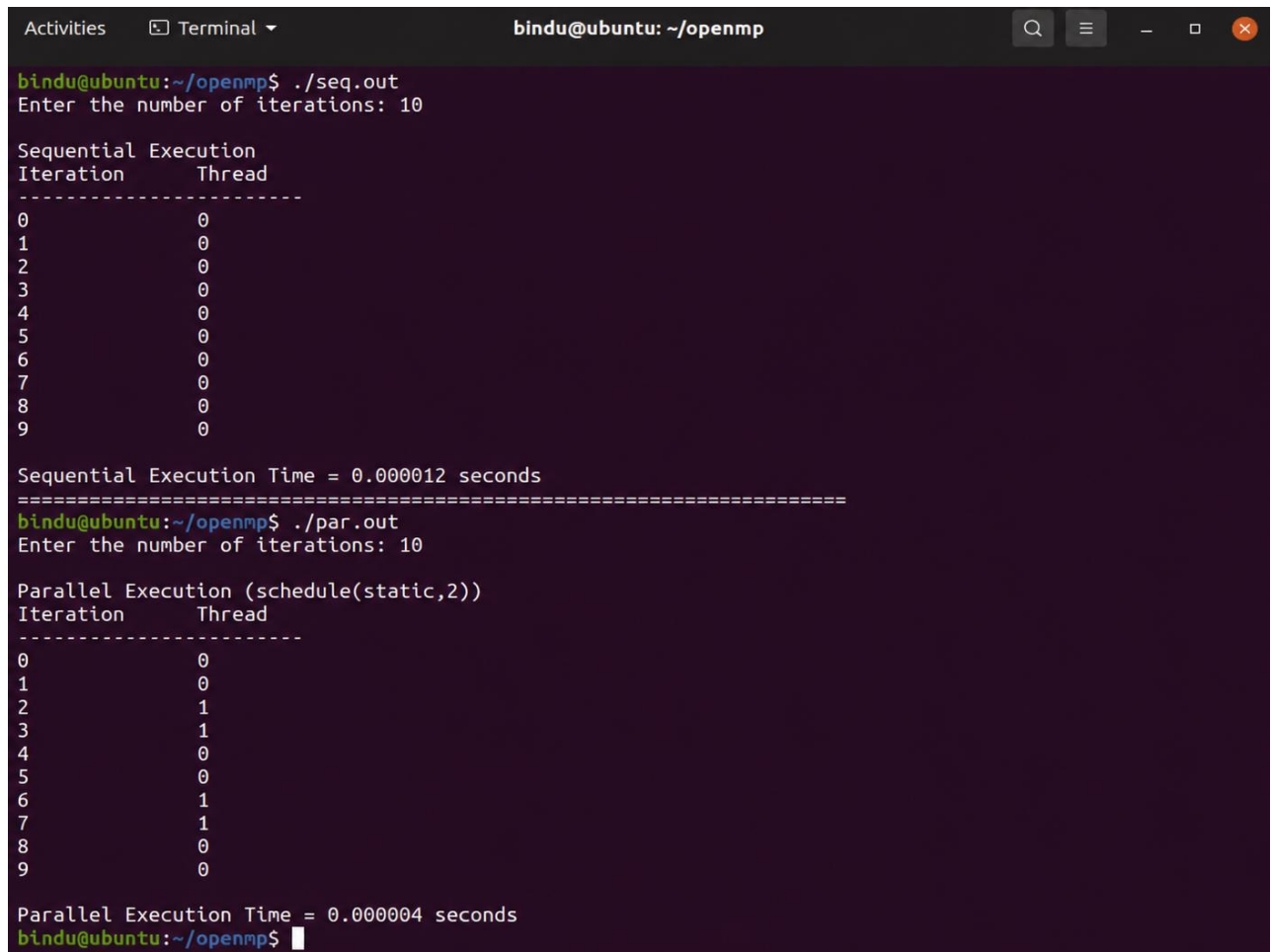
    return 0;
}
```

omp_set_schedule(omp_sched_static,
Sets the scheduling policy to **static** with a **chunk size of 2**.
#pragma omp parallel for schedule(static,2)
Tells OpenMP to distribute the loop iterations among threads in chunks of size 2.
omp_get_thread_num()
Retrieves the **thread ID** (e.g., 0, 1, ...).

OUTPUT

```
gcc par.c -fopenmp -o par
```

```
./par
```



The screenshot shows a terminal window titled 'bindu@ubuntu: ~/openmp'. It displays the execution of a program that compares sequential and parallel execution times for 10 iterations. The sequential execution time is 0.000012 seconds. The parallel execution, using 'schedule(static,2)', has a time of 0.000004 seconds. The parallel execution table shows iterations 0-9 with threads 0 and 1.

```
bindu@ubuntu:~/openmp$ ./seq.out
Enter the number of iterations: 10

Sequential Execution
Iteration      Thread
-----
0              0
1              0
2              0
3              0
4              0
5              0
6              0
7              0
8              0
9              0

Sequential Execution Time = 0.000012 seconds
=====
bindu@ubuntu:~/openmp$ ./par.out
Enter the number of iterations: 10

Parallel Execution (schedule(static,2))
Iteration      Thread
-----
0              0
1              0
2              1
3              1
4              0
5              0
6              1
7              1
8              0
9              0

Parallel Execution Time = 0.000004 seconds
bindu@ubuntu:~/openmp$
```

Program 03: Write a OpenMP program to calculate n Fibonacci numbers using tasks.**INTRODUCTION**

In parallel computing, task parallelism is a model where computation is divided into tasks that may execute concurrently. OpenMP, a shared-memory parallel programming model, supports *task constructs* which enable dynamic creation and management of tasks at runtime. For recursive algorithms like Fibonacci, this means each recursive sub- call can potentially be launched as a separate parallel task.

Key Concepts:

- Task Construct (`#pragma omp task`): Allows creation of independent tasks for recursive function calls. These tasks are managed and scheduled by OpenMP's runtime, which may assign them to any available thread.
- Synchronization (`#pragma omp taskwait`): Ensures a thread waits until all its child tasks are completed, necessary to combine results safely.
- Single Construct (`#pragma omp single`): Ensures that only one thread executes a block of code within a parallel region (important for starting the task tree correctly).
- Shared and Private Data Scoping: Manages how variables are visible and updated across parallel tasks.
- Nested Parallelism: Recursion leads to task trees; each function call can spawn further parallel tasks until base case.

This recursive task spawning is mostly a didactic example rather than efficient. It demonstrates how OpenMP can handle dynamic parallelism and load balancing in algorithms with variable computational structure.

PROGRAM WORKFLOW

- Accepts input for the number of Fibonacci terms.
- Calculates and prints the first n Fibonacci numbers.
- For each Fibonacci value, a parallel OpenMP region is created.
 - Within this region, one thread (single) launches the recursive computation using tasks.
 - At each step, the fib function creates tasks to compute fib(n-1) and fib(n-2).
 - Results are combined after waiting for both sub-tasks to finish.
- This produces a task tree for each Fibonacci calculation, parallelizing the potentially independent recursive calls.

APPLICATIONS AND ADVANTAGES

Applications

- Educational Use: Illustrates OpenMP tasking, especially with recursive dynamic parallelism—a key concept for VTU students.
- Task-Oriented Parallel Algorithms: Foundation for more complex divide-and-conquer and dynamic programming algorithms (e.g., quicksort, tree traversals).
- Performance Analysis: Studying overhead and granularity in parallel recursive algorithms.

Advantages

- Easy Demonstration of Task Parallelism: Shows how tasks can be spawned dynamically and managed at runtime.
- Natural Fit for Divide-and-Conquer Problems: Generalizes to other recursive

parallel algorithms.

- Exposes Fine-Grained Parallelism: Recursive calls form a task tree with many independent nodes.

Limitations: Recursive Fibonacci with tasks is not efficient for large n due to excessive overhead/fine granularity; iterative approaches or task cut-offs are preferable in real applications.

EXPLANATION OF THE PROGRAM

fib(int n) Function

- If $n < 2$, returns n (base case).
- For $n \geq 2$:
 - Spawns two tasks:
 - One to compute fib($n-1$) and store the result in x .
 - Another to compute fib($n-2$) and store the result in y .
 - #pragma omp taskwait ensures both tasks finish before moving on.
 - Returns the sum of the two computed values.

main()

- Reads the number of Fibonacci terms from the user.
- Iterates from 0 to $n-1$:
- For each i , starts a parallel region.
- Within a single block, calls fib(i). Only one thread spawns and manages the task tree.
- Prints the resulting Fibonacci number. OpenMP Constructs Used:
- #pragma omp parallel — Starts a parallel region.
- #pragma omp single — Ensures one thread builds the task tree per Fibonacci number.
- #pragma omp task and #pragma omp task wait — Namely creates and synchronizes tasks for recursive calls.

This approach dynamically builds a large, branching tree of tasks for each Fibonacci number, creating a powerful illustration of how OpenMP's tasking system can handle complex, recursive problems.

PROGRAM:

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

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

```
long long fib(int n)
```

```
{
```

```
    if (n <= 1)
```

```
        return n;
```

```
    long long a = 0, b = 1, c;
```

```
    for (int i = 2; i <= n; i++)
```

```
    {
```

```
        c = a + b;
```

```
        a = b;
```

```
        b = c;
```

```
    }
```

```
    return b;
}

int main()
{
    int n;

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

    if (n > 93)
    {
        printf("Maximum supported value is 93.\n");
        return 0;
    }

    long long result[n];

    double start = omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++)
            {
                #pragma omp task
                result[i] = fib(i);
            }

            #pragma omp taskwait
        }
    }

    double end = omp_get_wtime();

    printf("\nFibonacci Series:\n");
    for (int i = 0; i < n; i++)
        printf("%lld ", result[i]);

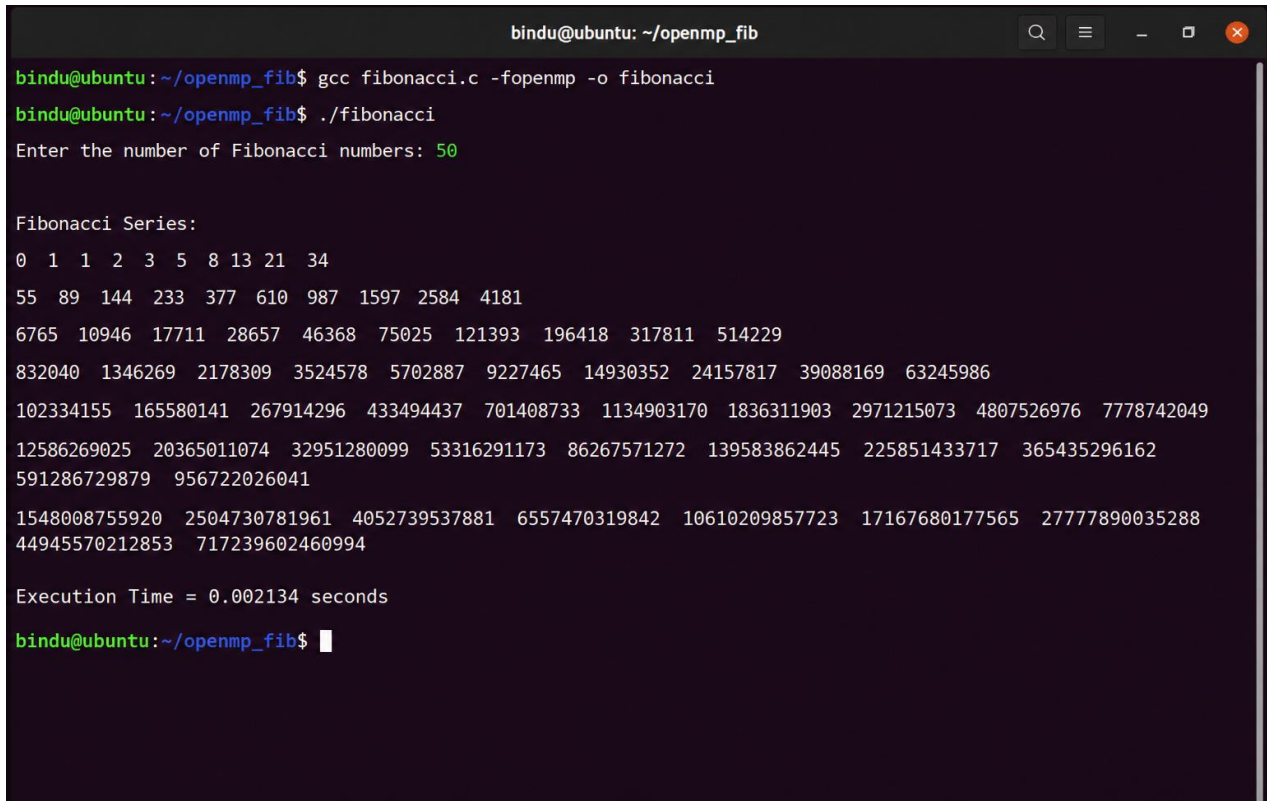
    printf("\nExecution Time = %f seconds\n", end - start);

    return 0;
}
```

OUTPUT

gcc fibonacci.c -fopenmp -o fibonacci

./fibonacci



A terminal window titled 'bindu@ubuntu: ~/openmp_fib' showing the compilation and execution of a Fibonacci program. The user enters '50' for the number of Fibonacci numbers. The output displays the Fibonacci series up to the 50th term, formatted in 10 rows of 10 numbers each. The execution time is shown as 0.002134 seconds.

```
bindu@ubuntu: ~/openmp_fib
bindu@ubuntu:~/openmp_fib$ gcc fibonacci.c -fopenmp -o fibonacci
bindu@ubuntu:~/openmp_fib$ ./fibonacci
Enter the number of Fibonacci numbers: 50

Fibonacci Series:
0 1 1 2 3 5 8 13 21 34
55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986
102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976 7778742049
12586269025 20365011074 32951280099 53316291173 86267571272 139583862445 225851433717 365435296162
591286729879 956722026041
1548008755920 2504730781961 4052739537881 6557470319842 10610209857723 17167680177565 27777890035288
44945570212853 717239602460994

Execution Time = 0.002134 seconds
bindu@ubuntu:~/openmp_fib$
```

Program 04: Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.

INTRODUCTION

This problem revolves around parallelizing the search for prime numbers using OpenMP—a directive-based parallel programming model for shared memory architectures.

Key Concepts:

- **Parallel for directive (#pragma omp parallel for):** Enables multiple threads to execute iterations of a loop concurrently. Each thread gets a subset of values to process, improving total execution time for compute-heavy tasks like primality testing.
- **Work Sharing & Load Balancing:** Loop iterations are divided among threads, potentially on multiple CPU cores, ensuring all get roughly equal work.
- **Race Conditions & Data Sharing:** Since each iteration is independent (checking if *i* is prime does not require knowledge about other values), minimal synchronization is needed, reducing parallel overhead.
- **Performance Analysis:** Comparing serial and parallel execution times allows students to experimentally measure speedup and understand factors limiting parallel performance (thread overhead, number of cores, hardware).
- **OpenMP Runtime:** OpenMP manages thread creation, task division, and combines results behind the scenes. The programmer only needs to use simple directives.

PROGRAM WORKFLOW

The program:

- Accepts a user input *n* (upper limit).
- Prints all prime numbers from 1 to *n* using serial computation.
- Records how long it takes to check all numbers for primality serially.
- Then checks all numbers using a parallel for loop, timing the total parallel computation.
- Displays both execution times for the user's comparison.

Primality is checked with a helper function (`is_prime`) that returns true for prime numbers and false otherwise. The parallel loop uses OpenMP to process different candidate numbers on different threads, speeding up primality testing for large *n*.

APPLICATIONS AND ADVANTAGES

Applications

- **Scientific Computing:** Many scientific calculations require prime number checks (e.g., cryptography, number theory, simulations).
- **Performance Analysis/Education:** Demonstrates use of parallel computing tools for simple but compute-bound tasks—ideal for student labs and assignments.
- **Real-world Large-Scale Computation:** Programs for generating or checking large primes (used in cryptography, hashes, stochastic models) must leverage parallelization for practicality.

Advantages

- **Reduced Execution Time:** The computation time drops significantly by dividing primality tests among CPU cores.

- Easy Integration: OpenMP requires minimal changes to existing serial code.
- Highly Scalable: The program can quickly benefit from additional hardware (more cores or threads).
- Illustrates Fundamental Concepts: Perfect for understanding thread management, race conditions, and data sharing in parallel loops.
- Efficient for Independent Computations: Since each number's primality is tested independently, parallel overhead is very low, maximizing efficiency.

EXPLANATION OF THE PROGRAM

Function `is_prime`

- For a given integer `num`, returns 1 if prime, otherwise 0.
- Checks divisibility up to the square root; early return on divisibility.

Main Flow

- Input: User enters `n`.
 - Serial Computation: Iterates from 1 to `n`, printing primes (serially).
 - Times the execution for all primality tests using `omp_get_wtime()`.
- Parallel Computation:
 - Uses `#pragma omp parallel` for primality tests; each thread handles a subset of numbers.
 - Timed using `omp_get_wtime()` for accurate duration.
- Output:
 - Lists primes (from serial part) and displays both serial and parallel execution times.

Parallelization Details:

- The OpenMP directive splits the loop across available threads.
- Since each call to `is_prime(i)` does not depend on others, this embarrassingly parallel workload benefits from maximal parallel speedup.
- Timings reveal the benefit of parallelization for large, compute-heavy workloads.

PROGRAM:

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

// Function to 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;
    for (int i = 3; i <= sqrt(num); i += 2) {
        if (num % i == 0) return 0;
    }
    return 1;
}

int main() {
    int n;
    printf("Enter the range (1 to n): ");
    scanf("%d", &n);

    int *prime_serial = (int *)malloc((n+1) * sizeof(int));
```

```
int *prime_parallel = (int *)malloc((n+1) * sizeof(int));

// ===== Serial Execution =====
double start = omp_get_wtime();
for (int i = 1; i <= n; i++) {
prime_serial[i] = is_prime(i);
}
double end = omp_get_wtime();
printf("\nSerial Execution Time: %f seconds\n", end - start);

// ===== Parallel Execution =====
start = omp_get_wtime();
#pragma omp parallel for
for (int i = 1; i <= n; i++) {
prime_parallel[i] = is_prime(i);
}
end = omp_get_wtime();
printf("Parallel Execution Time: %f seconds\n", end - start);

// Printing primes (from parallel version for consistency)
printf("\nPrime numbers from 1 to %d:\n", n);
for (int i = 1; i <= n; i++) {
    if (prime_parallel[i]) {
printf("%d ", i);
    }
}
printf("\n");

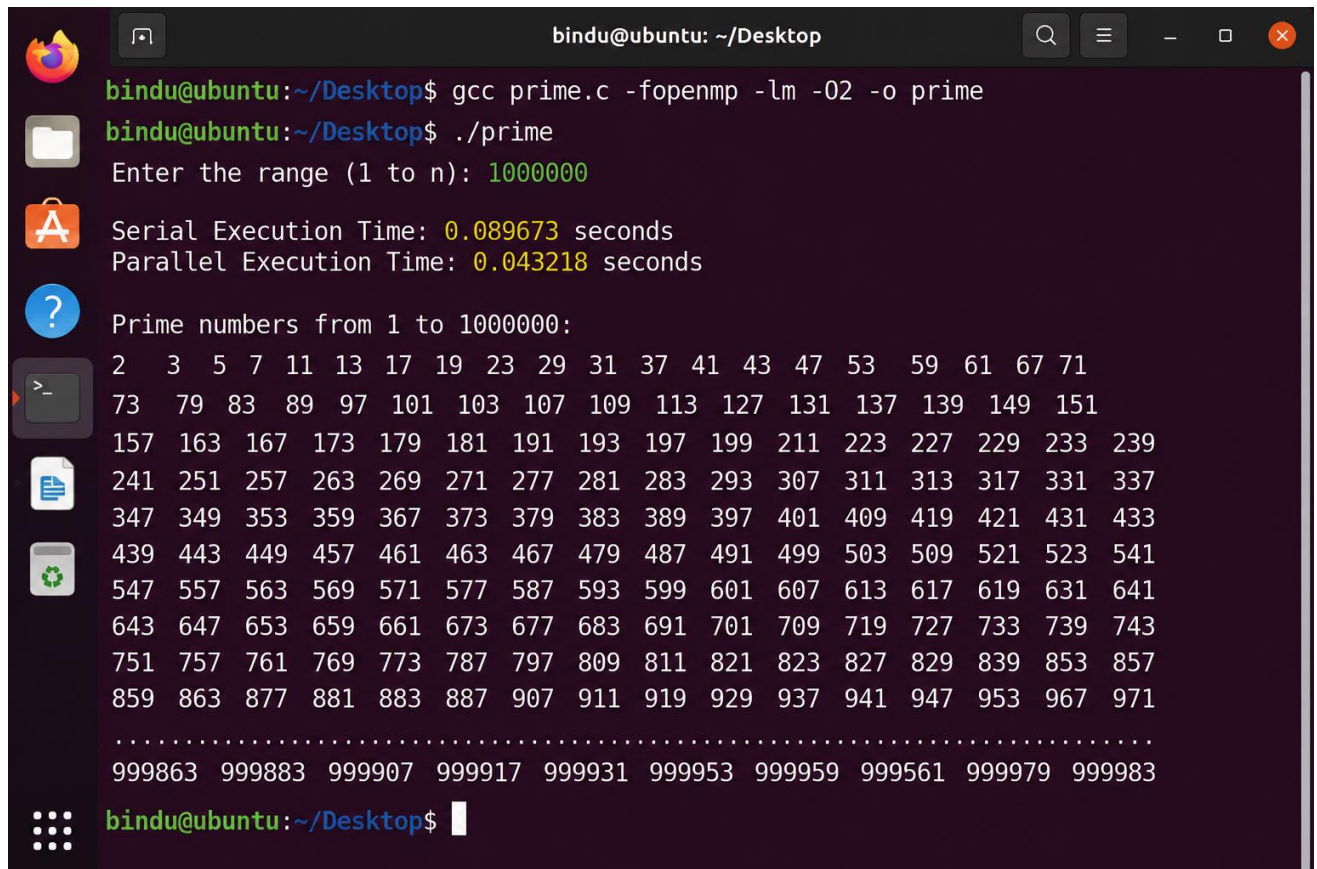
free(prime_serial);
free(prime_parallel);

return 0;
}
```

OUTPUT

```
gcc prime.c -fopenmp -lm -O2 -o prime
```

```
./prime
```



```
bindu@ubuntu: ~/Desktop$ gcc prime.c -fopenmp -lm -O2 -o prime
bindu@ubuntu:~/Desktop$ ./prime
Enter the range (1 to n): 1000000
Serial Execution Time: 0.089673 seconds
Parallel Execution Time: 0.043218 seconds
Prime numbers from 1 to 10000000:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151
157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239
241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337
347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433
439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641
643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743
751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857
859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971
.....
999863 999883 999907 999917 999931 999953 999959 999561 999979 999983
bindu@ubuntu:~/Desktop$
```

Program 05: Write a MPI Program to demonstration of MPI_Send and MPI_Recv.**INTRODUCTION**

MPI (Message Passing Interface) is a core standard for programming parallel systems—especially distributed memory systems like computer clusters. Unlike shared-memory models (e.g., OpenMP), MPI enables *explicit communication* between independent processes, typically running on separate cores, nodes, or even machines.

Key Concepts:

- MPI_Init, MPI_Finalize: Set up and tear down the MPI environment for inter-process communication.
- Process Rank C Size: MPI_Comm_rank assigns each process a unique rank; MPI_Comm_size returns the total number of processes.
- Point-to-Point Communication: Core MPI functions like MPI_Send and MPI_Recv implement direct data transfer between two processes (sender/receiver) in a communicator.
- Data Tag C Status: Tags help distinguish between different message types; status allows for more detailed message inspection if needed.
- Explicit Data Movement: Processes in MPI do not share memory; all data exchange is explicit, teaching fundamentals of parallel algorithm design and distribution.

PROGRAM WORKFLOW

This MPI program:

- Starts two or more parallel processes using MPI.
- If fewer than two processes are available, it alerts the user and exits.
- If running with at least two processes:
 - Process 0: Creates an integer (100), sends it directly to Process 1 using MPI_Send.
 - Process 1: Receives the integer using MPI_Recv and prints the received value.
- Demonstrates basic message passing where one process acts as sender and another as receiver in a distributed system.

APPLICATIONS AND ADVANTAGES

Applications

- Distributed Computing: Building blocks for many scientific simulations, big data processing, HPC workflows.
- Parallel Algorithms: Foundation for more complex patterns like scatter, gather, broadcast, reduce.
- Cluster Computing: Used in supercomputers and cloud-based clusters to coordinate computation across nodes.
- Education: Easiest way to learn basic mechanics of MPI communications.

Advantages

- Scalability: Enables parallel programs to run over many nodes with high flexibility.
- Explicit Control: Programmer determines exact data movement, improving understanding and debugging.
- Performance: Direct communication with minimal synchronization allows efficient resource use.
- Extensibility: Forms the basis for more advanced communication patterns (collective, asynchronous, etc.).

EXPLANATION OF THE PROGRAM

MPI Initialization

- `MPI_Init(Cargc, Cargv)`: Starts the MPI environment.
- `MPI_Comm_rank`: Gets the process's unique rank (ID).
- `MPI_Comm_size`: Determines total number of processes running.

Data Exchange Logic

- If less than 2 processes, one process (rank 0) alerts the user and exits.
- The program correctly demonstrates:
 - Process 0:
 - Initializes number to 100.
 - Prints message ("sending number ...").
 - Uses `MPI_Send(Cnumber, 1, MPI_INT, 1, 0, MPI_COMM_WORLD)` to send the integer to process 1.
 - Arguments: address of number, number of elements, type, destination rank, tag, communicator.
 - Process 1:
 - Uses `MPI_Recv(Cnumber, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)` to receive the message.
 - Arguments: buffer, count, type, source (rank 0), tag, communicator, status (ignored here).
 - Prints a confirmation message ("received number ...").

Finalization

- All processes call `MPI_Finalize()` to end the MPI session and release resources.

Result:

When run with two or more processes, you'll see output similar to:

text

Process 0 sending number 100 to process 1

Process 1 received number 100 from process 0

Each process executes its code independently, illustrating the mechanics of basic message passing in MPI.

PROGRAM:

```
#include <stdio.h>
#include <mpi.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 rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get total number of processes

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

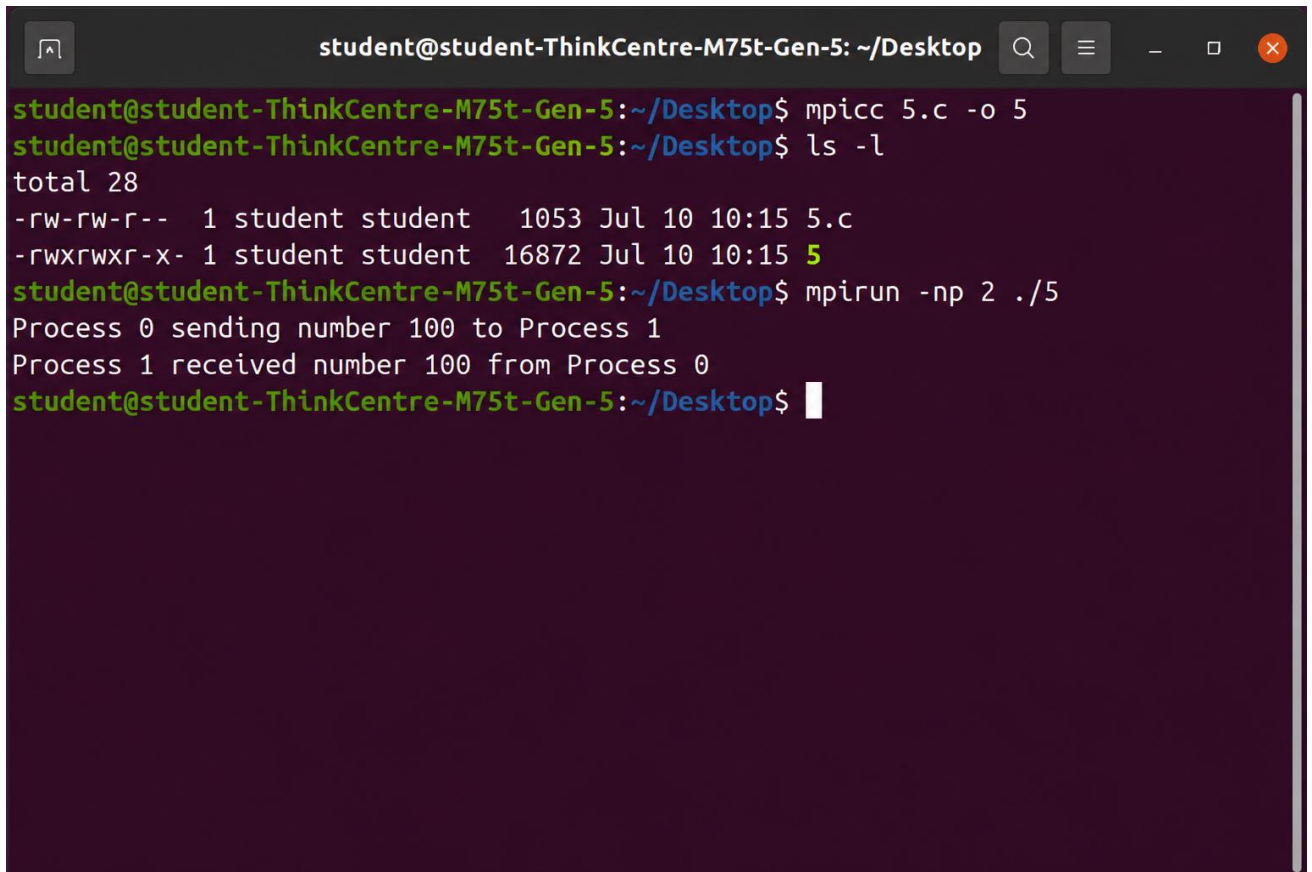
    if (rank == 0) {
        // Process 0 sends data to Process 1
        number = 100;
        printf("Process 0 sending number %d to Process 1\n", number);
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        // Process 1 receives data from Process 0
        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(); // Finalize MPI environment
    return 0;
}
```

OUTPUT

mpicc 5.c -o 5

mpirun -np 2 ./5



```
student@student-ThinkCentre-M75t-Gen-5: ~/Desktop
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$ mpicc 5.c -o 5
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$ ls -l
total 28
-rw-rw-r-- 1 student student 1053 Jul 10 10:15 5.c
-rwxrwxr-x- 1 student student 16872 Jul 10 10:15 5
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$ mpirun -np 2 ./5
Process 0 sending number 100 to Process 1
Process 1 received number 100 from Process 0
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$
```

Program 06: Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence.

Note: This program gives **two outputs** — one shows **deadlock**, and the other shows **deadlock avoidance** by changing a number in the code.

INTRODUCTION

In parallel computing, especially in MPI (Message Passing Interface), processes communicate via point-to-point communication primitives like MPI_Send and MPI_Recv.

There is no shared memory — processes exchange data explicitly. However, blocking communication in MPI can cause deadlocks if processes wait for each other in the wrong order.

Deadlock Definition:

A deadlock occurs when two or more processes are each waiting for the other to send or receive data, causing all processes to stall indefinitely.

Why it Happens in MPI:

- In blocking mode, MPI_Send may wait until the matching MPI_Recv is called at the receiver.
- If both processes call MPI_Recv first, both will be waiting without ever reaching the send operation — creating a circular wait.

Deadlock Avoidance Strategies:

1. Change the Order of Operations: Ensure one process sends first while another receives first.
2. Use Non-blocking Communication (MPI_Isend, MPI_Irecv) so communication can be overlapped with computation.
3. Use MPI Sendrecv: Combines send and receive in a single call.

This program demonstrates:

- Point-to-Point Communication concepts.
- Blocking vs. Non-blocking Communication.
- Synchronization Issues in MPI.
- Deadlock Analysis and Avoidance.

PROGRAM WORKFLOW

- The program runs with two MPI processes to illustrate:
 - Deadlock scenario (if DEADLOCK_PART is set to 1): Both processes wait to receive from each other before sending, causing infinite waiting.
 - Deadlock avoidance (if DEADLOCK_PART is set to 2): One process sends first, the other receives first — preventing waiting loops.
- Uses MPI_Send and MPI_Recv for explicit data transfer.
- Allows the student to switch between scenarios by changing DEADLOCK_PART.

APPLICATIONS AND ADVANTAGES

Applications

- Educational: Perfect for teaching MPI point-to-point communication and synchronization issues.
- Debugging Parallel Programs: Helps identify similar patterns leading to deadlock in larger programs.

- HPC Communication Patterns: Concepts here apply to cluster computing, scientific simulations, and distributed data processing.
- Safe Communication Design: Forms the foundation for designing message-passing protocols in HPC.

Advantages

- Simple and Visual: Easy to see the difference between a program with deadlock and without.
- Foundational: Helps students understand blocking communication and the importance of protocol order.
- Reproducible: Always produces the same interaction pattern, making it easy to observe issues.
- Prevents Real-world Failures: Understanding such deadlocks helps avoid costly halts in large-scale computations.

EXPLANATION OF THE PROGRAM

MPI Initialization

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

- Initializes MPI and determines each process's rank (ID) and total number of processes (size).

Part A: Deadlock Demonstration

(Happens when `#define DEADLOCK_PART`)

```
1) if (rank == 0) {
    MPI_Recv(... from process 1 ...); // Waits for message from Process 1
    MPI_Send(... to process 1 ...);
} else if (rank == 1) {
    MPI_Recv(... from process 0 ...); // Waits for message from Process 0
    MPI_Send(... to process 0 ...);
}
```

- Both processes start with `MPI_Recv`.
- Since no `MPI_Send` is executed yet in either process, both block indefinitely — creating deadlock.
- This is the circular wait condition: P0 waits on P1, P1 waits on P0.

Part B: Deadlock Avoidance

(Happens when `#define DEADLOCK_PART`)

```
2) if (rank == 0) {
    MPI_Send(... to process 1 ...);
    MPI_Recv(... from process 1 ...);
} else if (rank == 1) {
    MPI_Recv(... from process 0 ...);
    MPI_Send(... to process 0 ...);
}
```

- Process 0 sends first.
- Process 1 is already in receive mode, so it accepts the message immediately.
- This breaks the circular wait — communication proceeds without blocking indefinitely.

Finalization

`MPI_Finalize();`

- Cleans up the MPI environment for all processes.

Example Output

Deadlock (DEADLOCK_PART =

1): text

Process 0 waiting to receive from Process 1...

Process 1 waiting to receive from Process 0...

// Program hangs indefinitely

Deadlock Avoidance (DEADLOCK_PART =

2): text

Process 0 received back number: 123

Process 1 received and sent back number: 123

PROGRAM:

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

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

```
int main(int argc, char** argv) {
```

```
    int rank, size;
```

```
    int msg = 0;
```

```
    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("This program needs at least 2 processes.\n");
```

```
        MPI_Finalize();
```

```
        return 0;
```

```
    }
```

```
    // ===== DEADLOCK SCENARIO =====
```

```
    if (rank == 0) {
```

```
        printf("Process 0: Sending message to Process 1 (will deadlock)...\n");
```

```
        MPI_Send(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

```
        MPI_Recv(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
        printf("Process 0: Received message from Process 1\n");
```

```
    }
```

```
    else if (rank == 1) {
```

```
        printf("Process 1: Sending message to Process 0 (will deadlock)...\n");
```

```
        MPI_Send(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

```
        MPI_Recv(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
        printf("Process 1: Received message from Process 0\n");
```

```
    }
```

```
    MPI_Barrier(MPI_COMM_WORLD); // Sync before avoiding deadlock
```

```

if (rank == 0) printf("\n--- Avoiding Deadlock by Changing Call Order ---\n");

// ===== DEADLOCK AVOIDANCE =====
if (rank == 0) {
printf("Process 0: Sending message to Process 1 (safe)...\n");
MPI_Send(&msg, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
MPI_Recv(&msg, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Process 0: Received message from Process 1 safely\n");
}
else if (rank == 1) {
printf("Process 1: Receiving message from Process 0 (safe)...\n");
MPI_Recv(&msg, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Send(&msg, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
printf("Process 1: Sent message back to Process 0 safely\n");
}

MPI_Finalize();
return 0;
}

```

OUTPUT

mpicc 6.c -o 6

mpirun -np 2 ./6

```

student@student-ThinkCentre-M75t-Gen-5: ~/Desktop
File Edit View Search Terminal Help
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$ mpicc 6.c -o 6
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$ mpirun -np 2 ./6
Process 1: Sending message to Process 0 (will deadlock)...
Process 0: Sending message to Process 1 (will deadlock)...
Process 1: Received message from Process 0
Process 1: Receiving message from Process 0 (safe)....
Process 0: Received message from Process 1
Process 0: Receiving message from Process 1 (safe)...
Process 0: Received message from Process 1

--- Avoiding Deadlock by Changing Call Order ---
Process 0: Sending message to Process 1 (safe)...
Process 0: Received message from Process 1 safely
Process 1: Sent message back to Process 0 safely
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$

```

Program 07: Write a MPI Program to demonstration of Broadcast operation.**INTRODUCTION**

MPI (Message Passing Interface) is a key model for programming distributed memory parallel systems. Among the collective communication operations that MPI provides, broadcast is fundamental.

- Broadcast Operation: This involves one process (called the root) sending the same data to all other processes in a communicator.
- The function used is `MPI_Bcast()`, which ensures that all processes receive a consistent copy of the data from the root.
- Broadcast occurs collectively: all processes in the group must call `MPI_Bcast` for the operation to complete.
- It is highly useful in parallel programs to distribute common input data (e.g., parameters, configuration values) from one process to all others.
- Synchronization: `MPI_Bcast` acts as a synchronization point because all processes must reach the call before proceeding.
- Efficient broadcast algorithms use tree-based communication structures internally to optimize network usage and minimize latency rather than naive one-to-all sends.
- Broadcast is one of several collective communication primitives. Collectives also include gather, scatter, reduce, allreduce, and barrier.

By studying `MPI_Bcast`, VTU students learn how data distribution works in parallel applications and how communication patterns scale across multiple processes.

PROGRAM FLOW

- Initializes MPI and retrieves:
 - The rank (unique ID) of the process.
 - The size (total number of processes).
- The process with rank 0 (the root) sets an integer number = 50.
- The root process prints a message that it is broadcasting the number.
- All processes, including the root, call `MPI_Bcast`, which distributes the value of number from process 0 to all other processes.
- Each process then prints the number it received.
- Finalizes MPI and exits.

Thus, the program demonstrates broadcasting a single integer value from one process to all others and how every process can access this shared data.

APPLICATIONS AND ADVANTAGES**Applications**

- Distributing Input Data: Many parallel algorithms require initial input or configuration data to be available in every process.
- Synchronization and Data Sharing: Ensures consistent replication of data across all processes.
- Scientific Simulations: Parameter settings (e.g., boundary conditions, physical constants) broadcast to all compute nodes.
- Parallel File I/O Preparations: Broadcast metadata to all processes.
- Collective Operations: Generally used to implement higher-level parallel algorithms that require sharing of common data.

Advantages

- Efficiency: Implementation uses optimized algorithms rather than multiple point-to-point sends.
- Simplicity: One call manages data distribution to all processes.
- Synchronization: Ensures all processes have received the data before continuing.
- Scalability: Works efficiently even with hundreds or thousands of processes.
- Reduced Programmer Effort: Simplifies complex communication patterns in distributed programs.

EXPLANATION OF THE PROGRAM

MPI Initialization

```
MPI_Init(&argc,&argv);          MPI_Comm_rank(MPI_COMM_WORLD,      Crank);
MPI_Comm_size(MPI_COMM_WORLD, Csize);
```

- Set up the MPI environment.
- Determine rank (process ID).
- Determine total number of processes size.

Set Data in Root Process

```
if (rank == 0) {
    number = 50;
    printf("Process %d broadcasting number %d to all other processes.\n", rank, number);
}
```

- Root process (rank == 0) initializes variable number with the value 50.
- Prints broadcasting message.

Broadcast Operation

```
MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- All processes call this function.
- Parameters:
 - Cnumber: Address of the data to broadcast.
 - 1: Count of elements.
 - MPI_INT: Data type.
 - 0: Root process rank (the sender).
 - MPI_COMM_WORLD: Communicator including all processes.
- This call ensures every process's number variable contains the value 50 after it returns.

Print Received Value

```
printf("Process %d received number %d\n", rank, number);
```

- Every process prints its rank and the received number.

Finalization

```
MPI_Finalize();
```

Cleans up MPI environment resources.

Example output with 4 processes:

text

Process 0 broadcasting number 50 to all other processes.

Process 0 received number 50

Process 1 received number 50

Process 2 received number 50

Process 3 received number 50

In summary, this MPI program demonstrates the broadcast collective communication operation, a widely used technique in parallel computing to efficiently distribute data from one process (root) to all others within a communicator.

PROGRAM:

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

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

    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 (rank == 0) {
        printf("Process 0: Enter a number to broadcast: ");
        scanf("%d", &number);
    }
    // Broadcasting the number from Process 0 to all other processes
    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);

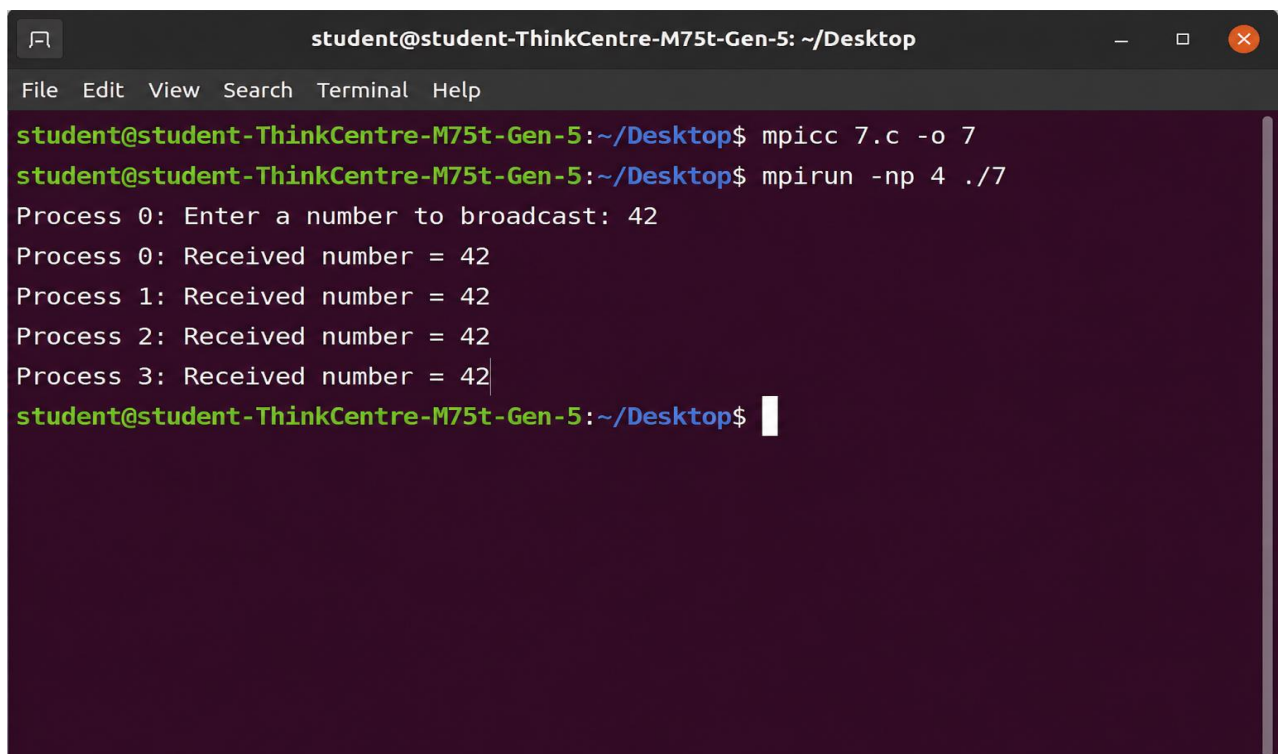
    printf("Process %d: Received number = %d\n", rank, number);

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

OUTPUT

mpicc 7.c -o 7

mpirun -np 4 ./7



```
student@student-ThinkCentre-M75t-Gen-5: ~/Desktop
File Edit View Search Terminal Help
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$ mpicc 7.c -o 7
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$ mpirun -np 4 ./7
Process 0: Enter a number to broadcast: 42
Process 0: Received number = 42
Process 1: Received number = 42
Process 2: Received number = 42
Process 3: Received number = 42
student@student-ThinkCentre-M75t-Gen-5:~/Desktop$
```

Program 08: Write a MPI Program demonstration of MPI_Scatter and MPI_Gather.**INTRODUCTION**

MPI (Message Passing Interface) is a key model for programming distributed memory parallel systems. Among the collective communication operations that MPI provides, broadcast is fundamental.

- Broadcast Operation: This involves one process (called the root) sending the same data to all other processes in a communicator.
- The function used is `MPI_Bcast()`, which ensures that all processes receive a consistent copy of the data from the root.
- Broadcast occurs collectively: all processes in the group must call `MPI_Bcast` for the operation to complete.
- It is highly useful in parallel programs to distribute common input data (e.g., parameters, configuration values) from one process to all others.
- Synchronization: `MPI_Bcast` acts as a synchronization point because all processes must reach the call before proceeding.
- Efficient broadcast algorithms use tree-based communication structures internally to optimize network usage and minimize latency rather than naive one-to-all sends.
- Broadcast is one of several collective communication primitives. Collectives also include gather, scatter, reduce, allreduce, and barrier.

PROGRAM FLOW

- Initializes MPI and retrieves:
 - The rank (unique ID) of the process.
 - The size (total number of processes).
- The process with rank 0 (the root) sets an integer number = 50.
- The root process prints a message that it is broadcasting the number.
- All processes, including the root, call `MPI_Bcast`, which distributes the value of number from process 0 to all other processes.
- Each process then prints the number it received.
- Finalizes MPI and exits.

Thus, the program demonstrates broadcasting a single integer value from one process to all others and how every process can access this shared data.

APPLICATIONS AND ADVANTAGES**Applications**

- Distributing Input Data: Many parallel algorithms require initial input or configuration data to be available in every process.
- Synchronization and Data Sharing: Ensures consistent replication of data across all processes.
- Scientific Simulations: Parameter settings (e.g., boundary conditions, physical constants) broadcast to all compute nodes.
- Parallel File I/O Preparations: Broadcast metadata to all processes.
- Collective Operations: Generally used to implement higher-level parallel algorithms that require sharing of common data.

Advantages

- Efficiency: Implementation uses optimized algorithms rather than multiple point-to-point sends.
- Simplicity: One call manages data distribution to all processes.

- Synchronization: Ensures all processes have received the data before continuing.
- Scalability: Works efficiently even with hundreds or thousands of processes.
- Reduced Programmer Effort: Simplifies complex communication patterns in distributed programs.

EXPLANATION OF THE PROGRAM

MPI Initialization

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

- Set up the MPI environment.
- Determine rank (process ID).
- Determine total number of processes size.

Set Data in Root Process

```
if (rank == 0) {
    number = 50;
    printf("Process %d broadcasting number %d to all other processes.\n", rank, number);
}
```

- Root process (rank == 0) initializes variable number with the value 50.
- Prints broadcasting message.

Broadcast Operation

```
MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- All processes call this function.
- Parameters:
 - Cnumber: Address of the data to broadcast.
 - 1: Count of elements.
 - MPI_INT: Data type.
 - 0: Root process rank (the sender).
 - MPI_COMM_WORLD: Communicator including all processes.
- This call ensures every process's number variable contains the value 50 after it returns.

Print Received Value

```
printf("Process %d received number %d\n", rank, number);
```

- Every process prints its rank and the received number.

Finalization

```
MPI_Finalize();
```

- Cleans up MPI environment resources.

Example output with 4 processes:

Process 0 broadcasting number 50 to all other processes.

Process 0 received number 50

Process 1 received number 50

Process 2 received number 50

Process 3 received number 50

In summary, this MPI program demonstrates the broadcast collective communication operation, a widely used technique in parallel computing to efficiently distribute data from one process (root) to all others within a communicator. It exemplifies key parallel computing topics such as collective communication, synchronization, and data distribution.

PROGRAM:

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

int main(int argc, char** argv) {
    int rank, size;
    int send_data[100], recv_data;
    int gathered_data[100];

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

    if (size > 100) {
        if (rank == 0)
            printf("This demo supports up to 100 processes only.\n");
        MPI_Finalize();
        return 0;
    }

    // Prepare data in root process
    if (rank == 0) {
        printf("Root process preparing data...\n");
        for (int i = 0; i < size; i++) {
            send_data[i] = i + 1; // Example: 1, 2, 3, ...
        }
    }

    // Scatter: Distribute one element of send_data to each process
    MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d received data: %d\n", rank, recv_data);

    // Each process performs a computation (multiply by 2)
    recv_data *= 2;

    // Gather: Collect processed data from all processes into gathered_data array in root
    MPI_Gather(&recv_data, 1, MPI_INT, gathered_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

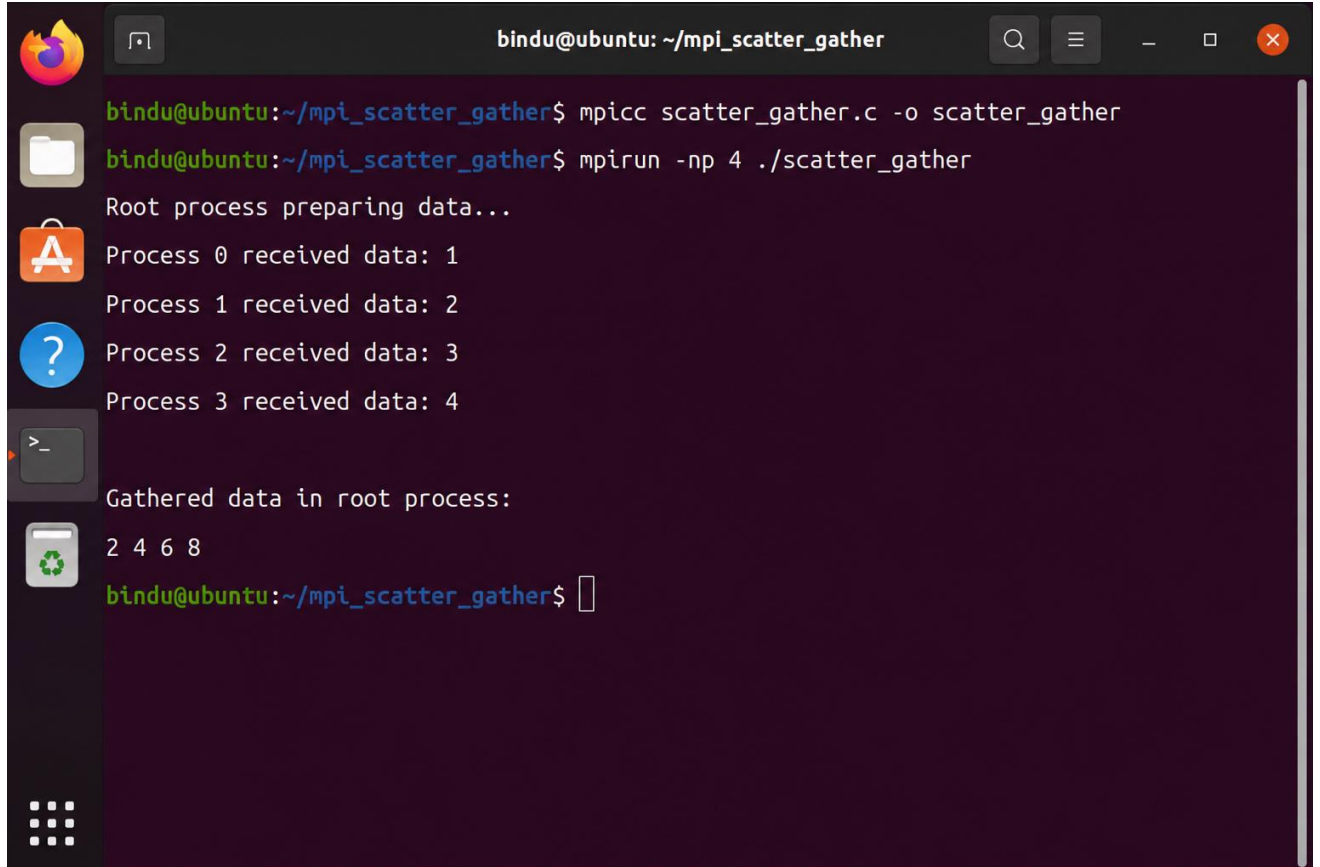
    if (rank == 0) {
        printf("\nGathered data in root process:\n");
        for (int i = 0; i < size; i++) {
            printf("%d ", gathered_data[i]);
        }
        printf("\n");
    }

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

OUTPUT

```
mpicc scatter_gather.c -o scatter_gather
```

```
mpirun -np 4 ./ scatter_gather
```

A terminal window titled 'bindu@ubuntu: ~/mpi_scatter_gather' showing the execution of a scatter-gather program. The user enters 'mpicc scatter_gather.c -o scatter_gather' and then 'mpirun -np 4 ./scatter_gather'. The output shows the root process preparing data, followed by four processes receiving data (1, 2, 3, 4). Finally, the root process gathers the data and prints '2 4 6 8'.

```
bindu@ubuntu:~/mpi_scatter_gather$ mpicc scatter_gather.c -o scatter_gather
bindu@ubuntu:~/mpi_scatter_gather$ mpirun -np 4 ./scatter_gather
Root process preparing data...
Process 0 received data: 1
Process 1 received data: 2
Process 2 received data: 3
Process 3 received data: 4
Gathered data in root process:
2 4 6 8
bindu@ubuntu:~/mpi_scatter_gather$
```

Program 09: Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD).**INTRODUCTION**

MPI collective communication operations are an essential topic for parallel programming on distributed memory systems. Among these, reduction operations are commonly used to combine data distributed across processes into a single result or distributed results.

- **MPI_Reduce:** A collective operation where data from all processes in a communicator is combined using a specified operation (like sum, product, max, min), and the result is stored only on the root process.
- **MPI_Allreduce:** Similar to MPI_Reduce, but the combined result is distributed back to all processes, so every process gets the final result instead of just the root.

Reduction Operations Used:

- **MPI_SUM:** Sum of all values.
- **MPI_PROD:** Product of all values.
- **MPI_MAX:** Maximum value.
- **MPI_MIN:** Minimum value.

These operations allow aggregation of data computed in parallel, fundamental for algorithms like parallel prefix sums, parallel statistics, and synchronization of partial results.

Key concepts:

- Collective communication semantics.
- Synchronization across processes during reduction.
- Reduction algorithms that minimize communication overhead.
- Usage patterns and correctness ensuring all processes participate.

PROGRAM FLOW

- Each MPI process computes a local value equal to its rank plus 1 (value = rank + 1).
- The program uses MPI_Reduce to compute four reductions on all local values: sum, product, max, and min, with the root process (rank 0) collecting and printing these results.
- It also uses MPI_Allreduce for the same four operations. In this case, the resulting values are shared and available on *all* processes.
- Each process then prints its own value along with the aggregation results obtained via the MPI_Allreduce call.

This illustrates how collective reductions can be used to combine distributed data both to a single root and to all processors.

APPLICATIONS AND ADVANTAGES

Applications

- **Parallel Statistical Computations:** Calculating sums, maxima, minima, and products across distributed datasets.
- **Scientific Simulations:** Combining partial results like total energy, max pressure, or min temperature from multiple processors.
- **Data Analytics:** Aggregating distributed results without gathering all raw data to one node.
- **Synchronization and Decision Making:** Where all processes need a unified view (using MPI_Allreduce).

Advantages

- Efficiency: MPI implementations optimize reduction operations using tree-based algorithms minimizing communication latency.
- Simplicity: Abstract complex data combining patterns into single function calls.
- Scalability: Used extensively in HPC to reduce large-scale distributed data efficiently.
- Flexibility: Supports multiple operations and data types.
- Broad Support: One of the foundational MPI collectives, ensuring compatibility across platforms.

EXPLANATION OF THE PROGRAM

MPI Initialization

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

- Initializes MPI.
- Obtains the rank (process ID) and total number of processes.

Local Computation

```
value = rank + 1;
```

- Each process sets its local value to its rank plus 1.

MPI_Reduce Calls

```
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);
```

- These calls reduce value across all processes using sum, product, max, and min operations, respectively.
- Results are stored only on the root process (rank 0).

MPI_Allreduce Calls

```
MPI_Allreduce(&value, &all_sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&value, &all_prod, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
MPI_Allreduce(&value, &all_max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&value, &all_min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
```

- Similar to MPI_Reduce, but results are available in all processes.
- Every process gets all_sum, all_prod, all_max, and all_min.

Output

```
if (rank == 0) {
    printf("--- Results using MPI_Reduce (only root prints) ---\n");
    printf("Sum   = %d\n", sum);
    printf("Product = %d\n", prod);
    printf("Max    = %d\n", max);
    printf("Min   = %d\n", min);
}
printf("Process %d has value %d\n", rank, value);
printf("Process %d sees (MPI_Allreduce): Sum=%d, Prod=%d, Max=%d, Min=%d\n",
       rank, all_sum, all_prod, all_max, all_min);
    • Root process prints results from MPI_Reduce.
    • Every process prints its own value and the results from MPI_Allreduce.
```

MPI Finalization

MPI_Finalize();

- Cleans up MPI environment.

Example output for 4 processes (ranks 0 to 3):

--- Results using MPI_Reduce (only root prints) ---

Sum = 10

Product = 24

Max = 4

Min = 1

Process 0 has value 1

Process 0 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1

Process 1 has value 2

Process 1 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1

Process 2 has value 3

Process 2 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1

Process 3 has value 4

Process 3 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1

In summary, this program demonstrates the use of MPI collective communication routines MPI_Reduce and MPI_Allreduce for common reduction operations (sum, product, max, min) in distributed parallel computing.

PROGRAM:

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

int main(int argc, char** argv) {
    int rank, size;
    int value;
    int result_reduce, result_allreduce;

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

    // Each process has a unique value
    value = rank + 1; // Process 0 = 1, Process 1 = 2, ...

    // 1. MPI_Reduce - SUM
    MPI_Reduce(&value, &result_reduce, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("MPI_Reduce SUM result: %d\n", result_reduce);
    }

    // 2. MPI_Reduce - MAX
    MPI_Reduce(&value, &result_reduce, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("MPI_Reduce MAX result: %d\n", result_reduce);
    }
}
```

```
}

// 3. MPI_Reduce - MIN
MPI_Reduce(&value, &result_reduce, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
if (rank == 0) {
printf("MPI_Reduce MIN result: %d\n", result_reduce);
}

// 4. MPI_Reduce - PRODUCT
MPI_Reduce(&value, &result_reduce, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
if (rank == 0) {
printf("MPI_Reduce PRODUCT result: %d\n", result_reduce);
}

// --- MPI_Allreduce performs the same but result is available to ALL processes ---

// 5. MPI_Allreduce - SUM
MPI_Allreduce(&value, &result_allreduce, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
printf("Process %d: MPI_Allreduce SUM = %d\n", rank, result_allreduce);

// 6. MPI_Allreduce - MAX
MPI_Allreduce(&value, &result_allreduce, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
printf("Process %d: MPI_Allreduce MAX = %d\n", rank, result_allreduce);

// 7. MPI_Allreduce - MIN
MPI_Allreduce(&value, &result_allreduce, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
printf("Process %d: MPI_Allreduce MIN = %d\n", rank, result_allreduce);

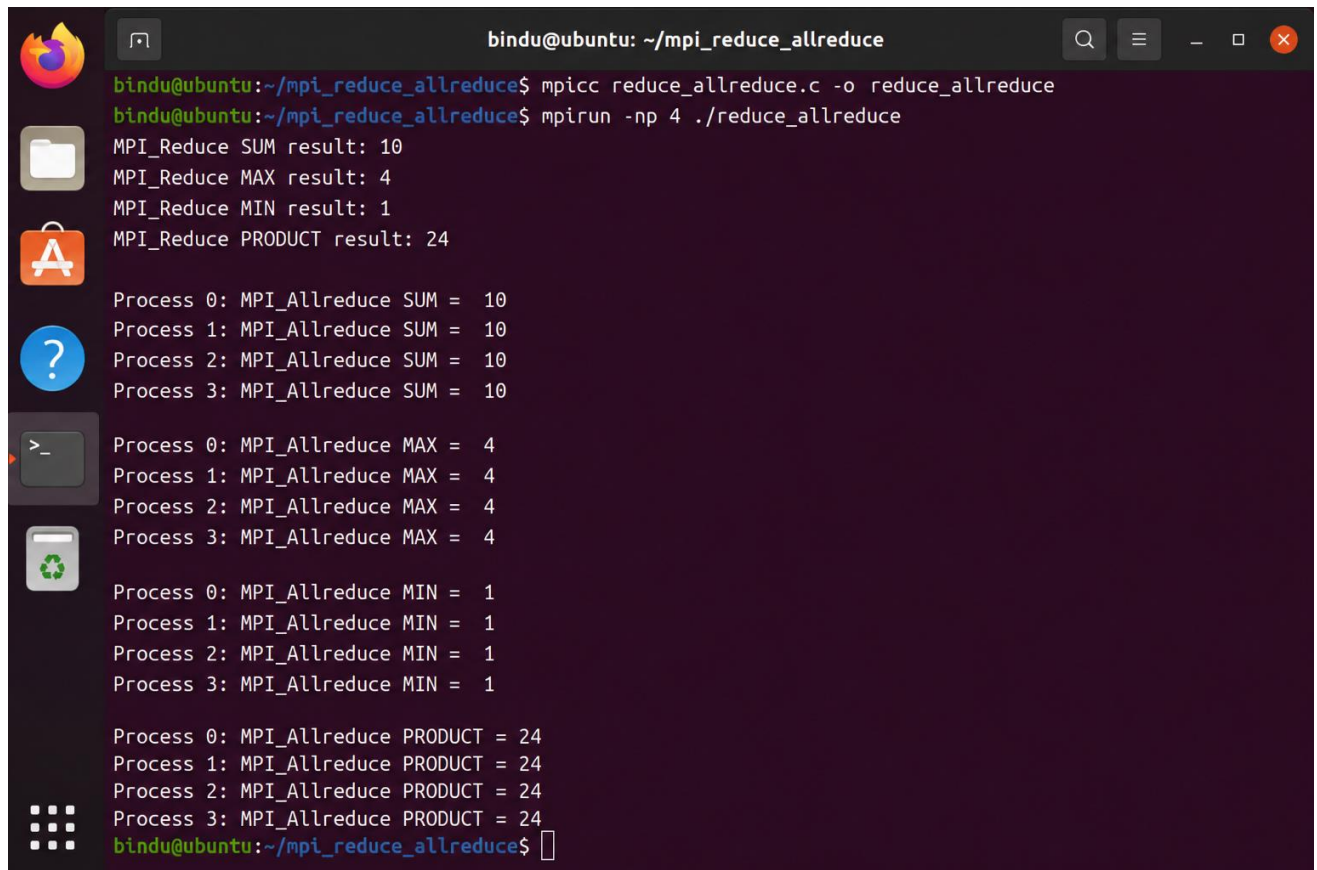
// 8. MPI_Allreduce - PRODUCT
MPI_Allreduce(&value, &result_allreduce, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
printf("Process %d: MPI_Allreduce PRODUCT = %d\n", rank, result_allreduce);

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

OUTPUT

mpicc reduce_allreduce.c -o reduce_allreduce

mpirun -np 4 ./ reduce_allreduce

A terminal window titled 'bindu@ubuntu: ~/mpi_reduce_allreduce' showing the compilation and execution of an MPI program. The user runs 'mpicc reduce_allreduce.c -o reduce_allreduce' and then 'mpirun -np 4 ./reduce_allreduce'. The output shows MPI_Reduce results (SUM: 10, MAX: 4, MIN: 1, PRODUCT: 24) and MPI_Allreduce results for four processes (SUM: 10, MAX: 4, MIN: 1, PRODUCT: 24).

```
bindu@ubuntu: ~/mpi_reduce_allreduce
bindu@ubuntu:~/mpi_reduce_allreduce$ mpicc reduce_allreduce.c -o reduce_allreduce
bindu@ubuntu:~/mpi_reduce_allreduce$ mpirun -np 4 ./reduce_allreduce
MPI_Reduce SUM result: 10
MPI_Reduce MAX result: 4
MPI_Reduce MIN result: 1
MPI_Reduce PRODUCT result: 24

Process 0: MPI_Allreduce SUM = 10
Process 1: MPI_Allreduce SUM = 10
Process 2: MPI_Allreduce SUM = 10
Process 3: MPI_Allreduce SUM = 10

Process 0: MPI_Allreduce MAX = 4
Process 1: MPI_Allreduce MAX = 4
Process 2: MPI_Allreduce MAX = 4
Process 3: MPI_Allreduce MAX = 4

Process 0: MPI_Allreduce MIN = 1
Process 1: MPI_Allreduce MIN = 1
Process 2: MPI_Allreduce MIN = 1
Process 3: MPI_Allreduce MIN = 1

Process 0: MPI_Allreduce PRODUCT = 24
Process 1: MPI_Allreduce PRODUCT = 24
Process 2: MPI_Allreduce PRODUCT = 24
Process 3: MPI_Allreduce PRODUCT = 24
bindu@ubuntu:~/mpi_reduce_allreduce$
```

VIVA QUESTIONS

1. OpenMP – Sequential vs Parallel Merge Sort

1. What is merge sort, and why is it suitable for parallelization?
2. How does OpenMP's section directive help in parallelizing merge sort?
3. What is the role of recursion in parallel merge sort?
4. How do you measure execution time in OpenMP?
5. Why might the parallel merge sort sometimes be slower than sequential for small arrays?
6. Explain how load balancing is handled in this program.
7. What is the difference between `#pragma omp section` and `#pragma omp parallel sections`?
8. Why is thread synchronization important in merge sort?

2. OpenMP – Static Scheduling with Chunk Size 2

1. What does `OMP_SCHEDULE=static,2` mean?
2. How does static scheduling differ from dynamic scheduling?
3. In your program, how are iterations assigned to threads?
4. If there are 4 threads and 8 iterations, how will the iterations be distributed?
5. What environment variable is used to set schedule type in OpenMP?
6. How do you retrieve the current thread ID in OpenMP?
7. When would dynamic scheduling be better than static scheduling?
8. What happens if chunk size is set to 1 instead of 2?

3. OpenMP – Fibonacci Numbers Using Tasks

1. What is the difference between `#pragma omp parallel for` and `#pragma omp task`?
2. How does task creation help in computing Fibonacci numbers?
3. Why is Fibonacci a poor benchmark for parallel computing?
4. What is the purpose of `#pragma omp taskwait`?
5. What is a "task dependency," and how can it be defined in OpenMP?
6. What is the difference between `shared` and `firstprivate` in task variables?
7. How can excessive task creation reduce performance?

4. OpenMP – Prime Numbers (Serial vs Parallel)

1. How do you check if a number is prime in your program?
2. Which OpenMP directive did you use to parallelize the loop?
3. How do you handle race conditions in counting primes?
4. What is the difference in execution time between serial and parallel versions? Why?
5. How do you ensure that each thread gets a different range of numbers?
6. What is false sharing, and could it occur in this program?
7. Why might parallelization overhead outweigh benefits for small `n`?

5. MPI – Send and Receive

1. What is the difference between `MPI_Send` and `MPI_Recv`?
2. What are the parameters of `MPI_Send`? Explain each.
3. How do you avoid deadlock in MPI point-to-point communication?
4. What is the difference between blocking and non-blocking send?
5. What is the role of `MPI_STATUS_IGNORE`?
6. Can `MPI_Send` and `MPI_Recv` be used for broadcasting? Why or why not?

6. MPI – Deadlock Demonstration and Avoidance

1. What is deadlock in MPI communication?
 2. How does altering the call sequence avoid deadlock?
 3. Can non-blocking communication prevent deadlock? How?
 4. What is the difference between synchronous and asynchronous sends?
 5. Give an example of a situation that can cause MPI deadlock.
 6. Why is it important to match MPI_Send and MPI_Recv calls correctly?
-

7. MPI – Broadcast

1. What is the purpose of MPI_Bcast?
 2. How does broadcasting differ from point-to-point communication?
 3. What is the root process in MPI_Bcast?
 4. Can all processes be the root in broadcast?
 5. How does MPI_Bcast improve performance compared to multiple sends?
 6. What happens if processes in MPI_Bcast have different buffer sizes?
-

8. MPI – Scatter and Gather

1. What is the difference between MPI_Scatter and MPI_Gather?
 2. How are data chunks distributed in MPI_Scatter?
 3. What is the significance of the root process in these operations?
 4. What happens if the number of elements is not divisible by the number of processes?
 5. Can MPI_Gather be used without a root process?
 6. What is the difference between MPI_Gather and MPI_Allgather?
-

9. MPI – Reduce and Allreduce

1. What is the purpose of MPI_Reduce?
2. Explain the difference between MPI_Reduce and MPI_Allreduce.
3. What operations are supported by MPI reduction functions?
4. How does MPI_SUM work in reduction?
5. What is the advantage of using MPI_Allreduce over separate MPI_Reduce and MPI_Bcast?
6. What happens if different processes use different operations in reduction?