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

using namespace std;
using namespace std::chrono;

class Graph {
public:
    int V;
    vector<vector<int>> adj;

    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

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

    void generateRandomGraph(int edges);
    void sequentialBFS(int start);
    void parallelBFS(int start);
    void sequentialDFS(int start);
    void parallelDFS(int start);
```

```
};

void Graph::generateRandomGraph(int edges) {
    srand(time(0));
    for (int i = 0; i < edges; i++) {
        int u = rand() % V;
        int v = rand() % V;
        if (u != v) {
            addEdge(u, v);
        }
    }
}

void Graph::sequentialBFS(int start) {
    vector<bool> visited(V, false);
    queue<int> q;
    visited[start] = true;
    q.push(start);

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

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

```
void Graph::parallelBFS(int start) {
  vector<bool> visited(V, false);
  queue<int> q;
  visited[start] = true;
  q.push(start);

  while (!q.empty()) {
    int size = q.size();
    vector<int> levelNodes;

    #pragma omp parallel for shared(visited, q)
    for (int i = 0; i < size; i++) {
      int node;
      #pragma omp critical
      {
        if (!q.empty()) {
          node = q.front();
          q.pop();
        }
      }


      for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
          visited[neighbor] = true;
          #pragma omp critical
          levelNodes.push_back(neighbor);
```

```
            }
        }
    }

    for (int node : levelNodes) {
        q.push(node);
    }
  }
}

void Graph::sequentialDFS(int start) {
    vector<bool> visited(V, false);
    stack<int> s;
    s.push(start);

    while (!s.empty()) {
        int node = s.top();
        s.pop();
        if (!visited[node]) {
            visited[node] = true;
            for (int neighbor : adj[node]) {
                if (!visited[neighbor]) {
                    s.push(neighbor);
                }
            }
        }
    }
}
```

```
void Graph::parallelDFS(int start) {
    vector<bool> visited(V, false);
    stack<int> s;
    s.push(start);

    #pragma omp parallel
    {
        while (!s.empty()) {
            int node;
            #pragma omp critical
            {
                if (!s.empty()) {
                    node = s.top();
                    s.pop();
                }
            }
            if (!visited[node]) {
                visited[node] = true;
                #pragma omp parallel for
                for (int i = 0; i < adj[node].size(); i++) {
                    int neighbor = adj[node][i];
                    if (!visited[neighbor]) {
                        #pragma omp critical
                        s.push(neighbor);
                    }
                }
            }
        }
```

```
    }
}

int main() {
    int V, E;

    cout << "Enter the number of vertices: ";
    cin >> V;

    cout << "Enter the number of edges: ";
    cin >> E;

    if (E > (V * (V - 1)) / 2) {
        cout << "Too many edges for the given number of vertices. Adjusting to maximum possible edges.\n";
        E = (V * (V - 1)) / 2;
    }

    Graph g(V);
    g.generateRandomGraph(E);

    auto start = high_resolution_clock::now();
    g.sequentialBFS(0);
    auto stop = high_resolution_clock::now();
    auto seqBFS_time = duration_cast<microseconds>(stop - start);

    start = high_resolution_clock::now();
    g.parallelBFS(0);
    stop = high_resolution_clock::now();
    auto parBFS_time = duration_cast<microseconds>(stop - start);
```

```
    start = high_resolution_clock::now();
    g.sequentialDFS(0);
    stop = high_resolution_clock::now();
    auto seqDFS_time = duration_cast<microseconds>(stop - start);

    start = high_resolution_clock::now();
    g.parallelDFS(0);
    stop = high_resolution_clock::now();
    auto parDFS_time = duration_cast<microseconds>(stop - start);

    cout << "Sequential BFS Time: " << seqBFS_time.count() << " microseconds" << endl;
    cout << "Parallel BFS Time: " << parBFS_time.count() << " microseconds" << endl;
    cout << "Speedup for BFS: " << (double)seqBFS_time.count() / parBFS_time.count() << endl;

    cout << "Sequential DFS Time: " << seqDFS_time.count() << " microseconds" << endl;
    cout << "Parallel DFS Time: " << parDFS_time.count() << " microseconds" << endl;
    cout << "Speedup for DFS: " << (double)seqDFS_time.count() / parDFS_time.count() << endl;

    return 0;
}
```

Output

Clear

```
Enter the number of vertices: 500
Enter the number of edges: 42000
Sequential BFS Time: 5656 microseconds
Parallel BFS Time: 5574 microseconds
Speedup for BFS: 1.01471
Sequential DFS Time: 11177 microseconds
Parallel DFS Time: 9949 microseconds
Speedup for DFS: 1.12343


=== Code Execution Successful ===
```

```
#include <iostream>
#include <chrono>
#include <omp.h>
#include <vector>
using namespace std;
using namespace std::chrono;

class Sorting {
private:
    vector<int> arr;
    int n;

    void merge(vector<int>& arr, int start, int mid, int end) {
        vector<int> left(arr.begin() + start, arr.begin() + mid + 1);
        vector<int> right(arr.begin() + mid + 1, arr.begin() + end + 1);
        int i = 0, j = 0, k = start;
        while (i < left.size() && j < right.size()) {
            if (left[i] <= right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }
        while (i < left.size()) arr[k++] = left[i++];
        while (j < right.size()) arr[k++] = right[j++];
    }

public:
    Sorting(vector<int> inputArr) : arr(inputArr), n(inputArr.size()) {}

    void bubbleSort() {
```

```
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void mergeSort(int start, int end) {
    if (start < end) {
        int mid = start + (end - start) / 2;
        mergeSort(start, mid);
        mergeSort(mid + 1, end);
        merge(arr, start, mid, end);
    }
}

void parallelBubbleSort() {
    bool sorted = false;
    while (!sorted) {
        sorted = true;
        #pragma omp parallel for shared(arr, sorted)
        for (int i = 0; i < n - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                sorted = false;
            }
        }
        #pragma omp parallel for shared(arr, sorted)
        for (int i = 1; i < n - 1; i += 2) {
```

```
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                sorted = false;
            }
        }
    }
}

void parallelMergeSort(int start, int end) {
    if (start < end) {
        int mid = start + (end - start) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            parallelMergeSort(start, mid);
            #pragma omp section
            parallelMergeSort(mid + 1, end);
        }
        merge(arr, start, mid, end);
    }
}

void displayArray() {
    for (int i : arr) {
        cout << i << " ";
    }
    cout << endl;
}

vector<int> getArray() {
    return arr;
```

```
    }
};

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

    vector<int> inputArr(n);
    for (int i = 0; i < n; i++) {
        inputArr[i] = rand() % 100;
        cout << inputArr[i] << " ";
    }
    cout << "\n\n";

    Sorting sorter(inputArr);

    vector<int> originalArr = sorter.getArray();

    cout << "Sequential Execution:\n\n";

    cout << "Bubble Sort: ";
    auto start = high_resolution_clock::now();
    sorter.bubbleSort();
    auto end = high_resolution_clock::now();
    sorter.displayArray();
    double seq_bubble_time = duration<double, milli>(end - start).count();
    cout << "TIME TAKEN: " << seq_bubble_time << " ms\n";

    sorter = Sorting(originalArr);
    cout << "\nMerge Sort: ";
```

```
    start = high_resolution_clock::now();
    sorter.mergeSort(0, n - 1);
    end = high_resolution_clock::now();
    sorter.displayArray();
    double seq_merge_time = duration<double, milli>(end - start).count();
    cout << "TIME TAKEN: " << seq_merge_time << " ms\n";

    cout << "\nParallel Execution:\n\n";

    sorter = Sorting(originalArr);
    cout << "Parallel Bubble Sort: ";
    start = high_resolution_clock::now();
    sorter.parallelBubbleSort();
    end = high_resolution_clock::now();
    sorter.displayArray();
    double par_bubble_time = duration<double, milli>(end - start).count();
    cout << "TIME TAKEN: " << par_bubble_time << " ms\n";
    cout << "Speedup Factor (Bubble Sort): " << seq_bubble_time / par_bubble_time << "\n";

    sorter = Sorting(originalArr);
    cout << "\nParallel Merge Sort: ";
    start = high_resolution_clock::now();
    sorter.parallelMergeSort(0, n - 1);
    end = high_resolution_clock::now();
    sorter.displayArray();
    double par_merge_time = duration<double, milli>(end - start).count();
    cout << "TIME TAKEN: " << par_merge_time << " ms\n";
    cout << "Speedup Factor (Merge Sort): " << seq_merge_time / par_merge_time << "\n";

    return 0;
}
```

```
Enter number of elements: 500
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57
 24 95 82 45 14 67 34 64 43 50 87 8 76 78 88 84 3 51 54 99 32 60 76 68 39 12 26 86 94 39 95 70 34 78 67 1 97 2 17 92 52 56 1 80 86 41 65 89 44 19 40 29 31 17 97 71 81 75 9 27 67 56 97 53 86 65 6 83 19 24 28
 71 32 29 3 19 70 68 8 15 40 49 96 23 18 45 46 51 21 55 79 88 64 28 41 50 93 0 34 64 24 14 87 56 43 91 27 65 59 36 32 51 37 28 75 7 74 21 58 95 29 37 35 93 18 28 43 11 28 29 76 4 43 63 13 38 6 40 4 18 28 88
 69 17 17 96 24 43 70 83 90 99 72 25 44 90 5 39 54 86 69 82 42 64 97 7 55 4 48 11 22 28 99 43 46 68 40 22 11 10 5 1 61 30 78 5 20 36 44 26 22 65 8 16 82 58 24 37 62 24 0 36 52 99 79 50 68 71 73 31 81 30 33
94 60 63 99 81 99 96 59 73 13 68 90 95 26 66 84 40 90 84 76 42 36 7 45 56 79 18 87 12 48 72 59 9 36 10 42 87 6 1 13 72 21 55 19 99 21 4 39 11 40 67 5 28 27 50 84 58 20 24 22 69 96 81 30 84 92 72 72 50 25 85
 22 99 40 42 98 13 98 90 24 90 9 81 19 36 32 55 94 4 79 69 73 76 50 55 60 42 79 84 93 5 21 67 4 13 61 54 26 59 44 2 2 6 84 21 42 68 28 89 72 8 58 98 36 8 53 48 3 33 33 48 90 54 67 46 68 29 0 46 88 97 49 90
3 33 63 97 53 92 86 25 52 96 75 88 57 29 36 60 14 21 60 4 28 27 50 48 56 2 94 97 99 43 39 2 28 3 0 81 47 38 59 51 35 34 39 92 15 27 4 29 49 64 85 29 43 35 77 0 38 71 49 89 67 88 92 95 43 44 29 90 82 40 41 6
9 26 32

Sequential Execution:

Bubble Sort: 0 0 0 0 0 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 7 7 7 8 8 8 8 8 9 9 9 10 10 11 11 11 11 11 11 12 12 13 13 13 13 13 13 13 14 14 14 15 15 15 15 16 17 17 17 17 18 18
18 18 19 19 19 19 19 19 20 20 21 21 21 21 21 21 21 21 21 22 22 22 22 22 22 23 23 24 24 24 24 24 24 24 24 24 25 25 25 25 26 26 26 26 26 26 26 26 27 27 27 27 27 27 27 28 28 28 28 28 28 28 28 28 28 28 29 29 29
 29 29 29 29 29 29 29 29 29 29 30 30 30 30 31 31 32 32 32 32 32 33 33 33 33 34 34 34 34 35 35 35 35 35 36 36 36 36 36 36 36 36 36 36 37 37 37 37 38 38 38 39 39 39 39 39 39 40 40 40 40 40 40 40 40 40 41 41 4
1 42 42 42 42 42 42 42 43 43 43 43 43 43 43 43 43 44 44 44 44 44 45 45 45 46 46 46 46 46 47 48 48 48 48 48 49 49 49 49 49 50 50 50 50 50 50 50 51 51 51 51 52 52 52 53 53 53 54 54 54 54 55 55 55 55 55 56 56
56 56 56 56 56 57 57 58 58 58 58 58 59 59 59 59 59 59 60 60 60 60 60 61 61 62 62 62 62 63 63 63 63 64 64 64 64 64 65 65 65 65 66 67 67 67 67 67 67 67 67 67 67 68 68 68 68 68 68 68 68 69 69 69 69 69 69 70 70
 70 70 70 71 71 71 71 72 72 72 72 72 72 72 73 73 73 73 73 74 75 75 75 76 76 76 76 76 77 77 78 78 78 79 79 79 79 79 80 80 81 81 81 81 81 81 81 82 82 82 82 82 83 83 83 84 84 84 84 84 84 84 84 84 85 85 86 86 8
6 86 86 86 86 87 87 87 87 88 88 88 88 88 88 89 89 89 90 90 90 90 90 90 90 90 90 90 91 91 92 92 92 92 92 92 93 93 93 93 93 94 94 94 94 95 95 95 95 95 96 96 96 96 96 96 97 97 97 97 97 97 97 98 98 98 98 99 99
99 99 99 99 99 99 99
TIME TAKEN: 2.13815 ms

Merge Sort: 0 0 0 0 0 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 7 7 7 8 8 8 8 8 9 9 9 10 10 11 11 11 11 11 11 12 12 13 13 13 13 13 13 13 14 14 14 15 15 15 15 16 17 17 17 17 18 18 1
8 18 19 19 19 19 19 19 20 20 21 21 21 21 21 21 21 21 21 22 22 22 22 22 22 23 23 24 24 24 24 24 24 24 24 24 25 25 25 25 26 26 26 26 26 26 26 26 27 27 27 27 27 27 27 28 28 28 28 28 28 28 28 28 28 28 29 29 29
29 29 29 29 29 29 29 29 29 29 30 30 30 30 31 31 32 32 32 32 32 33 33 33 33 34 34 34 34 35 35 35 35 35 36 36 36 36 36 36 36 36 36 36 37 37 37 37 38 38 38 39 39 39 39 39 39 40 40 40 40 40 40 40 40 40 41 41 41
 42 42 42 42 42 42 42 43 43 43 43 43 43 43 43 43 44 44 44 44 44 45 45 45 46 46 46 46 46 47 48 48 48 48 48 49 49 49 49 49 50 50 50 50 50 50 50 51 51 51 51 52 52 52 53 53 53 54 54 54 54 55 55 55 55 55 56 56 5
6 56 56 56 56 57 57 58 58 58 58 58 59 59 59 59 59 59 60 60 60 60 60 61 61 62 62 62 62 63 63 63 63 64 64 64 64 64 65 65 65 65 66 67 67 67 67 67 67 67 67 67 67 68 68 68 68 68 68 68 68 69 69 69 69 69 69 70 70
70 70 70 71 71 71 71 72 72 72 72 72 72 72 73 73 73 73 73 74 75 75 75 76 76 76 76 76 77 77 78 78 78 79 79 79 79 79 80 80 81 81 81 81 81 81 81 82 82 82 82 82 83 83 83 84 84 84 84 84 84 84 84 84 85 85 86 86 86
 86 86 86 86 87 87 87 87 88 88 88 88 88 88 89 89 89 90 90 90 90 90 90 90 90 90 90 91 91 92 92 92 92 92 92 93 93 93 93 93 94 94 94 94 95 95 95 95 95 96 96 96 96 96 96 97 97 97 97 97 97 97 98 98 98 98 99 99 9
9 99 99 99 99 99 99
TIME TAKEN: 0.615425 ms

Parallel Execution:

Parallel Bubble Sort: 0 0 0 0 0 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 7 7 7 8 8 8 8 8 9 9 9 10 10 11 11 11 11 11 11 12 12 13 13 13 13 13 13 13 14 14 14 15 15 15 15 16 17 17 17
17 18 18 18 18 19 19 19 19 19 19 20 20 21 21 21 21 21 21 21 21 21 22 22 22 22 22 22 23 23 24 24 24 24 24 24 24 24 24 25 25 25 25 26 26 26 26 26 26 26 26 27 27 27 27 27 27 27 28 28 28 28 28 28 28 28 28 28 28
 29 29 29 29 29 29 29 29 29 29 29 29 29 30 30 30 30 31 31 32 32 32 32 32 33 33 33 33 34 34 34 34 35 35 35 35 35 36 36 36 36 36 36 36 36 36 36 37 37 37 37 38 38 38 39 39 39 39 39 39 40 40 40 40 40 40 40 40 4
0 41 41 41 42 42 42 42 42 42 42 43 43 43 43 43 43 43 43 43 44 44 44 44 44 45 45 45 46 46 46 46 46 47 48 48 48 48 48 49 49 49 49 49 50 50 50 50 50 50 50 51 51 51 51 52 52 52 53 53 53 54 54 54 54 55 55 55 55
55 56 56 56 56 56 56 56 57 57 58 58 58 58 58 59 59 59 59 59 59 60 60 60 60 60 61 61 62 62 62 62 63 63 63 63 64 64 64 64 64 65 65 65 65 66 67 67 67 67 67 67 67 67 67 67 68 68 68 68 68 68 68 68 69 69 69 69 69
 69 70 70 70 70 70 71 71 71 71 72 72 72 72 72 72 72 73 73 73 73 73 74 75 75 75 76 76 76 76 76 77 77 78 78 78 79 79 79 79 79 80 80 81 81 81 81 81 81 81 82 82 82 82 82 83 83 83 84 84 84 84 84 84 84 84 84 85 8
5 86 86 86 86 86 86 86 87 87 87 87 88 88 88 88 88 88 89 89 89 90 90 90 90 90 90 90 90 90 90 91 91 92 92 92 92 92 92 93 93 93 93 93 94 94 94 94 95 95 95 95 95 96 96 96 96 96 96 97 97 97 97 97 97 97 98 98 98
98 99 99 99 99 99 99 99 99 99
TIME TAKEN: 2.65423 ms
Speedup Factor (Bubble Sort): 0.805562

Parallel Merge Sort: 0 0 0 0 0 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 7 7 7 8 8 8 8 8 9 9 9 10 10 11 11 11 11 11 11 12 12 13 13 13 13 13 13 13 14 14 14 15 15 15 15 16 17 17 17 1
7 18 18 18 18 19 19 19 19 19 19 20 20 21 21 21 21 21 21 21 21 21 22 22 22 22 22 22 23 23 24 24 24 24 24 24 24 24 24 25 25 25 25 26 26 26 26 26 26 26 26 27 27 27 27 27 27 27 28 28 28 28 28 28 28 28 28 28 28
29 29 29 29 29 29 29 29 29 29 29 29 29 30 30 30 30 31 31 32 32 32 32 32 33 33 33 33 34 34 34 34 35 35 35 35 35 36 36 36 36 36 36 36 36 36 36 37 37 37 37 38 38 38 39 39 39 39 39 39 40 40 40 40 40 40 40 40 40
 41 41 41 42 42 42 42 42 42 42 43 43 43 43 43 43 43 43 43 44 44 44 44 44 45 45 45 46 46 46 46 46 47 48 48 48 48 48 49 49 49 49 49 50 50 50 50 50 50 50 51 51 51 51 52 52 52 53 53 53 54 54 54 54 55 55 55 55 5
5 56 56 56 56 56 56 56 57 57 58 58 58 58 58 59 59 59 59 59 59 60 60 60 60 60 61 61 62 62 62 62 63 63 63 63 64 64 64 64 64 65 65 65 65 66 67 67 67 67 67 67 67 67 67 67 68 68 68 68 68 68 68 68 69 69 69 69 69
69 70 70 70 70 70 71 71 71 71 72 72 72 72 72 72 72 73 73 73 73 73 74 75 75 75 76 76 76 76 76 77 77 78 78 78 79 79 79 79 79 80 80 81 81 81 81 81 81 81 82 82 82 82 82 83 83 83 84 84 84 84 84 84 84 84 84 85 85
 86 86 86 86 86 86 86 87 87 87 87 88 88 88 88 88 88 89 89 89 90 90 90 90 90 90 90 90 90 90 91 91 92 92 92 92 92 92 93 93 93 93 93 94 94 94 94 95 95 95 95 95 96 96 96 96 96 96 97 97 97 97 97 97 97 98 98 98 9
8 99 99 99 99 99 99 99 99 99
TIME TAKEN: 0.37403 ms
Speedup Factor (Merge Sort): 1.64539
```

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <omp.h>

using namespace std;

class ParallelMinMax {
private:
    vector<int> arr;
    int size;
    int min_seq, max_seq, min_par, max_par;
    long long sum_seq, sum_par;
    double avg_seq, avg_par;
    double time_min_seq, time_max_seq, time_sum_seq, time_avg_seq;
    double time_min_par, time_max_par, time_sum_par, time_avg_par;

public:
    ParallelMinMax(int size) : size(size), min_seq(0), max_seq(0), min_par(0), max_par(0), sum_seq(0),
sum_par(0), avg_seq(0), avg_par(0),
        time_min_seq(0), time_max_seq(0), time_sum_seq(0), time_avg_seq(0), time_min_par(0),
time_max_par(0), time_sum_par(0), time_avg_par(0) {
        arr.resize(size);
    }

    void generateRandomArray() {
        srand(time(0));
        for (int i = 0; i < size; i++) {
            arr[i] = rand() % 100 + 1;
        }
```

```
}

void computeMinSequential() {
    auto start = chrono::high_resolution_clock::now();
    min_seq = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] < min_seq) min_seq = arr[i];
    }
    auto end = chrono::high_resolution_clock::now();
    time_min_seq = chrono::duration<double>(end - start).count();
}

void computeMaxSequential() {
    auto start = chrono::high_resolution_clock::now();
    max_seq = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max_seq) max_seq = arr[i];
    }
    auto end = chrono::high_resolution_clock::now();
    time_max_seq = chrono::duration<double>(end - start).count();
}

void computeSumSequential() {
    auto start = chrono::high_resolution_clock::now();
    sum_seq = 0;
    for (int i = 0; i < size; i++) {
        sum_seq += arr[i];
    }
    auto end = chrono::high_resolution_clock::now();
    time_sum_seq = chrono::duration<double>(end - start).count();
}
```

```
void computeAvgSequential() {

    auto start = chrono::high_resolution_clock::now();

    avg_seq = static_cast<double>(sum_seq) / size;

    auto end = chrono::high_resolution_clock::now();

    time_avg_seq = chrono::duration<double>(end - start).count();

}


void computeMinParallel() {

    auto start = chrono::high_resolution_clock::now();

    min_par = arr[0];

    #pragma omp parallel for reduction(min:min_par)

    for (int i = 1; i < size; i++) {

        if (arr[i] < min_par) min_par = arr[i];

    }

    auto end = chrono::high_resolution_clock::now();

    time_min_par = chrono::duration<double>(end - start).count();

}


void computeMaxParallel() {

    auto start = chrono::high_resolution_clock::now();

    max_par = arr[0];

    #pragma omp parallel for reduction(max:max_par)

    for (int i = 1; i < size; i++) {

        if (arr[i] > max_par) max_par = arr[i];

    }

    auto end = chrono::high_resolution_clock::now();

    time_max_par = chrono::duration<double>(end - start).count();

}


void computeSumParallel() {
```

```
    auto start = chrono::high_resolution_clock::now();
    sum_par = 0;
    #pragma omp parallel for reduction(+:sum_par)
    for (int i = 0; i < size; i++) {
        sum_par += arr[i];
    }
    auto end = chrono::high_resolution_clock::now();
    time_sum_par = chrono::duration<double>(end - start).count();
}

void computeAvgParallel() {
    auto start = chrono::high_resolution_clock::now();
    avg_par = static_cast<double>(sum_par) / size;
    auto end = chrono::high_resolution_clock::now();
    time_avg_par = chrono::duration<double>(end - start).count();
}

void sequentialComputation() {
    computeMinSequential();
    computeMaxSequential();
    computeSumSequential();
    computeAvgSequential();
}

void parallelComputation() {
    computeMinParallel();
    computeMaxParallel();
    computeSumParallel();
    computeAvgParallel();
}
```

```
    void displayResults() {
        cout << "---- Sequential Computation ----\n";
        cout << "Min: " << min_seq << " | Time: " << time_min_seq << " sec\n";
        cout << "Max: " << max_seq << " | Time: " << time_max_seq << " sec\n";
        cout << "Sum: " << sum_seq << " | Time: " << time_sum_seq << " sec\n";
        cout << "Average: " << avg_seq << " | Time: " << time_avg_seq << " sec\n\n";

        cout << "---- Parallel Computation ----\n";
        cout << "Min: " << min_par << " | Time: " << time_min_par << " sec\n";
        cout << "Max: " << max_par << " | Time: " << time_max_par << " sec\n";
        cout << "Sum: " << sum_par << " | Time: " << time_sum_par << " sec\n";
        cout << "Average: " << avg_par << " | Time: " << time_avg_par << " sec\n\n";

        cout << "---- Speedup Factors ----\n";
        cout << "Speedup (Min): " << (time_min_seq / time_min_par) << "x\n";
        cout << "Speedup (Max): " << (time_max_seq / time_max_par) << "x\n";
        cout << "Speedup (Sum): " << (time_sum_seq / time_sum_par) << "x\n";
        cout << "Speedup (Average): " << (time_avg_seq / time_avg_par) << "x\n";
    }
};

int main() {
    int size;
    cout << "Enter array size: ";
    cin >> size;
    if (size <= 0) {
        cout << "Invalid size!" << endl;
        return 1;
    }

    ParallelMinMax pm(size);
```

```
    pm.generateRandomArray();
    pm.sequentialComputation();
    pm.parallelComputation();
    pm.displayResults();

    return 0;
}
```

Output   Clear

```
Enter array size: 1000
---- Sequential Computation ----
Min: 1 | Time: 4.45e-06 sec
Max: 100 | Time: 3.94e-06 sec
Sum: 51026 | Time: 3.83e-06 sec
Average: 51.026 | Time: 3e-08 sec

---- Parallel Computation ----
Min: 1 | Time: 4.33e-06 sec
Max: 100 | Time: 3.85e-06 sec
Sum: 51026 | Time: 4.01e-06 sec
Average: 51.026 | Time: 3e-08 sec

---- Speedup Factors ----
Speedup (Min): 1.02771x
Speedup (Max): 1.02338x
Speedup (Sum): 0.955112x
Speedup (Average): 1x


=== Code Execution Successful ===
```

```
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include "cuda_runtime.h"

using namespace std;

#define N 1000

__global__ void vectorAdd(int *d_a, int *d_b, int *d_c, int n) {
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        if (i < n) {
        d_c[i] = d_a[i] + d_b[i];
        }
}

__global__ void matrixMultiplyKernel(float *a, float *b, float *c) {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        if (row < N && col < N) {
        float sum = 0;
        for (int i = 0; i < N; i++) {
        sum += a[row * N + i] * b[i * N + col];
        }
        c[row * N + col] = sum;
        }
}
```

```
class VectorAddition {
public:
	void performVectorAddition() {
	int *a, *b, *c, *d;
	int *d_a, *d_b, *d_c;
	size_t size = N * sizeof(int);

	a = (int *)malloc(size);
	b = (int *)malloc(size);
	c = (int *)malloc(size);
	d = (int *)malloc(size);

	srand(time(NULL));
	for (int i = 0; i < N; i++) {
	a[i] = rand() % 100;
	b[i] = rand() % 100;
	}

	clock_t start_cpu = clock();
	for (int i = 0; i < N; i++) {
	c[i] = a[i] + b[i];
	}
	clock_t end_cpu = clock();
	double cpu_time = (double)(end_cpu - start_cpu) / CLOCKS_PER_SEC;

	cudaMalloc((void **)&d_a, size);
	cudaMalloc((void **)&d_b, size);
	cudaMalloc((void **)&d_c, size);
	cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
	cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);
cudaEventRecord(stop);

cudaMemcpy(d, d_c, size, cudaMemcpyDeviceToHost);
cudaEventSynchronize(stop);
float gpu_time = 0;
cudaEventElapsedTime(&gpu_time, start, stop);

bool match = true;
for (int i = 0; i < N; i++) {
if (c[i] != d[i]) {
match = false;
break;
}
}

printf("CPU Time: %.6f s\n", cpu_time);
printf("GPU Time: %.6f ms\n", gpu_time);
printf("Speedup Factor: %.2f\n", (cpu_time) *10000/ gpu_time);
printf("Arrays Match: %s\n", match ? "Yes" : "No");

free(a); free(b); free(c); free(d);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

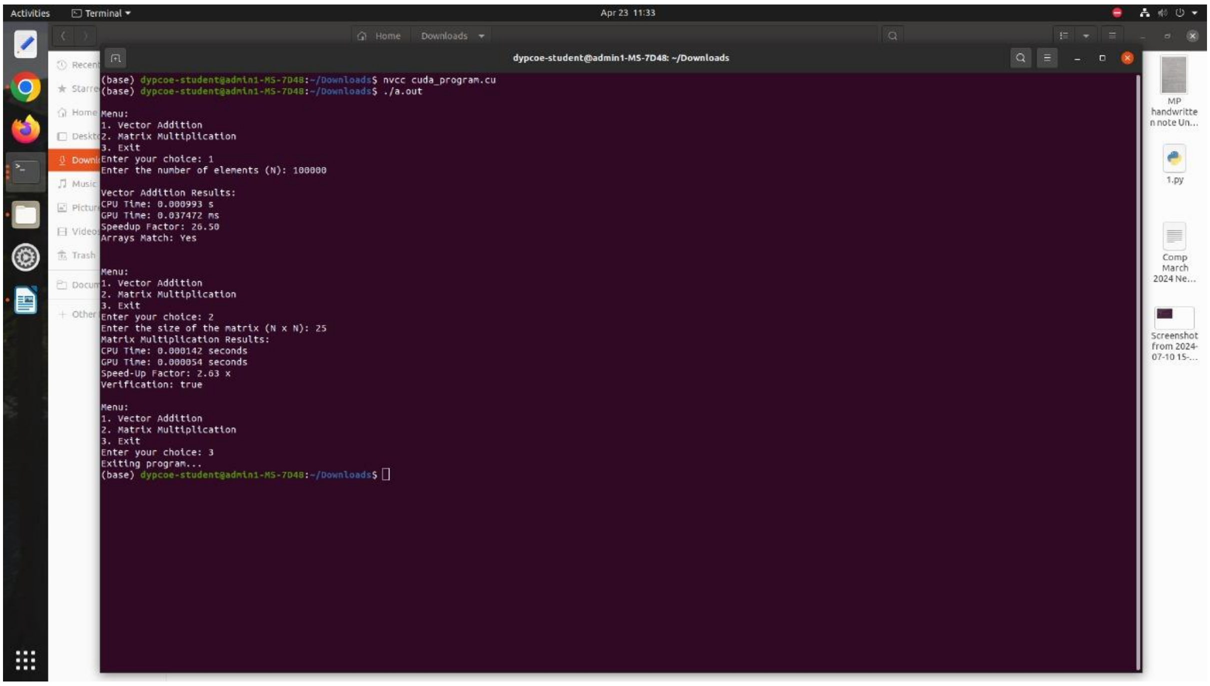
```
            cudaEventDestroy(start);

            cudaEventDestroy(stop);

            }

};

class MatrixMultiplier {

private:

            float *hostA, *hostB, *hostC, *hostD;

            float *devA, *devB, *devC;

            int size;

            float cpuTime, gpuTime;

public:

            MatrixMultiplier() {

            size = N * N * sizeof(float);

            hostA = (float *)malloc(size);

            hostB = (float *)malloc(size);

            hostC = (float *)malloc(size);

            hostD = (float *)malloc(size);

            cudaMalloc((void **)&devA, size);

            cudaMalloc((void **)&devB, size);

            cudaMalloc((void **)&devC, size);

            cpuTime = gpuTime = 0.0;

            }

            ~MatrixMultiplier() {

            free(hostA);

            free(hostB);

            free(hostC);

            free(hostD);

            cudaFree(devA);

            cudaFree(devB);
```

```
cudaFree(devC);
}

void initializeMatrices() {
for (int i = 0; i < N * N; i++) {
hostA[i] = rand() % 100;
hostB[i] = rand() % 100;
}
}

void gpuMatrixMultiplication() {
cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);
cudaMemcpy(devB, hostB, size, cudaMemcpyHostToDevice);

dim3 dimBlock(16, 16);
dim3 dimGrid((N + 15) / 16, (N + 15) / 16);

clock_t tic = clock();
matrixMultiplyKernel<<<dimGrid, dimBlock>>>(devA, devB, devC);
cudaDeviceSynchronize();
clock_t toc = clock();

gpuTime = ((float)(toc - tic)) / CLOCKS_PER_SEC;
cudaMemcpy(hostC, devC, size, cudaMemcpyDeviceToHost);
}

void cpuMatrixMultiplication() {
clock_t tic = clock();
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
float sum = 0;
```

```
for (int k = 0; k < N; k++) {
        sum += hostA[i * N + k] * hostB[k * N + j];
}
hostD[i * N + j] = sum;
}
}

clock_t toc = clock();
cpuTime = ((float)(toc - tic)) / CLOCKS_PER_SEC;
}

bool verifyEquality() {
float tolerance = 1e-5;
for (int i = 0; i < N * N; i++) {
if (fabs(hostC[i] - hostD[i]) > tolerance) {
printf("Mismatch at index %d: GPU = %f, CPU = %f\n", i, hostC[i], hostD[i]);
return false;

}
}
return true;
}

void printResults() {
printf("CPU Time: %f seconds\n", cpuTime);
printf("GPU Time: %f seconds\n", gpuTime);
if (gpuTime > 0) {
printf("Speed-Up Factor: %.2f x\n", (cpuTime) / gpuTime);
} else {
printf("Speed-Up Factor: N/A (GPU time too small)\n");
}
```

```
    }
};

int main() {
    int choice;

    VectorAddition vectorAdder;

    MatrixMultiplier matrixMultiplier;
    matrixMultiplier.initializeMatrices();
    matrixMultiplier.cpuMatrixMultiplication();
    matrixMultiplier.gpuMatrixMultiplication();
    bool success = matrixMultiplier.verifyEquality();

    do {
    cout << "\nMenu:" << endl;
    cout << "1. Vector Addition" << endl;
    cout << "2. Matirx Multiplication" << endl;
    cout << "3. Exit" << endl;
    cout << "Enter your choice: ";
    cin >> choice;
    switch (choice) {
    case 1:
    vectorAdder.performVectorAddition();
    break;
    case 2:

    matrixMultiplier.printResults();
    printf("Verification: %s\n", success ? "true" : "false");
    break;
    case 3:
```

```
cout << "Exiting..." << endl;
break;
default:
cout << "Invalid choice!" << endl;
}
} while (choice != 3);
return 0;
}
```



```
#include <iostream>
#include <vector>
#include <cmath>
#include <chrono>
#include <cuda.h>

#define N 1000  // number of data points

// CUDA Kernel for parallel sum calculation
__global__ void computeSums(const float* x, const float* y, float* sumX, float* sumY, float* sumXY,
float* sumX2, int n) {
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	if (i < n) {
	atomicAdd(sumX, x[i]);
	atomicAdd(sumY, y[i]);
	atomicAdd(sumXY, x[i] * y[i]);
	atomicAdd(sumX2, x[i] * x[i]);
	}
}

// CPU implementation of linear regression
void linearRegressionCPU(const float* x, const float* y, int n, float& b0, float& b1) {
	float sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;
	for (int i = 0; i < n; i++) {
	sumX += x[i];
	sumY += y[i];
	sumXY += x[i] * y[i];
	sumX2 += x[i] * x[i];
	}
	b1 = (n * sumXY - sumX * sumY) / (n * sumX2 - sumX * sumX);
	b0 = (sumY - b1 * sumX) / n;
```

```
}

int main() {
    // Host data
    std::vector<float> h_x(N), h_y(N);
    float b0_cpu, b1_cpu;
    float b0_gpu, b1_gpu;

    // Generate synthetic data: y = 2.5 + 1.2x
    for (int i = 0; i < N; ++i) {
    h_x[i] = i;
    h_y[i] = 2.5f + 1.2f * i;
    }

    // === CPU Linear Regression ===
    auto start_cpu = std::chrono::high_resolution_clock::now();
    linearRegressionCPU(h_x.data(), h_y.data(), N, b0_cpu, b1_cpu);
    auto end_cpu = std::chrono::high_resolution_clock::now();
    float cpu_time = std::chrono::duration<float, std::milli>(end_cpu - start_cpu).count();

    // === GPU Linear Regression ===
    float *d_x, *d_y, *d_sumX, *d_sumY, *d_sumXY, *d_sumX2;
    cudaMalloc(&d_x, N * sizeof(float));
    cudaMalloc(&d_y, N * sizeof(float));
    cudaMalloc(&d_sumX, sizeof(float));
    cudaMalloc(&d_sumY, sizeof(float));
    cudaMalloc(&d_sumXY, sizeof(float));
    cudaMalloc(&d_sumX2, sizeof(float));

    // Initialize device memory for sums
    cudaMemset(d_sumX, 0, sizeof(float));
```

```
    cudaMemset(d_sumY, 0, sizeof(float));
    cudaMemset(d_sumXY, 0, sizeof(float));
    cudaMemset(d_sumX2, 0, sizeof(float));

    cudaMemcpy(d_x, h_x.data(), N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y.data(), N * sizeof(float), cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    cudaEvent_t start_gpu, stop_gpu;
    cudaEventCreate(&start_gpu);
    cudaEventCreate(&stop_gpu);
    cudaEventRecord(start_gpu);
computeSums<<<blocksPerGrid, threadsPerBlock>>>(d_x, d_y, d_sumX, d_sumY, d_sumXY, d_sumX2,
N);

    cudaEventRecord(stop_gpu);
    cudaEventSynchronize(stop_gpu);

    float gpu_time = 0;
    cudaEventElapsedTime(&gpu_time, start_gpu, stop_gpu);

    float sumX, sumY, sumXY, sumX2;
    cudaMemcpy(&sumX, d_sumX, sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(&sumY, d_sumY, sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(&sumXY, d_sumXY, sizeof(float), cudaMemcpyDeviceToHost);
    cudaMemcpy(&sumX2, d_sumX2, sizeof(float), cudaMemcpyDeviceToHost);
    b1_gpu = (N * sumXY - sumX * sumY) / (N * sumX2 - sumX * sumX);
    b0_gpu = (sumY - b1_gpu * sumX) / N;
    bool verified = fabs(b0_cpu - b0_gpu) < 1e-2 && fabs(b1_cpu - b1_gpu) < 1e-2;
```

```
    std::cout << "=== Linear Regression using CUDA ===\n";
    std::cout << "CPU Time: " << cpu_time << " ms\n";
    std::cout << "GPU Time: " << gpu_time << " ms\n";
    std::cout << "Speedup Factor: " << cpu_time / gpu_time << "x\n";
    std::cout << "Verification: " << (verified ? "PASSED ✅" : "FAILED ❌") << "\n";
    std::cout << "Equation (CPU): y = " << b0_cpu << " + " << b1_cpu << " * x\n";
    std::cout << "Equation (GPU): y = " << b0_gpu << " + " << b1_gpu << " * x\n";
    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_sumX);
    cudaFree(d_sumY);
    cudaFree(d_sumXY);
    cudaFree(d_sumX2);
    cudaEventDestroy(start_gpu);
    cudaEventDestroy(stop_gpu);

    return 0;
}
```