

Parallel Breadth-First Search (BFS) in Graphs

Zein Al-Hashimi, Roxana Ramazanova

Introduction

This project demonstrates a parallel implementation of the Breadth-First Search (BFS) algorithm using OpenMP in C++. BFS is a fundamental graph traversal algorithm, and parallelizing it helps improve performance on large graphs by exploring nodes at the same level concurrently.

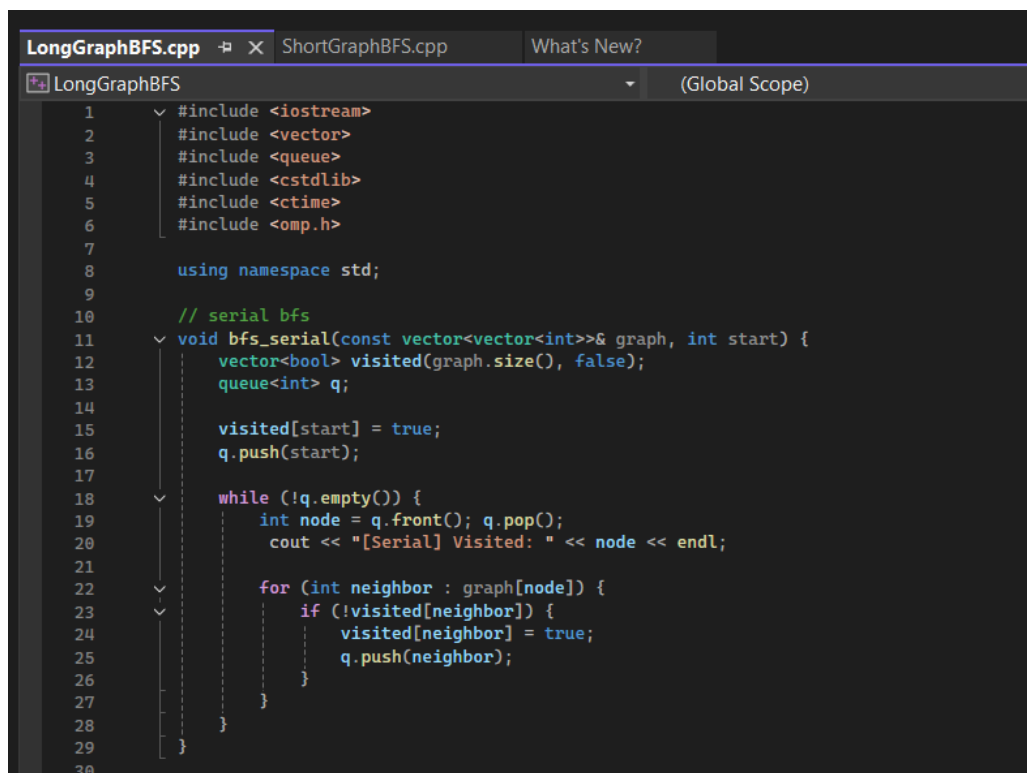
Serial BFS Implementation

We started the project by implementing the standard (serial) Breadth-First Search (BFS) algorithm. This version utilizes a **queue** to manage the nodes to visit and a **visited** array to prevent revisiting the same node multiple times.

To make sure the implementation worked correctly, we first tested it on a **small graph with 4 nodes**, where we could easily check the output by hand.

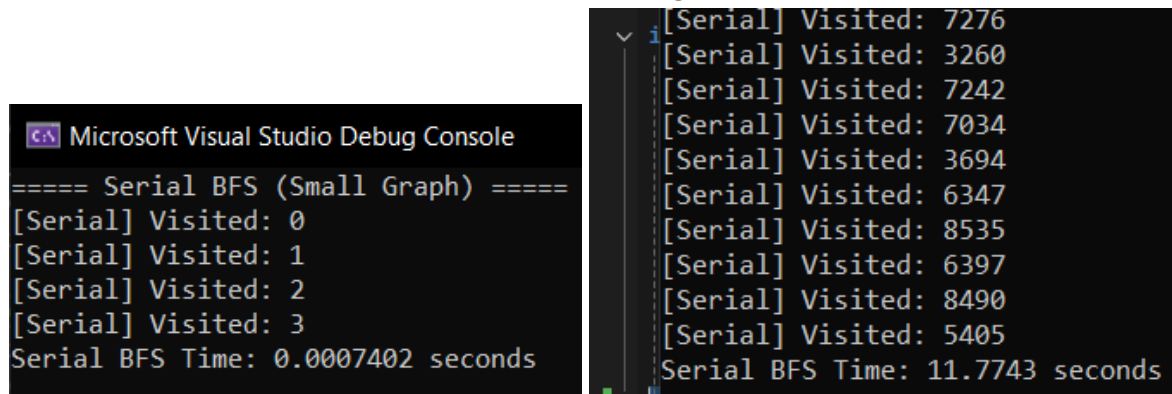
After confirming the correctness, we also tested the serial version on a **large random graph** with 10,000 nodes and 10 edges per node to see how long it would take.

Serial BFS code:



```
LongGraphBFS.cpp  ShortGraphBFS.cpp  What's New?
LongGraphBFS (Global Scope)
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <cstdlib>
5  #include <ctime>
6  #include <omp.h>
7
8  using namespace std;
9
10 // serial bfs
11 void bfs_serial(const vector<vector<int>>& graph, int start) {
12     vector<bool> visited(graph.size(), false);
13     queue<int> q;
14
15     visited[start] = true;
16     q.push(start);
17
18     while (!q.empty()) {
19         int node = q.front(); q.pop();
20         cout << "[Serial] Visited: " << node << endl;
21
22         for (int neighbor : graph[node]) {
23             if (!visited[neighbor]) {
24                 visited[neighbor] = true;
25                 q.push(neighbor);
26             }
27         }
28     }
29 }
30
```

Output of a Small Graph with 4 Nodes and a Large Random Graph:



```
Microsoft Visual Studio Debug Console

===== Serial BFS (Small Graph) =====
[Serial] Visited: 0
[Serial] Visited: 1
[Serial] Visited: 2
[Serial] Visited: 3
Serial BFS Time: 0.0007402 seconds

[Serial] Visited: 7276
[Serial] Visited: 3260
[Serial] Visited: 7242
[Serial] Visited: 7034
[Serial] Visited: 3694
[Serial] Visited: 6347
[Serial] Visited: 8535
[Serial] Visited: 6397
[Serial] Visited: 8490
[Serial] Visited: 5405
Serial BFS Time: 11.7743 seconds
```

Parallel BFS Implementation

After testing the serial version, we created a parallel BFS version using **OpenMP**. We parallelized the part where nodes at the current level are processed. This allows multiple threads to work at the same time, which improves performance.

We used **#pragma omp parallel for** to process the current level in parallel. To avoid errors from threads accessing the same data, we used **#pragma omp critical** when updating the shared queue and **visited** array. This helped us prevent race conditions.

We tested the parallel version on both the small graph (to ensure the output remained correct) and on the large graph (to compare performance).

Parallel BFS code:



```
30 void bfs_parallel(const vector<vector<int>>& graph, int start) {
31     vector<bool> visited(graph.size(), false);
32     queue<int> q;
33
34     visited[start] = true;
35     q.push(start);
36
37     while (!q.empty()) {
38         int size = q.size();
39         vector<int> currentLevel;
40
41         while (size-- > 0) {
42             int node = q.front(); q.pop();
43             cout << "[Parallel] Visited: " << node << endl;
44             currentLevel.push_back(node);
45         }
46
47         #pragma omp parallel for schedule(dynamic)
48         for (int i = 0; i < currentLevel.size(); ++i) {
49             int node = currentLevel[i];
50             for (int neighbor : graph[node]) {
51                 #pragma omp critical
52                 {
53                     if (!visited[neighbor]) {
54                         visited[neighbor] = true;
55                         q.push(neighbor);
56                     }
57                 }
58             }
59         }
60     }
61 }
```

Output of a Small Graph with 4 Nodes and a Large Random Graph:

```
===== Parallel BFS (Small Graph) =====  
[Parallel] Visited: 0  
[Parallel] Visited: 1  
[Parallel] Visited: 2  
[Parallel] Visited: 3  
Parallel BFS Time: 0.0004981 seconds
```

```
[Parallel] Visited: 4307  
[Parallel] Visited: 5234  
[Parallel] Visited: 9497  
[Parallel] Visited: 3724  
[Parallel] Visited: 4848  
[Parallel] Visited: 9986  
[Parallel] Visited: 9095  
[Parallel] Visited: 9889  
[Parallel] Visited: 5446  
[Parallel] Visited: 3509  
[Parallel] Visited: 9911  
Parallel BFS Time: 2.60173 seconds
```

Graph Generation

To test the BFS on a large graph, we wrote a custom function to **generate a random graph**. The function takes two inputs: the number of nodes and the number of edges per node. For our main test, we used **10,000 nodes** and **10 edges per node**.

We also included a **small example graph** directly in the code to test the basic behavior of both BFS versions.

Large Graph Generation:

```
// generating a large random graph  
vector<vector<int>> generate_random_graph(int nodes, int edges_per_node) {  
    vector<vector<int>> graph(nodes);  
    srand(time(nullptr));  
  
    for (int i = 0; i < nodes; ++i) {  
        for (int j = 0; j < edges_per_node; ++j) {  
            int neighbor = rand() % nodes;  
            if (neighbor != i) {  
                graph[i].push_back(neighbor);  
            }  
        }  
    }  
  
    return graph;  
}  
  
int main() {  
    int num_nodes = 10000;  
    int edges_per_node = 10;  
    vector<vector<int>> graph = generate_random_graph(num_nodes, edges_per_node);  
}
```

Small Graph Generation and cout of Graphs:

```
int main() {
    // small graph
    vector<vector<int>> graph = {
        {1, 2}, // Node 0
        {0, 3}, // Node 1
        {0, 3}, // Node 2
        {1, 2}  // Node 3
    };

    // serial
    cout << "==== Serial BFS (Small Graph) =====< endl;
    double t1 = omp_get_wtime();
    bfs_serial(graph, 0);
    double t2 = omp_get_wtime();
    cout << "Serial BFS Time: " << (t2 - t1) << " seconds" << endl;

    // parallel
    cout << "\n==== Parallel BFS (Small Graph) =====< endl;
    double t3 = omp_get_wtime();
    bfs_parallel(graph, 0);
    double t4 = omp_get_wtime();
    cout << "Parallel BFS Time: " << (t4 - t3) << " seconds" << endl;

    return 0;
}
```

Performance Results

To evaluate the performance improvement from parallelization, we measured the execution time of both the serial and parallel BFS implementations on a large random graph with 10,000 nodes and 10 edges per node. The `omp_get_wtime()` function was used to calculate the execution time in seconds.

Timer Codes for Serial and Parallel BFS:

```
// serial
cout << "==== Serial BFS (Large Graph) =====< endl;
double t1 = omp_get_wtime();
bfs_serial(graph, 0);
double t2 = omp_get_wtime();
cout << "Serial BFS Time: " << (t2 - t1) << " seconds" << endl;

// parallel
cout << "\n==== Parallel BFS (Large Graph) =====< endl;
double t3 = omp_get_wtime();
bfs_parallel(graph, 0);
double t4 = omp_get_wtime();
cout << "Parallel BFS Time: " << (t4 - t3) << " seconds" << endl;
```

The results show that the parallel BFS was approximately **4.5 times faster** than the serial version, demonstrating the effectiveness of OpenMP parallelization in large-scale graph traversal.

Test Type	Graph Size	Execution Time (seconds)
Serial BFS	10,000	11.7743
Parallel BFS	10,000	2.60173

Reflection

This final project helped us understand how to parallelize a graph algorithm using OpenMP. We started by implementing a serial version of the Breadth-First Search (BFS) algorithm using a queue and a visited array. The serial version worked well for small graphs but became very slow when tested with large graphs containing thousands of nodes.

To improve performance, we implemented a parallel version of BFS using OpenMP. We parallelized the processing of nodes at each level of the BFS using `#pragma omp parallel for`. We also used `#pragma omp critical` to ensure that the shared queue and visited array were updated safely. One challenge was dealing with synchronization, as race conditions could occur if multiple threads modified shared data at the same time. Adding the critical section solved this issue, but we learned that it can slow down performance if overused.

We also wrote a function to generate a large random graph, which allowed us to test the algorithm on a realistic dataset. By measuring execution time using `omp_get_wtime()`, we observed a significant performance improvement. For example, on a graph with 10,000 nodes, the parallel BFS ran about four to five times faster than the serial version.

Through this project, we gained hands-on experience with OpenMP, parallel loops, synchronization techniques, and performance measurement. We also improved our understanding of how shared-memory programming models work. Overall, we found this project very useful and practical. It showed us that even simple algorithms like BFS can benefit from parallelism when applied to large datasets. This knowledge will help us in future work involving performance optimization and parallel computing.