Sequential and Parallel Triangle Counting Algorithm on Undirected Graphs

Riccardo Zuliani 875532

March 26, 2023

Assignment Topic

The topic of this assignment is to perform some benchmarks comparing the sequential and parallel execution of an algorithm implementation to count the number of triangles in a given graph using C++.

Proposed Solution and Keys Choices

The first step was to be able to read the edges list of a given graph from a .csv file (obtained from the Stanford Datasets site) and store it in data structure, I decided to use the *std::vector<std::pair<int, int>>* since was the easiest choice and the most reasonable. Next I moved to set up the *UndirectedGraph* object with its initial private attributes like the number of edges, number of vertices and the graph density, but also moved the creation of the edges list in the object constructor. Note that all .csv files had been manually pre-processed inserting as top row the number of vertices, replacing the tabs or spaces between two vertices index with a comma. For this assignment I decided to use the adjacency lists to maintain the vertices neighbours since with the standard C++ library it is was much intuitive to build rather than the incidence matrix, in fact I represent it as a *std::vector<std::vector<int>>*. The implemented *TriangleCounter* function iterates through all the edges and sums the length of the intersection of each pair of vertices and at the end it divides the resulting sum by three since I count each single triangle three times. One of the main improvement was to move modify the intersections step since in order to be able to apply the *std::set_intersect* function the two adjacency list must be sorted, thus rather than sorting at each time I intersect two list I moved this step in the adjacency list creation, by making an ordered insertion. Furthermore, in order to conduct a deep analysis on the algorithm performance I decided to develop two functions to create random undirected graphs given the number of edges and vertices, one for sparse graphs and the other for dense graphs.

Parallelization topics:

- *OpenMP*: I decided to use OpenMP after develop the whole application with the *std::thread* since the code looks much clean and is easy to debug.
- *TriangleCounter*: I have added the OpenMP *parallel for* compiler directive specifying the number of threads, the schedule dynamic and the reduction.
- *Adjacency lists*: Similarly like the previous step except the reduction. Indeed here since there is a race condition that could happen in case multiple threads want to write in the same adjacency list. I decided to use a $n-$dimensional vector of *omp_lock* with n equal to the number of vertices, since each adjacency list can be updated by a single thread at time. I have decided to adopt this strategy instead of the compiler directive *#pragma omp critical* since I want that in case there is no lock in a specific adjacency list, the thread that wants to access and update that vector can perform the operation. Instead with the compiler directive only a single thread at time can execute the respective code block.

Solution Structure

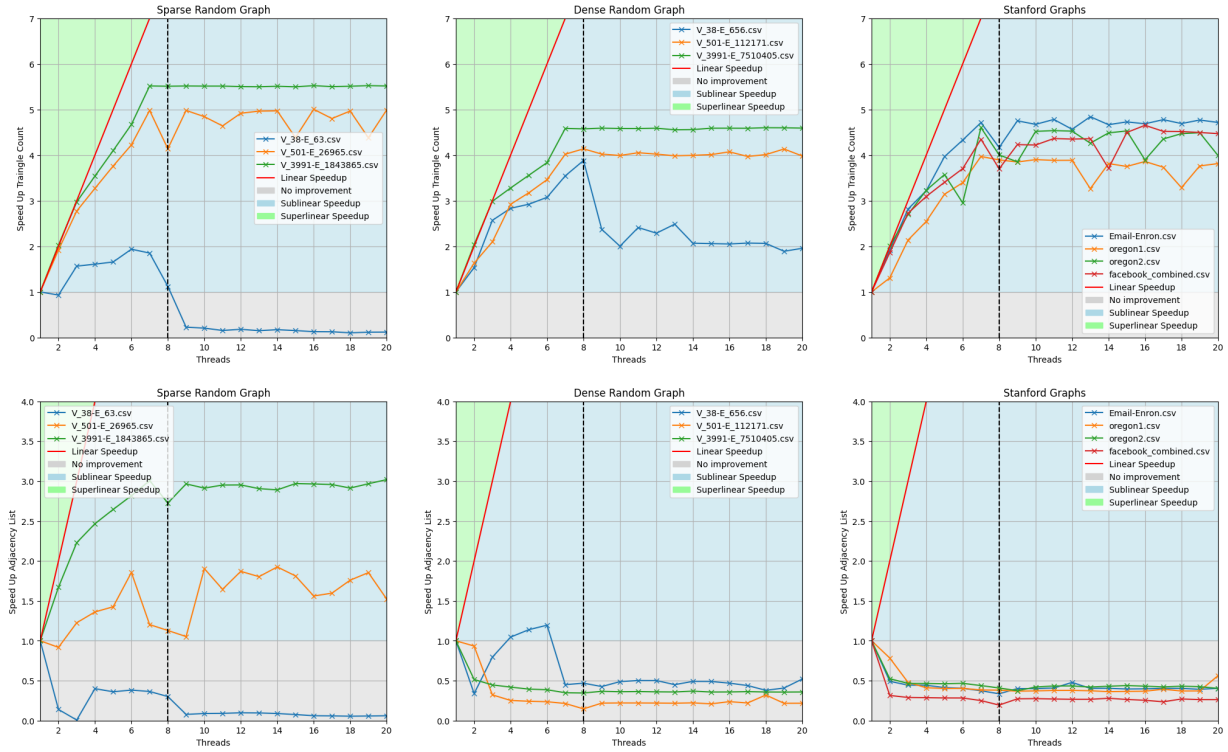
The structure of my implementation is very simple and is defined by 5 main folders:

- **bin**: contains the executable file, created after compilation.
- **datasets**: contains other two directories one for the *random_graphs* and the other for the *stanford_graphs*.
- **include**: contains *UndirectedGraph.hpp* that implement the Graph object so the reading process from the file, the creation of the adjacency list and the counting triangle function, and *Utils.hpp* that contains the function for the random grahs generation and the intersection fuction that return the length of the intersection between two adjacency lists.
- **results**: contains the .csv files named with the respective data and time stamp.
- **src**: contains the *Main.cpp* and here you can run the following command in the terminal to compile the project: *g++ -std=c++2a -fopenmp -O3 -o ../bin/app Main.cpp*

Further Improvement

One of the possible further improvements that I want to apply is the implementation of the Multicore Triangle Computations Without Tuning, where the authors adopted the Compressed Sparse Row (CSR) variant for the adjacency lists that stores the row pointer array as a compact bitmap instead of a dense array. Moreover the edge list of the graph can be sorted by the source vertex of each edge and this procedure can be speeded up by taking into account the degree of each vertex. For example by sorting the edges in the edge list by the source vertex, and then sorting the neighbors of each vertex by their degree. This will improve cache locality and potentially speed up graph processing algorithms that rely on the adjacency list.

Benchmarks



All these results were obtained from a HP EliteDesk 800 G1 TWR supplied with 8 cores i7-4790 CPU. In the first row there are the graphs that explain the speed up for the triangle counting problem whereas in the second row the speedup for the adjacency list creation. The first and second columns identify the *random sparse* and *dense* graphs, 3 graphs each for range of number of vertices, **small:** 10 - 100, **medium:** 101 - 1000, **big:** 3001 - 4100. And Finally the third column represents the *Stanford graphs*. In the first row as expected the performance of the triangle counting algorithm in all graphs continues more or less to grow until we reach the maximum number of available threads (8 in our case). The only exception is for the smaller sparse and dense graph in which at some point the performance starts to decrease, maybe due to the overhead of the thread management. In the second row we note that the adjacency list speedup of the bigger sparse graph reaches up to 3x the performance with respect to the sequential version, whereas the medium sparse graph got slightly worse performance and the small graph was even worse than the performance of the sequential execution. Continuing with the other two plot of the second row as expected the speedup for the adjacency list creation for all dense graph is bellow the performance of the sequential execution. This happens when as the density of the graph approaches 1, the probability that a thread wants to insert a vertex into a specific adjacency list where there is another thread working on it is close to 1. Finally the speed up for the adjacency list creation for the Stanford graph is below the sequential execution since every graph is very sparse.

Dataset Name	# Edges	# Vertices	Density	# Triangles
facebook_combined.csv	88234	4039	0.01082	1612010
Email-Enron.csv	183831	36692	0.000273098	727044
oregon1.csv	23409	11174	0.000375003	19894
oregon2.csv	32730	11461	0.000498389	89541
V_38-E_63.csv	63	38	0.0896159	6
V_38-E_656.csv	656	38	0.933144	6844
V_501-E_26965.csv	26965	501	0.215289	208012
V_501-E_112171.csv	112171	501	0.895577	14981964
V_3991-E_1843865.csv	1843865	3991	0.231582	131499899
V_3991-E_7510405.csv	7510405	3991	0.943276	8886671265