Shahid Beheshti University

Faculty of Mathematical Science

Department of Computer Science

Concurrent Programming – Winter 2022 – Assignment 2 – DFS implementation with OpenMP

By:

Arman Davoodi

Introduction

In this assignment, the iterative DFS graph traversal algorithm was written in C++ in both parallel and sequential format. Also the parallel implementation is done using OpenMP library, and the graph data structure is an array of nodes, each having an array containing their children indexes. The main Objective of this assignment, other than the implementation, is to compare both sequential and parallel methods in terms of time efficiency.

The results given in this report are based on several runs on a system with Intel Core I7-8550U CPU and Windows 10 operating system.

Sequential Implementation

In sequential implementation, a vector data structure is used as a stack to make use of data dependency and reduce the time needed to read data from the main memory. Then the first node is pushed to the stack and its visited flag is set true. After that, there is a while loop that runs until the stack has at least one element inside it. In the loop, a node is popped from the stack and then all children of that node _which are not pushed to the stack before_ are pushed to the stack and their visited flag is set true _so that the algorithm knows that they have been already pushed to the stack_. At the end of the loop, if the stack is empty, then it means that all nodes which are reachable from the first node pushed to the stack are traversed. So the algorithm checks whether there are any nodes in the graph that it hasn't seen or not; and if there are, one of them is pushed to the stack.

Parallel Implementation

In parallel implementation, each thread has its own stack and starts by pushing a node _which is not previously visited_ into its stack, then it starts to traverse the graph from the visited node. Since in this implementation, the nodes' visited flags are read and modified by all threads, a lock is associated with each node to prevent race condition.

To implement this algorithm, an OpenMP parallel-for on all nodes is used which checks whether a node is visited or not, then if not, it pushes that node to the thread's stack and then the thread will start traversing the sub-graph reachable by the visited node. Moreover, due to load balancing, dynamic scheduling is used for this loop.

Random Graph Generator

For testing and comparing time efficiency of the methods, an algorithm was implemented to generate a random graph of arbitrary size. This method gets the size of graph _number of nodes_ as a parameter and generates random integers for the value of each node.

Let N be the set of all nodes in the graph. To build the arcs, the algorithm generates a random number for all elements of the Cartesian product of N by itself like (n_i, n_j). If the random number was not divisible by two, it considers that an arc exists from the node n_i to the node n_j.

Chunk-Size tuning

As it was mentioned, parallel implementation of this algorithm uses dynamic scheduling. In addition, since the system that the code is run on has only 8 cores, 8 threads are used for the parallel algorithm. To find an appropriate chunk-size, the algorithm was run 100 times on a graph of size 10000 for each chunk-size and the results are presented in *Table 1*, where we can conclude that the best chunk-size in this situation is 1.

Table 1

Chunk-size	Average Time in seconds
1	1.247516
10	1.278765
20	1.343930
100	1.427512

Comparison

To compare time efficiency for these implementations, both algorithms were run on 3 graphs of size 13, 10000 and, 20000. The result of each one is illustrated respectively on *Table 2*, *Table 3* and, *Table 4*. The times section of each table represents how many times the algorithms were run continuously on the same graph and AVG. Duration is the average time of all of those runs.

Table 2: the data on this table are associated with a graph of size 13

times	1		10		100	
	Duration(s)	Speedup	AVG. Duration(s)	Speedup	AVG. Duration(s)	Speedup

Sequential	0.000065	0.01	0.000026	0.25	0.000041	0.77
Parallel	0.006038		0.000106		0.000053	

It can be seen that since the number of nodes for the graph associated with *Table 2* is too low, the overhead of the parallelization is more than the speedup that it gives. Therefore, the total speedups represented in *Table 2* are less than 1.

Table 3: the data on this table are associated with a graph of size 10000

times	1		10		100	
	Duration(s)	Speedup	AVG. Duration(s)	Speedup	AVG. Duration(s)	Speedup
Sequential	1.462731	1.18	1.447521	1.19	1.432969	1.11
Parallel	1.238838		1.221330		1.287413	

Table 4: the data on this table are associated with a graph of size 20000

times	1		10		100	
	Duration(s)	Speedup	AVG. Duration(s)	Speedup	AVG. Duration(s)	Speedup
Sequential	5.714347	1.24	5.731135	1.23	5.790773	1.06
Parallel	4.598362		4.654803		5.438028	

From *Table 3* and *Table 4* we can see that in total, the parallel implementation gives a speedup more than 1 for large enough graphs. However, somehow the speedup in the last section of *Table 4* is much lower than other sections' speedups.

Increased Runtime with Higher Number of Tests

From a statistical point of view, it is better to run the same test more than once to have a higher confidence in the result of the test. However, while doing this assignment, the average time of 100 runs of the parallel algorithm on the graph with 20000 nodes was much higher than the runtime of a single run of the same algorithm on the same graph. Therefore, the time it takes to run both parallel and sequential algorithms on the graph for ith time was measured and the results are shown in *Figure 1*.

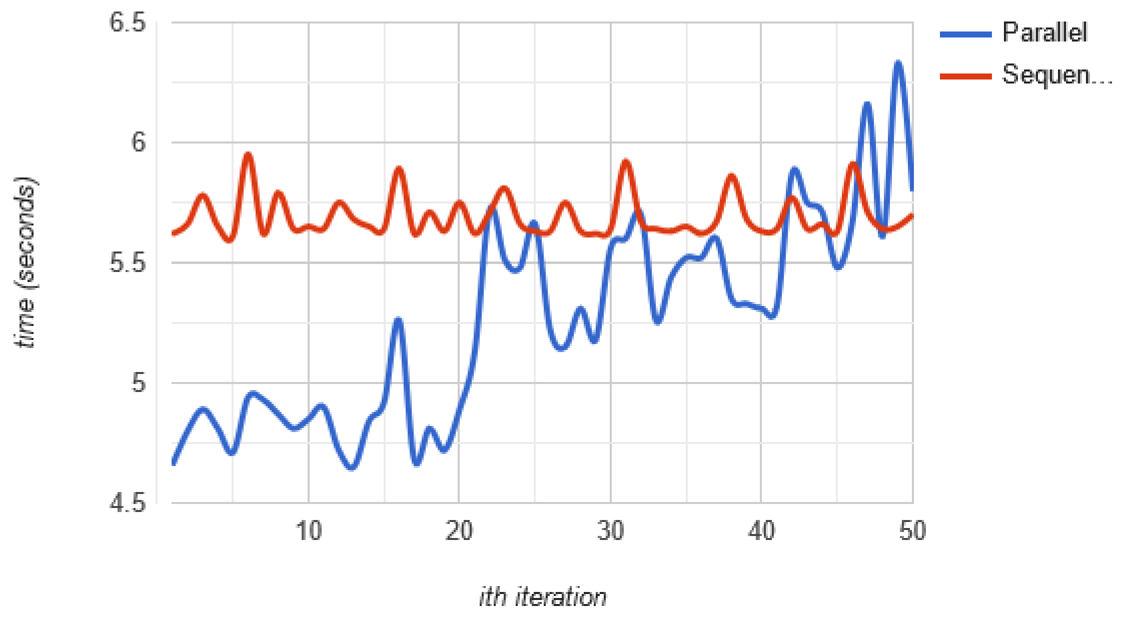

Figure 1: the red line represents sequential and the blue line represents parallel algorithm. The graph shows the runtime in seconds for the ith time each algorithm was run continuously.

Considering this graph, we can conclude that it is probable that the operating system reduces the degree of parallelism for the parallel algorithm when it is run multiple times. Therefore, on large iterations for large graphs, the parallel algorithm's performance will be the same as the sequential algorithm.

Conclusion

In the end, we can conclude that this parallel implementation of iterative DFS graph traversal algorithm can give us a speedup higher than 1.20 for large enough graphs.