LP 5 VIVA PREP

PRACTICAL 1: Parallel BFS and DFS using OpenMP

To implement Breadth-First Search (BFS) and Depth-First Search (DFS) using OpenMP in C++ to demonstrate parallel graph traversal**.**

✅ Objective:

To implement Breadth-First Search (BFS) and Depth-First Search (DFS) using OpenMP in C++ to demonstrate parallel graph traversal.

📚 Theory:

🔹 **Breadth-First Search (BFS):**

- A graph traversal algorithm that explores neighbours level by level.
- Uses a **queue** to store nodes to be visited.
- Good for finding shortest path in unweighted graphs.

🔹 **Depth-First Search (DFS):**

- Explores as far as possible along a branch before backtracking.
- Uses a **stack** or recursion.
- Good for checking connectivity or detecting cycles.

🔹 **OpenMP:**

- API for multi-threading in C/C++ and Fortran.
- Allows easy parallelism using compiler directives.
- Main directives used:
    - #pragma omp parallel
    - #pragma omp parallel for
    - #pragma omp critical

🧠 Parallel Strategy:

For BFS:

- The outer loop (queue-based) must remain sequential because it depends on the current level.
- But checking neighbors can be **parallelized**, since visiting neighbors can be done concurrently.

For DFS:

- Recursive nature makes it harder to parallelize.
- Still, we can parallelize the loop where we explore adjacent nodes.

🎙️ **Viva Questions (Detailed)**

Q1: Why is BFS more suitable for parallelism than DFS?

🗣️ *BFS explores nodes in layers, and at each layer, all neighbors can be processed independently. DFS is recursive and sequential in nature, making parallelism harder to implement effectively.*

Q2: What is the role of #pragma omp critical?

🗣️ *It ensures that only one thread modifies shared resources like the visited array or the queue at a time, avoiding data races.*

Q3: What data structures are used in BFS and DFS?

🗣️ *BFS uses a queue, DFS uses a stack (or recursion).*

Q4: Can we remove #pragma omp critical and still run the program?

🗣️ *Yes, but it may produce incorrect output due to race conditions where multiple threads try to update the same data.*

Q5: How do you measure speedup with OpenMP?

🗣️ *By measuring execution time with and without OpenMP and comparing them using the formula: Speedup = Time_sequential / Time_parallel.*

📜 **Key Terms to Remember:**

Term	Meaning
OpenMP	Open Multi-Processing API for parallel computing
#pragma	Compiler directive to instruct OpenMP
Parallel for	Distributes iterations among threads
Critical	Ensures mutual exclusion for shared resources
Race condition	Error caused by unsynchronized access to shared data
BFS / DFS	Graph traversal algorithms

Great! Let's now focus specifically on **OpenMP**, which is crucial for your HPC (High Performance Computing) subject and viva.

🔧 OpenMP – Overview for Viva & Practicals (SPPU BE Comp Sem 8)

✅ What is OpenMP?

OpenMP stands for **Open Multi-Processing**.
It is an **API (Application Programming Interface)** that supports **multi-platform shared memory multiprocessing** programming in **C, C++, and Fortran**.

📌 Why Use OpenMP?

- Makes it easy to write **parallel programs**.
- Takes advantage of **multi-core CPUs**.
- Improves **performance** and reduces **execution time**.
- Simple to implement with just **compiler directives (pragma)**.

⚙️ How It Works?

OpenMP uses a **fork-join model**:

- The **master thread** begins execution.
- When it hits a parallel region, it **forks** multiple threads.

- After the region, all threads **join back**.

📝 **Key Directives in OpenMP (C/C++):**

Directive	Use
#pragma omp parallel	Starts parallel region
#pragma omp for	Parallelizes a for loop
#pragma omp sections	Divides tasks into independent sections
#pragma omp critical	Ensures mutual exclusion (avoids race condition)
#pragma omp single	Block executed by only one thread
#pragma omp barrier	Synchronization point for all threads

💡 **Important OpenMP Functions:**

Function	Description
omp_get_thread_num()	Returns ID of current thread
omp_get_num_threads()	Returns total number of threads
omp_set_num_threads(n)	Sets number of threads to n

🎙️ **OpenMP Viva Questions with Answers**

Q1: What is OpenMP?

🗣️ *It is an API used for parallel programming in shared-memory environments using threads.*

Q2: What is the role of #pragma omp parallel?

🗣️ *It creates a block of code that runs in parallel across multiple threads.*

Q3: What is a race condition?

🗣️ *A race condition occurs when multiple threads access and modify shared data simultaneously, leading to unpredictable results.*

Q4: How does #pragma omp critical help?

🗣️ *It ensures that only one thread executes the enclosed block at a time, preventing race conditions.*

Q5: How do you specify the number of threads in OpenMP?

🗣️ *Using omp_set_num_threads(n); or by setting the environment variable OMP_NUM_THREADS.*

Q6: Difference between parallel and sequential execution?

🗣️ *Sequential: One instruction at a time.*
Parallel: Multiple instructions run simultaneously using multiple threads.

Q7: Can OpenMP be used with recursive functions?

🗣️ *It's not ideal. OpenMP works best with loops and block-level parallelism. Recursive parallelism is better handled using other frameworks like Cilk or TBB.*

📌 **Real-Life Applications of OpenMP:**

- Simulations (weather, physics)
- Image processing
- Data analysis
- Scientific computing
- AI/ML matrix computations

Here's a **simple, step-by-step explanation** of your C++ code that implements **Parallel BFS and DFS** using **OpenMP**:

🔷 **Overview**

This program:

- Creates a graph using an **adjacency list**.
- Implements **BFS** (Breadth-First Search) and **DFS** (Depth-First Search) traversals.
- Uses **OpenMP** to make parts of the algorithm **parallel**, improving performance.

🔶 1. Header Files

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>
```

- iostream – For input/output.
- vector – To represent adjacency list.
- queue – Used in BFS.
- omp.h – OpenMP for parallel programming.

🔶 2. Graph Class

```
class Graph {
   int V;
   vector<vector<int> > adj;
```

- V: Number of vertices.
- adj: A list of lists (vector<vector<int>>) that stores edges.

🔶 3. Constructor

```
Graph(int vertices) {
   V = vertices;
   adj.resize(V);
}
```

- Initializes the graph with V vertices.

🔶 4. Add Edge

```
void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u); // Undirected graph
}
```

- Adds an edge between u and v.
- Since the graph is **undirected**, both directions are added.

🔶 5. BFS (Breadth-First Search)

```
void BFS(int start) {
    vector<bool> visited(V, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);
```

- Marks starting node as visited.
- Pushes it into the queue.

```
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " ";

        #pragma omp parallel for
        for (int i = 0; i < adj[current].size(); i++) {
            int neighbor = adj[current][i];
            if (!visited[neighbor]) {
                #pragma omp critical
```

```
            {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
    }
}
```

🧠 **Key Points:**

- Each neighbor is checked **in parallel** using #pragma omp parallel for.
- Since multiple threads may try to access shared variables (visited, queue), we use:
    - #pragma omp critical to **avoid race conditions**.
- visited[neighbor] is checked again inside critical section to ensure correctness.

🔶 **6. DFS (Depth-First Search)**

```
void DFSUtil(int current, vector<bool>& visited) {
    visited[current] = true;
    cout << current << " ";

    #pragma omp parallel for
    for (int i = 0; i < adj[current].size(); i++) {
        int neighbor = adj[current][i];
        if (!visited[neighbor]) {
            #pragma omp critical
            {
                if (!visited[neighbor]) {
                    DFSUtil(neighbor, visited);
```

```
            }
        }
    }
}
```

- Recursive function that marks the current node visited.
- For each neighbor:
    - Uses **OpenMP parallel for** for parallel traversal.
    - Uses #pragma omp critical to safely call recursion.

🔶 7. DFS Wrapper Function

```
void DFS(int start) {
    vector<bool> visited(V, false);
    cout << "DFS: ";
    DFSUtil(start, visited);
    cout << endl;
}
```

- Initializes visited vector and starts DFS from given node.

🔶 8. Main Function

```
int main() {
    int V, E;
    cin >> V >> E;

    Graph g(V);

    for (int i = 0; i < E; i++) {
        int u, v;
```

```
        cin >> u >> v;
        g.addEdge(u, v);
    }

    g.BFS(0); // Start from node 0
    g.DFS(0);
    return 0;
}
```

🌟 **What it does:**

1. Accepts number of vertices and edges.
2. Takes edge input (as pairs of vertices).
3. Runs **BFS** and **DFS** starting from node 0.

✅ **Output Example:**

If you enter:

Enter number of vertices: 4

Enter number of edges: 4

0 1

0 2

1 2

2 3

Output:

BFS: 0 1 2 3

DFS: 0 1 2 3

💬 **In Simple Terms:**

- A graph is built using edges.
- BFS explores nodes **level by level**.

- DFS explores **deep down one path before backtracking**.
- OpenMP makes neighbor processing **faster** by doing it **in parallel**.
- #pragma omp critical ensures threads **don't conflict** while updating shared data.

Here is a **viva explanation script** for your BFS and DFS program using OpenMP. This script is simple, clear, and suitable for viva or presentation purposes:

🎤 Viva Explanation Script – BFS and DFS using OpenMP

Q: What is the aim of this program?
A:
The aim of this program is to implement **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** traversal on a graph using **OpenMP** to perform certain operations in **parallel**, which can help improve execution speed in larger graphs.

Q: What is a graph and how is it represented here?
A:
A graph is a data structure consisting of **vertices (nodes)** and **edges (connections)**. In this program, we represent the graph using an **adjacency list**, which is implemented as a vector of vectors.

Q: What is OpenMP and why are we using it here?
A:
OpenMP is an API used for **parallel programming in C/C++**. It helps us to split certain tasks across multiple CPU threads.
We use it here to **parallelize the traversal of neighbors** of a node during BFS and DFS, to make those parts faster.

Q: How does BFS work in this program?
A:

- BFS starts from a given node and visits all its **immediate neighbors first**, then their neighbors, and so on – this is called **level-wise traversal**.
- We use a **queue** to store the nodes to be visited.
- In our BFS function, we check all neighbors using a #pragma omp parallel for loop.

- To avoid errors due to multiple threads accessing shared variables like visited[] and queue, we use #pragma omp critical.

Q: How does DFS work in this program?
A:

- DFS starts from a node and goes **as deep as possible** along one path before backtracking.
- It is implemented using **recursion**.
- We parallelize the loop that visits each neighbor using #pragma omp parallel for.
- Since recursion and visited array can cause conflicts in parallel execution, we protect recursive calls using #pragma omp critical.

Q: How is the graph built?
A:

- The user inputs the number of vertices and edges.
- Then for each edge, we call addEdge(u, v) to store the connection.
- Since it's an **undirected graph**, we store the edge in both directions.

Q: How do you run this program?
A:

1. Compile using a compiler that supports OpenMP, for example:
2. g++ -fopenmp bfs_dfs.cpp -o bfs_dfs
3. Run the program:
4. ./bfs_dfs
5. Enter the number of vertices and edges, followed by edge pairs.

Q: What OpenMP directives are used here?
A:
We use:

- #pragma omp parallel for – To run a loop in parallel.

- #pragma omp critical – To protect shared resources (like visited array and queue/recursion) from race conditions.

Q: Can you give an example input and output?
A:
Input:

4

4

0 1

0 2

1 2

2 3

Output:

BFS: 0 1 2 3

DFS: 0 1 2 3

Q: What is the advantage of using OpenMP in this code?
A:
Using OpenMP helps improve performance when traversing **large graphs**, especially when checking multiple neighbors in BFS or DFS. It allows these checks to be done **simultaneously** using multiple threads.

🟢 **Conclusion:**

This program shows how parallel programming can be applied even in simple algorithms like BFS and DFS to make them more efficient, using OpenMP in C++.

Here's a full explanation of **Practical 2** from your file hpc2(Bubble_Merge).cpp, including:

1. **Concepts covered**
2. **Code explanation**
3. **Viva questions with answers**
4. **Presentation/script to explain the code during practical**

✅ PRACTICAL 2 OVERVIEW:

Title:
Comparison of sequential and parallel implementations of Bubble Sort and Merge Sort using OpenMP.

Objective:
To study and compare sequential vs parallel versions of:

- **Bubble Sort**
- **Merge Sort**
  ...using OpenMP for parallelism and chrono to measure execution time.

💡 CONCEPTS EXPLAINED:

1. Bubble Sort:

- Repeatedly swaps adjacent elements if they are in the wrong order.
- **Time Complexity:** $O(n^2)$
- Works well for small datasets.

Parallel Bubble Sort (Odd-Even Transposition):

- Odd and even indexed elements are sorted in alternating phases using #pragma omp parallel for.

---

2. Merge Sort:

- Divide and conquer algorithm.
- Recursively splits the array and then merges sorted halves.
- **Time Complexity:** O(n log n)

Parallel Merge Sort:

- Uses #pragma omp parallel sections to divide recursive calls concurrently.

---

3. OpenMP:

- API for multi-platform shared-memory parallel programming.
- Enables you to run sections or loops concurrently on multiple threads.

---

4. Chrono Library:

- Measures high-resolution time (start, end) for performance comparison.

🧠 VIVA QUESTIONS & ANSWERS

Question	Answer
What is OpenMP?	OpenMP is an API for writing parallel programs using shared memory architecture.
What does #pragma omp parallel for do?	It distributes loop iterations among available threads.
How is merge sort better than bubble sort?	Merge sort has better time complexity O(n log n) vs $O(n^2)$ of bubble sort.
What is the need for parallel sorting?	To reduce execution time by utilizing multiple CPU cores.
How is performance measured in your code?	Using chrono library to measure execution time in seconds.
What is the significance of #pragma omp sections?	It allows independent code blocks to run in parallel.
Can merge sort be completely parallelized?	Not entirely, due to sequential merge step, but recursive calls can be parallel.

💻 CODE EXPLANATION SCRIPT (To Speak in Viva or Presentation)

Introduction:

“In this practical, we implemented and compared sequential and parallel versions of bubble sort and merge sort using OpenMP in C++. We used the chrono library to calculate the execution time for performance analysis.”

Bubble Sort:

void bubbleSort(vector<int>& arr)

"This is the sequential version. It runs two nested loops, comparing adjacent elements and swapping them if out of order."

void parallelBubbleSort(vector<int>& arr)

"This is the parallel version using the odd-even transposition approach. Here, we use #pragma omp parallel for inside each iteration to parallelize the swapping process."

Merge Sort:

void mergeSort(vector<int>& arr, int l, int r)

"This function splits the array into two halves recursively and merges them using the merge() function."

void parallelMergeSort(vector<int>& arr, int l, int r)

"This is the parallel version. The recursive calls to sort left and right halves are done in separate OpenMP sections, allowing them to execute concurrently."

Merge Function:

void merge(vector<int>& arr, int l, int m, int r)

"This function merges two sorted subarrays using a temporary vector."

Main Function:

“We first take user input and store it in a vector. Then we run all four sorting algorithms one by one — sequential bubble, parallel bubble, sequential merge, and parallel merge. After each sort, we print the sorted array and the time taken.”

high_resolution_clock::now()

"This is used to mark start and end times. We calculate the duration by subtracting start from end and displaying the result."

Output:

"The output shows the original array, the sorted arrays for all algorithms, and their execution time for comparison."

✅ **FINAL OUTPUT STRUCTURE**

Enter number of elements: 5

Enter elements: 9 3 7 1 2

Original Array:

9 3 7 1 2

Sequential Bubble Sort:

1 2 3 7 9

Time: 0.0001 seconds

Parallel Bubble Sort:

1 2 3 7 9

Time: 0.00006 seconds

Sequential Merge Sort:

1 2 3 7 9

Time: 0.00002 second

Parallel Merge Sort:

1 2 3 7 9

Time: 0.00001 seconds

📌 **CONCLUSION (to say in exam)**

“This practical clearly demonstrates the performance benefits of parallel computing. As the size of data increases, the parallel versions of sorting algorithms, especially merge sort, show significant speed improvements by utilizing multi-core processing with OpenMP.”

Great! You're referring to **Practical 3** from your LP5 manual, which focuses on:

Performing reduction operations like finding MIN, MAX, SUM, and AVERAGE using OpenMP.

You've already provided a working C++ code using OpenMP reduction. Let's now structure the **complete explanation of Practical 3** for viva, code walkthrough, and theoretical understanding just like we did for Practical 2.

✅ **PRACTICAL 3 OVERVIEW:**

Title:
Write a program to demonstrate use of reduction clause in OpenMP for calculating:

- Minimum
- Maximum
- Sum
- Average

💡 **CONCEPTS EXPLAINED:**

🔹 **1. OpenMP reduction Clause:**

- A special OpenMP directive used to **aggregate values across threads** safely.
- Syntax:
- #pragma omp parallel for reduction(operation: variable)
- Each thread performs its operation and combines the result at the end.

🔹 **2. Types of Reduction Operations Used:**

Operation	Purpose
+	Sum
min	Minimum value
max	Maximum value

Average is calculated using sum and dividing by total elements after parallel region.

🔹 **3. Parallel for Loop:**

- OpenMP distributes the loop iterations across multiple threads.

💻 **PRACTICAL 3 CODE (with explanation)**

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

using namespace std;

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int min_val = arr[0];
    int max_val = arr[0];
    int sum = 0;

    // Parallel region using reduction
```

```
    #pragma omp parallel for reduction(min:min_val) reduction(max:max_val)
reduction(+:sum)
    for (int i = 0; i < n; i++) {
        if (arr[i] < min_val)
            min_val = arr[i];
        if (arr[i] > max_val)
            max_val = arr[i];
        sum += arr[i];
    }

    double avg = (double)sum / n;

    cout << "\nResults:\n";
    cout << "Minimum = " << min_val << endl;
    cout << "Maximum = " << max_val << endl;
    cout << "Sum = " << sum << endl;
    cout << "Average = " << avg << endl;

    return 0;
}
```

CODE EXPLANATION SCRIPT (For Viva/Presentation)

“In this program, I used OpenMP to parallelize the operations of finding the minimum, maximum, sum, and average from an array.
We start by reading n numbers from the user into a vector.
We then use #pragma omp parallel for along with three reduction clauses:

- reduction(min:min_val) to compute the minimum,
- reduction(max:max_val) to compute the maximum, and

- reduction(+:sum) to add all elements in parallel.
  Finally, we calculate the average using the sum divided by n and display the results."

🧠 VIVA QUESTIONS & ANSWERS

Question	Answer
What is reduction in OpenMP?	It's a clause that combines results from all threads into a single final result using a specified operation.
Why do we need reduction?	To safely perform accumulative operations (like sum) in parallel without race conditions.
Can we use reduction for min and max?	Yes, OpenMP supports min and max reductions.
What happens if we don't use reduction?	The result may be incorrect due to concurrent access/modification by multiple threads.
How is average calculated in this program?	Average is computed as sum / n after summing all elements using reduction.
How many threads are used by default in OpenMP?	It depends on the system, but usually equals the number of available CPU cores.
Can we perform custom reductions?	Not directly in OpenMP C++ reduction clause, but we can simulate using critical or atomic if needed.

✅ SAMPLE OUTPUT

Enter number of elements: 6

Enter elements:

10 20 5 15 40 30

Results:

Minimum = 5

Maximum = 40

Sum = 120

Average = 20

📌 **CONCLUSION (to say at end)**

“This practical demonstrates how OpenMP can be effectively used to speed up reduction operations such as finding sum, minimum, and maximum. Using the reduction clause ensures thread-safe computation and improves performance for large datasets.”

Here is a **detailed explanation and in-depth conceptual breakdown** of **Practical 3: OpenMP Reduction Operations**—including theoretical background, code explanation line-by-line, and all internal working.

✅ **PRACTICAL 3: Objective**

Goal: Use OpenMP reduction clause to calculate:

- Minimum value
- Maximum value
- Sum
- Average

This demonstrates **parallel processing** using **OpenMP** where each thread works on part of the data and the results are then **combined safely**.

🔧 **Concepts Behind the Code**

🔹 **What is OpenMP?**

OpenMP (Open Multi-Processing) is an API that supports **multi-platform shared-memory multiprocessing programming** in C, C++, and Fortran.

It allows parallelizing tasks easily using **compiler directives**, like:

#pragma omp parallel

🔹 **What is Reduction?**

Reduction is a **parallel programming concept** where:

- Multiple threads **independently compute partial results**.
- After completion, these partial results are **combined (reduced)** into a final single result using an **operator** like:
    - + → Sum
    - min → Minimum
    - max → Maximum
    - * → Multiplication, etc.

This avoids **race conditions**.

⚙️ How Reduction Works Internally in OpenMP

Let's say we have an array: [10, 5, 7, 2, 8, 12]
And 3 threads:

Each thread:

1. **Owns a private copy** of the reduction variable (e.g., sum, min_val, etc.)
2. Performs operation on **its own chunk** of the array
3. At the end, OpenMP **combines** all private values into the **original variable**

🧠 Theory for Each Operation

Operation	OpenMP Directive	Behavior
sum	reduction(+:sum)	Adds partial sums from each thread
min	reduction(min:min_val)	Takes minimum value from all threads
max	reduction(max:max_val)	Takes maximum value from all threads
average	Not a direct reduction	Calculated as sum / n after summing

---

💻 Code with Line-by-Line Explanation

```
#include <iostream>        // for input/output
#include <vector>          // to use dynamic arrays
#include <omp.h>           // OpenMP header
```

```
using namespace std;

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;               // Input the size of the array

    vector<int> arr(n);     // Create dynamic array (vector)

    cout << "Enter elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];      // Input array elements
    }

    int min_val = arr[0];   // Initialize min to first element
    int max_val = arr[0];   // Initialize max to first element
    int sum = 0;            // Initialize sum to 0

    // Parallel section with reduction
    #pragma omp parallel for reduction(min:min_val) reduction(max:max_val)
reduction(+:sum)
    for (int i = 0; i < n; i++) {
        // Threads work independently on chunks of data
        if (arr[i] < min_val)
            min_val = arr[i];   // thread-local comparison
        if (arr[i] > max_val)
            max_val = arr[i];   // thread-local comparison
        sum += arr[i];          // thread-local summing
```

```
    }

    double avg = (double)sum / n; // Average calculation outside loop

    // Display final output
    cout << "\nResults:\n";
    cout << "Minimum = " << min_val << endl;
    cout << "Maximum = " << max_val << endl;
    cout << "Sum = " << sum << endl;
    cout << "Average = " << avg << endl;

    return 0;
}
```

What Happens Internally?

Suppose:

arr = [10, 30, 5, 8, 40, 2]

And 2 threads are created:

Thread	Elements Processed	Local Min	Local Max	Local Sum
T0	[10, 30, 5]	5	30	45
T1	[8, 40, 2]	2	40	50

OpenMP combines:

- **min**: min(5, 2) = 2
- **max**: max(30, 40) = 40
- **sum**: 45 + 50 = 95
- **avg**: 95 / 6 = 15.83

Output Example:

Enter number of elements: 5

Enter elements:

10 20 30 40 50

Results:

Minimum = 10

Maximum = 50

Sum = 150

Average = 30

🎙️ **Viva Questions (In-depth)**

Question	Detailed Answer
What is a race condition?	When two or more threads access shared memory simultaneously and try to change it without synchronization, leading to unpredictable results.
How does OpenMP handle race conditions?	Using constructs like critical, atomic, and reduction to ensure thread-safe access.
What is the benefit of reduction clause over critical?	reduction is more efficient because OpenMP optimizes it internally, whereas critical forces serialization.
Why is average not directly parallelized?	Because it depends on sum, which is reduced first. Then the final average is computed serially.
Can reduction be applied to arrays?	No, OpenMP does not support reduction on arrays directly—only on scalar variables.
What is thread-private copy in reduction?	It is a separate copy of the reduction variable used by each thread. At the end, OpenMP combines them into a single result.

📌 **Conclusion**

This practical demonstrates efficient **parallel aggregation operations** using OpenMP. By using reduction, we avoid complex synchronization and improve performance.

It also reinforces key parallel programming concepts like **data sharing**, **thread safety**, and **load distribution**.

Practical 4 code for **vector addition and matrix multiplication using CPU and GPU**. This will cover:

✅ 1. PRACTICAL OBJECTIVE

Goal:
Write a Python program that performs:

- Vector addition
- Matrix multiplication

And do it in a way that allows you to **switch between CPU and GPU** by changing a single variable.

We achieve this using:

- **NumPy** for CPU-based operations
- **CuPy** for GPU-based operations

🔧 2. CODE WITH EXPLANATION

USE_GPU = False

- This flag decides whether we want to use the GPU or CPU.
- You can change it to True if a GPU and CUDA are available.

```
if USE_GPU:
    import cupy as xp
else:
    import numpy as xp
```

- cupy is GPU-compatible and performs computations using CUDA.
- numpy is CPU-based.

- We import either of them **as xp**, so we can write the rest of the code **once** and reuse xp for both.

🔹 **VECTOR ADDITION**

```
N = int(input("Enter size of vector/matrix (N): "))
```

- Takes user input for the size of vectors and matrices.

```
print("Enter elements for Vector A:")
A = xp.array([int(input(f"A[{i}]: ")) for i in range(N)])
```

- Takes N inputs from the user and stores them in **vector A** using either NumPy or CuPy.

```
print("Enter elements for Vector B:")
B = xp.array([int(input(f"B[{i}]: ")) for i in range(N)])
```

- Takes N inputs for **vector B**.

```
C = A + B
```

- Element-wise vector addition using array broadcasting.

```
print("\nVector A:", A)
print("Vector B:", B)
print("Addition (A + B):", C)
```

- Displays the original vectors and their result.

🔸 **MATRIX MULTIPLICATION**

```
print("\nEnter elements for Matrix D:")
D = xp.zeros((N, N), dtype=int)
for i in range(N):
```

```
    for j in range(N):
        D[i, j] = int(input(f"D[{i}][{j}]: "))
```

- Creates an N×N matrix initialized with zeros.
- Uses nested loops to get values from the user and populate **Matrix D**.

```
print("Enter elements for Matrix E:")
E = xp.zeros((N, N), dtype=int)
for i in range(N):
    for j in range(N):
        E[i, j] = int(input(f"E[{i}][{j}]: "))
```

- Same logic for creating **Matrix E**.

```
F = D @ E
```

- Performs matrix multiplication using @ operator (also works as matmul in NumPy/CuPy).
- For two matrices D and E, this calculates D[i][k] * E[k][j] summed over k.

```
print("\nMatrix D:")
print(D)

print("\nMatrix E:")
print(E)

print("\nMultiplication Result (D x E):")
print(F)
```

- Prints all matrices and their product.

⚙️ 3. HOW NumPy vs CuPy WORK INTERNALLY

Feature	NumPy	CuPy
Used for	CPU-based math	GPU-based math with CUDA
Speed	Slower for large data	Faster due to parallel cores
Syntax	np.array()	cp.array()
Compatibility	Built-in Python	Needs NVIDIA GPU + CUDA Toolkit
Installation	pip install numpy	pip install cupy-cuda11x

With USE_GPU and xp, we can switch between them seamlessly!

🧪 4. SAMPLE OUTPUT

Suppose you run with:

USE_GPU = False

Then input:

Enter size of vector/matrix (N): 2

Enter elements for Vector A:

A[0]: 1

A[1]: 2

Enter elements for Vector B:

B[0]: 3

B[1]: 4

Vector A: [1 2]

Vector B: [3 4]

Addition (A + B): [4 6]

Enter elements for Matrix D:

D[0][0]: 1

D[0][1]: 2

D[1][0]: 3

D[1][1]: 4

Enter elements for Matrix E:

E[0][0]: 5

E[0][1]: 6

E[1][0]: 7

E[1][1]: 8

Matrix D:

[[1 2]

 [3 4]]

Matrix E:

[[5 6]

 [7 8]]

Multiplication Result (D x E):

[[19 22]

 [43 50]]

🎓 5. VIVA QUESTIONS

Question	Answer
What is NumPy used for?	Numerical computations using CPU.
What is CuPy?	GPU-accelerated library compatible with NumPy.
What is the advantage of using GPU?	Parallel computation makes it much faster for large datasets.
What does @ operator do?	It performs matrix multiplication.

Question	Answer
What happens if I run CuPy code without a GPU?	You will get a RuntimeError unless you use NumPy instead.
What is the difference between zeros() and array()?	zeros() creates an array with all values as 0, array() converts input data to an array.

📘 6. SUMMARY

Task	Performed using
Vector Addition	A + B
Matrix Multiplication	D @ E
CPU computation	NumPy
GPU computation	CuPy
Abstraction	xp for both

DLL

Here is the **detailed explanation for all Deep Learning practicals** with **theoretical background, key concepts, step-by-step explanation of the code**, and **what each part of the practical script is doing**.

✅ PRACTICAL 1: Boston Housing Price Prediction using ANN

🧠 Theoretical Concepts:

- **Regression Problem:** Predicting continuous value (house price).
- **ANN (Artificial Neural Network):** Network of connected neurons with layers:
  - Input Layer
  - Hidden Layers
  - Output Layer
- **Activation Function:** ReLU in hidden layers, Linear for regression output.
- **Loss Function:** Mean Squared Error (MSE)
- **Optimizer:** Adam for adaptive learning

🔍 What Happens in the Code:

1. **Load Dataset:**
2. from sklearn.datasets import load_boston

→ Fetches Boston dataset with 13 features and median house prices.

3. **Preprocessing:**
4. scaler = StandardScaler()
5. X_scaled = scaler.fit_transform(X)

→ Scales features to zero mean and unit variance for better convergence.

6. **Model Creation:**
7. model = Sequential()
8. model.add(Dense(64, activation='relu', input_shape=(13,)))
9. model.add(Dense(1))

→ Creates ANN with one hidden layer (64 neurons) and one output neuron for price.

10. **Compilation:**
11. model.compile(optimizer='adam', loss='mse', metrics=['mae'])
12. **Training:**
13. model.fit(X_train, y_train, epochs=100, validation_split=0.2)
14. **Evaluation:**
15. loss, mae = model.evaluate(X_test, y_test)
16. print(f"Mean Absolute Error: {mae}")

✅ PRACTICAL 2: Fashion MNIST Classification using ANN

🧠 Theoretical Concepts:

- **Fashion MNIST:** Dataset with 28x28 grayscale images of clothes.
- **Classification:** Predict which item (T-shirt, sneaker, etc.).
- **ANN Model:** Uses flattened image as 1D array.

🔍 Code Breakdown:

1. **Load Dataset:**
2. (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
3. **Normalize & Flatten:**
4. x_train = x_train / 255.0
5. model.add(Flatten(input_shape=(28, 28)))
6. **Create Model:**
7. model.add(Dense(128, activation='relu'))
8. model.add(Dense(10, activation='softmax'))
9. **Compile:**
10. model.compile(optimizer='adam',
11. loss='sparse_categorical_crossentropy',
12. metrics=['accuracy'])
13. **Train & Evaluate:**
14. model.fit(x_train, y_train, epochs=10)

15. model.evaluate(x_test, y_test)

✅ PRACTICAL 3: Fashion MNIST Classification using CNN

🧠 Theoretical Concepts:

- **CNN (Convolutional Neural Networks):**
  - Extract features using filters.
  - More efficient than ANN for images.
- **Pooling:** Reduces size and computation.
- **Dropout:** Prevents overfitting.

🔍 Code Explanation:

1. **Preprocess Images:**
2. x_train = x_train.reshape(-1, 28, 28, 1) / 255.0
3. **Model Creation:**
4. model = Sequential([
5. Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
6. MaxPooling2D((2,2)),
7. Conv2D(64, (3,3), activation='relu'),
8. MaxPooling2D((2,2)),
9. Flatten(),
10. Dense(128, activation='relu'),
11. Dropout(0.3),
12. Dense(10, activation='softmax')
13.])
14. **Compile:**
15. model.compile(optimizer='adam',
16. loss='sparse_categorical_crossentropy',
17. metrics=['accuracy'])
18. **Train & Evaluate:**

19. model.fit(x_train, y_train, epochs=10)
20. model.evaluate(x_test, y_test)

✅ PRACTICAL 4: CNN for Handwritten Digit Classification (MNIST)

🧠 Theoretical Concepts:

- **MNIST Dataset:** 28x28 pixel images of digits (0–9)
- **CNN used to classify digits.**
- Same as Fashion MNIST structure but different dataset.

🔍 Steps in Code:

1. **Load MNIST:**
2. (x_train, y_train), (x_test, y_test) = mnist.load_data()
3. **Normalize & Reshape:**
4. x_train = x_train.reshape(-1, 28, 28, 1) / 255.0
5. **CNN Architecture:**
6. Conv2D(32, (3,3), activation='relu'),
7. MaxPooling2D(2,2),
8. Flatten(),
9. Dense(128, activation='relu'),
10. Dense(10, activation='softmax')
11. **Compile, Train & Evaluate.**

✅ PRACTICAL 5: OCR using CNN (Optical Character Recognition)

🧠 Theoretical Concepts:

- **OCR:** Recognizes text from image.
- **Use Case:** Read characters (like CAPTCHA, scanned forms).
- **CNN:** Detects patterns in character images.

🔍 Script Explanation:

1. **Load OCR Dataset:** Could be EMNIST or custom.
2. **Reshape and Normalize Images**
3. **Build CNN:**
4. Conv2D → MaxPooling → Dropout → Flatten → Dense → Output
5. **Compile:**
    - Loss: categorical_crossentropy
    - Optimizer: adam
    - Metric: accuracy
6. **Train & Predict**

📝 **Summary of What You Learn:**

Practical	Dataset	Type	Model	Outcome
Boston	Tabular	Regression	ANN	Predict house price
Fashion ANN	Images	Classification	ANN	Clothing item
Fashion CNN	Images	Classification	CNN	Clothing item
MNIST	Digits	Classification	CNN	Handwritten digit
OCR	Character Images	Classification	CNN	Recognize letters

DLL Notes :

Here are the **unit-wise detailed viva answers** for **Deep Learning (SPPU syllabus)**:

✅ Unit 1: Introduction to Deep Learning

1. **What is Deep Learning?**
   Answer:
   Deep Learning is a subset of Machine Learning that uses artificial neural networks with multiple layers (also known as deep neural networks) to model and understand complex patterns in data. It automatically learns features from data without the need for manual feature extraction, which is often required in traditional ML.

2. **Difference between shallow and deep networks?**
   Answer:
   - **Shallow Networks** have 1-2 hidden layers and are suitable for simple tasks.
   - **Deep Networks** have multiple hidden layers and can learn hierarchical representations, making them better for complex tasks like image recognition and NLP.

3. **What is an activation function? Why is it needed?**
   Answer:
   An activation function adds non-linearity to the neural network, allowing it to learn complex patterns. Without it, the network would behave like a linear regression model. Common activation functions:
   - Sigmoid: σ(x)=11+e−x\sigma(x) = \frac{1}{1 + e^{-x}}
   - Tanh: Scales between -1 and 1.
   - ReLU: ReLU(x)=max⁡(0,x)\text{ReLU}(x) = \max(0, x)

4. **Explain gradient descent.**
   Answer:
   Gradient descent is an optimization algorithm used to minimize the loss function by iteratively updating the weights in the opposite direction of the gradient. Learning rate controls the step size during each update.

5. **What is backpropagation?**
   Answer:
   Backpropagation is an algorithm used to calculate the gradient of the loss function with respect to weights. It uses the chain rule to propagate errors backward from the output layer to update weights layer by layer.

✅ Unit 2: Deep Feedforward Networks

1. **What is a Multilayer Perceptron (MLP)?**
   Answer:
   An MLP is a fully connected feedforward neural network with one or more hidden layers. It takes fixed-size input, passes it through hidden layers using activation functions, and produces an output. MLPs are used in classification and regression tasks.

2. **What is overfitting? How can it be avoided?**
   Answer:
   Overfitting happens when a model learns not only the patterns but also the noise in the training data, leading to poor generalization.
   Prevention methods:
   - Regularization (L1/L2)
   - Dropout
   - Data augmentation
   - Early stopping

3. **Explain the concept of weight initialization.**
   Answer:
   Initializing weights properly ensures efficient convergence during training. Poor initialization can cause vanishing or exploding gradients. Common methods:
   - Xavier initialization
   - He initialization

4. **What is the role of learning rate?**
   Answer:
   It determines how much the weights are adjusted during training. A small learning rate can result in slow convergence, while a large one may overshoot the minimum.

5. **What is the Adam optimizer?**
   Answer:
   Adam (Adaptive Moment Estimation) is an optimization algorithm that computes adaptive learning rates for each parameter by storing the exponentially decaying average of past gradients and squared gradients.

✅ Unit 3: Convolutional Neural Networks (CNNs)

1. **What is a convolution operation in CNNs?**
   Answer:
   Convolution applies a filter (kernel) across the input image to extract local features like edges, textures, and shapes. It reduces the number of parameters compared to fully connected layers.

2. **What is pooling and why is it used?**
   Answer:
   Pooling reduces the spatial dimensions (width, height) of the feature maps.
   - Max Pooling: Takes the maximum value in a patch.
   - Average Pooling: Takes the average value.
     It provides translational invariance and reduces computation.

3. **Explain padding in CNNs.**
   Answer:
   Padding adds extra pixels (usually zeros) to the input image's border to control the output size and preserve spatial dimensions after convolution.

4. **Popular CNN architectures?**
   Answer:
   - **LeNet-5**: One of the first CNNs for digit recognition.
   - **AlexNet**: Won ImageNet in 2012, deeper than LeNet.
   - **VGGNet**: Used small filters (3×3), increased depth.
   - **ResNet**: Introduced skip connections to solve vanishing gradient issues.

5. **Applications of CNNs?**
   Answer:
   - Image classification
   - Object detection
   - Facial recognition
   - Medical image analysis

✅ Unit 4: Sequence Modeling (RNNs, LSTMs, GRUs)

1. **What is a Recurrent Neural Network (RNN)?**
   Answer:
   RNNs are neural networks designed for sequential data. They maintain a hidden state

that captures information from previous time steps. Used in language modeling, speech recognition, etc.

2. **What are the issues with RNNs?**
   Answer:
   - **Vanishing gradients**: Gradients become very small during backpropagation, making learning difficult.
   - **Exploding gradients**: Gradients become too large, causing unstable updates.
3. **What is an LSTM and how does it work?**
   Answer:
   Long Short-Term Memory (LSTM) is a type of RNN that can capture long-term dependencies using:
   - **Forget gate**: Decides what to discard from cell state
   - **Input gate**: Updates cell state
   - **Output gate**: Generates output
4. **What is a GRU?**
   Answer:
   A Gated Recurrent Unit is a simpler version of LSTM with fewer gates (reset and update gates) but performs similarly in many tasks.
5. **Applications of RNNs and LSTMs?**
   Answer:
   - Text generation
   - Machine translation
   - Speech synthesis
   - Time-series forecasting

✅ Unit 5: Generative Models and Autoencoders

1. **What is an autoencoder?**
   Answer:
   An autoencoder is a neural network used for unsupervised learning that learns to compress input data into a latent-space representation (encoding) and then reconstruct it (decoding). Useful for dimensionality reduction and noise removal.
2. **What is a Variational Autoencoder (VAE)?**
   Answer:

A VAE introduces a probabilistic approach, where the encoder learns a distribution (mean and variance) instead of fixed values. It allows generating new, similar samples from the learned distribution.

3. **What is a GAN (Generative Adversarial Network)?**
   Answer:
   A GAN consists of:
   - **Generator**: Produces fake data from random noise.
   - **Discriminator**: Classifies real vs fake data.
     Both are trained together in a game-theoretic setting until the generator produces realistic data.
4. **Applications of GANs?**
   Answer:
   - Synthetic image generation
   - Image-to-image translation
   - Super-resolution
   - Art generation
5. **What is a denoising autoencoder?**
   Answer:
   A denoising autoencoder is trained to remove noise from the input. It learns to reconstruct the original, clean data from noisy versions, thus improving robustness.

HPC NOTES :

Here's a **detailed and concise list** to help you with **quick revision** on **High Performance Computing (HPC)**:

Unit 1: Introduction to High Performance Computing

- **HPC**: Use of parallel computing resources to perform high-speed processing for complex problems.
- **Architectural Models**:
  - **Shared Memory**: All processors access a common memory space.
  - **Distributed Memory**: Each processor has its own memory; data communication occurs through message passing.

    - **Hybrid**: Combines both shared and distributed memory architectures for enhanced performance.

- **Performance Metrics**:
    - **Speedup**: Ratio of execution time of the serial program to the parallel program.
    - **Efficiency**: Ratio of speedup to the number of processors; measures how effectively processors are used.
    - **Scalability**: The system's ability to maintain performance when the number of processors is increased.
- **Moore's Law**: Predicted doubling of transistors every two years, enabling growth in parallel computing capabilities.
- **Applications of HPC**: Weather forecasting, drug discovery, AI research, space exploration.

Unit 2: Parallel Programming Platforms and Principles

- **Flynn's Taxonomy**: Classifies systems based on instruction and data streams.
    - **SISD**: Single Instruction, Single Data (traditional sequential processing).
    - **SIMD**: Single Instruction, Multiple Data (vector processors).
    - **MIMD**: Multiple Instruction, Multiple Data (parallel processors).
    - **MISD**: Multiple Instruction, Single Data (rare in practice).
- **Parallel Programming Models**:
    - **Data Parallelism**: Distributes data across processors for simultaneous processing.
    - **Task Parallelism**: Different tasks are executed on different processors.
- **Performance Laws**:
    - **Amdahl's Law**: Limits the speedup achievable due to the serial portion of the program.
    - **Gustafson's Law**: Claims that scaling the problem size increases parallelism, thus improving performance.
- **Parallel Algorithm Design Principles**:
    - **Partitioning**: Dividing the task into sub-tasks.

- **Communication**: Minimizing communication overhead in parallel systems.
- **Agglomeration**: Merging tasks to reduce overhead.
- **Mapping**: Efficient allocation of tasks to processors.

Unit 3: Parallel Programming with MPI

- **MPI Basics**: Standard API for parallel computing with message-passing for distributed systems.
- **Point-to-Point Communication**:
  - **MPI_Send**: Sending data to a specific process.
  - **MPI_Recv**: Receiving data from a specific process.
- **Collective Communication**:
  - **MPI_Bcast**: Broadcasting data from one process to all.
  - **MPI_Reduce**: Performing operations (e.g., summing values) across processes.
  - **MPI_Scatter**: Distributing chunks of data to different processes.
  - **MPI_Gather**: Gathering data from all processes into one.
- **Synchronization**:
  - **MPI_Barrier**: Synchronizes processes at a specific point in the program.

Unit 4: Shared Memory Programming with OpenMP

- **OpenMP Basics**: A shared-memory parallel programming model for multi-threaded CPU programming.
- **Key Constructs**:
  - #pragma omp parallel: Defines a parallel region where threads are created.
  - #pragma omp for: Divides loop iterations among threads.
  - #pragma omp sections: Divides different parts of code for parallel execution.
  - #pragma omp single: Ensures that a code block runs only once by one thread.
- **Synchronization**:
  - **Critical Section**: Ensures that only one thread accesses a critical section of code at a time.

    - **Atomic Operations**: Ensures that a variable is updated atomically without interruption.
    - **Barriers**: Forces threads to wait until all have reached a specific point before continuing.

Unit 5: GPU Programming with CUDA

- **CUDA Basics**: CUDA is a parallel computing platform and programming model for NVIDIA GPUs.
- **GPU vs CPU**: GPUs have thousands of smaller cores optimized for parallel tasks, while CPUs are better at sequential tasks.
- **Thread Hierarchy**:
    - **Grids**: Collection of blocks.
    - **Blocks**: Collection of threads.
    - **Threads**: Individual parallel units of execution.
- **Memory Hierarchy**:
    - **Global Memory**: Slow and large memory accessible by all threads.
    - **Shared Memory**: Fast, small memory within a block.
    - **Constant Memory**: Read-only memory shared by all threads.
    - **Local Memory**: Private memory for each thread.
- **Synchronization**: CUDA uses **__syncthreads()** to synchronize threads within a block.

Unit 6: Performance Optimization and Applications

- **Bottlenecks**: Issues that limit performance, such as slow memory access or inefficient CPU usage.
- **Memory Access Patterns**: Accessing memory in a sequential pattern improves cache utilization.
- **Loop Optimization**:
    - **Unrolling**: Reduces the overhead of loop control by expanding the loop.
    - **Blocking**: Breaks a large data set into smaller blocks to improve cache utilization.

- **Cache Optimization**: Organizing data access patterns to minimize cache misses.
- **Profiling Tools**:
    - **gprof**: A profiling tool to measure where time is spent in the code.
    - **Valgrind**: Helps detect memory leaks and optimize memory usage.
- **Real-World Applications**:
    - **Weather forecasting**: Computational simulations requiring high performance.
    - **Molecular dynamics**: Uses HPC for simulating the interactions between atoms and molecules.
    - **Machine Learning/AI**: Training large models in parallel across GPUs.
    - **Engineering simulations**: Simulating structures under various conditions using computational fluid dynamics (CFD).

Key Takeaways for Viva

- **Understand key performance metrics** (speedup, efficiency, scalability).
- **Know the differences between MPI and OpenMP** (message passing vs shared memory).
- **Understand GPU architecture and CUDA basics**.
- **Know how to identify and resolve performance bottlenecks**.
