Università
Ca'Foscari
Venezia

Master's Degree Course in
Computer Science and Information Technology

Artificial Intelligence and Data Engineering

Learning with Massive Data

Assignment 1

Sequential and parallel counting of undirected graph triangles

Author
Giovanni Costa
Student ID
880892

Academic Year
2022-2023

This report describes a C++ parallel implementation (with OpenMP library) of one of the most advanced algorithms[1] for counting the number of triangles in an undirected graph and related empirical values obtained. The experiment[2] is performed on three different type of graphs: one quite sparse but with low dimension named *Little graph*, one very big and sparse named *Sparse graph*, and one very dense named *Dense graph*.

1 Implementation

The development is realized according to the object-oriented programming paradigm; so *Graph*, that use $std :: map < T, Set < T >>$, and *Set* that is organized as $std :: vector < T >$ are made. Talking about the algorithm in details, this is conceptually based on two functions, *'rank_by_degree'* and *'triangle_counting'*, that can be also deeply explained looking at the original algorithm paper[1].
The first form a directed graph, where each undirected input edge gives rise to exactly one directed edge. The ranking helps to improve the asymptotic performance and ensures each triangle is counted only once using the criteria $adjacency\_list(v) = \{u \in N(v) \,|\, deg(v) \leq deg(u)\}$ and the breaking tie by ID rule. The second count triangles in the directed graph formed in the previous step by looking at the cardinality of the intersection between a vertex and neighbours of each of its neighbours.

2 Key points

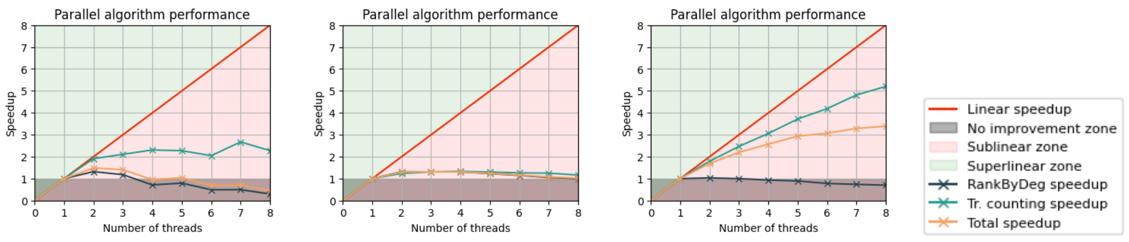
This experiment is not exempt from critical choices, both regarding performance and efficiency.
Talking about implementation, the adjacency list allows for many advantages over the other representations: first of all, the neighbours' lookup is very fast because the adjacent nodes of a vertex are all stored in the same list. Secondly, by definition, it's particularly suitable for sparse data storage and for the addition of a node or an edge. In particular, *std::map* that is physically organized as a red and black tree allows a constant insertion and logarithmic retrieval of the specific node in terms of time complexity.
Considering instead the lists of neighbours, they are implemented with the custom version of *std::vector* as briefly anticipated, that respect element uniqueness and allow fast duplicate checking due to the fact that the node are inserted in order. Also, with *std::vector*, element are stored continuously so the algorithm can take advantage of a space locality. It's pointed out the fact that having no duplicates is important because it's not guaranteed how the input file is encoded and in which order the lists of vertices are written in it (sometimes edges are described as (u, v) assuming graph undirected, sometimes as (u, v) (v, u)). Also, for using OpenMP features data must be contiguous and the *std::set* implementation don't guarantee this.
Regarding the parallelization, some shrewdnesses are taken: firstly, the cache usage approach is constructive and this has permitted to achieve good global times result and low cache misses (see table 1). In fact, the thread creation is done before the *std::map* scan, so each subprocess has in its own memory the current vertex adjacency list and this minimize the cache misses because of common blocks usage. In addition, the iterations of the inner for that scan the neighbours of the current vertex in order to compute the

[1]"Multicore Triangle Computations Without Tuning" by Julian Shun, Kanat Tangwongsan

[2]All tests are based on the average of three program runs and the parameter 'OMP_DYNAMIC' is set to false due to test stability and reliability

intersection and check if there's a triangle, are split dynamically: it's intuitive to think that some nodes of the graph as more neighbours than others and consequently a larger list, so for balancing the workload and reducing the threads idling the specific iteration is assigned at runtime by system.
Lastly, for the concurrent creation of the directed graph synchronization technique is introduced: before the insertion of one element in one adjacency list it's defined a critical section. In this way, different thread can't access to the same memory address at the same time and race conditions are removed. To avoiding these type of problems also the increment of the attribute ‘numEdge‘ is declared as an atomic operation and the computation of the final number of triangle are managed by ‘reduction‘ clause of OpenMP.

3 Experimental evaluation



(a) *Little graph* **(b)** *Sparse graph* **(c)** *Dense graph*

Figure 1: *Speed-up plots*

Looking at all the plots in figure 1 and refering also to the summary table with graphs details (table 1) it can be clearly seen that the parallel creation of the directed graphs did not lead to substantial improvement. In particular, this could be traced to the fact that edges are encoded in group with same vertex ID, so by reading and inserting them sequentially the threads access very likely to the same memory point every time and given the mutual section this is not allowed and causes the scalability reduction. A solution could be shuffling the reading or the data structures insertion after their parsing.
Regarding the triangle counting speed-up, this is satisfying only for the plot 1c: in fact the space locality given by the usage of *std::vector* and the bigger lengths of the adjacency list due to graph density probably make the workload split convenient, and the overall process scale well. The other plots and in particular the one in the figure 1b further supports this hypothesis by the fact that when there are a lot of small adjacency lists the threads' work is quite low, so the benefits of using parallel version doesn't seam relevant with respect to sequential version.

	Little Graph	**Very sparse graph**	**Very dense graph**
No. nodes	4039	1696415	4039
No. edges	88234	11095298	8097333
Density	0.0108	$0.8{\cdot}10^{-5}$	0.993
Best total speedup	1.48	1.31	3.38
Best no. of threads	2	4	8
Best total time (in sec)	0.06	20.59	18.75
L1 cache misses	8.89%	8.74%	7.37%

Table 1: *Experiment's summary table*