VERTEXWISE TRIANGLE COUNTING

The topic of the project is triangle counting of large unweighted and undirected graphs, by calculating a vector *c3*, that express the number of triangles that every node takes part in. In graph theory, the triangle graph is the complete graph *K3*, consisting of three vertices and three edges. Triangle counting has gained increased popularity in the fields of network and graph analysis. The application of triangle detection, location, and counting are multi-fold: detection of minimal cycles, graph-theoretic clustering techniques, recognition of median, claw-free, and line graphs, and test of automorphism. Source code consist of two different algorithms (*v3* and *v4*) of calculating *c3* vector. For every version, there is a sequential implementation and some parallel ones, that use *Cilk*, OpenMP and *PThreads.* The parallel implementations were tested in the *AUTh High Performance Computing (HPC) infrastructure* and data of operation run time were exported. Afterwards, we used these data to analyze the behavior of our code.

At this point, we are going to explain some critical functions and points of our code.

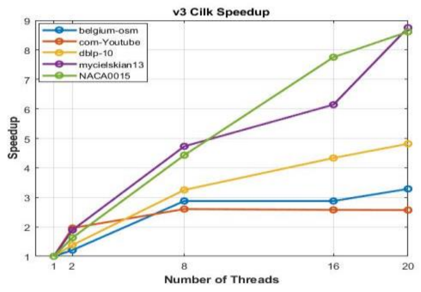
- `find_triangles (v3)`: This function is used to calculate c3 vector. We have already read the symmetric COO format of the graph and converted it to CSC. So, through this function we take every node i (from 0 to n-1) and find all the pairs (j, k) of its neighbors (=nodes that are connected with) and then, by calling *check_edge* function, we find out if these two are connected with an edge to each other too. If this happens, we have find a triangle, so we increase c3[i] by 1. We do not increase c3[j] and c3[k] in this iteration to avoid data race in parallel implementations. Note that we will find the same triangle three times with all the possible nodes as first, so we do not loose any triangle.
- `check_edge (v3)`: As already mentioned, it is used inside *find_triangles* to check if there is an edge between a pair of nodes. It uses binary search to make problems with high amount of data more efficient (O(logn) steps).
- `readmtxvalues, openmtxfile (v3 & v4)`: Used to read *mtx* file and convert it to two COO arrays.
- `find_triangles (v4)`: In this version, the function is different. Here, we take every pair of nodes that are connected to each other (masking with adjacent A) and we are counting their common neighbors (array A*A) with *common_neighbors* function. The result is the number of triangles that they both take part in. We only increase c3[i] though, for the same reason we explained earlier.
- `common_neighbors (v4)`: Counts the common neighbors of two given nodes in a very efficient way, taking advantage of the fact that the two lists are shorted. We implement a method similar to "merging of two shorted lists" and we replace a (length1*length2) complexity algorithm with an O(length1+length2) algorithm.
- `Cilk implementation (v3 & v4)`: We chose to use only one *"cilk_for"*, as we suppose that $n \gg$ *num_of_threads*. In this way, each thread takes over one iteration (of n total) and when it finishes with that, it takes the next available of the queue, working dynamically. So we utilize all of our threads every moment and we maximize the speedup.
- `OpenMP implementation (v3 & v4)`: Same implementation with *Cilk*.
- `PThreads implementation (v4)`: Here, we implemented dynamic scheduling with *pThreads*. Each thread, locks the global variable "iteration" to avoid data race that would cause multiple computations of the same iteration. The thread keeps the value of iteration using a local variable *"t_it"* and increase the global "iteration" by one. So, this thread computes a specific iteration, while the next thread that will be available will take over the next iteration of the virtual queue in the same way.

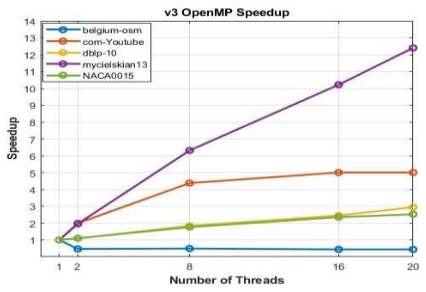
As we have mentioned, we also used the *AUTh High Performance Computing (HPC) infrastructure* to test our code behavior for specific graphs given as input. We present our graphs through the following table:

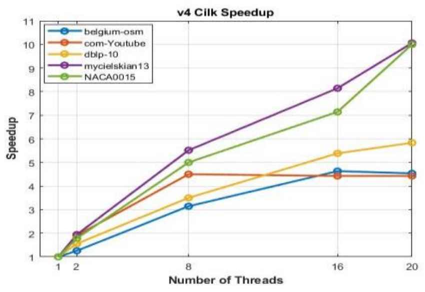
graph	n (nodes)	m (edges)	m/n (density)	m×n (complexity)	triangles
belgium_osm	1,441,295	1,549,970	1.07	2.23×10^12	2420
com-Youtube	1,134,890	2,987,624	2.63	3.39×10^12	3,056,386
dblp-2010	326,186	807,700	2.47	2.63×10^11	1,676,652
mycielskian13	6,143	613,871	99.9	3.77×10^9	0
NACA0015	1,039,183	3,114,818	3	3.23×10^12	2,075,635

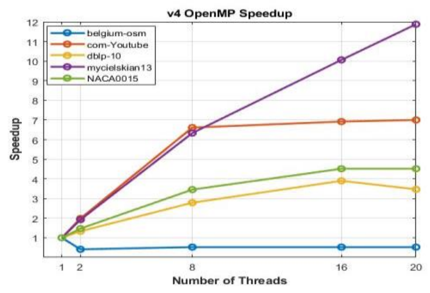
We notice that the size of the problem is determined of n and m, but we cannot short the graphs by size, as the exact dependence is not known, but we could compare the graphs according to their density and complexity, as we defined them. So, we could say that *com-Youtube* is the most complex one, followed by *NACA0015* and *belgium_osm*, while *mycielskian13* is extremely dense and *belgium_osm* extremely sparse. It is really remarkable the fact that *mycielskian13* does not have any triangles despite its density. *Dblp-2010* is a balanced graph in both criteria.

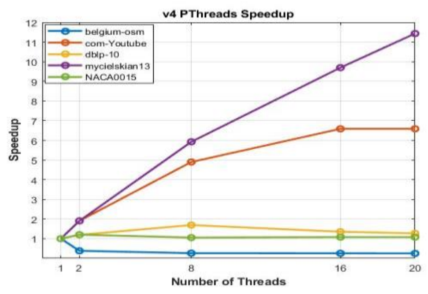
In our experiment in *HPC,* we used every single version of our code, for every single graph and with different amount of threads each time (1, 2, 8, 16, 20). So, we recorded the time of every operation and created the speedup diagrams that are presented below. We calculated speedup through the formula: *speedup(n_threads) = operation_time(1_thread) / operation_time(n_threads).*









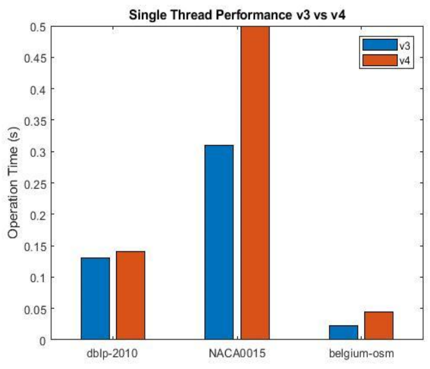


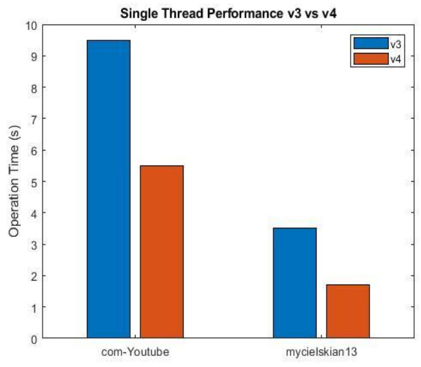
Amdahl's law:

In computer architecture, Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. Amdahl's law gives an **upper bound** of speedup of every parallel implementation, as there is a part of every problem that has to be computed sequentially.

Amdahl's law is confirmed through our diagrams. We can notice that almost every process meets an upper bound on its speedup in the range of 20 cpu threads. Also, according to Amdahl's law, the processes that have not met the upper bound in the 20 thread experiment, are going to do for some wider range of threads.

Moreover, it is clear that the parallel implementation is really efficient in most of the cases. *Mycielskian13* shows the greatest speedup in all of the methods. Its high density could be factor of that, as every thread has a huge amount of work to do in each iteration and the work is almost equally distributed. On the other hand, *belgium_osm*, who is a complex and sparse graph, shows very bad parallel behavior in *OpenMP* and *PThreads*, as the distribution of work is not done properly. However, *Cilk* looks to make much better scheduling with *belgium_osm. Dblp-10,* who is an average graph regarding to our criteria, density and complexity, displays an average behavior regarding to its speedup too, in comparison to the other graphs. Overall, v4 looks to present better speedup performance in comparison to v3. Last but not least, our attempt to implement dynamic scheduling in *PThreads* looks to perform great in the 'dense' *mycielskian13* and the 'complex' *com-Youtube,* but not that good on the other graphs (almost equally to single thread).





# TODO comments v3 vs v4 and operation time between graphs