(Approved by AICTE |Affiliated to VTU | Recognized by UGC with 2(f) & 12(B) status |Accredited by NBA and NAAC)

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

ACCREDITED BY NBA

VII SEMESTER

ACADEMIC YEAR 2025–2026[ODD]

PARALLEL COMPUTING

MVJ22IS72 LABORATORY MANUAL

NAM E OF THE STUDENT :

BRANCH :

UNIVERSITY SEAT NO. :

SEMESTER & SECTION :

BATCH :

Department of Information Science and Engineering

VISION OF THE INSTITUTE

Be an Institution of Excellence with International Standards

MISSION OF THE INSTITUTE

- Impart quality education, along with industrial exposure.
- Provide world-class facilities to undertake research activities relevant to industrial and professional needs.
- Promote Innovation, Entrepreneurship and Value-added education that is socially relevant, along with economic benefits.

VISION OF THE DEPARTMENT

To be recognized as a department of repute in Information Science and Engineering by adopting quality teaching learning process and impart knowledge to make students equipped with capabilities required for professional, Industrial and research areas to serve society.

MISSION OF THE DEPARTMENT

M1: Innovation and technically competent: To impart quality education in Information Science and Engineering by adopting modern teaching learning processes using innovation techniques that enable them to become technically competent.

M2: Competitive Software Professionals: Providing training Programs that bridges gap between industry and academia, to produce competitive software professionals.

M3: Personal and Professional growth: To provide scholarly environment that enables value addition to staff and students to achieve personal and profession growth.

Program Outcomes (PO):

PO1: Engineering Knowledge: Apply knowledge of mathematics, natural science, computing, engineering fundamentals and an engineering specialization as specified in WK1 to WK4 respectively to develop to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature and analyze

complex engineering problems reaching substantiated conclusions with consideration for sustainable development. (WK1 to WK4)

PO3: Design/Development of Solutions: Design creative solutions for complex engineering problems and design/develop systems/components/processes to meet identified needs with consideration for the public health and safety, whole-life cost, net zero carbon, culture, society and environment as required. (WK5)

PO4: Conduct Investigations of Complex Problems: Conduct investigations of complex engineering problems using research-based knowledge including design of experiments, modelling, analysis & interpretation of data to provide valid conclusions. (WK8).

PO5: Engineering Tool Usage: Create, select and apply appropriate techniques, resources and modern engineering & IT tools, including prediction and modelling recognizing their limitations to solve complex engineering problems. (WK2 and WK6) PO6: The Engineer and The World: Analyze and evaluate societal and environmental aspects while solving complex engineering problems for its impact on sustainability with reference to economy, health, safety, legal framework, culture and environment. (WK1, WK5, and WK7).

PO7: Ethics: Apply ethical principles and commit to professional ethics, human values, diversity and inclusion; adhere to national & international laws. (WK9)

PO8: Individual and Collaborative Team work: Function effectively as an individual, and as a member or leader in diverse/multi-disciplinary teams.

PO9: Communication: Communicate effectively and inclusively within the engineering community and society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations considering cultural, language, and learning differences

PO10: Project Management and Finance: Apply knowledge and understanding of engineering management principles and economic decision-making and apply these to one's own work, as a member and leader in a team, and to manage projects and in multidisciplinary environments.

PO11: Life-Long Learning: Recognize the need for, and have the preparation and ability for i) independent and life-long leaming ii) adaptability to new and emerging technologies and iii) critical thinking in the broadest context of technological change. (WK8)

Program Educational Objectives (PEOs):

PEO1: IT Proficiency :Graduates will excel as IT Experts with extensive knowledge to analyze and design solutions to Information Engineering problems.

PEO2: Social & moral principles: Graduates will work in a team,show case professionalism, ethical values expose themselves to current trends and become responsible Engineers.

PEO3: Higher education: Graduates will pursue higher studies with the sound knowledge of fundamental concepts and skills in basic sciences and IT disciplines.

Program Specific Outcomes (PSO):

PSO1. **Software professional expertise:** An ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, DBMS, and networking for efficient design of computer-based systems of varying complexity.

PSO2. **Core competence:** An ability to compete practically to provide solutions for real world problems with a broad range of programming language and open source platforms in various computing domains

Course outcomes (CO):

On the completion of this laboratory course, the students will be able to:

- Foundation knowledge in database concepts, technology and practice to groom students into well-informed database application developers.
- Strong practice in SQL programming through a variety of database problems.
- Develop database applications using front-end tools and back-end DBMS.
- Create, Update and query on the database.
- Demonstrate the working of different concepts of DBMS
- Implement, analyze and evaluate the project developed for an application.

SEMESTER – VII

Laboratory Code: MVJ22IS72 IA Marks: 50

Exam Marks: 50 Exam Hours: 03

Exp 1:	Familiarization with HPC programming paradigms: Single program multiple data (SPMD) & MPMD	2 hrs
Exp 2:	To interface Speeding up C/Fortran/Python programs: Vectorization; Compiler optimizations.	2 hrs
Exp 3:	Programming in Message Passing Interface (MPI): Point-to-point and collective communications; Parallel I/O; MPI for Python and C/Fortran.	2 hrs
Exp 4:	Programming in OpenMP.	2 hrs
Exp 5:	Programming GPUs using OpenACC.	2 hrs
Exp 6:	Programming GPUs using CuPy and CUDA	2 hrs
Exp 7:	Reduction clause in OpenMP	2 hrs
Exp 8:	Scheduling loops in OpenMP-odd even transposition sort	2 hrs
Exp 9:	Synchronization in OpenMp – Producer – Consumer problem	2 hrs
Exp 10:	OpenMP program for fork join model	2 hrs

Experiment no 1

Familiarization with HPC programming paradigms: Single program multiple data (SPMD) & MPMD

AIM:

To familiarize with HPC programming paradigms: Single program multiple data (SPMD) & MPMD

Explanation: #include <mpi.h>
This includes the MPI library header. mpi.h contains all necessary declarations for MPI functions and constants.
#include <stdio.h>: Standard C header for input/output operations (like printf).
int main(int argc, char** argv) { The main function is the entry point of the program. argc and argv are command-line arguments. They're passed to MPI for internal configuration.
int rank, size; Declares two integers: rank: The unique ID of the process (from 0 to size - 1).
size: Total number of processes running the program.
MPI_Init(&argc, &argv);
Initializes the MPI environment. This must be called before any other MPI function.
It sets up communication between all processes. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
Gets the rank (ID) of the current process within the default communicator (MPI_COMM_WORLD).
Each process will get a different rank from 0 up to size - 1 MPI_Comm_size(MPI_COMM_WORLD, &size); Gets the total number of processes involved in MPI_COMM_WORLD. This is the same for every process. printf("Hello from process %d of %d\n", rank, size); Each process prints: Its own rank (rank)
Total number of processes (size) MPI_Finalize(); Cleans up the MPI environment. No MPI function should be called after this. return 0; } Standard return statement indicating the program completed successfully.

Concept	Description
MPI	A standard for parallel programming, especially in distributed memory systems.
Process	A separate instance of a running program. Each has its own memory space.
Rank	Unique ID of each process.
MPI_COMM_WORLD	Default communicator containing all processes.
Parallel Execution	This program runs the same code in multiple processes at the same time.

Objective: Understand SPMD and MPMD models.

Description: Simple examples demonstrating parallel execution with SPMD and MPMD using pseudo code or basic MPI/OpenMP.
Understand the difference between SPMD (Single Program Multiple Data) and MPMD (Multiple Program Multiple Data) using simple MPI programs.

SPMD (Single Program Multiple Data)
In SPMD, all processors execute the same program, but work on different pieces of data.

SPMD Program in C using MPI
sudo apt install lam4-dev
sudo apt install libmpich-dev
sudo apt install libopenmpi-dev

Program Code: // spmd_example.c

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
   int rank, size;
   MPI_Init(&argc, &argv);              // Initialize MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get process rank
   MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get total number of processes
   printf("Hello from process %d of %d\n", rank, size);
   MPI_Finalize(); // Finalize MPI
   return 0;
}
```

Execution Steps (SPMD)

Compile the program:
mpicc spmd_example.c -o spmd_example
Run the program with 2 processes:
mpirun -np 2 ./spmd_example

Sample Output (SPMD)

Hello from process 0 of 4
Hello from process 1 of 4

Explanation (SPMD)

All 2 processes run the same binary.

Each gets a unique rank and performs the same code, but can differentiate behavior using rank.

MPMD (Multiple Program Multiple Data)

In MPMD, different programs (executables) are assigned to different processors.

Program Code: Program 1: master.c

```
// master.c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
   int rank;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   printf("I am MASTER process %d\n", rank);
   MPI_Finalize();
   return 0;
}
```

Program Code: Program 2: worker.c

```
// worker.c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
   int rank;
   MPI_Init(&argc, &argv);        //Step 1: Initialize MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);      // Step 2: Get process rank
   printf("I am WORKER process %d\n", rank);            //Step 3: Print message
   MPI_Finalize();                //Step 4: Finalize MPI
   return 0;
}
```

Execution Steps (MPMD)

Compile both programs:

```
mpicc master.c -o master
mpicc worker.c -o worker
```

Run using MPMD layout:

```
mpirun -np 1 ./master : -np 1 ./worker
```

Output (MPMD)

```
I am MASTER process 0
I am WORKER process 1
```

Explanation (MPMD)

One process executes master, three execute worker.

Different programs are used, suited for master-worker models or heterogenous task roles.

#include <mpi.h> Includes the MPI library needed for parallel programming.

#include <stdio.h> Standard C library for input/output functions like printf.

MPI_Init Starts the MPI environment — must be called before any MPI functions.

MPI_Comm_rank Gets the current process's unique rank (an ID between 0 and n-1).

printf(...) Prints "I am MASTER process X" where X is the rank.

MPI_Finalize Ends the MPI environment — all MPI programs must call this before exiting.

END

Experiment no 2

To interface Speeding up C/Fortran/Python programs: Vectorization; Compiler optimizations.

AIM: Speeding Up C/Fortran/Python Programs, Implement vectorization and apply compiler optimizations.

Explanation : It shows how fast vectorized operations (using NumPy) can be when working with large arrays, compared to traditional for-loops. It adds two large arrays element by element using NumPy's vectorized addition, which is much faster.

import numpy as np

This line imports the NumPy library, which is used for fast numerical operations, especially with large arrays or matrices.

import time

Imports the time module to measure how long the array addition takes.

Creating large arrays

N = 10_000_000

We are defining the size of the arrays — 10 million elements! This is to test performance on big data.

a = np.random.rand(N)

b = np.random.rand(N)

Creates two arrays a and b of size 10 million each.

Each element is a random floating-point number between 0 and 1.

Vectorized addition using NumPy

start = time.time()

c = a + b

end = time.time()

Starts a timer before the operation using start = time.time().

c = a + b: This is vectorized addition — NumPy adds each element of array a to the corresponding element in array b in one efficient step (no loop needed).

Stops the timer using end = time.time().

print("Time taken (Vectorized):", end - start, "seconds")

Prints how long the vectorized addition took.

print("First 5 elements of result:", c[:5])

Displays the first 5 elements of the result array c, so we can see the output without printing all 10 million numbers.

Vectorized operations with NumPy are extremely fast and efficient, especially for large data.

This program demonstrates performance measurement using time.time().

Using libraries like NumPy avoids the need for slow loops and makes code both cleaner and faster.

Vectorization in Python (with NumPy)

Program Code: Example Program a) : Element-wise Array Addition

```
$] gedit ex2.py
import numpy as np
import time
# Creating large arrays
N = 10_000_000
a = np.random.rand(N)
b = np.random.rand(N)
# Vectorized addition using NumPy
start = time.time()
c = a + b
```

```
end = time.time()
print("Time taken (Vectorized):", end - start, "seconds")
print("First 5 elements of result:", c[:5])
```

Compilation: python3 ex2.py

Output:

Time taken (Vectorized): 0.042978763580322266 seconds

First 5 elements of result: [0.61763085 1.09881922 1.07841423 0.51643988 1.03757299]

Program b) Non-vectorized Version (Loop-based)

Explanation:

shows how long it takes to add two large arrays element by element using a for loop in Python. It helps students understand why vectorized operations (like in NumPy) are preferred for large data.

import numpy as np

Imports the NumPy library to generate random arrays.

import time

Imports the time module to measure how long the addition process takes.

Creating large arrays

N = 10_000_000

Sets N to 10 million. This means we are working with very large arrays (to test performance).

a = np.random.rand(N)

b = np.random.rand(N)

Creates two arrays a and b, each with 10 million random values between 0 and 1.

c = np.zeros(N)

Creates a result array c filled with zeros, which will store the sum of a[i] + b[i] for each index i.

Loop-based addition

```
start = time.time()
for i in range(N):
    c[i] = a[i] + b[i]
end = time.time()
```

Starts the timer before the loop with start = time.time().

A for loop goes through each index from 0 to 9,999,999.

At each step, it adds a[i] + b[i] and stores the result in c[i].

Stops the timer with end = time.time() after the loop finishes.

print("Time taken (Loop-based):", end - start, "seconds")

Prints how long the loop took to perform the addition.

print("First 5 elements of result:", c[:5])

Prints the first 5 values of the result array c to show that addition was done correctly.

Key Takeaways for Students:

A simple for loop works correctly but is very slow for large arrays (like 10 million elements).

This approach processes one element at a time, which takes a lot of time in Python.

This program helps you see the performance difference between loop-based and vectorized operations (like in NumPy).

Comparison:

Feature	Loop-Based	NumPy Vectorized
Code simplicity	Longer (loop needed)	Shorter (just a + b)
Speed (for large N)	Slow	Very Fast
Performance efficiency	Low (due to Python overhead)	High (due to optimized C code)

Program Code: b) $]gedit loop_add.py

```
import numpy as np
import time
# Creating large arrays
N = 10_000_000
a = np.random.rand(N)
b = np.random.rand(N)
c = np.zeros(N)
# Loop-based addition
start = time.time()
for i in range(N):
    c[i] = a[i] + b[i]
end = time.time()
print("Time taken (Loop-based):", end - start, "seconds")
print("First 5 elements of result:", c[:5])
```

Compilation: python3 loop_add.py

Output: Time taken (Loop-based): 8.661064147949219 seconds

First 5 elements of result: [0.43177569 1.33515382 0.15372426 1.64221627 0.76275284]

Program c) Ubuntu or Linux : Compiler Optimizations in C

Explanation of ex1.c : demonstrates how to add two large arrays (vectors) in C using a for loop and measure how long the addition takes.

How arrays are dynamically allocated in C, How random numbers are generated

How to measure execution time in C, The importance of memory management

#include <stdio.h>

Includes the standard input/output library to use printf() for displaying output.

#include <stdlib.h>

Includes the standard library for dynamic memory allocation (malloc) and random number generation (drand48()).

#include <time.h>

Needed to measure how long the operation takes using clock().

#define N 100000000

Defines N as 100 million — the number of elements in each array. This simulates a large data problem.

int main() {

The main function where the program starts executing.

1. Memory Allocation:

double *a = malloc(N * sizeof(double));

double *b = malloc(N * sizeof(double));

double *c = malloc(N * sizeof(double));

Dynamically allocates memory for three large arrays: a, b, and c, each with N elements.

malloc returns a pointer to a block of memory large enough to hold N double values.

2. Filling Arrays with Random Values:

```
for (int i = 0; i < N; i++) {
    a[i] = drand48();
    b[i] = drand48();
}
```

Fills arrays a and b with random floating-point numbers between 0 and 1 using drand48().

3. Measuring Time for Addition:

```
clock_t start = clock();
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
clock_t end = clock();
```

clock() records the CPU time before and after the addition loop.

The loop adds corresponding elements of a and b and stores the results in c.

4. Output the Result:

```
printf("Time taken: %f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);
```

Calculates and prints the time taken in seconds.

CLOCKS_PER_SEC is a constant that represents the number of clock ticks per second.

```
printf("First 5: %f %f %f %f %f\n", c[0], c[1], c[2], c[3], c[4]);
```

Prints the first 5 elements of the result array c to check the output.

5. Freeing the Memory: free(a); free(b); free(c);

Frees the dynamically allocated memory to avoid memory leaks.

Concept	Explanation
Dynamic memory	Used malloc to allocate big arrays at runtime.
Random numbers	Used drand48() to fill arrays with values between 0 and 1.
Loop-based addition	Manually added each element from arrays a and b into c.
Time measurement	Used clock() to measure performance.
Memory management	Used free() to release the memory when done.

This is a low-level approach where you have full control over memory and performance.

While it's fast, it requires more careful memory management than in Python or NumPy.

Program Code: $] **gedit ex1.c**

```
 #include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 100000000
int main() {
        double *a = malloc(N * sizeof(double));
         double *b = malloc(N * sizeof(double));
         double *c = malloc(N * sizeof(double));
         for (int i = 0; i < N; i++)
                {
            a[i] = drand48();
            b[i] = drand48();
```

```
                }
               clock_t start = clock();
                for (int i = 0; i < N; i++) {
                c[i] = a[i] + b[i];
                  }
                clock_t end = clock();
        printf("Time taken: %f seconds\n", (double)(end - start) /CLOCKS_PER_SEC);
                 printf("First 5: %f %f %f %f %f\n", c[0], c[1], c[2], c[3], c[4]);
                 free(a); free(b); free(c);
                 return 0;
}
```

Output:

Without optimization flags:

```
kalyani@DESKTOP-9HEJFCC:~/lab2$ gcc ex1.c -o ex1
kalyani@DESKTOP-9HEJFCC:~/lab2$ ./ex1
Time taken: 1.453125 seconds
First 5: 0.000985 0.218274 0.455933 0.579515 0.981184
```

With optimization flags:

```
kalyani@DESKTOP-9HEJFCC:~/lab2$ gcc -O3 -march=native -ffast-math ex1.c -o ex1
kalyani@DESKTOP-9HEJFCC:~/lab2$ ./ex1
Time taken: 1.125000 seconds
First 5: 0.000985 0.218274 0.455933 0.579515 0.981184
```

Experiment Number - 3:

Programming in Message Passing Interface (MPI): Point-to-point and collective communications; Parallel I/O; MPI for Python and C/Fortran.

Message Passing Interface (MPI) Programming

Objective: Use MPI for inter-process communication.
Tasks:
- Point-to-point: Send/Receive
- Collective: Broadcast, Scatter, Gather, Reduce
- Parallel I/O
- MPI4Py and MPI in C/Fortran examples

```
sudo apt install -y mpich gfortran python3-pip
pip3 install mpi4py
# Compile C programs
mpicc program.c -o program
mpirun -np <n> ./program
# Run Python programs
mpirun -np <n> python3 program.py
```

Overview of MPI Topics

Point-to-Point Communication (Send/Recv)

Collective Communication (Broadcast, Scatter, Gather, Reduce)

Parallel I/O using MPI

MPI Programs in Python (mpi4py) and C

Step 1: Install MPI Libraries on Ubuntu

Open a terminal and run the following:

In Ubuntu terminal:

```
sudo apt update
sudo apt install -y mpich gfortran python3-pip
pip3 install mpi4py
```

lab3a.c program Explanation:

This program demonstrates basic communication between two processes using MPI.
Purpose of the Program:
To send a number from one process to another using point-to-point communication in MPI (used in parallel computing environments like clusters or supercomputers).
It runs on two processes.
Process 0 sends the number 42 to Process 1.

Process 1 receives the number and prints it.

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

Includes the MPI library and the standard input/output library.
mpi.h provides all the necessary functions and constants for MPI programs.

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

Main function that accepts command-line arguments — required by MPI to initialize properly.

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

Initializes the MPI environment.
This must be the first MPI call in any MPI program.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Gets the rank (ID) of the current process.
Each process is assigned a unique rank (starting from 0).
MPI_COMM_WORLD is the default group that includes all processes.

```
int number;
```

Declares a variable to hold the number to be sent or received.

```
if (rank == 0) {
  number = 42;
MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
```

If the current process is rank 0:
It sets number = 42.
Then it sends this number to Process 1 using MPI_Send.
&number – the address of the variable to send
1 – number of elements
MPI_INT – data type
1 – destination process rank
0 – tag (used to identify message type)
MPI_COMM_WORLD – communicator (group of processes)

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

If the current process is rank 1:
It receives the number sent by process 0 using MPI_Recv.
&number – where to store received data
1 – number of elements
MPI_INT – data type
0 – source process rank
0 – tag
MPI_COMM_WORLD – communicator
MPI_STATUS_IGNORE – we ignore the status of the message
Then, it prints the received number.

```
MPI_Finalize();
```

Finalizes the MPI environment. No MPI calls can be made after this.

```
return 0;
```

Ends the program successfully.

Concept	Meaning
MPI_Init	Starts MPI program
MPI_Comm_rank	Gets the current process's ID (rank)
MPI_Send	Sends data from one process to another
MPI_Recv	Receives data from another process
MPI_Finalize	Closes MPI environment
Point-to-point communication	One-to-one communication using Send and Recv

Program Code: lab3a.c

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   int number;
   if (rank == 0) {
      number = 42;
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   } else if (rank == 1) {
      MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Process 1 received number %d from Process 0\n", number);
   }
   MPI_Finalize();
   return 0;
}
```

Compile and run:

mpicc lab3a.c -o lab3a

mpirun -np 2 ./lab3a

Output:

Process 1 received number 42 from Process 0

Program b) Explanation of lab3b:

It is a simple Python MPI program using the mpi4py library. It demonstrates basic point-to-point communication between two processes.
purpose of the Program:
To send a number (42) from Process 0 to Process 1 using MPI in Python.
This example helps students understand:
MPI concepts in Python
How to send and receive messages between processes
The role of process rank

from mpi4py import MPI
Imports the mpi4py module, which allows us to use MPI functions in Python.
comm = MPI.COMM_WORLD
COMM_WORLD is the default communicator that includes all MPI processes.
comm is used to send and receive messages between these processes.
rank = comm.Get_rank()
Retrieves the rank (ID) of the current process in COMM_WORLD.
If you're running with 2 processes, rank will be either 0 or 1.
conditional Code Based on Rank:

```
if rank == 0:
    data = 42
    comm.send(data, dest=1, tag=0)
```

If the process is rank 0, it:
Sets data = 42
Sends this value to rank 1 using comm.send
dest=1: destination process
tag=0: a label to identify the message

```
elif rank == 1:
    data = comm.recv(source=0, tag=0)
```

If the process is rank 1, it:
Receives the message from rank 0 using comm.recv
source=0: the sending process
tag=0: must match the tag used in send
print(f"Process 1 received data: {data}")
All processes run this line.

Term	Meaning
MPI.COMM_WORLD	Default communicator including all processes
comm.Get_rank()	Gets the process ID (rank)
comm.send()	Sends data to another process
comm.recv()	Receives data from another process
tag	Identifier for matching messages

Program Code: $]lab3b.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = 42
    comm.send(data, dest=1, tag=0)
elif rank == 1:
    data = comm.recv(source=0, tag=0)
print(f"Process 1 received data: {data}")
```

RUN in terminal:

```
 mpirun -np 2 python3 lab3b.py
/*Visually
Process 0              Process 1
----------             ----------
data = 42
```

send ---> [42] ----> receive
print "Process 1 received data: 42" */
Output:
Process 1 received data: 42

```
kalyani@DESKTOP-9HEJFCC:~/lab3$ mpirun -np 2 python3 lab3b.py
--------------------------------------------------------------------------
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

  Local host: DESKTOP-9HEJFCC
--------------------------------------------------------------------------
Process 1 received data: 42
[DESKTOP-9HEJFCC:00037] 1 more process has sent help message help-btl-vader.txt / cma-permission-denied
[DESKTOP-9HEJFCC:00037] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
```

2. Collective Communication (Broadcast, Scatter, Gather)

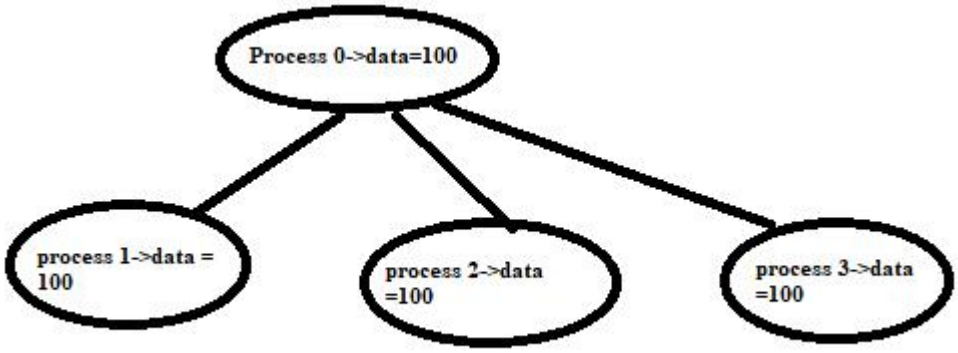
C (Broadcast)

Explanation : lab3d.c

C MPI program which demonstrates the Broadcast (Bcast) collective communication in MPI.
Purpose of the Program:
To send the same data (100) from one process (rank 0) to all other processes using MPI_Bcast.
#include <mpi.h>
#include <stdio.h>
Include necessary headers for using MPI functions and standard input/output.
int main(int argc, char** argv) {
MPI_Init(&argc, &argv);
Initialize the MPI environment. This must be called before any other MPI functions.
int rank, data;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
rank stores the unique ID of the current process.
If 4 processes are launched, ranks will be 0, 1, 2, and 3.
if (rank == 0) data = 100;
Only process 0 sets the value of data = 100.
MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
Broadcasts the value of data from process 0 to all other processes.
After this line, every process (even non-zero ranks) will have data = 100.
&data: address of the data being shared
1: number of items
MPI_INT: data type
0: root process (sender)
MPI_COMM_WORLD: communicator (all processes)
printf("Process %d received data %d\n", rank, data);
Each process prints the value it received.
MPI_Finalize();
return 0;
Shuts down the MPI environment. All processes must call this before exiting.
Output (if 4 processes are used):
Process 0 received data 100
Process 1 received data 100

Process 2 received data 100
Process 3 received data 100
(Note: Output order may vary because processes print independently.)
Visual Representation:

Broadcast Operation



MPI Concept Meaning
MPI_Bcast Sends the same data from one root process to all other processes
MPI_COMM_WORLD Default group of all processes
rank Unique ID assigned to each process

Program Code: vi lab3d.c

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
     int rank, data;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     if (rank == 0) data = 100;
   MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
      printf("Process %d received data %d\n", rank, data);
                MPI_Finalize();
            return 0;
}
```

Compilation:
mpicc lab3d.c -o lab3d
Run: mpirun -np 2 ./lab3d
Output:
Process 0 received data 100
Process 1 received data 100

```
kalyani@DESKTOP-9HEJFCC: ~/lab3
kalyani@DESKTOP-9HEJFCC:~/lab3$ vi lab3d.c
kalyani@DESKTOP-9HEJFCC:~/lab3$ mpicc lab3d.c -o lab3d
kalyani@DESKTOP-9HEJFCC:~/lab3$ mpirun -np 4 ./lab3d
--------------------------------------------------------------------------
WARNING: Linux kernel CMA support was requested via the
btl_vader_single_copy_mechanism MCA variable, but CMA support is
not available due to restrictive ptrace settings.

The vader shared memory BTL will fall back on another single-copy
mechanism if one is available. This may result in lower performance.

  Local host: DESKTOP-9HEJFCC
--------------------------------------------------------------------------
Process 0 received data 100
Process 1 received data 100
Process 2 received data 100
Process 3 received data 100
[DESKTOP-9HEJFCC:00056] 3 more processes have sent help message help-btl-vader.txt / cma-permission-denied
[DESKTOP-9HEJFCC:00056] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
kalyani@DESKTOP-9HEJFCC:~/lab3$
```

Program Code: Python (Broadcast):

lab3e.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
data = None
if rank == 0:
    data = 100
data = comm.bcast(data, root=0)
print(f"Process {rank} received data: {data}")
```

Run the program : **mpirun -np 2 python3 lab3e.py**

Output:

Process 0 received data: 100

Process 1 received data: 100

Python (gather):

Explanation gather_example.py:

To demonstrate how the gather() function is used to collect data from multiple processes into a single root process (usually rank 0).

MPI Operation	Purpose
gather()	Collects data from all processes to a single root process

from mpi4py import MPI

Imports the MPI module in Python. This enables us to use MPI functions like gather().

comm = MPI.COMM_WORLD

COMM_WORLD is the default communicator that includes all the processes involved in the MPI program.

rank = comm.Get_rank()

Each process gets its unique ID, called rank.

If 4 processes are running, ranks will be 0, 1, 2, 3.

size = comm.Get_size()

Returns the total number of processes running in the communicator.

send_data = rank ** 2
Each process calculates the square of its rank.

Process 0 → 0 ** 2 = 0

Process 1 → 1 ** 2 = 1

Process 2 → 2 ** 2 = 4

Process 3 → 3 ** 2 = 9

gathered_data = comm.gather(send_data, root=0)
Collects send_data from all processes and sends it to root process (rank 0).

The root receives a list of data: [0, 1, 4, 9].

```
if rank == 0:
    print(f"Process {rank} gathered data from all processes: {gathered_data}")
```

Only rank 0 prints the result.

The gathered list shows data from all processes in order of their ranks.

Output (when run with 4 processes):

Process 0 gathered data from all processes: [0, 1, 4, 9]

Visual Representation:

```
  [Process 0] send_data = 0  \
  [Process 1] send_data = 1   \
  [Process 2] send_data = 4    ====>  gathered_data = [0, 1, 4, 9]
  [Process 3] send_data = 9   /
                        (collected by root process)
```

MPI Concept	Description
comm.gather()	Collects data from all processes to the root process
rank	Unique ID for each process
root=0	The process that will collect and store all gathered data

How to Run This Program:
Save as gather_example.py, and run with 4 processes:

```
mpiexec -n 4 python gather_example.py
```

Program Code: $]gather_example.py

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Each process creates a value (e.g., its rank squared)
send_data = rank ** 2

# Root process will collect data from all
gathered_data = comm.gather(send_data, root=0)

if rank == 0:
    print(f"Process {rank} gathered data from all processes: {gathered_data}")
```

How It Works:
Each process computes rank ** 2.

comm.gather() sends this value from each process to the root process (rank 0).

Only rank 0 receives the full list of gathered data.

Save as gather_example.py and run using:

mpiexec -n 4 python gather_example.py
Output (for 4 processes):

Process 0 gathered data from all processes: [0, 1, 4, 9]
Explanation:

Process 0 sends 0**2 = 0

Process 1 sends 1**2 = 1

Process 2 sends 2**2 = 4

Process 3 sends 3**2 = 9

The root (rank 0) collects: [0, 1, 4, 9]

Python (scatter):
Scatter: simple and clear Python MPI program using mpi4py that demonstrates the Scatter collective communication pattern, along with the expected output when run with 4 processes.

MPI Operation	Purpose
scatter()	Splits a list into parts and distributes each part to each process

Python MPI program using mpi4py that demonstrates the Scatter operation.
Purpose of the Program:
To demonstrate how a list of values can be divided and distributed among multiple processes using comm.scatter().

```
from mpi4py import MPI
```
Import the mpi4py module to use MPI functionalities in Python.

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```
comm: the communicator that includes all processes.

rank: the unique ID of each process (0, 1, 2, ...).

size: the total number of processes running.

```
# Only the root process prepares the data to scatter
if rank == 0:
    data = [10, 20, 30, 40]  # One value for each process
else:
    data = None
```
Only process 0 creates the data list [10, 20, 30, 40], which has one item per process.

Other processes set data = None because they will receive their part, not create it.

```
# Scatter the data: each process gets one item
recv_data = comm.scatter(data, root=0)
```
Scatter operation begins:

Data is divided evenly across all processes.

Each process gets one item:

Rank 0 → gets 10

Rank 1 → gets 20

Rank 2 → gets 30

Rank 3 → gets 40

```
print(f"Process {rank} received data: {recv_data}")
```
Each process prints what it received.

Output (with 4 processes):

```
Process 0 received data: 10
Process 1 received data: 20
Process 2 received data: 30
Process 3 received data: 40
```

Note: Output order may vary since processes run in parallel.

```
        [Root Process: Rank 0]
          data = [10, 20, 30, 40]
                |
    ---------------------------------
    |       |       |        |
[Rank 0]  [Rank 1]  [Rank 2]  [Rank 3]
  10        20        30        40
```

Concept	Meaning
scatter()	Distributes parts of a list from root to each process
rank	Unique ID of each process
root=0	The process that holds the full data list
recv_data	The part of the list each process receives

How to Run This Program:
Save as scatter_example.py and run with 4 processes:

```
mpirun -np 4 python3 scatter_example.py
```

MPI Scatter Example in Python
scatter_example.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Only the root process prepares the data to scatter
if rank == 0:
    data = [10, 20, 30, 40]  # One value for each process
else:
    data = None

# Scatter the data: each process gets one item
recv_data = comm.scatter(data, root=0)

print(f"Process {rank} received data: {recv_data}")
```

How It Works:
rank 0 prepares a list: [10, 20, 30, 40]

comm.scatter() sends:

10 to process 0

20 to process 1

30 to process 2

40 to process 3

Each process receives a single item.

How to Run:
Save as scatter_example.py and run using:

mpiexec -n 4 python scatter_example.py
Output (for 4 processes):
The output order may vary (since each process prints independently), but you will see:

Process 0 received data: 10
Process 1 received data: 20
Process 2 received data: 30
Process 3 received data: 40

Program Code: 3. Parallel I/O (C and Python)

➢ **C (MPI I/O)**
Explanation of lab3f.c: C program using MPI for parallel file I/O. This example shows how multiple processes write independently to a shared file using MPI_File_write_at.

Purpose of the Program:
This program uses MPI I/O to let each process write its own message to a specific position (offset) in a common file (output.txt) without interfering with each other.

```
#include <mpi.h>
#include <stdio.h>
```
Include MPI and standard I/O libraries.

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
```
Initializes the MPI environment.

```
   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```
Get the rank (ID) of the current process. For example, if 4 processes run, ranks are 0, 1, 2, 3.

```
   MPI_File fh;
   MPI_File_open(MPI_COMM_WORLD, "output.txt", MPI_MODE_CREATE |
```

MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
Open (or create) a shared file named output.txt for writing.

All processes use the same communicator MPI_COMM_WORLD.

```
    char buf[50];
    sprintf(buf, "Hello from process %d\n", rank);
```

Each process creates a message in buf, like "Hello from process 0\n", "Hello from process 1\n", etc.

The buffer size is 50 bytes, ensuring space for padding or future alignment.

```
    MPI_Offset offset = rank * 50;
```

Calculate a unique file offset for each process:

Rank 0 writes at byte 0

Rank 1 writes at byte 50

Rank 2 writes at byte 100 and so on.

This prevents data from overlapping.

```
    MPI_File_write_at(fh, offset, buf, 50, MPI_CHAR, MPI_STATUS_IGNORE);
```

Each process writes exactly 50 characters at its own offset in the file.

MPI_File_write_at allows parallel non-conflicting writes to a shared file.

```
    MPI_File_close(&fh);
    MPI_Finalize();
    return 0;
}
```

Close the file and shut down the MPI environment.

Output (in output.txt after execution):
If run with 4 processes (mpirun -np 4 ./a.out), the file output.txt will contain:

```
Hello from process 0
Hello from process 1
Hello from process 2
Hello from process 3
```

(Each message starts at a separate 50-byte block.)

Concept Explanation
MPI_File_open Opens or creates a file for parallel I/O
MPI_File_write_at Writes data at a specific offset in the file
offset Ensures that each process writes to a different part of the file
MPI_COMM_WORLD All processes participate in file access
Visual Representation:

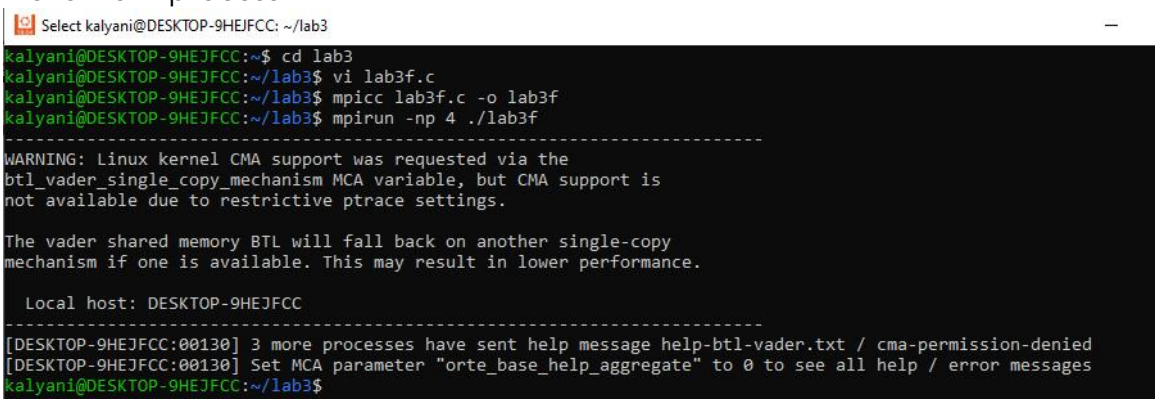
File: output.txt

[Offset 0] → "Hello from process 0"
[Offset 50] → "Hello from process 1"
[Offset 100] → "Hello from process 2"
[Offset 150] → "Hello from process 3"

Program Code: $]gedit lab3f.c

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_File fh;
   MPI_File_open(MPI_COMM_WORLD, "output.txt", MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
   char buf[50];
   sprintf(buf, "Hello from process %d\n", rank);
   MPI_Offset offset = rank * 50;
   MPI_File_write_at(fh, offset, buf, 50, MPI_CHAR, MPI_STATUS_IGNORE);
   MPI_File_close(&fh);
   MPI_Finalize();
   return 0;
}
```

Compile and run: **mpicc lab3f.c -o lab3f**
mpirun -2 4 ./lab3f
Output: output.txt file will be generated with the following output. To open the file , execute the following command
cat output.txt
Hello from process 0
Hello from process 1



Python (mpi4py I/O)

Explanation of lparallel_io.py:

Python MPI program using mpi4py for parallel file I/O. The program demonstrates how multiple processes write to different positions in the same file simultaneously using Write_at.

Purpose of the Program:

To let each MPI process write a separate line into a shared file (output_py.txt), at non-overlapping positions, using MPI I/O.

from mpi4py import MPI

Import the mpi4py library to use MPI features in Python.

comm = MPI.COMM_WORLD

rank = comm.Get_rank()

comm: Communicator that includes all processes.

rank: Unique identifier (ID) of the current process (e.g., 0, 1, 2, ...).

fh = MPI.File.Open(comm, "output_py.txt", MPI.MODE_CREATE | MPI.MODE_WRONLY)

Open (or create) a shared file called output_py.txt in write-only mode.

All processes participate in the file opening.

data = f"Hello from process {rank}\n".ljust(50)

Each process prepares a message, like:

"Hello from process 0\n"

"Hello from process 1\n"

.ljust(50) ensures the message is exactly 50 characters, filling with spaces if needed — this is critical for calculating unique offsets.

offset = rank * 50

Each process calculates a unique byte position in the file to write:

Rank 0 → offset 0

Rank 1 → offset 50

Rank 2 → offset 100

And so on...

Prevents overlapping file writes.

fh.Write_at(offset, data.encode())

Each process writes its data to the file at its own offset.

.encode() converts the string into bytes before writing (as required by MPI).

fh.Close()
Closes the shared file after writing.

Output in output_py.txt: (if run with 4 processes)

Hello from process 0
Hello from process 1
Hello from process 2
Hello from process 3
Each message is padded to 50 bytes, and they are written in fixed-size blocks, one after another.

Run Command:

mpirun -np 4 python3 parallel_io.py
(Assuming the file is saved as parallel_io.py)

Concept	Explanation
MPI.File.Open()	Opens a shared file among all MPI processes
Write_at()	Writes data at a specific location in the file
offset	Ensures each process writes in a unique section
.ljust(50)	Pads string to fixed size (important in parallel I/O)
.encode()	Converts the string to bytes before writing

Visual Representation of File Writes:

File: output_py.txt

Bytes 0–49 → "Hello from process 0 ..."
Bytes 50–99 → "Hello from process 1 ..."
Bytes 100–149 → "Hello from process 2 ..."
Bytes 150–199 → "Hello from process 3 ..."
Each message occupies exactly 50 bytes, which ensures non-overlapping parallel writes.

Program code of parallel_io.py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
fh = MPI.File.Open(comm, "output_py.txt", MPI.MODE_CREATE | MPI.MODE_WRONLY)
data = f"Hello from process {rank}\n".ljust(50)
offset = rank * 50
fh.Write_at(offset, data.encode())
fh.Close()
```

mpirun -np 2 python3 lab3g.py

Output: output_py.txt file will be generated with the following output. To open the file , execute the following command

```
cat output_py.txt

Hello from process 0
Hello from process 1
```

Experiment no 4

Programming in OpenMP.

AIM:

Programming in OpenMP.

Explanation of add.c :

OpenMP parallel sum program in C:

Purpose of the Program:
Calculate the sum of all elements in an array using parallel programming with OpenMP to speed up the summation.

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

Include standard input/output and OpenMP libraries for parallel programming.

```
int main() {
   int i;
   const int N = 1000;
   int array[N];
   long long sum = 0;
```

Declare variables:

N = 1000 is the size of the array.

array[N] is an integer array of size 1000.

sum is a variable to store the final sum, declared as long long to hold large values.

```
// Initialize array
for(i = 0; i < N; i++) {
   array[i] = 1; // All elements are 1
}
```

Initialize the array: each element is set to 1.

```
// Parallel region for sum using reduction
#pragma omp parallel for reduction(+:sum)
for(i = 0; i < N; i++) {
   sum += array[i];
}
```

#pragma omp parallel for reduction(+:sum):
This is an OpenMP directive that tells the compiler to:

Run the for loop in parallel across multiple threads.

Use reduction on sum to safely accumulate values from all threads.

How it works:

Each thread calculates a partial sum of some part of the array.

At the end, all partial sums are added together automatically to get the total sum.

```
printf("Sum of array elements = %lld\n", sum);
return 0;
}
```

Print the total sum.

End the program.

Term	Explanation
OpenMP	A library for easy multi-threading in C/C++ programs.
parallel for	Run the loop iterations in parallel on multiple threads.
reduction(+:sum)	Combine partial sums from all threads safely.
long long	Used to store large sums to avoid overflow.

Compilation
Output:

Sum of array elements = 1000
Since all elements are 1 and the array has 1000 elements, the sum is 1000.

Why Use Parallelism Here?
If the array was very large, computing the sum in a single thread might be slow.

OpenMP speeds up the process by dividing the work among multiple CPU cores

Program Code: $]gedit add.c

```
#include <stdio.h>
#include <omp.h>
int main() {
   int i;
   const int N = 1000;
   int array[N];
   long long sum = 0;
   // Initialize array
   for(i = 0; i < N; i++) {
      array[i] = 1; // All elements are 1
   }
   // Parallel region for sum using reduction
   #pragma omp parallel for reduction(+:sum)
   for(i = 0; i < N; i++) {
      sum += array[i];
   }
   printf("Sum of array elements = %lld\n", sum);
   return 0;
}
```

Compilation: **gcc -fopenmp add.c -o add**
Run: **./add**

OUTPUT:

```
kalyani@DESKTOP-9HEJFCC:~$ gcc -fopenmp add.c -o add
kalyani@DESKTOP-9HEJFCC:~$ ./add
Sum of array elements = 1000
```

Complete Program with (C and OpenACC)
Explanation of openmp.c

OpenACC program in C, which demonstrates parallel computation on the GPU.

To perform element-wise addition of two arrays using OpenACC, which allows offloading loops to a GPU for acceleration.

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

Includes standard input/output and OpenACC header.

openacc.h provides functions and macros for OpenACC.

```
#define N 3
```

Defines the size of the arrays: 1000 elements.

```
int main() {
   int i;
   float a[N], b[N], c[N];
```

Declare:

Loop variable i

Arrays a, b, and c of size 1000

float is used for decimal values

```
// Initialize arrays
for (i = 0; i < N; i++) {
   a[i] = i * 1.0f;
   b[i] = i * 2.0f;
}
```

Populates arrays:

a[i] gets values like 0.0, 1.0, 2.0, ...

b[i] gets values like 0.0, 2.0, 4.0, ...

1.0f and 2.0f ensure the values are floats

```
// Parallel region - use OpenACC to compute c[i] = a[i] + b[i]
```

```
#pragma acc parallel loop
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

#pragma acc parallel loop tells the compiler:

Run this for loop in parallel on the GPU

It will automatically copy data to/from GPU memory as needed

Performs c[i] = a[i] + b[i] for each element

```
// Print some results to verify
printf("Sample Output:\n");
for (i = 0; i < 10; i++) {
    printf("c[%d] = %f\n", i, c[i]);
}
```

Outputs the first 10 results to verify correctness

Expected output:

c[0] = 0.000000

c[1] = 3.000000

c[2] = 6.000000

etc.

```
return 0;
}
```

Ends the program

Concept	Meaning
OpenACC	A directive-based API to parallelize C/C++/Fortran code for GPUs
#pragma acc	Instructs compiler to parallelize specific regions
Offloading	Sending computations to another processor (here, a GPU)
parallel loop	Executes each loop iteration on a separate GPU thread

Easy way to use GPU acceleration in C/C++ programs

Great for scientific computing, simulations, and big data processing

Less complex than CUDA or OpenCL for basic parallel tasks

program code: gedit openmp.c

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

```
#include <stdlib.h>
#include <openacc.h>
#define N 1000
int main() {
   int i;
   float a[N], b[N], c[N];
   // Initialize arrays
   for (i = 0; i < N; i++) {
      a[i] = i * 1.0f;
      b[i] = i * 2.0f;
   }

   // Parallel region - use OpenACC to compute c[i] = a[i] + b[i]
   #pragma acc parallel loop
   for (i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
   }
   // Print some results to verify
   printf("Sample Output:\n");
   for (i = 0; i < 10; i++) {
      printf("c[%d] = %f\n", i, c[i]);
   }
   return 0;
}
```

Compilation:
kalyani@DESKTOP-9HEJFCC:~/Lab5$ **mpicc openmp.c -o openmp**
Run:
kalyani@DESKTOP-9HEJFCC:~/Lab5$ **mpirun -np 4 ./openmp**

Output:
c[0] = 0.000000
c[1] = 3.000000
c[2] = 6.000000
c[3] = 9.000000
c[4] = 12.000000
c[5] = 15.000000
c[6] = 18.000000
c[7] = 21.000000
c[8] = 24.000000
c[9] = 27.000000

Experiment No 5

Programming GPUs using OpenACC.

Open https://developer.nvidia.com/hpc-sdk-downloads

Linux x86_64 (tar file)

Click it

We find these commands:

wget https://developer.download.nvidia.com/hpc-sdk/25.3/nvhpc_2025_253_Linux_x86_64_cuda_multi.tar.gz

tar xpzf nvhpc_2025_253_Linux_x86_64_cuda_multi.tar.gz

cd nvhpc_2025_253_Linux_x86_64_cuda_12.8

./install

export PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/25.3/compilers/bin:$PATH

export LD_LIBRARY_PATH=/opt/nvidia/hpc_sdk/Linux_x86_64/25.3/compilers/lib:$LD_LIBRARY_PATH

verify the installation: nvaccelinfo

nvhpc_2025_253_Linux_x86_64_cuda_multi/install

kalyani@DESKTOP-9HEJFCC: ~/Lab5

```
Try: sudo apt install <deb name>

kalyani@DESKTOP-9HEJFCC:~/Lab5$ wget https://developer.download.nvidia.com/hpc-sdk/25.3/nvhpc_2025_253_Linux_x86_64_cud
_multi.tar.gz
--2025-05-15 00:01:04--  https://developer.download.nvidia.com/hpc-sdk/25.3/nvhpc_2025_253_Linux_x86_64_cuda_multi.tar.
z
Resolving developer.download.nvidia.com (developer.download.nvidia.com)... 23.193.165.88, 23.193.165.81
Connecting to developer.download.nvidia.com (developer.download.nvidia.com)|23.193.165.88|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13131786953 (12G) [application/x-gzip]
Saving to: 'nvhpc_2025_253_Linux_x86_64_cuda_multi.tar.gz'

        nvhpc_2025_253_Linux_   1%[                                          ] 206.16M  1.45MB/s    eta 2h 31m
```

simple example of a program using OpenACC to parallelize a loop on a GPU:

Explanation of lab5.c

To show how to use OpenACC to parallelize a simple addition of two integer arrays on the GPU and verify the results.

#include <stdio.h>

#include <openacc.h>

stdio.h: For input/output (like printf)

openacc.h: Enables use of OpenACC directives

#define N 10

Defines the size of the arrays a, b, and c as 10 elements

int main() {

```
  int a[N], b[N], c[N];
```

Declares three integer arrays:

a[]: First input array

b[]: Second input array

c[]: Result array

```
// Initialize arrays
for (int i = 0; i < N; i++) {
  a[i] = i;
  b[i] = 2 * i;
}
```

Fills:

a[] with 0, 1, 2, ..., 9

b[] with 0, 2, 4, ..., 18

```
// Parallelize loop using OpenACC
#pragma acc parallel loop copyin(a[0:N], b[0:N]) copyout(c[0:N])
for (int i = 0; i < N; i++) {
  c[i] = a[i] + b[i];
}
```

This is the key line:

#pragma acc parallel loop: Instructs the compiler to run this loop in parallel on GPU

copyin(a[0:N], b[0:N]): Copy arrays a and b from CPU to GPU

copyout(c[0:N]): Copy result c from GPU to CPU after computation

The actual computation: c[i] = a[i] + b[i]

```
// Verify results
for (int i = 0; i < N; i++) {
  printf("%d + %d = %d\n", a[i], b[i], c[i]);
}
```

Prints the result to ensure everything worked correctly

OpenACC Directive-based way to run C/C++ code on GPU without deep CUDA programming
parallel loop Runs each loop iteration independently on the GPU
copyin / copyout Manages data transfer between CPU and GPU memory
Speedup Running on GPU can be much faster for large loops

Very small data size (N = 10) → easy to understand output

Simple loop-based computation

Clear use of OpenACC pragmas

Demonstrates data movement between CPU and GPU

program code : lab5.c

```
#include <stdio.h>
#include <openacc.h>
#define N 10
int main() {
  int a[N], b[N], c[N];
  // Initialize arrays
  for (int i = 0; i < N; i++) {
    a[i] = i;
    b[i] = 2 * i;
  }
  // Parallelize loop using OpenACC
  #pragma acc parallel loop copyin(a[0:N], b[0:N]) copyout(c[0:N])
  for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
  }
  // Verify results
  for (int i = 0; i < N; i++) {
    printf("%d + %d = %d\n", a[i], b[i], c[i]);
  }
  return 0;
}
```

This program uses OpenACC to parallelize a simple loop that adds two arrays a and b and stores the result in array c. The #pragma acc parallel loop directive tells the compiler to parallelize the loop on the GPU.
To compile this program, you'll need to use a compiler that supports OpenACC, such as the PGI compiler or GCC. The compilation command will depend on your specific compiler and system.
For example, with the mpicc compiler, you might use:
Compilation: **mpicc lab5.c -o lab5,**
To run: **mpirun -np 2 ./lab5**

OUTPUT:
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15

6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27

/***This will compile the program and generate an executable that can run on the GPU.

Note that this is just a simple example to illustrate the basics of OpenACC programming. In a real-world application, you'd likely want to add more error checking, optimize data transfers between the host and device, and so on
simple example of a program using Numba and OpenACC-like functionality (via Numba's @cuda.jit decorator is not exactly OpenACC, but we can use Numba's @njit with parallel=True for CPU parallelization or use Cupy for GPU acceleration) to parallelize a loop:
$]pip install numpy numba cupy
If this command fails, install CUDA from https://developer.nvidia.com/cuda-downloads operating system linux, architecture x86_64 distribution ubuntu version 20.04 installer type runfile(local) download installer for Linux Ubuntu 20.04 x86_64

sudo apt install nvidia-cuda-toolkit
nvidia-smi
sudo apt install nvidia-driver-535
export NUMBA_CUDA_DRIVER=/usr/lib/x86_64-linux-gnu/libcuda.so
conda activate myenv
export NUMBA_CUDA_DRIVER=/usr/lib/x86_64-linux-gnu/libcuda.so
pip uninstall numba cupy
pip install numba cupy-cuda12x
import cupy as cp
a = cp.array([1, 2, 3])
print(a)

sudo apt update && sudo apt upgrade -y
sudo apt install build-essential python3 python3-pip
sudo apt install nvidia-cuda-toolkit
nvcc –version
/* we will get like this nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation

```
Built on Thu_Nov_18_09:45:30_PST_2021
Cuda compilation tools, release 11.5, V11.5.119
Build cuda_11.5.r11.5/compiler.30672275_0
administrator@LAB226-PC01:~$
*/
pip install numpy numba
* $]    pip install cupy-cuda12x
from numba import cuda
print(cuda.detect())
python your_script.py
nvidia-smi


/*wget
https://developer.download.nvidia.com/compute/cuda/12.9.0/local_installers/cuda_12.9.0_575.51.03_linux.run

sudo sh cuda_12.9.0_575.51.03_linux.run
*/
/*wget
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-keyring_1.1-1_all.debsudo dpkg -i cuda-keyring_1.1-1_all.debsudo apt-get updatesudo apt-get -y install cuda-toolkit-12-9
sudo dpkg -i cuda-keyring_1.1-1_all.deb
sudo apt-get update
sudo apt-get -y install cuda-toolkit-12-9
wget
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64/cuda-ubuntu2204.pin
sudo mv cuda-ubuntu2204.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget
https://developer.download.nvidia.com/compute/cuda/12.9.0/local_installers/cuda-repo-ubuntu2204-12-9-local_12.9.0-575.51.03-1_amd64.deb

sudo dpkg -i cuda-repo-ubuntu2204-12-9-local_12.9.0-575.51.03-1_amd64.deb
sudo cp /var/cuda-repo-ubuntu2204-12-9-local/cuda-*-keyring.gpg /usr/share/keyrings/
sudo apt-get update
sudo apt-get -y install cuda-toolkit-12-9
sudo apt-get install -y nvidia-open
sudo apt-get install -y cuda-drivers
/*sudo apt update && sudo apt install -y build-essential git python3
git clone https://github.com/spack/spack.git ~/spack
. ~/spack/share/spack/setup-env.sh
spack –version
spack install gcc+nvptx*/
```

Experiment no 6

GPU Programming with CuPy and CUDA

AIM:

Objective: Use Python (CuPy) and CUDA C to perform GPU computation.

Install cupy and cuda

To perform vector addition (i.e., c[i] = a[i] + b[i]) using GPU parallelism with CUDA.

Explanation of lab6.cu

1. Header Files

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

stdio.h: Used for standard input/output.

cuda_runtime.h: Gives access to CUDA functions (e.g., cudaMalloc, cudaMemcpy).

2. CUDA Kernel

```
__global__ void vector_add(float *a, float *b, float *c, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
```

__global__: This is a CUDA kernel, meaning it runs on the GPU and is called from the CPU.

threadIdx.x: Index of the thread within a block.

blockIdx.x * blockDim.x: Position of the block in the grid.

idx: The global thread index — ensures each thread works on a different array element.

if (idx < n): Prevents accessing out-of-bounds memory if threads > n.

3. Main Program (on CPU)

Initialize data

```
int n = 10;
float a[n], b[n], c[n];
```

Defines n = 10 elements in each array.

```
for (int i = 0; i < n; i++) {
    a[i] = i * 1.0f;
    b[i] = i * 2.0f;
}
```

a = [0.0, 1.0, 2.0, ..., 9.0]

b = [0.0, 2.0, 4.0, ..., 18.0]

Allocate GPU memory

```
cudaMalloc((void **)&d_a, n * sizeof(float));
cudaMalloc((void **)&d_b, n * sizeof(float));
cudaMalloc((void **)&d_c, n * sizeof(float));
```

cudaMalloc: Reserves memory on the GPU for arrays a, b, and c.

Copy data from CPU to GPU

```
cudaMemcpy(d_a, a, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, n * sizeof(float), cudaMemcpyHostToDevice);
```

cudaMemcpy: Transfers data from CPU (Host) to GPU (Device).

Launch the kernel (GPU function)

```
vector_add<<<1, n>>>(d_a, d_b, d_c, n);
```

<<<1, n>>>: Launches 1 block with n threads

Each thread computes one element of c[i] = a[i] + b[i].

```
cudaMemcpy(c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost);
```

Retrieves the result from GPU memory d_c into CPU array c.

Print results

```
for (int i = 0; i < n; i++) {
    printf("%f ", c[i]);
}
```

Output:

Vector addition result:

0.000000 3.000000 6.000000 9.000000 ... 27.000000

Free GPU memory

cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

Frees up the allocated GPU memory.

CUDA KernelA function that runs on the GPU. Marked with __global__.

Threading Model Each thread handles one element of the array.

Memory Transfer Data must be copied from CPU to GPU before computation and back after.

GPU Acceleration Parallelizing work over many threads can be faster than CPU loops (especially for large arrays).

How to Compile and Run

Use the NVIDIA compiler (nvcc):

Uses simple array operations to demonstrate parallelism

Program Code: $]gedit lab6.cu

```
#include <stdio.h>
#include <cuda_runtime.h>
__global__ void vector_add(float *a, float *b, float *c, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
int main() {
    int n = 10;
    float a[n], b[n], c[n];
    float *d_a, *d_b, *d_c;
    // Initialize arrays
    for (int i = 0; i < n; i++) {
```

```
        a[i] = i * 1.0f;
        b[i] = i * 2.0f;
    }
    // Allocate GPU memory
    cudaMalloc((void **)&d_a, n * sizeof(float));
    cudaMalloc((void **)&d_b, n * sizeof(float));
    cudaMalloc((void **)&d_c, n * sizeof(float));
    // Copy data to GPU
    cudaMemcpy(d_a, a, n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, n * sizeof(float), cudaMemcpyHostToDevice);
    // Launch kernel
    vector_add<<<1, n>>>(d_a, d_b, d_c, n);
    // Copy result back to CPU
    cudaMemcpy(c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost);
    // Print result
    printf("Vector addition result:\n");
    for (int i = 0; i < n; i++) {
        printf("%f ", c[i]);
    }
    printf("\n");

    // Free GPU memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Compilation: **nvcc lab6.cu -o lab6**

To run: **/lab6**

Vector addition result:

0.000000 3.000000 6.000000 9.000000 12.000000 15.000000 18.000000 21.000000 24.000000 27.000000

End

Experiment No 7

Reduction Clause in OpenMP

AIM:

Explanation of lab7.c:
Includes the standard I/O and OpenMP header.

This confirms that the reduction was successful across all threads

What is Reduction in OpenMP?

The reduction(operator : variable) clause:

Initializes a private copy of variable for each thread.

Applies the specified operator (e.g., +, *, max, etc.) after the parallel region to combine the results.

Ensures thread-safe aggregation of results in parallel loops.

SIZE is defined as 1000, which means we will work with an array of 1000 elements.

Array Initialization:

```
int sum = 0;
int array[SIZE];
for (i = 0; i < SIZE; i++) {
    array[i] = 1;
}
```

sum will store the final result.

Every element in the array is set to 1.

So, the expected sum is 1000 * 1 = 1000.

parallel Sum using OpenMP:

```
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < SIZE; i++) {
        sum += array[i];
}
```

#pragma omp parallel for: Tells the compiler to parallelize the for loop using OpenMP.

reduction(+:sum): Each thread calculates its partial sum, and OpenMP automatically adds them together at the end.

Use reduction
Without reduction, multiple threads would try to update the shared sum at the same time,

leading to a race condition and incorrect results.

printing the Output
printf("Total Sum = %d\n", sum);
Displays the sum of all array elements.

Expected output:

Total Sum = 1000
What Students Should Learn from This: Basics of OpenMP parallel loops.

Use of reduction to avoid race conditions.

Importance of thread safety in parallel programming.

Program code of lab7.c

```
#include <stdio.h>
#include <omp.h>
#define SIZE 1000
int main() {
    int i;
    int sum = 0;
    int array[SIZE];
    // Initialize array
    for (i = 0; i < SIZE; i++) {
        array[i] = 1;  // Simple case where the sum should be 1000
    }
    / /Parallel region with reduction
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < SIZE; i++) {
        sum += array[i];
    }
    printf("Total Sum = %d\n", sum);
    return 0;
}
```

Save the code to a file: lab7.c

Compilation: mpicc lab7.c -o lab7
Run: mpirun ./lab7
Output:
Total Sum = 1000

End

Experiment No 8

AIM:

Scheduling Loops in OpenMP - Odd-Even Transposition Sort

Implement and compare static, dynamic, guided schedules

Odd-Even Transposition Sort algorithm in parallel using OpenMP, while demonstrating loop scheduling techniques (static, dynamic, guided).

2. Algorithm Overview: Odd-Even Transposition Sort

A comparison-based sort, ideal for parallelization.

Alternates between odd and even phases:

Even phase: Compare indices 0 and 1, 2 and 3, etc.

Odd phase: Compare indices 1 and 2, 3 and 4, etc.

Repeat for n passes (where n is array length).

3. C Program with OpenMP Scheduling

OpenMP-based C program that performs parallel odd-even sorting (also called Brick Sort):

Program Overview:

This program sorts an array using the odd-even transposition sort algorithm in parallel using OpenMP. It divides the sorting into "even" and "odd" phases and runs comparisons in parallel.

Schedule Type	Description
static	Divides iterations evenly beforehand. Fastest for regular loops.
dynamic	Assigns small chunks to threads dynamically, reducing idle time for uneven workloads.
guided	Similar to dynamic but starts with large chunks and reduces. Useful when iteration time varies.

Explanation of lab8.c

```
#include <omp.h>
#define N 8
```

Includes standard libraries and the OpenMP library.

N is the size of the array (8 elements).

printArray Function:

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

Utility function to display the array elements.

Main Function – Initialization:

```
int arr[N] = {9, 7, 3, 5, 1, 6, 2, 4};
```

The unsorted array is hardcoded.

```
printf("Original array:\n");
printArray(arr);
```

Displays the initial array.

Odd-Even Sort with Parallelism:

```
for (phase = 0; phase < N; phase++) {
    if (phase % 2 == 0) {
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < N - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                // swap
            }
        }
    } else {
        #pragma omp parallel for schedule(static)
        for (int i = 1; i < N - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                // swap
            }
        }
    }
}
```

Odd-Even Transposition Sort involves multiple phases.

Even phase compares index pairs like (0,1), (2,3), etc.

Odd phase compares index pairs like (1,2), (3,4), etc.

Each comparison and swap can be done in parallel using #pragma omp parallel for.

OpenMP Directive Explanation:

#pragma omp parallel for schedule(static)

Tells OpenMP to run the for loop in parallel.

schedule(static) means iterations are divided equally among threads.

You can try replacing it with schedule(dynamic) or schedule(guided) for performance experimentation.

Final Output:

printf("Sorted array:\n");
printArray(arr);
After all phases, the array is sorted and displayed.

Original array:
9 7 3 5 1 6 2 4
Sorted array:
1 2 3 4 5 6 7 9

How odd-even sorting works.

Using OpenMP to parallelize sections of code.

The effect of parallel for and schedule clauses.

The concept of safe parallel swaps (non-overlapping).

Program code : $]gedit lab8.c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 8
void printArray(int arr[]) {
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main() {
    int arr[N] = {9, 7, 3, 5, 1, 6, 2, 4};
    int phase;
    printf("Original array:\n");
    printArray(arr);
    //omp_set_num_threads(4); // Set number of threads
    for (phase = 0; phase < N; phase++) {
        // Even phase
        if (phase % 2 == 0) {
            #pragma omp parallel for schedule(static) // Try dynamic or guided
            for (int i = 0; i < N - 1; i += 2) {
                if (arr[i] > arr[i + 1]) {
                    int temp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = temp;
```

```
            }
        }
    }
    // Odd phase
    else {
        #pragma omp parallel for schedule(static)
        for (int i = 1; i < N - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                int temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
            }
        }
    }
  }

  printf("Sorted array:\n");
  printArray(arr);
  return 0;
}
```

mpicc lab8.c -o lab8
mpirun ./lab8
Output:

```
Original array:
9 7 3 5 1 6 2 4
Sorted array:
1 2 3 4 5 6 7 9
Original array:
9 7 3 5 1 6 2 4
Sorted array:
1 2 3 4 5 6 7 9
```

End

Experiment No 9

Synchronization in OpenMP – Producer-Consumer Problem

AIM:

Producer-Consumer Problem using OpenMP in C:

Program Purpose:
This program demonstrates the classic Producer-Consumer problem using OpenMP parallel sections. It simulates a shared buffer with a single producer and a single consumer running in parallel.

Demonstrate synchronization mechanisms (critical, atomic, barrier).
Task: Simulate producer-consumer using OpenMP threads.

Producer-Consumer problem using a shared buffer (e.g., array or queue). A producer generates items and places them into the buffer, while a consumer removes them for processing. The synchronization is necessary to prevent data races and ensure mutual exclusion.

Requirements

Shared buffer of fixed size

One producer thread

One consumer thread

Mutual exclusion (#pragma omp critical)

Buffer index management

Correct item count tracking

Key Concepts Introduced:
Shared buffer

Parallel sections (OpenMP)

Critical sections

Circular buffer

Busy waiting (simple synchronization)

Global Variables:

```
#define BUFFER_SIZE 5
#define NUM_ITEMS 10
int buffer[BUFFER_SIZE];
```

```
int count = 0;
int in = 0;    // Index for producer
int out = 0;   // Index for consumer
```

buffer: Shared array of fixed size (5 items).

count: Number of items currently in the buffer.

in: Index where the producer will place the next item.

out: Index where the consumer will take the next item.

NUM_ITEMS: Total items to be produced/consumed.

Helper Functions:

```
void produce(int item) { printf("Produced: %d\n", item); }
void consume(int item) { printf("Consumed: %d\n", item); }
```

These simulate production and consumption with simple print statements.

Parallel Sections with OpenMP:

```
#pragma omp parallel sections
```

Starts two independent sections that run in parallel threads.

Producer Section:

```
#pragma omp section
{
    for (i = 1; i <= NUM_ITEMS; i++) {
        while (count == BUFFER_SIZE); // wait if buffer full

        #pragma omp critical
        {
            buffer[in] = i;
            in = (in + 1) % BUFFER_SIZE;
            count++;
            produce(i);
        }
        sleep(1);
    }
}
```

Produces 10 items (i = 1 to 10).

If the buffer is full, it waits (count == BUFFER_SIZE).

Uses a critical section to safely update buffer and index.

sleep(1) simulates a 1-second delay in producing.

Consumer Section:

```
#pragma omp section
{
    for (i = 1; i <= NUM_ITEMS; i++) {
        while (count == 0); // wait if buffer empty

        #pragma omp critical
        {
            int item = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            count--;
            consume(item);
        }
        sleep(2);
    }
}
```

Consumes 10 items.

If the buffer is empty, it waits (count == 0).

Again, a critical section is used to access shared data safely.

sleep(2) simulates slower consumption.

Circular Buffer Handling:
Both in and out are updated using:

(in + 1) % BUFFER_SIZE
This ensures wrap-around indexing, so the buffer acts like a circular queue.

Synchronization Mechanism:
Critical sections (#pragma omp critical) ensure that only one thread updates the buffer or related indices at a time.

Busy waiting is used for simplicity (not ideal for real systems).

Output :

Produced: 1
Consumed: 1
Produced: 2
Produced: 3
Consumed: 2
Produced: 4
Produced: 5
Consumed: 3
...
The producer runs faster than the consumer (sleep(1) vs sleep(2)), so you'll notice that the buffer might get full, and the producer waits.

How OpenMP handles parallel execution with sections.

Importance of critical sections in shared-memory synchronization.

Basics of bounded buffer (producer-consumer).

Circular buffer logic using modulo.

Real-world issues like deadlock, busy waiting, and race conditions.

Program code : lab9.c

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h> // for sleep function
#define BUFFER_SIZE 5#define NUM_ITEMS 10
int buffer[BUFFER_SIZE];int count = 0; // number of items in the bufferint in = 0;     //
index for producerint out = 0;   // index for consumer
void produce(int item) {
   printf("Produced: %d\n", item);
}
void consume(int item) {
   printf("Consumed: %d\n", item);
}
int main() {
   int i;
   #pragma omp parallel sections
   {
      // Producer section
      #pragma omp section
      {
         for (i = 1; i <= NUM_ITEMS; i++) {
            int produced_item = i;
            // Wait until buffer has space
            while (count == BUFFER_SIZE) {
               // Busy wait (could use OpenMP locks or condition vars in real apps)
            }
            #pragma omp critical
            {
               buffer[in] = produced_item;
               in = (in + 1) % BUFFER_SIZE;
               count++;
               produce(produced_item);
            }
            sleep(1); // Simulate time delay for production
         }
      }
      // Consumer section
      #pragma omp section
```

```
        {
            for (i = 1; i <= NUM_ITEMS; i++) {
                int consumed_item;
                // Wait until buffer has items
                while (count == 0) {
                    // Busy wait
                }
                #pragma omp critical
                {
                    consumed_item = buffer[out];
                    out = (out + 1) % BUFFER_SIZE;
                    count--;
                    consume(consumed_item);
                }
               sleep(2); // Simulate time delay for consumption
            }
        }
    }
    return 0;
}
```

Compilation: mpicc lab9.c -o ./lab9
Run: mpirun -np 1 lab9.c
OUTPUT:
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5

End

Experiment No 10

Fork-Join Model in OpenMP

AIM:

Objective: Understand thread creation and joining.
Task: Write a program to demonstrate sequential and parallel sections.

Fork-Join Model, along with execution steps, explanation, and sample output.

To demonstrate the basic structure of an OpenMP parallel region and show how multiple threads can execute a block of code in parallel.

Explanation

Fork-Join Model in OpenMP:

The program begins with a single thread (master).

Upon reaching the #pragma omp parallel directive, it forks into multiple threads.

All threads execute the block of code inside the parallel region.

Once done, the threads join, and execution continues with the master thread.

omp_get_thread_num() returns the thread ID (0 to N-1, where N is the number of threads).

printf before and after the parallel region is executed only once by the master thread.

OpenMP: A library for parallel programming in C/C++ and Fortran.

#pragma omp parallel: A directive that tells the compiler to execute the following block using multiple threads.

omp_get_thread_num(): Returns the ID of the current thread (0 for master, others for worker threads).

```
#include <stdio.h>

#include <omp.h>
```

Includes standard I/O and OpenMP header (required for OpenMP functions like omp_get_thread_num()).

```
int main() {
```

```
    printf("Before parallel region (executed by master thread)\n");
```

This is printed once by the master thread (i.e., the default single thread before parallel region).

```
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        printf("Inside parallel region - Thread ID: %d\n", tid);
    }
```

The #pragma omp parallel creates a team of threads.

Each thread executes the code inside the block independently.

omp_get_thread_num() gives each thread a unique ID.

The printf statement will be executed by each thread, so you will see multiple lines printed.

Note: The number of threads depends on system settings or environment variable OMP_NUM_THREADS.

```
    printf("After parallel region (executed by master thread)\n");
    return 0;
}
```

After the parallel region, only the master thread continues and prints this message once.

Before parallel region (executed by master thread)

Inside parallel region - Thread ID: 0

Inside parallel region - Thread ID: 1

Inside parallel region - Thread ID: 2

Inside parallel region - Thread ID: 3

After parallel region (executed by master thread)

Note: The order of thread outputs may vary because threads run concurrently.

OpenMP makes parallel programming easier using compiler directives.

The code inside #pragma omp parallel is run by multiple threads.

Always use omp_get_thread_num() to identify which thread is running.

Outputs from parallel threads can be interleaved or unordered.

openMP Program: Fork-Join Model

```
// lab10.c
#include <stdio.h>
#include <omp.h>
int main() {
   printf("Before parallel region (executed by master thread)\n");

   #pragma omp parallel
   {
      int tid = omp_get_thread_num();
      printf("Inside parallel region - Thread ID: %d\n", tid);
   }
   printf("After parallel region (executed by master thread)\n");
   return 0;
}
```

Execution Steps

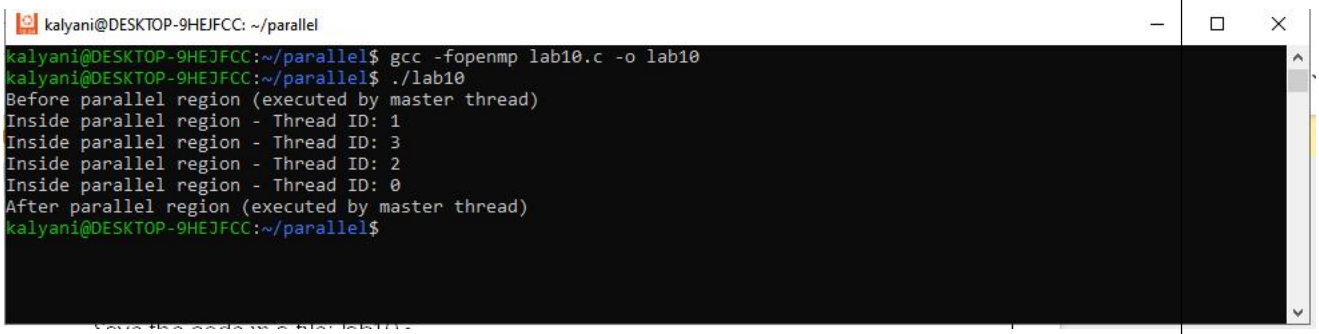
Save the code in a file: lab10.c.

Compile the program using an OpenMP-enabled compiler:

gcc -fopenmp lab10.c -o lab10

Run the program:

./lab10



OUTPUT:

```
Before parallel region (executed by master thread)
Inside parallel region - Thread ID: 1
Inside parallel region - Thread ID: 3
Inside parallel region - Thread ID: 2


Inside parallel region - Thread ID: 0
After parallel region (executed by master thread)

/*Note: The order of thread execution may vary each time due to
non-deterministic

scheduling.*/
```

END