Triangle Enumeration on Massive Graphs using AWS Lambda Functions

Tengkai Yu, Venkatesh Srinivasan, and Alex Thomo
{yutengkai,srinivas,thomo}@uvic.ca

University of Victoria, BC, Canada

Abstract. Triangle enumeration is a fundamental task in graph data analysis with many applications. Recently, Park et al. proposed a distributed algorithm, PTE (Pre-partitioned Triangle Enumeration), that, unlike previous works, scales well using multiple high end machines and can handle very large real-world networks.
This work presents a serverless implementation of the PTE algorithm using the AWS Lambda platform. Our experiments take advantage of the high concurrency of the lambda instances to compete with the expensive server-based experiments of Park et al. Our analysis shows the trade-off between the time and cost of triangle enumeration and the numbers of tasks generated by the distributed algorithm. Our results reveal the importance of using a higher number of tasks in order to improve the efficiency of PTE. Such an analysis can only be performed using a large number of workers which is indeed possible using AWS Lambda but not easy to achieve using few servers as in the case of Park et al.

Keywords: triangle enumeration · massive graphs · distributed system · AWS Lambda

1 Introduction

Triangle enumeration is an essential task when analyzing large graphs. Due to the booming of graph sizes in modern data science, it becomes almost impossible to fit the entire graph into the main memory of a single machine. Although there are several methods developed to use a single machine's resources efficiently (EM-NI [1], EM-CF [2], and MGT [3]), distributed algorithms show higher scalability and speed during experiments. The PTE algorithm of Park et al. [4] has stood out among the distributed algorithms due to its minimized shuffled data size. The algorithm separates the graph into subgraphs based on edge types. A higher number of tasks leads to smaller subgraph sizes. Park et al. implemented the PTE algorithm over 41 top tier servers and showed promising experimental results. However, server-based implementation limits the maximum number of tasks.

In this paper, we implement the PTE algorithm on the AWS Lambda platform. This highly scalable, on-demand computation service allows us to use 1000 concurrent instances at the same time, thus significantly increasing the maximum number of tasks allowed. This implementation turns out to be not only cost-effective but also achieved the same speed as the original PTE implementation. The experiment results show the sweet spot for numbers of tasks when enumerating smaller size graphs, which can be the guide for choosing the number of tasks for a similar size data. More importantly, for more massive graphs like the Twitter network, our charts show how increasing the number of tasks can help to increase the speed significantly.

1.1 Our Contribution

1. We present a serverless lambda implementation of the PTE triangle algorithm on the AWS Lambda platform, thus showing the possibility of efficiently enumerating triangles on the inexpensive AWS Lambda framework. Namely, we are able to enumerate more 34 billion triangles for *twitter-2010* in less than 2 minutes and spending less $4. This is in contrast to the original paper that achieved a similar run time but using an expensive high-end, multi-server environment.
2. We present a detailed trade-off of the time and cost of enumerating triangles versus the number of tasks used. This analysis was not available in the original PTE paper. We show that the run time reduced at a quadratic rate while the money cost increased at a lesser rate. Also, we determine the sweet spot for the number of tasks after which an increase in this number is not beneficial.
3. We also present a more detailed investigation of the scalability of the PTE algorithm enabled by the large number of lambda instances we can inexpensively spawn in AWS lambda. This analysis was not available in the original PTE paper. Furthermore, our results represent a careful design in terms of language and frameworks chosen in order to be able to scale to large datasets.
4. We show the benefit of using higher number of tasks to the boosting of the enumeration speed. This is hard to evaluate in the server-based system. However, in the serverless system like the AWS Lambda platform, we always have the flexibility to use more tasks when we are not satisfied with the current enumeration speed.

2 Related work

Before the PTE algorithm, there were a couple of approaches that tried to enumerate triangles in massive graphs. Their implementations run on either a single machine or distributed clusters. However, due to the considerable memory or disk cost of this problem, these implementations failed under certain extreme conditions. On the other hand, even when they are capable of completion, these approaches were more costly complexity-wise compared to the PTE algorithm.

2.1 Distributed-Memory Triangle Algorithms

Many distributed algorithms copied parts of the graph and sent them to distributed machines so that they can achieve parallel and exact enumerations.

The PATRIC algorithm [5] separated the original set of vertices into p disjoint subsets of core nodes where p is the number of their distributed machines. Then this algorithm sent subgraphs, each one containing a set of core vertices to a distributed machine along with all the neighbors of these core vertices, and enumerated the triangles in these subgraphs at the same time. On the other hand, the PDTL algorithm [6] implemented a multi-core version of MGT. We recall that the MGT algorithm [3] is an I/O efficient algorithm using external memory. So in the PDTL algorithm, every machine received a full copy of the graph in their external memory.

The disadvantages of these two approaches are undeniable. PATRIC requires the separated graphs to fit into the main memory on every machine while the division can be redundant. Nodes with high degrees replicate in too many machines, thus challenge the total memory size. The PDTL algorithm suffers the from same problem as the MGT algorithm that the entire graph can be much larger than the capacity of external memories of every distributed machine.

2.2 Map-reduce algorithms and shuffled data

The critical difference between Map-reduce algorithms and distributed algorithms from the previous subsection is that Map-reduce uses keys to group the data when shuffling data. This helps the machines retrieve and combine data when there exists stable storage available, thus fitting the cloud platforms most. In this case, the challenging factor is no longer the original graph, but the total amount of the shuffled data.

Cohen [7] uses a pair of nodes (u, v) as the key, and the shuffled data contains all 1 step or 2 step paths between u and v [7]. As a result, every edge has a chance to be replicated $n-1$ times which leads to the $O(|E|^{\frac{3}{2}})$ of shuffled data. The Graph Partition (GP) algorithm [8] uses the nodes' hashed value triples as the key, which reduces the quantity of shuffled data to $O(|E|^{\frac{3}{2}}/\sqrt{M})$ where M is the total data space of all machines. However, every triangle is still reported more than once and needs a weight function to count the correct value. The TTP algorithm [9] sorts the edge list, so every triangle with more than one hash value is enumerated only once. The CTTP algorithm [10] requires the number of rounds $R = O(|E|^{\frac{3}{2}}/(M\sqrt{m}))$ to ensure enough space for every round process, where m is the memory size of a single reducer. However, TTP and CTTP need $O(|E|^{\frac{3}{2}}/\sqrt{M})$ of shuffled data. This drawback leads us to the PTE algorithms [4], which only generates $O(|E|)$ shuffled data and will be discussed in more detail in Section 4.

3 Background

3.1 Preprocessing

There are two reasons for us to preprocess the data before the triangle enumeration. The first one is to ensure the counting accuracy. We have to remove

Table 1. Table of Symbols

Symbol	Definition
$G = (V, E)$	A graph G containing the set of vertices V and the set of edges E
u, v	Vertices
i, j, k	Colors of vertices
(u, v)	An edge from u to v
$d(u)$	The degree of node u
$id(u)$	The vertex index of u
$\prec$	The preceding order between two nodes
ρ	The number of colors
ξ	The coloring function for vertices
$E_{i,j}$	The edge between nodes u and v, where nodes colors are (i, j) or (j, i)
M	The total memory space available of the system
m	The memory size of a single machine

self-loop over any node u since it can form a triangle (u, u, u). Also, for any triangle with nodes (u, v, w), we wish to count it exactly once. For example, given directed edges $e_1 = (u, v)$, $e_2 = (v, u)$, $e_3 = (v, w)$ and $e_4 = (u, w)$, they can form two triangles over nodes (u, v, w) with edges sets (e_1, e_3, e_4) and (e_2, e_3, e_4). This fact requires us to remove all self-loops and only keep one edge between all connected nodes. As a solution, we remove all self-loops, symmetrize all edges between connected nodes pairs, and then remove one edge from all symmetric edge pairs.

Now, for any symmetric edge pair, which edge are we supposed to remove? This question leads to our second reason: we wish to increase our enumeration speed when using set intersections. In the PTE algorithm, for every edge (u, v), we intersect the outgoing edge sets of nodes u and v and add the length of the intersection set to the counting result. If there exists a node with a very high out-degree, this node involves a high volume of intersections, and each intersection computation takes a longer time than intersecting out-neighbours of two low out-degree nodes. Please note that, although the original PTE paper for-loops the intersection and applies if conditioning on every node, which we modify in our work by using separated type-1 workers, the time cost is affected by the out-degree of all nodes in the same way. As the solution, for any two nodes u and v, we denote $u \prec v$ if $d(u) < d(v)$ or $d(u) = d(v)$ and $id(u) < id(v)$, and we only keep the edge (u, v) if $u \prec v$. This prevents the existence of nodes whose degrees are much higher than others thus reducing the enumeration time significantly.

3.2 Amazon Website Service

AWS Lambda AWS Lambda is a serverless computing platform provided by Amazon Website Services [11]. Unlike EC2 servers, which is another computing product from AWS, Lambda provides higher scalability and convenience of implementation. Scientists do not need to calculate the proper server size to rent, which they have to do when using EC2, but they only need to judge the number of function instances required by the job. Due to the on-demand nature of Lambda function instances, the Lambda platform is more convenient to achieve the most resource-efficient and economic experiment.

Each function can configure the memory space (from 128 MB to 3008 MB) and the maximum running time (3 seconds to 900 seconds) for its instances [12]. The temporary disk size for any function instance is 512 MB. However, the AWS team does not reveal the exact computation power allocated to the functions but only informs the customers that the CPU power increases linearly with the configured memory. Although the documentation shows there is one full vCPU granted when the configured memory is 1792 MB, we could find the exact CPU power for our experiment [12].

Why are the Lambda functions more economical than renting EC2 servers? The EC2 servers, by the service of AWS [13], are rented by hours. If we wish to rent 41 machines as described in the original PTE paper, we can choose the machine type *t3a.2xlarge*. This machine charges us \$0.3008 per hour, and thus 41 of these machines can cost us more than 12 dollars per hour. This cost is more than our experiments cost for four graphs and five different numbers of colors. Also, the Lambda's low cost per function makes it inexpensive for the early implementation and debugging.

In our experiments, the invoker, or called the master function, is not a Lambda function. There are two reasons for this choice. Each one of the Lambda function has a lower computation power than most of the PCs', so implementing the invoker on the Lambda platform can generate a much higher invoking time cost, and sometimes the invoking time can be the majority time cost for the entire experiment. Second, each Lambda function instance cannot live for more than 15 minutes. If there is any experiment requiring more than 15 minutes to finish, the invoker implemented using the Lambda platform is powerless when handling this task.

S3 buckets S3 (Simple Storage Service) buckets are where we store the graphs and subgraphs' enumeration results. With a valid IAM role, a Lambda function instance can search, get, and put a target file from the S3 bucket efficiently Fig 1. This feature overcomes the difficulty of emitting data on a shared disk machine. Another benefit of S3 buckets is that they can provide a massive amount of data storage space at low cost. Local machines disk can run out quickly when partitioning the graphs into many while we do not need to worry about this when using S3 bucket.

4 PTE algorithm

The key idea of the PTE (Pre-partitioned Triangle Enumeration) algorithm is to partition vertices by assigning ρ colors to vertices before the enumeration starts.

There are three versions of PTE algorithms in [4]: PTE_{BASE}, PTE_{CD} and PTE_{SC}. The PTE_{BASE} is foundation of all of them, and it is the one we focus in this paper to illustrate the benefits of using AWS Lambda over a set of high-end servers.

The algorithm first assigns these ρ colors to all vertices uniformly at random using a function ξ. Since each edge has two endpoint nodes, edges are separated into $\rho + \binom{\rho}{2} = \frac{\rho(\rho+1)}{2}$ subsets by the algorithm. This coloring process uses a simple hash function to achieve uniform distribution, and the time cost is linear in the size of the edges set. Thus, there are 3 types of triangles. Type-1 triangles have all three nodes of the same color, type-2 triangles contain 2 different colors, and type-3 triangles require all three nodes have distinct colors. The type-1 triangles are the easiest to count. A single edge subset file E_{ii} is enough to count all type-1 triangles of color i. On the other hand, all type-2 or type-3 triangles require combining 3 color subsets to enumerate. We can use the same procedure to implement the function enumerating these two types of triangles, and the total number of the function instances is $\binom{\rho}{2} + \binom{\rho}{3}$.

The PTE algorithm achieves a significant decrease in the amount of shuffled data. For any edge E_{ij} or E_{ii}, it only creates ρ copies since there are only ρ choices of the third node. This helps the PTE algorithm to stand out against other distributed algorithms.

Note that, when we enumerate type-2 triangles of color i and j, we also enumerate two kinds of type-1 triangles: type-1 triangles of color i or j. This overlapping implies that all type-1 triangles are counted $(\rho - 1)$ times by enumerating type-2 triangles. Thus the final count of triangles should be

$$\sum_{i,j\in[\rho]} enum(i.j) + \sum_{i,j,k\in[\rho]} enum(i,j,k) - (\rho - 2) * \sum_{i\in[\rho]} enum(i)$$

This formula is an additional contribution of this work. In [4], type-1 triangles are enumerated as part of type-2 triangles. This increases the workload for the type-2 workers. Our formula above allows the type-1 workers to be separated from the type-2 and 3 workers and hence reduce the total running time of the algorithm.

5 Experiments

5.1 Data collection

We use four datasets in our experiments downloaded from the Webgraph site, and they are based on the real-world internet data [14] [15]. *uk-2005* and *indochina-2004* are the internet crawl from two regions. Thus their nodes represent URLs, and edges are directed links from one URL to another. On the other hand, *ljournal-2008* and *twitter-2010* are from the social network applications, Live Journal and Twitter. Their nodes are users, while every edge indicates that one user is following another. The sizes of graphs are presented in Table II. As described previously, we pre-processed and partitioned these graphs, then uploaded them to the S3 buckets.

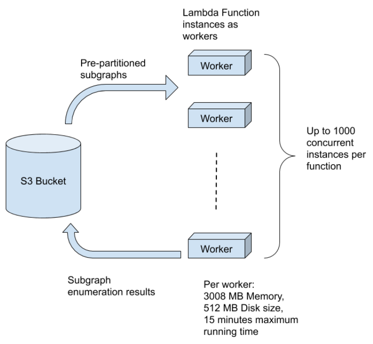
Lambda Function instances as workers
Pre-partitioned subgraphs
Worker
Worker
Worker
S3 Bucket
Up to 1000 concurrent instances per function
Subgraph enumeration results
Per worker: 3008 MB Memory, 512 MB Disk size, 15 minutes maximum running time

Fig. 1. Data Transfer between Lambda functions and the S3 bucket

Using the compression framework of Webgraph facilitated this process significantly and furthermore being able to run the algorithm on compressed partitions of the graphs in the Lambda functions made their footprint small enough to easily satisfy the AWS Lambda requirements. Notably, some other works that have successfully used Webgraph for scaling various algorithms to big graphs are [16–21].

Table 2. The summary of datasets

Dataset	**Vertices**	**Edges**	**Triangles**
ljournal-2008	5,363,260	79,023,142	411,155,444
indochina-2004	7,414,866	194,109,311	60,115,561,372
uk-2005	39,459,925	936,364,282	21,779,366,056
twitter-2010	41,652,230	1,468,365,182	34,824,916,864

5.2 Setup

We used two types of Lambda functions in our implementation. The Type-2-3 function is responsible for enumerating all the type 2 and 3 triangles. Thus its job includes getting 3 graphs from the S3 buckets, combining them, and enumerating the combination. On the other hand, the Type-1 function only gets one file from the S3 bucket and enumerates this relatively smaller subgraph. This difference between workloads leads to a significant running time difference. As we mentioned in the previous section, the original PTE paper counted the type-1 triangles as a part of the type-2 triangle, so they set up a condition to ensure all type-1 triangles are counted only once. We take advantage of the fact that type-1 enumerations cost much less time than the other two types according to Fig 2 3. Thus, we can call more tasks in parallel to reduce the total time cost. Every

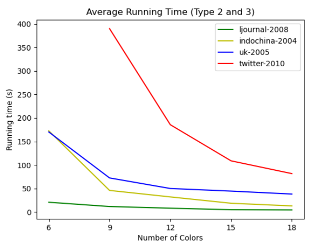

Fig. 2. Average Running Time (Type 2 and 3)

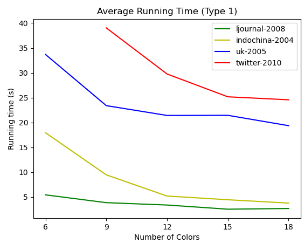

Fig. 3. Average Running Time (Type 1)

instance from either function can have 3,008 MB memory, 512 MB temporary disk size, and a maximum of 15 minutes running time. Unfortunately, we cannot provide the information about the CPU since it is hidden from customers, as we explained before. AWS Lambda allows a maximum of 1000 parallel instances from both functions.

We created a folder “graphs” inside an S3 bucket containing all graphs. All subgraphs partitioned from the same original graph belong to the same subfolder under the “graphs” folder (”graphs/twitter-2010” for example). Then this subfolder contains all sub subfolders of different numbers of color partitions. This structure makes sure that all subgraphs of a single experiment share the same prefix, and S3 buckets limit the number of requests of all files under the same prefix by 3500.

All pre-partitioned subgraphs contain three files: the .graph file, the .properties, and the .offsets. All three files' names are in the format *“graph name-i-j”* where i and j represent the color pair of edges inside this subgraph.

5.3 Experimental Results

Number of colors We started at 3 colors (1 or 2 colors are not sufficient for separating the graph) and chose steps of size 3 for increasing the colors. However, it turns out only *ljournal-2008* is enumerable with 3 colors. *indochina-2008* and *uk-2005* exceeded the Lambda function time limit (15 min) when using 3 colors. Thus our visualization starts at 6 colors. Also, *twitter-2010* is not enumerable

Algorithm 1: Triangle Enumeration on AWS Lambda

```
Data: problem = (i) or (i,j) or (i,j,k)
initialize $E'$
if problem is type $(i)$ then
    retrieve $E_{i,i}$ from the S3 bucket
    $E' = E_{i,i}$
else if problem is type $(i, j)$ then
    retrieve $E_{i,i}, E_{i,j}, E_{j,j}$ from the S3 bucket
    $E' = E_{i,i} \cap E_{i,j} \cap E_{j,j}$
else
    // problem is type $(i, j, k)$
    retrieve $E_{i,k}, E_{i,j}, E_{j,k}$ from the S3 bucket
    $E' = E_{i,k} \cap E_{i,j} \cap E_{j,k}$
end
enumerateTriangles($E'$)

Function enumerateTriangles($E$):
    result = 0
    for $(u, v) \in E$ do
        // inter: the intersection of two neighbor sets
        inter = $\{n_u|(u, n_u) \in E\} \cup \{n_v|(u, n_v) \in E\}$
        result += $len(inter)$;
    end
    return result
```

Table 3. Time cost (sec) using Intel I7 CPU and the AWS Lambda platform

Dataset	**Intel I7 CPU**	**AWS Lambda with 18 colors**
ljournal-2008	74	22
indochina-2004	2800	56
uk-2005	520	58
twitter-2010	15,456	117

with only 6 colors, so its line starts at 9 colors. Also, keep in mind that the AWS Lambda only allows 1000 instances running at the same time, and type-2 and type-3 function instances sum up to $\binom{\rho}{2} + \binom{\rho}{3}$ instances. The calculation of this formula shows that in order not to exceed the number 1000 of lambda instances we can start, the largest number of colors we can use is 18 (969 instances).

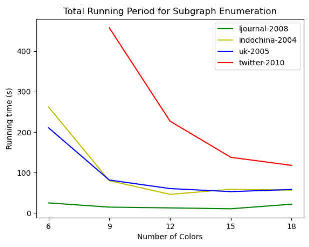

Fig. 4. Total Running Period for Subgraph Enumeration

Total running time The total running time is the time from the start to the end of enumeration. Fig 4 shows a significant decrease in running time when the number of colors increases. We make an interesting observation that, when the number of colors is 18, the time required to enumerate *twitter-2010* is only 117.865 seconds, which is very close to the Twitter enumeration time in the PTE paper. We note that these two Twitter graphs are slightly different: the *twitter-2010* dataset has 200M more edges than the Twitter graph used by the PTE paper. This fact means the we can achieve the same running time with a lower budget.

We can also locate the sweet spot quickly for every graph. All graphs except *twitter-2010* have their sweet spot at 9 colors. On the other hand, the running time on *twitter-2010* decreases well beyond 9 colors. This implies that, in an ideal implementation, a greater number of colors can achieve an even shorter time for its enumeration.

Time summation from all workers Fig 5 shows the sum of running time of all workers, which is directly related to the money charged by the AWS Lambda platform. As the number of colors grows, this sum increases quadratically. However, the rate of increase is slower than the rate at which the total running time decreases.

The most expensive experiment is enumerating *twitter-2010* with 18 colors. It contains 969 type-2-3 function instances and 18 type-1 instances. The total summation of time is 79,620 seconds in total. Given the charge rate from AWS Lambda, this experiment only costs 3.89806709729 dollars per run.

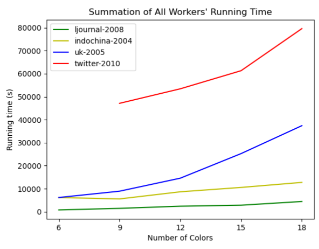

Fig. 5. Summation of All Workers Running Time

5.4 Insights

When the graph fits into the AWS Lambda platform, the speed of enumeration is very close or maybe even faster than the one from [4]. We achieve this at a much lower cost. They used 41 machines fully equipped with Intel Xeon E3-1230v3 CPUs and 32GB RAM [4]. Our result highlights the usefulness of on-demand services. A data scientist can turn to the AWS Lambda instead of purchasing or renting an expensive set of machines to either enumerate graphs fitting inside the platform, or running other algorithms that can separate the graph into different instances correctly.

Now assume scientists chose to enumerate triangles using the AWS Lambda, what is the relationship between time and money? If you want your program to run faster, you need to pay more money. This insight is straightforward from the Fig 5. As the number of colors grows, the real-world time cost decreases quadratically. However, the sum of the workers' time increases also quadratically, albeit at a lower rate. Since AWS Lambda charges mainly for the time cost of functions in this application, higher speed implies higher cost. Space in S3 Buckets, on the other hand, does not cost that much.

This leads us to the following question: Is higher number of colors always a better choice? In other words, given the 1000 concurrent instances limit, should the experiment always choose the number of colors to be 18? Fortunately, the answer is no. Remember, the total time charts contain the sweet spots for three graphs. These sweet spots imply that, for any graph with a size smaller or equal to *uk-2005*, 9 colors are enough efficiency-wise. A higher number of colors cannot increase the speed of enumeration significantly for these graphs but can cost much more. More experiments are needed to list sweet spots for different graph sizes, but this insight can help us choose the workload for this on-demand service wisely.

Is it disappointing that we failed to find the sweet spot for *twitter-2010*? On the contrary, this is our most important insight overall. Our chart shows the total time decreases even when the number of colors is 18. This fact reveals the contribution of a higher number of colors, which the original PTE paper did not consider. Park et al. only had 41 machines, and there are only 3 cores per machine used in their experiments. If they use one machine as the master, there are only 120 workers for the enumeration job, where the number 120 equals to $\binom{9}{2} + \binom{9}{3}$. This calculation implies that their concurrent number of colors cannot be higher than 9. However, since the total time cost still decreases fast with 18 colors, it motivates the use of higher number of colors in future experiments.

5.5 Python and Scipy Implementation

We also did experiments using Python and Scipy sparse matrix library [22]. Sparse matrices worked well with small graphs like Live Journal. However, the matrix multiplication operations cause inefficiency with space. The self dot product step can create positive entries in the locations which are going back to 0 during the entry-wise multiplication. Unlike set intersections, these redundant entries keep existing until the entire self dot product step finish. We also tried slice matrices to purge such memory over cost but slicing the matrices led to a significantly higher time cost.

6 Conclusions and Future Work

In this paper, we implemented the PTE algorithm using the AWS Lambda platform and ran experiments over large graphs like *twitter-2010*. The PTE algorithm partitions massive graphs efficiently and provides high scalability when enumerating triangles. This algorithm minimizes the shuffled data to increase the capability of the distributed system. We used the AWS Lambda platform to offer an economical implementation of the PTE algorithm and achieved accurate, high-speed enumeration.

Our experiments proved the strength of the AWS Lambda platform for running the distributed enumeration algorithms. None of the Lambda instances can hold the entire graph inside its space (memory or the tmp disk), but they can achieve the correct count if we properly separate the graph. High concurrency helped us to achieve scalability when running the experiments. We could locate the sweet spot for graphs shown in the previous section and choose the appropriate number of colors according to the size of any new graph. Our experiments showed that for any graphs of the size bigger than *twitter-2010*, it is helpful to use the highest number of colors possible.

What's next? In [4], the authors used 41 high-tier machines to enumerate triangles in much larger graphs like ClueWeb12, which has the size of 56G and contains more than 3 Trillion triangles. This workload certainly exceeds the AWS Lambda limit for personal usage. We are looking at ways to fit this monster-size graph into AWS Lambda with a more sophisticated algorithmic engineering, if possible.

Future work. We would like to extend distributed triangle enumeration to directed graphs. In those graphs we do not talk about triangles but "triads" (cf. [23]). There are 7 types of triads in directed graphs, and extending distributed enumeration to them might prove to be challenging.

We would also like to extend our work to distributed enumeration of four-node graphlets [24]. This is a more challenging problem but it is based on forming triangles and wedges (open triangles) first and then extending them to four-node graphlets. As such, we believe the techniques outlined in this paper could prove to be useful for enumeration of four-node graphlets as well.

One of the main applications of triangle enumeration is computing truss decomposition. In this problem we need to compute the number of triangles supporting each edge of the graph (cf. [25, 26]). Extending distributed triangle enumeration to an algorithm for distributed truss decomposition is also another avenue for our future research.

References

1. R. Dementiev, "Algorithm engineering for large data sets," Ph.D. dissertation, Verlag nicht ermittelbar, 2006.
2. B. Menegola, "An external memory algorithm for listing triangles," 2010.
3. X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, 2013, pp. 325–336.
4. H.-M. Park, S.-H. Myaeng, and U. Kang, "Pte: Enumerating trillion triangles on distributed systems," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1115–1124.
5. S. Arifuzzaman, M. Khan, and M. Marathe, "Patric: A parallel algorithm for counting triangles in massive networks," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 529–538.
6. I. Giechaskiel, G. Panagopoulos, and E. Yoneki, "Pdtl: Parallel and distributed triangle listing for massive graphs," in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 370–379.
7. J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.
8. S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 607–614.
9. H.-M. Park and C.-W. Chung, "An efficient mapreduce algorithm for counting triangles in a very large graph," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 539–548.
10. H.-M. Park, F. Silvestri, U. Kang, and R. Pagh, "Mapreduce triangle enumeration with guarantees," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, 2014, pp. 1739–1748.
11. Wikipedia contributors, "Aws lambda—wikipedia,the free encyclopedia," https://en.wikipedia.org/w/index.php?title=AWS_Lambda, 2020, [Online; accessed 10-April-2020].
12. Amazon Web Service, "Configuring functions in the aws lambda console," https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html, 2020.
13. ——, "Amazon ec2 pricing," https://aws.amazon.com/ec2/pricing/on-demand/, 2020.

14. P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.
15. P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.
16. S. Chen, R. Wei, D. Popova, and A. Thomo, "Efficient computation of importance based communities in web-scale networks using a single machine," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1553–1562.
17. F. Esfahani, V. Srinivasan, A. Thomo, and K. Wu, "Efficient computation of probabilistic core decomposition at web-scale." in *Advances in Database Technology-EDBT 2019, 22nd International Conference on Extending Database Technology*, 2019, pp. 325–336.
18. W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.
19. D. Popova, N. Ohsaka, K.-i. Kawarabayashi, and A. Thomo, "Nosingles: a space-efficient algorithm for influence maximization," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*. ACM, 2018, p. 18.
20. M. Simpson, V. Srinivasan, and A. Thomo, "Clearing contamination in large networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 6, pp. 1435–1448, June 2016.
21. ——, "Efficient computation of feedback arc set at web-scale," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 133–144, 2016.
22. P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.
23. Y. Santoso, A. Thomo, V. Srinivasan, and S. Chester, "Triad enumeration at trillion-scale using a single commodity machine," in *Advances in Database Technology-EDBT 2019, 22nd International Conference on Extending Database Technology*. OpenProceedings. org, 2019.
24. Y. Santoso, V. Srinivasan, and A. Thomo, "Efficient enumeration of four node graphlets at trillion-scale," in *Advances in Database Technology-EDBT 2020, 23rd International Conference on Extending Database Technology*, 2020, pp. 439–442.
25. F. Esfahani, J. Wu, V. Srinivasan, A. Thomo, and K. Wu, "Fast truss decomposition in large-scale probabilistic graphs." in *Advances in Database Technology-EDBT 2019, 22nd International Conference on Extending Database Technology*, 2019, pp. 722–725.
26. J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo, "K-truss decomposition of large networks on a single consumer-grade machine," in *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 2018, pp. 873–880.