Benchmark on Sequential and Parallel Graph Triangle Counting Problem

Riccardo Zuliani 875532

March 25, 2023

Assignment Topic

The topic of this assignment has the main objective to perform some benchmarks comparing the sequential and parallel execution of an algorithm implementation to count the number of triangle in a given graphs using C++. The parallel step could be done both using the classical *std::thread* library or using *OpenMP* a Cross-platform API for creating parallel applications on shared memory systems. Moreover at our disposal we have a large vastity of possible undirected graph available from the Stanford Large Network Dataset Collection

Proposed Solution and Keys Choices

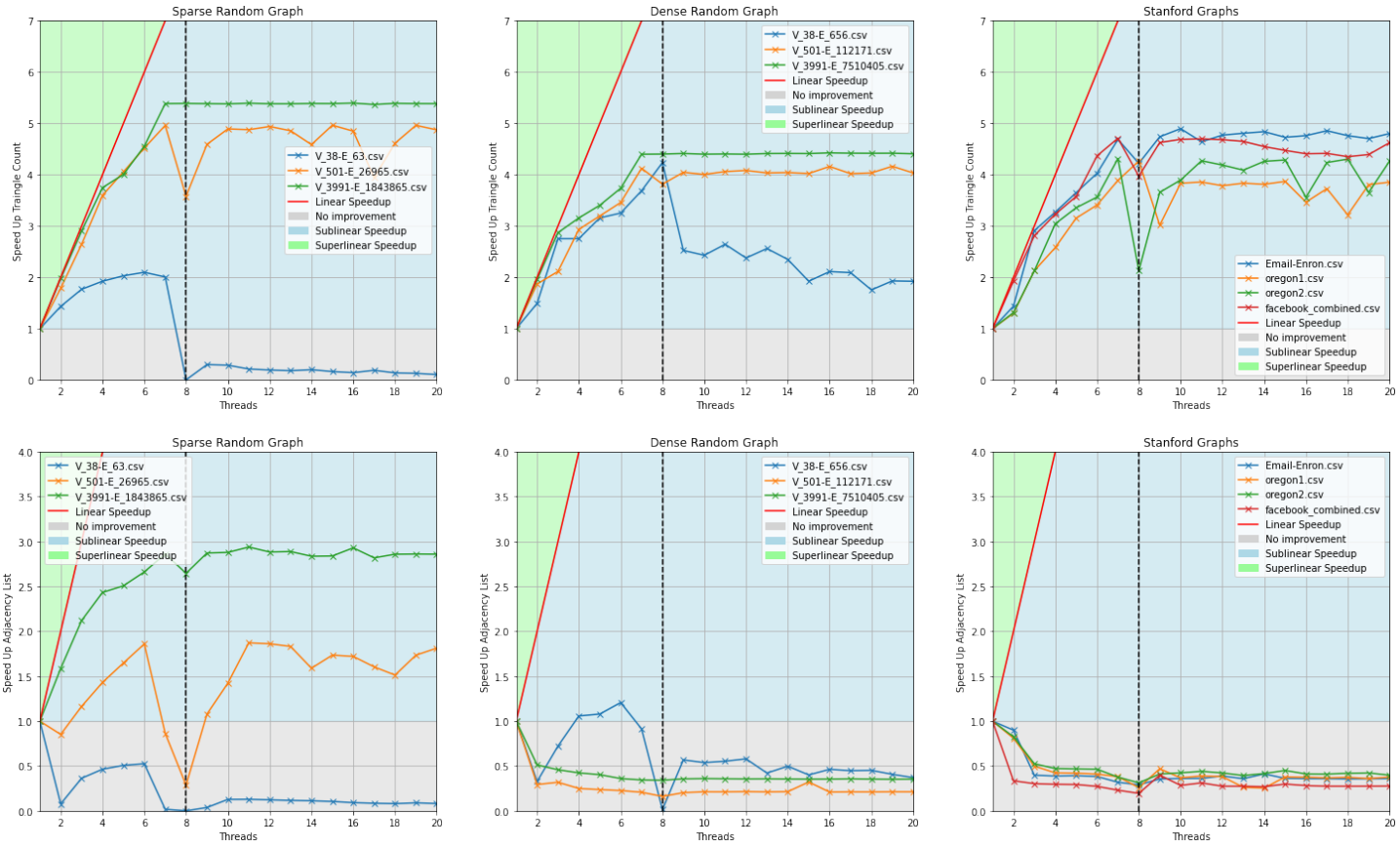
The first step was to be able to read the edges list of a given graph from a .csv file and store it in data structure, I decided to use the *std::vector<std::pair<int, int>>* since was the easiest choice and the most reasonable. Next the development moved to set up the *UndirectedGraph* object with its initial private attributes like the number of edges, number of vertices and the density. Moving on the creation of the edges list was moved on the constructor with given the file path. Note that the .csv files had been manually pre-processed inserting as top row the number of vertices and replacing the tab or spaces with the comma. In the next step I decide to opt for the adjacency list since is was much intuitive to build with the standard C++ library rather than the incidence matrix. At this point I developed the *TriangleCounter* function that iterate through all the edges and sum the length of the intersection of each pair of vertices and finally dividing the sum by three since we count the same triangle three times. Initially in the intersection step was present the ordering of both adjacency list in order to be able to apply the *std::set_intersect* function but further I decide to move the sorting process during the creation of the adjacency list, indeed there I make each insert of a new vertex adjacent to an other by first looking on the exact ordered position and the making the insertion. Up here I was adopting the *std::thread* to parallelize the triangle count, but then I decide to move to OpenMP since in my opinion the code looks nicer and much easy to read and thus debug. Refer to the code for further analysis. The first checkpoint for the application development was reached with the writing of useful statistics like the time take of find the number of triangle given a graph and the number of threads. Next since there are not many dense graph to perform a complete analysis, given a range of number of vertices I decided to develop a function to generate a sparse random graph and an other for the dense graph. With these random graphs I was able to conduct a complete benchmark on the performance of my implementation. At this point I move forward and decide also to parallelize also the create of the adjacency lists, there things become quite tricky since there is a race condition that could happen in case multiple threads want to write in the same adjacency list. To over comer this problem I decide to use implement a vector of OpenMP lock (*omp_lock*) with dimension 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 want to access and update that vector can do that. 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 two files, the first is *UndirectedGraph.hpp* that implement the Graph object containing the reading process from the file, the creation of the adjacency list and the counting triangle execution. Whereas the second is *Utils.hpp* that contains helpful functions for the correct execution of the application
- **results**: contains the .csv files organized named with the respective data and time stamp
- **src**: contains the *Main.cpp* and here we can run the following command line in the terminal to compile the project: *g++ -std=c++2a -fopenmp -O3 -o ../bin/app Main.cpp*

Benchmarks



All these results were obtained from a HP EliteDesk TW supplied with 8 cores CPU. The graph explain in the first row all 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 in which we run 3 graphs each for range of number of vertices, small: 10 - 100, medium: 101 - 1000, big: 3000 - 4300. And Finally the third column represent the Stanford graphs. In the first row as we can expect the performance of the triangle counting problem in all graph continue more on less to grew until we reach the maximum number of available threads (8 in our case), the 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. Furthermore in the second row more interesting we note that the adjacency list speedup of the bigger sparse graph reach up to 3x the performance of the sequential version whereas medium got slightly worst performance and the small even worst resulting below 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 happen in case density of the graph is closer to 1 the probability that a thread want to insert a vertices in a specific adjacency list where there is an other threat working on it is bigger and again closer to 1.

Main difficulties

One of the main difficulty that I have encountered developing this benchmark was with no clue the parallelization of the creation of the adjacency list. It took me more or less 1 week. The difficulty was to deciding the best path to make sure each thread is able to write to the same adjacency list where there is an other thread that is making some changes. As described previously to mitigate this race condition I opted to implement the lock in OpenMP via the cited constructors.

Further Improvements

Surely 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 adjancency 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. One way to do this would be to sort the edges in the edge list by the source vertex, and then sort the neighbors of each vertex by their degree. This would ensure that the neighbors of each vertex are ordered by their degree, which can improve cache locality and potentially speed up graph processing algorithms that rely on the adjacency list.