HPC Practical

Assignment 1

Q. Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS.

Amdahl's Law (Simple Explanation)

Amdahl's Law says:

> The speedup of a program using multiple processors is limited by the part of the program that cannot be parallelized.

📊 The Formula

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{N}}$$

Where:

- P = fraction of the program that **can be parallelized** (between 0 and 1)
- $(1 - P)$ = fraction that **must run serially**
- N = number of processors
- **Speedup** = how many times faster the parallel version is compared to the serial one

⚡ Example

Let's say:

- 80% of a program can be parallelized → $P = 0.8$
- 20% is serial → $1 - P = 0.2$
- You use 4 processors → $N = 4$

$$\text{Speedup} = \frac{1}{0.2 + \frac{0.8}{4}} = \frac{1}{0.2 + 0.2} = \frac{1}{0.4} = 2.5$$

So even with 4 processors, the max speedup is **2.5×**, not 4×!

Lets Learn Open MP

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

using namespace std;

int main()
{
    #pragma omp parallel
    {
        cout<<"Hello World!\n";
    }
}
```

What Is `#include <omp.h>`?

- This line includes the **OpenMP library header**.

- `omp.h` gives access to OpenMP features like:

    - parallel regions

    - thread management

    - synchronization tools (e.g., barriers, locks)

- It's necessary when you're using OpenMP in C or C++.

```
g++ -fopenmp code.cpp
```

What Is `#pragma omp parallel`?

- `#pragma` is a **compiler directive**—it tells the compiler to do something special.

- `#pragma omp parallel` tells the compiler:

    "Start a parallel region. Run the block of code below using **multiple threads**."

```cpp
#pragma omp parallel
{
    cout<<"Hello World!\n";
}
```

will be executed **once by each thread**, **in parallel**.

What Happens During Execution?

Let's say your system has 4 cores, and OpenMP is configured to use 4 threads. Then:

- All 4 threads will run the `cout << "Hello World!\n";` line **simultaneously**.

- You'll likely see "Hello World!" printed **4 times** (though the output order may vary).

```
sahil@b450-gaming-x:~/java/hpc$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

We can set how many threads to use in parallel

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

using namespace std;


int main()
{
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        cout<<"Hello World!\n";
    }

}
```



```
sahil@b450-gaming-x:~/java/hpc$ g++ -fopenmp code.cpp
sahil@b450-gaming-x:~/java/hpc$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
```
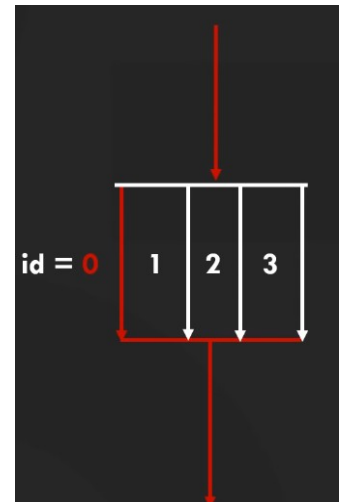
Thread ID

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

using namespace std;


int main()
{
    omp_set_num_threads(4);
    int id;
    #pragma omp parallel
    {
        id = omp_get_thread_num();
        cout<<"Thread : "<<id<<" Says : Hello World\n";
    }

}
```


Sometimes Parallel execution takes more time than serial execution

What Is False Sharing?

In parallel computing, **false sharing** happens when multiple threads **write to different variables that live on the same cache line** in memory.

Even though the threads are working on *different* elements of an array (like `thread_sum[id]`), if those elements are stored **next to each other in memory**, the CPU treats them as part of the same cache block. So:

> If two threads modify adjacent memory, they end up invalidating each other's cache unnecessarily. This causes a lot of slow memory traffic.

In Your Code:

```cpp
    int thread_sum[THREADS] = {0};  // initialize array to zero
```

Each thread writes to its own index:

```cpp
    thread_sum[id] += i;
```

BUT — those `thread_sum[0]`, `thread_sum[1]`, etc., are stored **right next to each other** in memory. So threads writing to different indices **still interfere** with each other's performance due to false sharing.

Result:

Even though the work is split across threads, they're slowing each other down due to cache thrashing. So, **you may see worse performance than the serial version**, especially if:

- The task is simple (like a plain sum)

- The number of iterations is not very large

- There's false sharing or thread overhead

Parallel For

Summation Example

```cpp
#include <iostream>
#include <omp.h>
#include <iomanip>
using namespace std;

const int N = 1000000;
const int THREADS = 4;

double serialSum(int& sum) {
    sum = 0;
    double start = omp_get_wtime();
    for (int i = 1; i <= N; i++) {
        sum += i;
    }
    double end = omp_get_wtime();
    return end - start;
}

double parallelSum(int& sum) {
    sum = 0;
    int thread_sum[THREADS] = {0};  // initialize array to zero

    double start = omp_get_wtime();
    omp_set_num_threads(THREADS);

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        thread_sum[id] = 0;

        #pragma omp for
        for (int i = 1; i <= N; i++) {
            thread_sum[id] += i;
        }
    }

    for (int i = 0; i < THREADS; i++) {
        sum += thread_sum[i];
    }

    double end = omp_get_wtime();
    return end - start;
}

int main() {
    int sum;
    double serial_time = serialSum(sum);
    cout << "Serial Sum: " << sum << "\n";
    cout << "Time taken (serial): " << fixed << setprecision(8) << serial_time << "
seconds\n\n";

    double parallel_time = parallelSum(sum);
    cout << "Parallel Sum: " << sum << "\n";
    cout << "Time taken (parallel): " << fixed << setprecision(8) << parallel_time <<
" seconds\n";

    return 0;
}
```
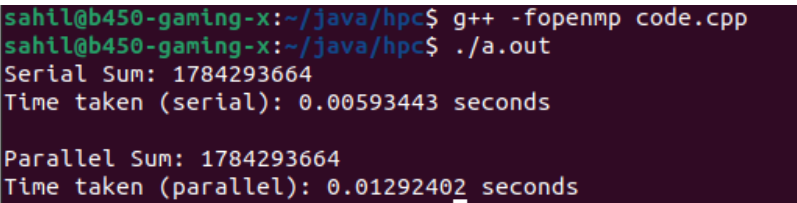
```
sahil@b450-gaming-x:~/java/hpc$ g++ -fopenmp code.cpp
sahil@b450-gaming-x:~/java/hpc$ ./a.out
Serial Sum: 1784293664
Time taken (serial): 0.00593443 seconds

Parallel Sum: 1784293664
Time taken (parallel): 0.01292402 seconds
```

OpenMP: Shared and Private Variables

In OpenMP, variables inside a `#pragma omp parallel` block can be either:

Type	Meaning
Shared	All threads **see and use the same** variable (shared memory).
Private	Each thread has its **own copy** of the variable (independent memory).

Default Behavior

By default:

- **Variables declared** *before* **the parallel block** are **shared**
- **Variables declared** *inside* **the parallel block** are **private**

But you can control it explicitly.

```
#pragma omp parallel shared(x) private(y)
```

- `x` is shared — all threads use the same memory location
- `y` is private — each thread gets its own copy


```
int n = 10;          // shared
int a = 7;           // shared

#pragma omp parallel for
for (int i = 0; i < n; i++)  // i private
{    int b = a + i;          // b private

     ...
}
```

It is better to declare the loop iteration variables inside the parallel region.

Shared variables increases the overhead, so minimize the number of shared variables.

EXPLICIT RULES
Private

```
#pragma omp parallel for shared(n, a) private(b)
for (int i = 0; i < n; i++)
{
        b = a + i;

        ...
}
```

The private(list) clause declares that all the variables in list are shared.

n and a are shared variables.
b is a private variable.

DEFAULT
Default (shared)

```
int a, b, c, n;
...

#pragma omp parallel for default(shared)
for (int i = 0; i < n; i++)
{
        // using a, b, c
}
```

```
int a, b, c, n;
…

#pragma omp parallel for default(shared) private(a,
b)
for (int i = 0; i < n; i++)
{
    // a and b are private variables
    // c and n are shared variables
}
```

DEFAULT
Default (none)

```
int n = 10;
std::vector<int> vector(n);
int a = 10;

#pragma omp parallel for default(none) shared(n,
vector,a)
for (int i = 0; i < n; i++)
{
        vector[i] = i * a;
}
```

SOME KEY POINTS:

Always write parallel regions with the default(none) clause.

Declare private variables inside parallel regions whenever possible.

Critical Section

Critical section prevents multiple threads from accessing a section of code at the same time.

Only one active thread can update the data referenced by the code.

Assignment 1

Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS.

Parallel DFS Code

```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <omp.h>
using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node)
{
    stack<int> s;
    s.push(node);

    while (!s.empty())
    {
        int curr_node = s.top();
        s.pop();

        if (!visited[curr_node])
         {
            visited[curr_node] = true;
            cout << curr_node << " ";

            // Push adjacent nodes to stack (in reverse order for correct DFS)
            for (int i = graph[curr_node].size() - 1; i >= 0; i--)
             {
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                    s.push(adj_node);
                }}}}
}

int main() {
    int n, m, start_node;
    cout << "Enter number of Nodes, Edges, and Starting Node of the graph:\n";
    cin >> n >> m >> start_node;

    cout << "Enter pairs of nodes representing edges:\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // Parallel initialization of the visited array
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }
    dfs(start_node);
    return 0;
}
```

Parallel BFS Code

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

using namespace std;

const int MAX = 100; // Maximum number of nodes in the graph
vector<int> graph[MAX]; // Adjacency list representation of the graph
bool visited[MAX]; // To keep track of visited nodes

// Function to add an edge to the graph
void addEdge(int u, int v) {
    graph[u].push_back(v);
    graph[v].push_back(u); // Since it's an undirected graph
}

// Parallel BFS using OpenMP
void parallelBFS(int start) {
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();

        // Print the current node
        cout << "\t" << node;

        // Parallelizing the exploration of adjacent nodes
        #pragma omp parallel for
        for (int i = 0; i < graph[node].size(); i++) {
            int neighbor = graph[node][i];

            // Check if the neighbor is visited
            if (!visited[neighbor]) {
                #pragma omp critical
                {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
    }
}

int main() {
    int n, m, start_node;

    // Input number of nodes, edges, and the starting node for BFS
    cout << "Enter number of nodes and edges: ";
    cin >> n >> m;

    cout << "Enter the edges (pairs of nodes):\n";
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        addEdge(u, v);
    }
```

```
    cout << "Enter the starting node for BFS: ";
    cin >> start_node;

    // Initialize visited array
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }

    // Perform parallel BFS
    cout << "Parallel BFS Traversal: ";
    parallelBFS(start_node);
    cout << endl;

    return 0;
}
```

# Basics of DFS (Depth-First Search):

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. The idea is to start from a node, explore one of its neighbors, then continue to explore that neighbor's neighbors, and so on. Once we reach a leaf node or a node with no unvisited neighbors, we backtrack and explore the next neighbor.

Sequential DFS:

In the **sequential version** of DFS, the algorithm explores one path completely before moving on to the next one. For example, given a tree, we explore a branch fully before switching to another branch.

Example of Sequential DFS on a tree:

Consider the following tree:

```
       1
     /   \
    2     3
   / \   / \
  4   5 6   7
```
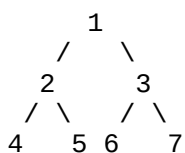
DFS traversal sequence might look like:
1 → 2 → 4 → 5 → 3 → 6 → 7

Parallel DFS Concept:

The main challenge in parallelizing DFS is that it is a recursive algorithm, and the traversal of one branch depends on the traversal of other branches. So, the main goal is to find **independent branches** in the graph that can be explored in parallel.

In **parallel DFS**, you can explore different branches (or different neighbors) of a node concurrently. This means that when you are exploring a node's neighbors, OpenMP can create multiple threads to explore different neighbors at the same time, thus reducing the time needed for traversal.

Parallel DFS with OpenMP:

Let's consider a graph and apply OpenMP to parallelize DFS.

Example of a Graph:

```
    1
   /   \
  2     3
 / \   / \
4   5 6   7
```
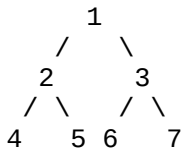
The goal is to parallelize DFS traversal so that the search of different branches can happen simultaneously.

Step-by-Step Parallel DFS Execution:

1. **Initialization**: We start by marking the starting node (let's say node 1) as visited.

2. **DFS on Node 1**: At node 1, we find two neighbors: node 2 and node 3.

3. **Parallel Execution**: OpenMP will create separate threads for each neighbor of node 1 (i.e., node 2 and node 3).

   - One thread will explore the subtree rooted at node 2.

   - Another thread will explore the subtree rooted at node 3.

4. **DFS on Node 2**: In parallel, thread 1 explores node 2, which has neighbors 4 and 5. OpenMP will again create threads for node 4 and node 5.

5. **DFS on Node 3**: At the same time, thread 2 explores node 3, which has neighbors 6 and 7. OpenMP will create threads for node 6 and node 7.

6. **Backtracking**: After exploring the neighbors of each node, the threads backtrack to their respective parent nodes.

Parallel BFS (Breadth-First Search) using OpenMP is about taking advantage of parallelism in the level-wise exploration of a graph. BFS explores a graph level by level, starting from a source node. At each level, it explores all the neighboring nodes of the current node and processes them before moving to the next level.

BFS (Breadth-First Search) Overview:

BFS explores a graph in breadth-first order, visiting all nodes at a distance d from the source before visiting nodes at distance d+1. It uses a queue to manage the nodes to be explored. BFS is ideal for finding the shortest path in an unweighted graph, and it works level by level.

Sequential BFS:

In a sequential BFS, you explore the neighbors of a node and then move on to the next level. The algorithm starts from the root node, enqueues all its neighbors, processes them one by one, then processes the next level.

Here's how BFS works sequentially:

1. **Initialize**: Start with a queue and enqueue the source node. Mark it as visited.

2. **Explore Neighbors**: Dequeue a node from the queue, explore all its unvisited neighbors, mark them as visited, and enqueue them.

3. **Repeat**: Continue dequeuing nodes and exploring their neighbors until the queue is empty.

Parallel BFS Concept:

Parallelizing BFS means that we can process multiple nodes in parallel, especially when exploring all the neighbors of nodes at the same level. Since all nodes at the same level are independent of each other, we can explore them simultaneously using multiple threads in parallel.

Challenges:

- **Queue Management**: A queue is a critical data structure in BFS. Parallelizing BFS involves handling concurrent access to the queue.

- **Synchronization**: Since multiple threads may attempt to enqueue or dequeue nodes simultaneously, we need to ensure that queue operations are done safely.

Parallel BFS Using OpenMP:

OpenMP can be used to parallelize the exploration of nodes at each level. Specifically, we can use OpenMP to parallelize the exploration of neighbors at each level. We will enqueue neighbors of a node, and multiple threads can process them simultaneously.

Steps to Parallelize BFS:

1. **Initialize**: Create a queue and enqueue the starting node. Also, mark the starting node as visited.

2. **Level-wise Parallelism**:

   - For each level, OpenMP can parallelize the exploration of all the neighbors of the nodes in the current level.

   - The threads in parallel will explore the neighbors of nodes in the current level.

   - After all nodes at the current level are processed, move to the next level.

3. **Synchronization**: After processing all nodes at a particular level, synchronize the threads before moving on to the next level.

Assignment 2

Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

```cpp
#include <iostream>
#include <iomanip>
#include <omp.h>
#include <cstdlib> // For rand() and srand()
#include <ctime>   // For time()

using namespace std;

// Function declarations
void mergesort(int a[], int i, int j);
void merge(int a[], int i1, int j1, int i2, int j2);

// Recursive Merge Sort with Parallelism
void mergesort(int a[], int i, int j) {
    if (i < j) {
        int mid = (i + j) / 2;

        // Cutoff to avoid too small tasks
        if (j - i < 1000) {
            mergesort(a, i, mid);
            mergesort(a, mid + 1, j);
        } else {
            #pragma omp task firstprivate(a, i, mid)
            {
                mergesort(a, i, mid);
            }
            #pragma omp task firstprivate(a, mid, j)
            {
                mergesort(a, mid + 1, j);
            }
            #pragma omp taskwait
        }

        merge(a, i, mid, mid + 1, j);
    }
}

// Merge two sorted subarrays
void merge(int a[], int i1, int j1, int i2, int j2) {
    int size = j2 - i1 + 1;
    int* temp = new int[size];
    int i = i1, j = i2, k = 0;

    while (i <= j1 && j <= j2) {
        if (a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }

    while (i <= j1)
        temp[k++] = a[i++];
    while (j <= j2)
        temp[k++] = a[j++];

    for (i = i1, k = 0; i <= j2; i++, k++)
        a[i] = temp[k];
```

```cpp
        delete[] temp;
}

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

    int* a = new int[n];
    int* b = new int[n];

    // Random array generation
    srand(time(0)); // Seed random number generator
    cout << "\nGenerating random array...\n";
    for (int i = 0; i < n; i++) {
        a[i] = rand() % 100000; // random numbers between 0 and 99999
        b[i] = a[i]; // Copy for parallel sorting
    }

    cout << "\nRandom array generated.\n";

    double start_time, end_time;

    // Sequential mergesort
    start_time = omp_get_wtime();
    mergesort(a, 0, n - 1);
    end_time = omp_get_wtime();
    double seq_time = end_time - start_time;

    cout << "\nSequential Time: " << fixed << setprecision(8) << seq_time << "
seconds" << endl;

    // Parallel mergesort
    start_time = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp single
        {
            mergesort(b, 0, n - 1);
        }
    }
    end_time = omp_get_wtime();
    double par_time = end_time - start_time;

    cout << "\nParallel Time: " << fixed << setprecision(8) << par_time << "
seconds" << endl;

    if (n <= 100) {
        cout << "\nSorted array:\n";
        for (int i = 0; i < n; i++) {
            cout << b[i] << " ";
        }
        cout << endl;
    }

    delete[] a;
    delete[] b;

    return 0;
}
```

Why This Happens:

1. **Without the `if` cutoff:**

   - Even **very small subarrays** (size 1, 2, 5 elements) are **creating OpenMP tasks**.

   - Creating and managing tasks **has overhead** — time needed to set up, schedule, and clean up tasks.

   - **Overhead > Work** for small arrays.

   - Result: **Parallel becomes slower**.

2. **With the `if (j-i<1000)` cutoff:**

   - Only **big subarrays** create tasks (big useful jobs).

   - **Small parts** are **handled sequentially ➜** No useless task creation.

   - So **task overhead is minimized** and **real parallelism** is achieved.

   - Result: **Parallel becomes faster**.

Part 1: What does this mean?

```
#pragma omp task firstprivate(a, i, mid)
{
    mergesort(a, i, mid);
}
#pragma omp task firstprivate(a, mid, j)
{
    mergesort(a, mid + 1, j);
}
#pragma omp taskwait
```

Meaning:

- `#pragma omp task`:
  ➜ **Tells OpenMP to create a new task. ➜** This task will **run independently** from the parent.

- `firstprivate(a, i, mid)`:
  ➜ **Copies** the variables `a`, `i`, and `mid` into the task **by value** (i.e., each task gets its own copy of `i`, `mid`, but points to the same array `a`).

- `mergesort(a, i, mid);`:
  ➜ The task runs **mergesort** on the **first half** of the array.

- Similarly, the second `#pragma omp task` sorts the **second half**.

- `#pragma omp taskwait`:
  ➜ **Waits** for both tasks to finish **before merging** the two sorted halves.

Short Summary:

Directive	Meaning
`#pragma omp task`	Create a new independent unit of work.
`firstprivate`	Pass variables safely into the task.
`#pragma omp taskwait`	Wait for all child tasks to finish.

Part 2: How Sequential MergeSort Works?

```
start_time = omp_get_wtime();
mergesort(a, 0, n - 1);
end_time = omp_get_wtime();
```

Here **no `#pragma omp parallel` is used**.

So it **calls `mergesort` normally**:

- `mergesort` splits the array in two recursively.

- **Each call happens sequentially.**

- **No threads**, no tasks, just normal recursion.

- After sorting two halves, **merge them**.

Timeline of Sequential Execution:

```
Main Thread
└── mergesort(a, 0, n-1)
      ├── mergesort(a, 0, mid)
      │     ├── mergesort(a, 0, mid1)
      │     └── mergesort(a, mid1+1, mid)
      └── mergesort(a, mid+1, n-1)
```

(All functions wait for their child calls to finish.)

Part 3: How Parallel MergeSort Works?

```
start_time = omp_get_wtime();
#pragma omp parallel
{
    #pragma omp single
    {
        mergesort(b, 0, n - 1);
    }
}
end_time = omp_get_wtime();
```

Here **parallel region** is created first:

```
#pragma omp parallel
```

➔ Spawns multiple threads (e.g., 4, 8, or however many cores you have).

Then inside, a **single thread** starts the work:

```
#pragma omp single
```

➜ Only **one thread** calls `mergesort(b, 0, n-1)`.

But **inside `mergesort`**, wherever it sees:

```
#pragma omp task
```

the current thread **creates new tasks** that **other threads** can **pick up and execute**!

Threads **dynamically steal** and execute tasks — this is called **work sharing**.

Simple Example: Parallel Merge Sort with OpenMP

We will use an array of 8 integers and **2 threads** (you can increase the number of threads depending on your machine's capabilities).

Array Example:

```
[38, 27, 43, 3, 9, 82, 10, 22]
```

Step-by-Step Parallel Execution

1. **Initial Split**:

   - The array `[38, 27, 43, 3, 9, 82, 10, 22]` is split into two halves:

     - Left: `[38, 27, 43, 3]`

     - Right: `[9, 82, 10, 22]`

2. **Parallel Sorting**:

   - These two halves are sorted **in parallel** using tasks.

   - **Thread 1** will handle the sorting of the left half `[38, 27, 43, 3]`.

   - **Thread 2** will handle the sorting of the right half `[9, 82, 10, 22]`.

3. **Recursive Sorting of Subarrays**:

   - The left half `[38, 27, 43, 3]` is split further into `[38, 27]` and `[43, 3]`, which will be sorted by the same threads.

   - The right half `[9, 82, 10, 22]` is split further into `[9, 82]` and `[10, 22]`, which will also be sorted by the threads.

4. **Merging**:

   - Once the subarrays are sorted, the **merge** step combines them into sorted subarrays. This step is performed sequentially (as merging is inherently sequential).

Bubble Sort

```cpp
#include <iostream>
#include <iomanip>
#include <omp.h>
#include <cstdlib> // For rand()
#include <ctime>   // For time()

using namespace std;

// Sequential Bubble Sort
void sequentialBubbleSort(int a[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (a[j] > a[j+1]) {
                swap(a[j], a[j+1]);
            }
        }
    }
}

// Parallel Bubble Sort
void parallelBubbleSort(int a[], int n) {
    for (int i = 0; i < n; i++) {
        // Even phase
        if (i % 2 == 0) {
            #pragma omp parallel for
            for (int j = 0; j < n-1; j += 2) {
                if (a[j] > a[j+1]) {
                    swap(a[j], a[j+1]);
                }
            }
        }
        // Odd phase
        else {
            #pragma omp parallel for
            for (int j = 1; j < n-1; j += 2) {
                if (a[j] > a[j+1]) {
                    swap(a[j], a[j+1]);
                }
            }
        }
    }
}

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

    int* a = new int[n];
    int* b = new int[n];

    // Random array generation
    srand(time(0)); // Seed random number generator
    cout << "\nGenerating random array...\n";
    for (int i = 0; i < n; i++) {
        a[i] = rand() % 100000; // Random numbers between 0 and 99999
        b[i] = a[i];            // Copy for parallel sort
    }

    cout << "\nRandom array generated.\n";
```

```cpp
    double start_time, end_time;

    // Sequential Bubble Sort
    start_time = omp_get_wtime();
    sequentialBubbleSort(a, n);
    end_time = omp_get_wtime();
    double seq_time = end_time - start_time;

    cout << "\nSequential Bubble Sort Time: " << fixed << setprecision(8) <<
seq_time << " seconds" << endl;

    // Parallel Bubble Sort
    start_time = omp_get_wtime();
    parallelBubbleSort(b, n);
    end_time = omp_get_wtime();
    double par_time = end_time - start_time;

    cout << "\nParallel Bubble Sort Time: " << fixed << setprecision(8) <<
par_time << " seconds" << endl;

    // Display sorted array (only if small size)
    if (n <= 100) {
        cout << "\nSorted array:\n";
        for (int i = 0; i < n; i++) {
            cout << b[i] << " ";
        }
        cout << endl;
    }

    delete[] a;
    delete[] b;

    return 0;
}
```

Let's break down **how the parallel bubble sort** will work when you have **10 elements** in the array and **4 threads** available.

1. The Array

Let's assume you have this example array of 10 elements:

`[9, 7, 3, 1, 2, 8, 5, 4, 6, 0]`

2. Thread Assignment

The goal is to **parallelize the bubble sort** using **4 threads**. We'll use the even-odd technique to split the work into two phases (even and odd), ensuring that no two threads overlap on the same element.

3. Even-Odd Phases

As explained earlier, the bubble sort process consists of **even** and **odd** phases:

- **Even phase**: Compare adjacent pairs `(0,1), (2,3), (4,5), (6,7), (8,9)`

- **Odd phase**: Compare adjacent pairs `(1,2), (3,4), (5,6), (7,8)`

4. Execution Steps with 4 Threads

Let's assume you have 4 threads available. Since we want to **maximize parallel execution**, we'll divide the work of comparing pairs across these 4 threads. This is how the work can be distributed:

Step-by-Step Execution:

1. First Phase: Even Indexes (`i % 2 == 0`)

We'll split this phase into **two parallel loops** (with 4 threads). So, the even-phase loop for comparisons is:

Thread	Pairs Compared
Thread 1	(0,1) → compare elements 0 and 1 (swap if needed)
Thread 2	(2,3) → compare elements 2 and 3 (swap if needed)
Thread 3	(4,5) → compare elements 4 and 5 (swap if needed)
Thread 4	(6,7) → compare elements 6 and 7 (swap if needed)
Main Thread (if needed)	(8,9) → compare elements 8 and 9 (swap if needed)

- Thread 1 will work on pair (0,1).
- Thread 2 will work on pair (2,3).
- Thread 3 will work on pair (4,5).
- Thread 4 will work on pair (6,7).

After that, the **main thread** can compare the last pair (8,9) (if OpenMP scheduling is not fully loaded).

Once all threads finish comparing their respective pairs in the even phase, the array will look like this (assuming swaps were needed):

```
[7, 9, 1, 3, 2, 8, 4, 5, 0, 6]
```

2. Second Phase: Odd Indexes (`i % 2 == 1`)

Now, we move to the **odd-phase** comparison. This is where we compare pairs `(1,2), (3,4), (5,6), (7,8)`:

Thread	Pairs Compared
Thread 1	(1,2) → compare elements 1 and 2 (swap if needed)
Thread 2	(3,4) → compare elements 3 and 4 (swap if needed)
Thread 3	(5,6) → compare elements 5 and 6 (swap if needed)
Thread 4	(7,8) → compare elements 7 and 8 (swap if needed)

Main Thread (if needed) —

Here, each thread works on an **odd-indexed pair**:

- Thread 1 compares elements (1,2)
- Thread 2 compares elements (3,4)

- Thread 3 compares elements (5,6)

- Thread 4 compares elements (7,8)

After completing the odd-phase, the array may look like:

`[7, 1, 9, 2, 3, 4, 5, 0, 6, 8]`

3. Repeating Phases

- This **even-odd comparison loop** will **repeat several times**. Each time, you'll bubble up the largest element towards the end of the array.

- You would **alternate between even and odd phases** and continue until the array is fully sorted.

5. Parallel Execution with 4 Threads:

With **4 threads**, **parallelization** takes place in two main parts:

1. Each phase (even and odd) is broken down into smaller comparisons.

2. Each comparison is executed in parallel using OpenMP's `#pragma omp parallel for` directive.

For example, in the **even phase**:

- 4 threads can handle 4 comparisons simultaneously (pairs `(0,1)`, `(2,3)`, `(4,5)`, `(6,7)`).

- The **odd phase** works similarly, with each thread handling one comparison at a time (pairs `(1,2)`, `(3,4)`, `(5,6)`, `(7,8)`).

6. The Parallelization Efficiency:

The **degree of parallelism** (number of threads) can make a significant impact on the speed of the algorithm. However, for **Bubble Sort**, the **parallelization speedup** is not as dramatic as other algorithms like **Merge Sort** or **Quick Sort** because:

- **Bubble Sort** still needs multiple phases to complete sorting.

- The number of comparisons reduces dramatically as elements are ordered.

Assignment 3

Implement Min, Max, Sum, and Average operations using Parallel Reduction

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

using namespace std;

void min_reduction(vector<int>& arr) {
  int min_value = INT_MAX;
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < arr.size(); i++) {
    if (arr[i] < min_value) {
      min_value = arr[i];
    }
  }
  cout << "Minimum value: " << min_value << endl;
}

void max_reduction(vector<int>& arr) {
  int max_value = INT_MIN;
  #pragma omp parallel for reduction(max: max_value)
  for (int i = 0; i < arr.size(); i++) {
    if (arr[i] > max_value) {
      max_value = arr[i];
    }
  }
  cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(vector<int>& arr) {
  int sum = 0;
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < arr.size(); i++) {
    sum += arr[i];
  }
  cout << "Sum: " << sum << endl;
}

void average_reduction(vector<int>& arr) {
  int sum = 0;
  #pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < arr.size(); i++) {
    sum += arr[i];
  }
  cout << "Average: " << (double)sum / arr.size() << endl;
}

int main() {
  vector<int> arr;
  arr.push_back(5);
  arr.push_back(2);
  arr.push_back(9);
  arr.push_back(1);
  arr.push_back(7);
  arr.push_back(6);
  arr.push_back(8);
  arr.push_back(3);
  arr.push_back(4);
```

```
  min_reduction(arr);
  max_reduction(arr);
  sum_reduction(arr);
  average_reduction(arr);
}
```

The provided code demonstrates how to use OpenMP reduction operations to perform parallelized operations on an array. Specifically, it calculates the minimum value, maximum value, sum, and average of the elements in the array, all in parallel. The use of OpenMP reduction ensures thread-safety and efficient parallelism for these operations.

OpenMP provides the `reduction` clause to safely perform operations on variables across multiple threads. The `reduction` operation ensures that the result is computed correctly by each thread and then combined at the end.

This function computes the **minimum** value in the array using parallelism.

```
void min_reduction(vector<int>& arr) {
  int min_value = INT_MAX;  // Start with the highest possible value.
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < arr.size(); i++) {
    if (arr[i] < min_value) {
      min_value = arr[i];  // Update the min_value if a smaller value is found.
    }
  }
  cout << "Minimum value: " << min_value << endl;
}
```

- **INT_MAX** is used as an initial value for `min_value`, because any value in the array will be smaller than `INT_MAX`, so it will be replaced by the first array element.

- **#pragma omp parallel for reduction(min: min_value)**: This is an OpenMP directive that tells the compiler to parallelize the loop. The `reduction(min: min_value)` clause ensures that each thread has its local copy of `min_value`. After the parallel region, OpenMP combines these local results into the final result.

- In OpenMP, the **reduction** clause is used to specify a reduction operation on a variable across multiple threads. The `min:` part specifies the type of reduction operation to perform on the variable.

- Inside the loop, if the current array element (`arr[i]`) is smaller than the current `min_value`, it updates the `min_value`.

Assignment 4

Write a Program for Matrix Addition and Multiplication using CUDA C

Addition

What is CUDA?

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by **NVIDIA**.
It allows you to **write C/C++ code that runs directly on NVIDIA GPUs**.

Normally, your C++ code runs on the CPU (central processing unit).
With CUDA, you can offload *heavy computations* (like large matrix operations, image processing, deep learning tasks) onto the GPU, which can handle **thousands of threads in parallel** — much faster than the CPU for many types of tasks.

How CUDA works (at a high level):

CPU (Host)	GPU (Device)
Allocates memory (malloc)	Allocates memory (cudaMalloc)
Copies data to GPU (cudaMemcpy)	Executes code (kernels) on many threads
Calls kernel (special GPU function)	Each thread does small part of the work
Copies results back (cudaMemcpy)	
Frees memory	Frees memory

You **move data** to the GPU, **launch the computation** (kernel), **wait for it to finish**, and **move results back**.

Now, let's explain your code line by line:

1. Include headers and `using namespace std`

```
#include <iostream>
#include <cuda_runtime.h>
using namespace std;
```

You need:

- `<iostream>` for `cout`

- `<cuda_runtime.h>` for CUDA memory and kernel functions (`cudaMalloc`, `cudaMemcpy`, etc.)

2. Define the kernel function

```
__global__ void addVectors(int* A, int* B, int* C, int n)
```

- `__global__` tells the compiler: **this function runs on the GPU** and is **called from the CPU**.

- This function adds element `i` of arrays A and B and stores in C.

Inside the kernel:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Each GPU thread gets a unique `i`, based on:

- `blockIdx.x`: which block we are in

- `blockDim.x`: how many threads per block

- `threadIdx.x`: which thread inside the block

Then:

```
if (i < n) {
    C[i] = A[i] + B[i];
}
```

Each thread adds a pair of elements.

3. In main() — Setup memory

Host-side memory (CPU):

```
int *A, *B, *C;
int size = n * sizeof(int);
cudaMallocHost(&A, size);
cudaMallocHost(&B, size);
cudaMallocHost(&C, size);
```

- `cudaMallocHost()` allocates *pinned* (page-locked) memory on CPU, which makes transfers faster to GPU.

Initialize values:

```
for (int i = 0; i < n; i++) {
    A[i] = i;
    B[i] = i * 2;
}
```

- Fill A with [0,1,2,...]

- Fill B with [0,2,4,...]

Device-side memory (GPU):

```
cudaMalloc(&dev_A, size);
cudaMalloc(&dev_B, size);
cudaMalloc(&dev_C, size);
```

- Allocate arrays `dev_A`, `dev_B`, `dev_C` on GPU memory.

Copy from CPU to GPU:

```
cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_B, B, size, cudaMemcpyHostToDevice);
```

4. Launch the kernel

```
int blockSize = 256;
int numBlocks = (n + blockSize - 1) / blockSize;
addVectors<<<numBlocks, blockSize>>>(dev_A, dev_B, dev_C, n);
```

- **256 threads per block** is common for performance.

- **Enough blocks** so that all `n` elements are processed.

- Then launch `addVectors` on GPU!

After launching:

```
cudaDeviceSynchronize();
```

- **Wait** for the GPU to finish running before proceeding.

5. Copy results back to CPU

```
cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);
```

After the kernel finishes, copy the resulting vector back from GPU to CPU.

6. Print some results

```
for (int i = 0; i < 10; i++) {
    cout << C[i] << " ";
}
cout << endl;
```

Prints the first 10 elements to check the addition.

(Expect results like `0 3 6 9 12 15 18 21 24 27`)

7. Cleanup memory

```
cudaFree(dev_A);
cudaFree(dev_B);
cudaFree(dev_C);
cudaFreeHost(A);
cudaFreeHost(B);
cudaFreeHost(C);
```

Always free GPU and CPU memory after use.

CUDA architecture in a simple picture:

```
CPU (Host) → allocate & initialize → copy to GPU (Device)
```

```
GPU → launch kernel → thousands of threads run parallel addition
GPU → copy results back → CPU prints results
```

In short:

- You create and manage **memory** on both CPU and GPU.

- You **move data** to GPU.

- You **launch a kernel**, where each thread does a small part of work.

- You **move results** back to CPU.

- You **free memory** afterward.

Multiplication

What Does This CUDA Code Do?

This code **performs matrix multiplication** on the **GPU** using **CUDA**.

You have two matrices A and B (each of size N x N), and you compute their matrix product C = A × B.

The computation is:

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] \times B[k][j]$$

Each thread on the GPU calculates **one element** C[i][j] of the output matrix!

Full Step-by-Step Explanation of Your Code

1. Kernel: `matmul`

```
__global__ void matmul(int* A, int* B, int* C, int N)
{
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if (Row < N && Col < N)
    {
        int Pvalue = 0;
        for (int k = 0; k < N; k++)
        {
            Pvalue += A[Row * N + k] * B[k * N + Col];
        }
        C[Row * N + Col] = Pvalue;
    }
}
```

- `Row` and `Col` are computed from the **block** and **thread** indices.

- Each **thread** is responsible for **one output element** `C[Row][Col]`.

- Inside, it **loops over** `k` to compute the **dot product** of the row of A and column of B.

  Matrix elements are accessed in **row-major order**: `A[row * N + col]`.

2. Main Program

(a) Setup

```
int N = 512;
int size = N * N * sizeof(int);
int *A, *B, *C;
int *dev_A, *dev_B, *dev_C;
```

- Matrices of size **512 x 512** (so 512×512 = **262,144 elements**).

(b) Allocate Pinned Host Memory

```
cudaMallocHost((void**)&A, size);
cudaMallocHost((void**)&B, size);
cudaMallocHost((void**)&C, size);
```

- Allocates **page-locked (pinned)** memory on the **CPU**.

- Makes **GPU memory transfers faster**.

(c) Allocate Device (GPU) Memory

```
cudaMalloc((void**)&dev_A, size);
cudaMalloc((void**)&dev_B, size);
cudaMalloc((void**)&dev_C, size);
```

- Allocates memory for matrices on the **GPU**.

(d) Initialize Matrices A and B

```cpp
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        A[i * N + j] = i * N + j;
        B[i * N + j] = j * N + i;
    }
}
```

- Fills matrix A with increasing numbers row-wise.

- Fills matrix B with increasing numbers **column-wise**.

(e) Copy Matrices CPU ➜ GPU

```cpp
cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_B, B, size, cudaMemcpyHostToDevice);
```

- Sends matrices from **CPU** memory to **GPU** memory.

(f) Configure Grid and Block Sizes

```cpp
dim3 dimBlock(16, 16);
dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y - 1) /
dimBlock.y);
```

- **Each block** is 16×16 = 256 threads.

- **Grid** size is enough to cover all N×N elements.

  Notice the (N + blockDim.x - 1) / blockDim.x formula — it ensures even if N is not divisible by 16, you still cover all elements.

(g) Launch the Kernel

```cpp
matmul<<<dimGrid, dimBlock>>>(dev_A, dev_B, dev_C, N);
```

- Tells the GPU to start matrix multiplication.

(h) Copy Result GPU ➜ CPU

```cpp
cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);
```

- Transfers the computed matrix C back to the CPU.

(i) Print a Portion of the Result

```cpp
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        std::cout << C[i * N + j] << " ";
    }
    std::cout << std::endl;
}
```

- Only prints the **top-left 10×10** part of the matrix to check if the output looks correct.

(j) Free Memory

```
cudaFree(dev_A);
cudaFree(dev_B);
cudaFree(dev_C);
cudaFreeHost(A);
cudaFreeHost(B);
cudaFreeHost(C);
```

- Properly deallocates all memory used.

How Matrix Multiplication Happens on GPU

CPU	GPU
Launch kernel	Thousands of threads run in parallel
Each thread calculates	One element `C[Row][Col]`
Loop inside each thread	Sum of products over `k`
Result copied back	Matrix `C` to CPU

Each thread computes only **one element** = efficient use of GPU threads!