Learning With Massive Data
CLAUDIO LUCCHESE

ASSIGNMENT 1

a.y. 2022/23

Quintavalle Sebastiano
Student Number 878500

April 4, 2023

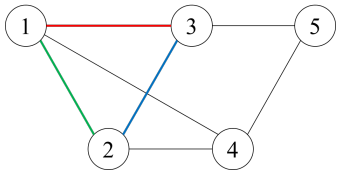
1 Triangle in a graph

1.1 The problem

Triangles in a graph The assignment consists in the problem of counting the number of triangles in an undirected graph $G = (V, E)$; V is a set of vertex with $|V| = n$ and $E \subseteq V \times V$ is a set of edges with $|E| = m$. In graph theory a triangle can be formally seen as a 3-clique: a subset of three vertices such that every two distinct vertices in the clique are adjacent.

Data structures Graphs are provided as datasets describing the list of edges as a pair integers which are node identifiers. The list of edges is implemented as a vector of pairs. The list is used to create an affinity matrix $A : n \times n$, a data structure such that $A[i, j] = 1 \iff (i, j) \in G[E]$.

Solver algorithm Counting triangles procedure exploits both edges list and affinity matrix; we inspect one edge at a time counting how many triangles is the edge contained in. In particular for edge $(a,\ b)$ we count how many entries in rows a and b of the affinity matrix are set to 1 at the same time: in so doing we find a common neighbor c and so the triangle abc. Each triangle is counted three times (one per side) and so the final summation must be divided by a factor of three. Let's consider in the following example the edge $(1,\ 3)$:



(a) Graph

1, 2
1, 3
1, 4
2, 3
2, 4
3, 5
4, 5

(b) Edges List

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	0
3	1	1	0	0	1
4	1	1	0	0	1
5	0	0	1	1	0

(c) Affinity Matrix

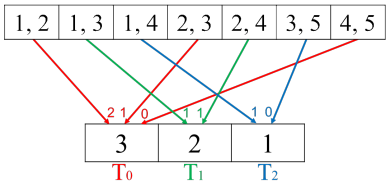
Complexity The lgorithm exploits both edges list and affinity matrix, so the total space occupation is $S(N, M) = M + N^2$. The construction of the affinity matrix only requires $\Theta(M)$ time; we linearly scan the list of edges and set two cells (because of the symmetry) of the affinity matrix to 1 in constant time. The solving algorithm linearly scans the list of edges; each edge requires in turn the linear scan of two rows of the affinity matrix for a total time of $T(N, M) = \Theta(M \cdot N)$.

1.2 Parallelization

Algorithm parallelization operates on the scan of the list of edges: the computation of the number of involved triangles for each edge is independent, so the process can be easily parallelized. Two different implementation are provided: the first using C++ thread standard library, the second using OpenMP API. Experimental results of the two implementation are compared in experimental evaluation section.

OpenMP OpenMP is a portable directive-based API that provides some tool to parallelize some common programming pattern, such as in our case for loops. The implementation uses the reduction directive to sum up partial results of each thread which saves in a private variable the partial number of triangles found. Parallelization is delegated to the API thanks to proper directives.

Thread Standard thread library needs for a more detailed implementation since we must specify ourselves each thread task and how to combine partial result. In the implementation edges are assigned to each thread in an interleaved manner: each of the T threads is responsible for one of every T edges. Moreover, each thread uses a single cell of an array to store results, so that we avoid conflicts. An example in the figure:



Final results is the summation of all cells in such array (divided by a factor of three). This implementation choice has important consequences especially for caching strategies, as will be examined in the next chapter.

2 Experimental evaluation

2.1 Problem setup

Dataset The algorithm is tested with five datasets from Stanford Large Network Dataset Collection. All available datasets are sparse (the number of nodes is linear with respect to number of edges). In order to study a possible different behavior with dense graphs an extra auto-generated full-graph was introduced.

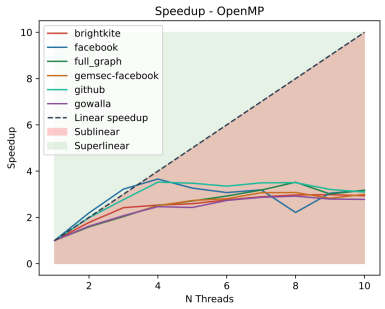
Dataset name	Nodes	Edges	Triangles	Data source
Brighkite	58'228	428'156	989'456	Ioc-Brightkite
Facebook	4'039	88'234	1'612'010	ego-Facebook
Gemsec-Facebook	50'515	1'380'293	4'065'768	facebook gemsec
Github	37'700	289'003	523'810	muasae-github
Gowalla	196'591	1'900'654	4'546'276	Ioc-Gowalla
Full graph	1'000	499'500	166'167'000	

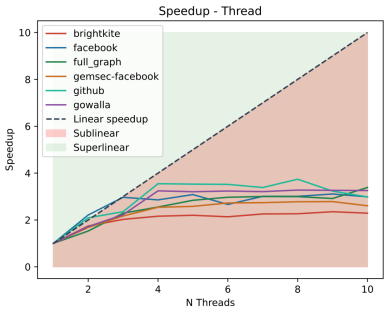
File organization Datasets are stored in a local directory called *data*; the directory contains a folder for each dataset (called with dataset name); each folder has two files: *edges.txt* which contains the list of edges: each line has a pair of integers divided by a space; the folder also contains *info.json* file describing graph properties: number of nodes, edges, and triangles. The program automatically works with these two files performing reading and parsing operations: it's sufficient to provide just dataset-name to the program. A tiny dataset example is included in assignment submission.

Hardware components Tests are performed using 11th generation processor Intel-Core i7-1165G7 (4 core) and 8 GB of RAM memory.

2.2 Results

The solving algorithm was evaluated for each dataset using from one to thirty threads. After reaching the breakout point of 4 threads (the number of cores of the machine) the speedup is quite stable: plots describe the trend from 1 to 10 threads to better show scalability.





Approaches comparison OpenMP version is in general more scalable than thread standard library one: some datasets have a super-linear speedup; the other which are sublinear are more close to linearity than the standard-thread-library version. OpenMP API is very powerful especially for some kind of programming pattern such as in our case aggregation of partial results over a for loop. It's reasonable that it produces better results rather than a simple own implementation using the standard library.

Graph density Speedup trends are on the overall comparable, the full-graph speedup doesn't seem to be affected by its huge density.

Caching Two datasets in particular seem to be more scalable: *facebook* and *github*. These two datasets have a peculiarity: the list of edges is totally ordered, while in the others the order is partial or in blocks. This fact can be very significant for caching: the affinity matrix is implemented as an array of pointers (each row is actually an array possibly not contiguous in memory with other close rows). Since edges are processed following edges list order, if they are sorted it is likely that the first node composing the edge is accessed multiple times in a short amount of time and we can exploit space locality. Moreover, the computation for a single edge in comparable in time and is likely for threads to move forward almost together and so to process the same range of nodes. The fact translates in multiple cache hits which really help performance; *facebook* dataset, which has a particular super linear speedup, is very small and probably it was possible to maintain an important part of memory in the cache. Furthermore, this can also be another motivation for standard-thread algorithm to be less scalable: as we have seen each thread process edges making "jumps" in the list of edges, so that this locality property is not so completely exploited.