1. 📂 hpc_bfs.cpp — *Parallel Breadth First Search (BFS)*

➔ WHAT:

This program performs **Breadth First Search (BFS)** traversal of a binary tree **using parallel processing**.

➔ HOW:

- A binary tree is created where users insert nodes.
- During BFS traversal:
  - At each tree level, nodes are collected.
  - Nodes of the same level are processed **in parallel** using `#pragma omp parallel for`.
  - Outputs and queue updates are protected using `#pragma omp critical` to avoid race conditions.

➔ WHY:

- In BFS, all nodes at the same level are **independent** and can be processed simultaneously.
- Using OpenMP parallelism speeds up BFS traversal for trees with a large number of nodes.

---

2. 📂 hpc_bubble.cpp — *Parallel Bubble Sort*

➔ WHAT:

This program performs **Bubble Sort** on an array in two ways: **sequentially** and **parallelly** using OpenMP.

➔ HOW:

- Array elements are compared and swapped:
  - **Even** indexed pairs are processed together.
  - **Odd** indexed pairs are processed together.
- `#pragma omp parallel for` is used to allow multiple comparisons/swaps in the same phase to happen **at the same time**.
- Execution time is measured and compared between sequential and parallel versions.

➔ WHY:

- Bubble Sort is naturally slow (`O(n^2)`) for large arrays.
- Parallelizing the comparisons reduces total time significantly by handling multiple adjacent swaps simultaneously.

---

3. 📂 hpc_dfs.cpp — *Parallel Depth First Search (DFS)*

➔ WHAT:

This program performs **Depth First Search (DFS)** traversal of a binary tree **using OpenMP parallel tasks**.

➔ HOW:

- Binary Search Tree (BST) is created by inserting nodes based on value.
- DFS traversal starts at the root:
  - A **parallel region** is created with a **single initial task**.
  - Each recursive call to left and right children is done as a **separate OpenMP task** (`#pragma omp task`).
  - `#pragma omp taskwait` ensures tasks are completed properly before moving up the tree.

➔ WHY:

- In DFS, traversing left and right subtrees can be done independently.
- Parallelizing recursive DFS improves performance, especially for **large and balanced trees**.

---

4. 📂 hpc_merge.cpp — *Parallel Merge Sort*

➔ WHAT:

This program sorts an array using **Merge Sort** both **sequentially** and **in parallel** with OpenMP.

➔ HOW:

- Array is divided into halves recursively.
- For the first few levels (`depth < 3`), **parallel sections** are used to sort the left and right halves simultaneously.
- After sorting, the halves are merged into a sorted array.
- Execution time for sequential and parallel versions is compared.

➔ WHY:

- Merge Sort naturally divides the problem into **independent subproblems**.
- Parallelizing the early levels reduces overall sorting time dramatically for **large arrays**.

---

5. 📂 hpc_operation.cpp — *Parallel Min, Max, Sum, Average Reduction*

➔ **WHAT:**

This program finds the **minimum**, **maximum**, **sum**, and **average** values of an array **using OpenMP reduction operations**.

➔ **HOW:**

- Array values are entered by the user.
- For each operation (min, max, sum, average):
  - OpenMP `reduction` clause is used to **combine partial results** calculated by different threads safely and efficiently.
- Results are printed after reduction.

➔ **WHY:**

- Reductions avoid manual locking or critical sections, making operations **faster and cleaner**.
- Useful when you need to **aggregate** results like sum, min, or max in **parallel loops**.

6. 📔 hpc_addition.ipynb — *Parallel Matrix Addition*

➔ WHAT:

This program performs **parallel addition of two matrices** using Python libraries like **NumPy** and **joblib**.

➔ HOW:

- Two matrices of the same size are generated randomly.
- Element-wise addition is normally very fast with NumPy.
- To simulate parallelism manually:
  - The matrix is divided into chunks (rows).
  - Each chunk is added in **parallel** using **joblib's `Parallel` and `delayed`** functions.
- Final result is the sum of two matrices, computed faster.

➔ WHY:

- Matrix operations are **computationally heavy** for very large matrices (1000x1000, 5000x5000, etc.).
- Parallel addition splits work between multiple CPU cores, significantly **reducing computation time** when matrix size is large.

---

7. 📔 hpc_multiplication.ipynb — *Parallel Matrix Multiplication*

➔ WHAT:

This program performs **parallel matrix multiplication** — multiplying two matrices together using Python with **parallel computing techniques**.

➔ HOW:

- Two matrices (`A` and `B`) are randomly generated.
- Normal matrix multiplication is done by:
  - Row of matrix A × Column of matrix B.
- For parallelism:
  - Each row's computation is split into **independent tasks** using **joblib** or **NumPy with parallel tricks**.
- Final product matrix contains the result of A × B multiplication.

➔ WHY:

- Matrix multiplication (`O(n^3)`) becomes **extremely slow** for large matrices.
- By distributing rows (or parts of rows) to different CPU cores, **parallelism speeds up the computation** drastically, especially useful in fields like AI, physics simulations, or graphics.

DL

8. 📔 DLPr1_updated.ipynb — *Basic Perceptron for Logic Gates*

➔ WHAT:

This program implements a simple **Perceptron model** to simulate basic **logic gates** (like AND, OR) using **Deep Learning** concepts.

➔ HOW:

- Inputs (x) and corresponding outputs (y) for a logic gate (like AND) are defined.
- A **simple neuron** (perceptron) is built with:
  - Random initial weights and bias.
  - Forward propagation to calculate output using a weighted sum.
  - An **activation function** (like step function) to decide final output (0 or 1).
  - Training loop where:
    - Error between predicted and actual output is calculated.
    - Weights and bias are **updated** based on error using **Perceptron learning rule**.
- After enough epochs (training cycles), the model correctly learns the logic gate behavior.

➔ WHY:

- Perceptron is the **fundamental building block** of Deep Learning.
- Simulating logic gates shows how neural networks can **mimic real logical decision-making**, and it teaches basics like weights, bias, activation, and learning in a very simple and visual way.

---

9. 📔 DLPr2.ipynb — *Single-Layer Neural Network (Sigmoid Activation)*

➔ WHAT:

This program builds a **single-layer neural network** that uses the **Sigmoid activation function** for predicting outputs from input data.

➔ HOW:

- Data points (x) and outputs (y) are set up (often non-linearly separable).
- Initialize random weights and bias.
- **Forward pass**:
  - Compute weighted sum.
  - Apply **Sigmoid function** to output a value between 0 and 1.
- **Backward pass** (training):
  - Calculate error between prediction and actual output.
  - Update weights and bias using **gradient descent** (using derivative of Sigmoid during weight adjustment).
- After training for several epochs, the network can make good predictions.

➔ WHY:

- Sigmoid activation helps to handle outputs that are **probabilistic** (between 0 and 1).
- It also introduces **non-linearity**, which simple perceptrons can't handle properly.
- This is the first step towards understanding **deep neural networks**.

---

10. 📔 DLPr3.ipynb — *Multi-Layer Perceptron (MLP) from Scratch*

➔ WHAT:

This program implements a **Multi-Layer Perceptron (MLP)** — a small **deep neural network** with one or more hidden layers manually coded from scratch.

➔ HOW:

- The input data is prepared.
- A neural network structure is defined:
  - **Input Layer** → **Hidden Layer(s)** → **Output Layer**.
- Forward Pass:
  - Each layer computes a weighted sum + bias.
  - Activation functions (like Sigmoid or ReLU) are applied between layers.
- Backward Pass (Training):
  - **Backpropagation** is manually implemented to compute errors layer-by-layer.
  - Gradients are calculated using derivatives.
  - Weights and biases are updated using **Gradient Descent**.
- After multiple epochs, the model learns the mapping between inputs and outputs.

➔ WHY:

- MLPs can **model complex relationships** that single-layer networks cannot.
- They are the base structure behind **modern deep learning architectures**.
- Building an MLP manually helps you deeply understand **how deep learning models learn** (forward and backward passes).