Code explanation:

Of course! Let’s go through your Colab setup step-by-step:

**🧹 Explanation of Your Google Colab Code**

**Step 1️⃣: Install GCC Compiler (with OpenMP support)**

!apt-get install g++

* ! means it's a **shell command** in Colab.
* apt-get install g++ installs the **G++ compiler** (C++ compiler).
* You need G++ because Colab runs Python by default, but your code is in C++.
* G++ can also **compile programs that use OpenMP** (OpenMP is for parallelism).

**Step 2️⃣: Create the C++ Program Dynamically**

code = """

// Entire C++ code (your Parallel Bubble Sort program)

"""

* You store the full C++ program inside a Python **multiline string** (""" ... """).

**Inside the C++ Program:**

* #include<iostream>: Standard input-output library.
* #include<omp.h>: OpenMP library for parallelism (#pragma omp uses this).

**Functions defined:**

1. **bubble(int array[], int n)**
   * Regular Bubble Sort (Sequential)
   * Two nested loops to compare adjacent elements and swap if needed.
2. **pBubble(int array[], int n)**
   * **Parallel Bubble Sort using OpenMP**:
   * It **sorts odd-indexed elements first**, then **even-indexed elements**.
   * #pragma omp parallel for tells compiler to **split loop iterations across threads**.
   * #pragma omp barrier forces synchronization — all threads must finish odd phase before even phase starts.
3. **printArray(int arr[], int n)**
   * Simple function to print the array.
4. **main()**
   * Creates an array of size 10 (with elements from 10 down to 1).
   * Measures **execution time** of:
     + Sequential Bubble Sort
     + Parallel Bubble Sort
   * Then prints sorted arrays and times.

**Step 3️⃣: Save the code into a .cpp file**

with open("parallel\_bubble\_sort.cpp", "w") as file:

file.write(code)

* open(..., "w"): Open file for writing.
* file.write(code): Writes the C++ code into a file called parallel\_bubble\_sort.cpp.
* ✅ Now, the C++ code is saved on Colab!

**Step 4️⃣: Print Success Message**

print("✅ C++ code written to parallel\_bubble\_sort.cpp")

* Just to inform you that writing was successful!

**🧠 Summary**

| **Python Part** | **Purpose** |
| --- | --- |
| Install G++ | To compile C++ code |
| Store C++ code in a string | Dynamic writing without manual upload |
| Write code to .cpp file | Save your program in Colab filesystem |
| Compile and run later using shell | (!g++ -fopenmp ..., !./a.out) |

**🔥 Bonus Tip:**

Once the code is written, you need to **compile and run** it like this:

# Compile with OpenMP support

!g++ -fopenmp parallel\_bubble\_sort.cpp -o parallel\_bubble\_sort

# Run the compiled program

!./parallel\_bubble\_sort

Would you also like me to explain **how OpenMP parallelism is working inside pBubble step-by-step**? (It’s super interesting!) 🚀

Theory:

Sure! I'll give you **proper theory** you can use for your **practical file** or **exam** based on your program.

**🔵 Title:**

**Parallel Bubble Sort using OpenMP**

**🔵 Aim:**

To implement **sequential** and **parallel** versions of **Bubble Sort** using **OpenMP** and compare their execution time.

**🔵 Theory:**

**✨ What is Bubble Sort?**

* Bubble Sort is a **simple sorting algorithm**.
* It works by **repeatedly swapping adjacent elements** if they are in the wrong order.
* After each pass, the largest element bubbles up to its correct position.

**✨ What is Parallel Bubble Sort?**

* In traditional Bubble Sort, comparisons happen sequentially.
* In **Parallel Bubble Sort**, **multiple comparisons** and **swaps** are performed **simultaneously** using **threads**.
* This reduces the total execution time, especially for large arrays.

**✨ Why OpenMP?**

* **OpenMP** is an API that supports **multi-threading** in C/C++ programs.
* It allows us to easily parallelize loops and sections of code using simple compiler directives (#pragma omp parallel for).

**✨ Parallelization Strategy:**

1. **Odd Phase:**
   * Compare and swap elements at positions (1,2), (3,4), (5,6), ...
2. **Even Phase:**
   * Compare and swap elements at positions (0,1), (2,3), (4,5), ...
3. **Barrier Synchronization:**
   * After completing odd-phase comparisons, all threads must wait before proceeding to even-phase.
   * This ensures correctness of sorting.

**🔵 Algorithm Steps:**

**1. Sequential Bubble Sort**

* Loop through the array.
* Compare each pair of adjacent elements.
* Swap if they are in wrong order.
* Repeat until the array is sorted.

**2. Parallel Bubble Sort**

* For each iteration:
  + In parallel, compare and swap odd indexed elements.
  + Synchronize (barrier).
  + In parallel, compare and swap even indexed elements.
* Repeat until the array is sorted.

**🔵 OpenMP Directives Used:**

| **Directive** | **Meaning** |
| --- | --- |
| #pragma omp parallel for | Splits loop iterations among threads |
| #pragma omp barrier | Forces all threads to wait until all have completed |

**🔵 Advantages of Parallel Bubble Sort:**

* **Reduced Time:** Multiple elements are sorted at once.
* **Better CPU Utilization:** Multithreading utilizes available cores.
* **Simple to Implement:** OpenMP makes it easy to parallelize loops.

**🔵 Limitations:**

* Bubble Sort is inherently **O(n²)** even with parallelism.
* For very large datasets, **better parallel algorithms** like **Merge Sort** or **Quick Sort** are preferred.
* Synchronization (barriers) can cause **overhead**.

**🔵 Conclusion:**

* Parallel Bubble Sort using OpenMP improves performance compared to sequential Bubble Sort.
* However, for extremely large datasets, more efficient parallel sorting algorithms should be considered.

**📚 Short Summary Table:**

| **Type** | **Time Complexity** | **Parallelism** |
| --- | --- | --- |
| Sequential Bubble Sort | O(n²) | No |
| Parallel Bubble Sort | O(n²) (but faster in practice) | Yes (using threads) |

Would you also like me to give a **small ready-to-paste "Observation Table"** and **"Result"** format 📄 for your practical file?  
(If yes, I can prepare it too!) ✅