Triangle counting in sparse graphs

Fotiou Dimitrios
AEM 9650
Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering
https://github.com/dimfot3/Triangle_Counting

***Abstract*—This document is a report for *Parallel and Distributed Systems*, subject of the Aristotle University of Thessaloniki. The main goal is to find and implement an efficient algorithm for triangle counting in sparse graphs and then try to optimize it with parallelism using pthreads, OpenMP and OpenCilk. The report starts with an introduction and related work upon triangle counting, then explains a proposed method, its implementation and finally presents the results of algorithms utilized in C as well as in Julia.**

***Index Terms*—traingle count, sparse graphs, parallelism, C, Julia**

I. Introduction

Triangle Count algorithms measure the triangles on a graph. Triangles are sets of three nodes that are connected. These algorithms find many applications in social networking like detection algorithms for spam activities and link recommendations [1]. Taking into consideration that real-life graphs can have a vast amount of nodes finding an efficient algorithm that can measure triangles in almost linear complexity, is a challenging task. There are two important key points that we need to evaluate before approaching the problem. The first is that most of these graphs have few connections between them and manipulating them as sparse matrices can save much memory and time. The second key is that nowadays, with the increase of technology even low-budget CPUs come with multiple cores and encourage using parallelism to solve efficiently time-consuming tasks.

In graph theory, the most common way of representing graphs is the adjacency matrices. In large datasets, these matrices can occupy a huge amount of space. Taking into consideration that many of the matrices, that we handle in social networks, are sparse, we can store and manipulate them in more compact ways like Coordinate list (COO) and CSR (Compressed sparse row) or CSC (Compressed sparse column). Here given that the graphs are undirected and so the matrices are symmetric we use a CSR format that is the same as CSC.

There are multiple ways to count triangles in graphs using adjacency matrices. We should distinguish, that counting the total amount of triangles and the triangles per node are slightly different tasks and can alter the complexity of the algorithm. The most simple and time-consuming algorithm with complexity $\Theta(2 \cdot N^3)$ that can find both the total and per node triangles is taking the diagonal of $A^3/6$, where A from now on is the adjacency matrix. Another more efficient way of calculating triangles is formula $A * (A^2)$, where $*$ is the Hadamard or element-wise product and the complexity is $\Theta(N^2 + N^3)$. However, exploiting the sparsity we can avoid the creation of dense matrix A^2, using masking and calculating only on the non-zero values of A. This technique produces a sparse matrix from which we can take the total triangles and per node. Professors of this project proposed this method to be used and parallelized in C and Julia and is analyzed more in the next section. However, it should be mentioned that the algorithm can be optimized even more using the Lower and Upper triangular matrices and calculating the $(LU) * A$ instead of $(AA) * A$ which reduces the amount of execution time when combined with masked multiplication. In case we care only about the number of total triangles and the graphs are undirected we can use algorithms proposed in [2] which calculates the $sum(L * (L^2))$, where L is the lower triangular matrix of A, and reduces significantly the amount of memory and time but ignores the info about triangles per node.

The second important aspect of an efficient triangle counting algorithm is parallelism. The most common approach and yet efficient is to split the rows of the outer loop into threads. This technique is partially aspired by different papers like [2] and [3] and is based on the logic that each thread has its own independent work, avoiding race conditions and mutex solutions that may add some delay. The only difficult part of this appoach is to verify that each thread takes equal amount of work. More about the implementation is the next section.

II. Proposed Method

The method which we will implement for triangle counting is the formula $A * (A^2)$. The first part of this project is to calculate the above formula using masking and avoid calculation in places where A is zero. The basic algorithm can be shown in 1. We iterate through every row i of A, and then for each non-zero column j of that row, we find the sum of common non-zero columns of rows i and j. While from first sight we see three loops and one searching algorithm, given the fact that, by definition of sparsity, the mean number of non zero columns in each row is small, the overall complexity is O(N). This can also be proved using a counter in the most inner loop.

Finding the common non-zero values of i and j rows can be approached as a searching problem. If the non-zero values are well distributed in a sparse matrix then the fastest searching algorithm is probably the linear as the per row non-zero values are small. However, there are sparse matrices that have the

non-zero values gathered and few rows contain almost all the non-zero elements. In that case, using a faster searching algorithms can be beneficial. One of the fastest searching algorithms is binary search. This algorithm implies that i, j rows have columns in CSR format sorted. From the matrix market, this statement holds true but to evaluate every situation a quicksort algorithm is needed to sort every row. In order to have the best searching algorithm for each situation, a dynamic approach is needed that determines the suitable algorithm for each row independently. In CSR format that's a trivial task and not costing as the row indices can reveal the number of non-zero columns per row.

Algorithm 1 Sequential Masked Triangle Counting

Input: CSR matrix A
Output: $C = A * (A^2)$

```
for each row i of A do
    for each each non zero col j of row i do
        for each each non zero col k of row j do
            if col k exists in row i then
                C(i,j) += 1
return C
```

As far as the parallelism each, implementation in pthreads, OpenMP, and OpenCilk approached slightly differently. Pthreads is the lowest level API and utilizing a parallel algorithm is flexible and challenging at the same time. The first implementation is splitting equally to N threads the total rows or paralleling the outer loop. While seems an easy and efficient approach, for parallelism to improve the overall sequential execution time, each thread should have a good amount of work to bypass the overhead of thread's creation. Again here the uniformity of distribution of non zero values comes to increase complexity. In nonuniform sparse matrices, some threads take by far more job than others putting a limit in the overall reduction of execution time. To avoid this unwanted behavior and split equally the amount of work, thread pooling is used. So instead of splitting the amount of work in the first place, we just start some threads that are running as soon as there are free rows that have not been processed. Of course, this needs some use of mutex to avoid race conditions and may increase a little the execution time on small uniform sparse matrices.

In OpenCilk api the parallelism were by far less complex. To parallelize loops there is a pretty handy command $cilk\_for$ that can easily do efficient parallelism with different number of threads that is set as environment variable. By default OpenCilk uses work stealing, a technique that solves the problem of unbalanced work like thread pooling. The only challenging task were the configuration of OpenCilk, as it has been deprecated in latest gcc compilers.

OpenMP API is similar to OpenCilk and a for command exists to make parallelism an easy task. In addition to this, gcc supports it and so compiling was trivial. OpenMP by default uses static scheduler and each thread takes a predefined amount of work, dealing with "lazy" threads in case of unbalanced matrices. To overcome this issue and make the OpenMP implementation competitive to pthreads and OpenCilk, scheduler should be set to dynamic and work stealing technique similar to OpenCilk solves the issue of unbalacing.

TABLE I
DATASETS

Dataset	$\|V\|$	$\|E\|$	Triangles
mycielskian13	6143	613871	0
dblp-2010	326186	807700	1676652
NACA0015	1039183	3114818	2075635
com-Youtube	1134890	2987624	3056386
belgium_osm	1441295	1549970	2420
italy_osm	6686493	7013978	7410
road_central	14081816	16933413	228918

Fig. 1. com-Youtube: Example of uneven distribution of non zero values

III. Implementation

A. Code and Validation

The code was written in C and as a building tool cmake was used, as it is cross-platform and more compact for bigger projects than raw Makefiles. For the validation of the results, total triangles were counted with simple and robust algorithms in Julia for each dataset, so we could easily evaluate the correctness of the more complex implementations. In addition to this, Test Driven Development (TDD) was followed and some Google tests [1] are used to verify the correctness of the basic utilities. All the code and scripts of this project can be found in github's repo [4].

B. Dataset

For the dataset, the recommended matrices from the matrix market [5] were used alongside some others. All the graphs were undirected, symmetric, and sparse. As can be seen in table I there is a variety in the specs of the datasets for the impementation to be tested in different situations.

C. Workspace

The code was first implemented in my device with Arch Linux, AMD Radeon Ryzen 5 2600 and 16GB RAM. However, the final results were calculated in Auth cluster [2] for fair comparisons on the same base with other colleagues that have the same project. The amount of maximum threads used in Auth cluster is 16 in one node.

[1] https://github.com/google/googletest
[2] https://hpc.it.auth.gr/

TABLE II
BEST AVERAGE EXECUTION TIMES(S) COMPARISONS ON CLUSTER WITH MAXIMUM 16 CORES

Dataset	Sequential	Pthreads	OpenMP	OpenCilk
mycielskian13	9.210542	1.152932	**1.078404**	1.174894
dblp-2010	0.391633	**0.072469**	0.073847	0.088613
NACA0015	0.391633	0.193115	0.152550	**0.138416**
com-Youtube	26.885117	3.655631	**3.345062**	3.818570
belgium_osm	0.068673	0.073876	0.089966	**0.012043**
italy_osm	0.305614	0.258632	0.434641	**0.043903**
road_central	2.087676	0.863764	0.924115	**0.275423**

TABLE III
BEST AVERAGE EXECUTION TIMES(S) FOR JULIA IMPLEMENTATION COMPARISONS ON CLUSTER WITH MAXIMUM 16 CORES

Dataset	Sequential	Threads	Speedup
mycielskian13	23.366	3.261	7.17
dblp-2010	0.575308	0.119422	4.82
NACA0015	0.858321	0.167907	5.14
com-Youtube	26.721	5.938	4.5
belgium_osm	0.084935	0.084063	1.01
italy_osm	0.359541	1.238	0.31
road_central	2.692	2.847	0.94

IV. RESULTS

In this sections we will present the results. In table II we have the best/lowest execution times of each dataset and in different implementations. As we can see, the sequential implementation is worse in all cases, even in datasets with small number of vertices. The parallel implementations are close enough but OpenCilk wins in most cases.

An informative plot is 2, where we can see that in half datasets the speedup is close enough in the three parallel implementations. However in datasets where the sequential implementation run in relatively small times the OpenCilk shows significant improvement in time.

In figures 3 and 4 we see two totally different effects of scalability in parallelism. In the first one when the execution time in sequential implementation were high, the increase of threads leads to an expodentail decrease in execution time. While on second dataset where the sequential implementation is fast, the overhead of thread's creation is significant and the increase of threads have none or negative effect to overall execution time.

Finally, as last optional task about this project were the sequential and parallel masked implementation in Julia. In table III we see again that in datasets like mycielskian13 and com-Youtube where sequential impelentation is costly, the parallel can show even 7.2 speedup. However, despite C's, impelementation we see that in some datasets like italy_osm and road_central the parallel implementation can be worse. As far as scalability, from the two plots 5 and 6 we see similar behavior with C's implementation.

V. CONCLUSION

In conclusion, triangle counting is a challenging problem with many applications. In this project, the algorithm for finding the triangles was almost given and the main goal

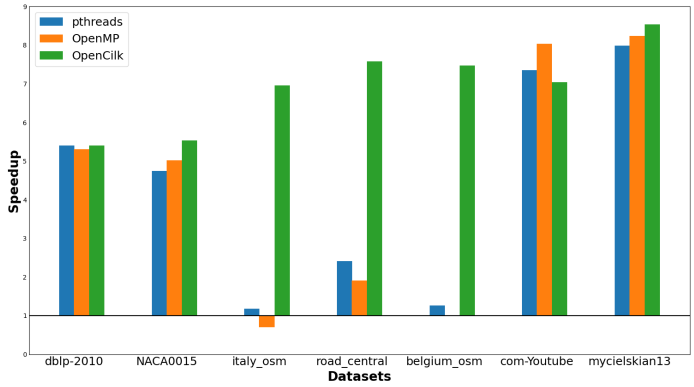

Fig. 2. Speedup comparisons Pthreads/OpenMP/OpenCilk at 16 threads with reference the sequential implementation

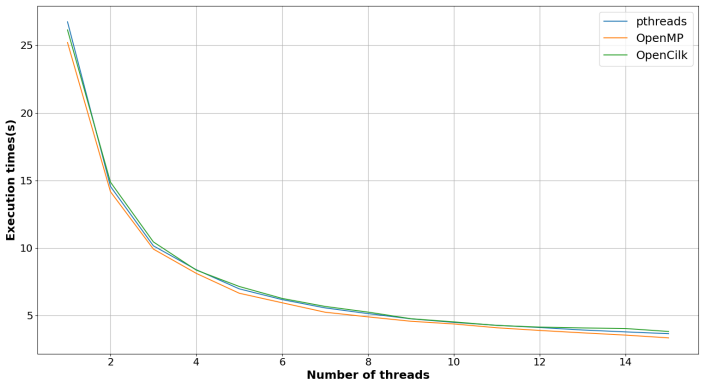

Fig. 3. Scalability in com-Youtube dataset

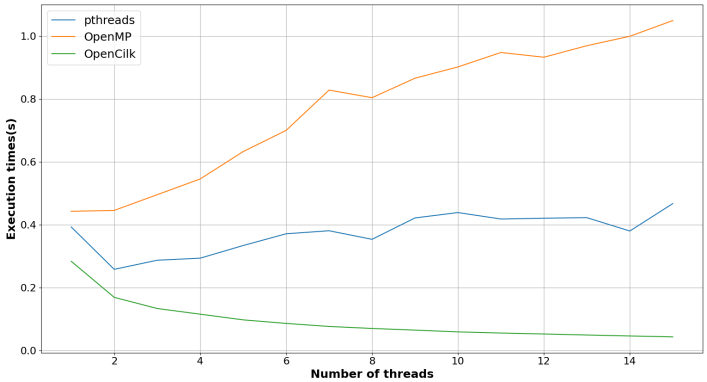

Fig. 4. Scalability in italy_osm dataset

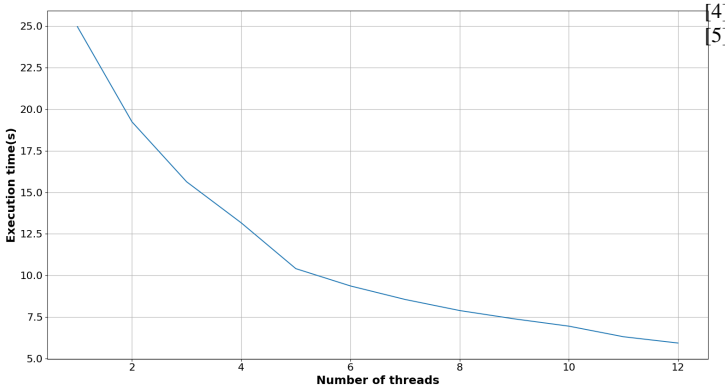

Fig. 5. Scalability in Julia's Threads with com-Youtube dataset

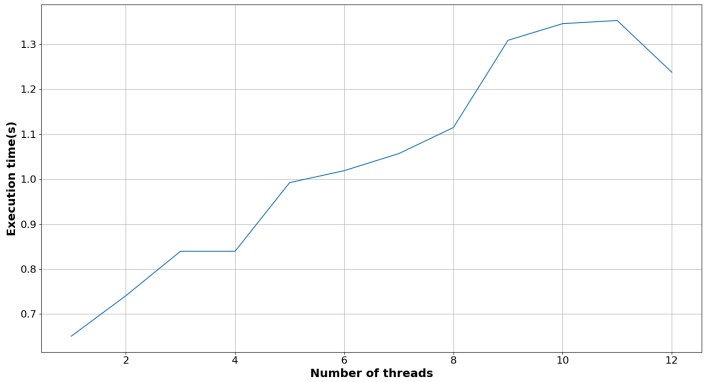

Fig. 6. Scalability in Julia's Threads with italy_osm dataset

was the parallelism of it. As far as, the parallelism the key to increase speedup were to split fairly the work to threads and keep them busy. This were implemented "manually" with thread pooling in case of C's pthreads and Julia's threads while in OpenMP and OpenCilk API similar behavior were more automated. The result's show us the importance of parallelism in slow sequential implementations and in the same time its limitations in cases where parallelim's oveheading overshadows the parallelism advantages.

ACKNOWLEDGMENT

Results presented in this work have been produced using the AUTH Compute Infrastructure and Resources

REFERENCES

[1] C. Tsourakakis, P. Drineas, P. Drineas, E. Michelakis, E. Michelakis, C. Faloutsos "Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation" April 2010 Social Network Analysis and Mining

[2] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, Sivasankaran Rajamanickam "Fast Linear Algebra-Based Triangle Counting with KokkosKernels" Center for Computing Research, Sandia National Laboratories

[3] Ariful Azad Aydın Buluc John Gilbert "Parallel Triangle Counting and Enumeration using Matrix Algebra" Lawrence Berkeley National Laboratory, University of California, Santa Barbara

[4] Fotiou Dimtrios Github https://github.com/dimfot3/Triangle_Counting

[5] Matrix Market https://math.nist.gov/MatrixMarket/