Accelerating Actor-based Distributed Triangle Counting

Aniruddha Mysore*, Kaushik Ravichandran*, Youssef Elmougy, Akihiro Hayashi, Vivek Sarkar
{animysore,kravicha3,yelmougy3,ahayashi,vsarkar}@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

Triangle counting is a cornerstone operation in large graph analytics. It has been a challenging problem historically, owing to the irregular and dynamic nature of the algorithm. This inhibits compile-time optimizations and also requires runtime optimizations such as message aggregation and load-imbalance mitigation. Popular triangle counting algorithms are either inherently slow, fail to take advantage of advanced hardware in modern processors, or involve sparse matrix operations.

With its support for fine-grained asynchronous messages, the Partitioned Global Address Space (PGAS) with the Actor model has been identified to be efficient for irregular applications [3, 11, 12]. However, few triangle counting implementations have been optimally implemented on top of PGAS Actor runtimes.

To address the above-mentioned challenges, we propose a set-intersection-based implementation of a distributed triangle counting algorithm atop the PGAS actor runtime. Evaluation of our approach on the PACE Phoenix cluster and the Perlmutter supercomputer showed encouraging results.

KEYWORDS

Triangle Counting, Distributed Graph Processing, Actors, PGAS

ACM Reference Format:
Aniruddha Mysore*, Kaushik Ravichandran*, Youssef Elmougy, Akihiro Hayashi, Vivek Sarkar. 2018. Accelerating Actor-based Distributed Triangle Counting. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION & BACKGROUND

A triangle is a set of three mutually adjacent vertices in a graph. It forms the foundation of a variety of graph algorithms. Additionally, it is a popular choice for high performance computing benchmarks and computer architecture design [14]. Significant efforts have been made to optimize triangle counting in large graphs [1, 4, 16] for it to serve these purposes better. A number of approaches to triangle counting exist, each with their own merits and demerits, which we defer to recent literature that discusses the same [13].

Similar to other irregular applications run on clusters, efforts have been made to express triangle counting algorithms using fine-grained asynchronous messages as per the Partitioned Global Address Space (PGAS) model [11, 12]. In past work, we identified that the actor model could be used to provide scalability for PGAS and introduced an actor runtime that allows fine-grained asynchronous message passing with automatic message aggregation [3, 11, 12]. We also demonstrated that our actor model is well suited for vertex-centric graph programming since fine-grained asynchronous message passing goes well with data propagation between vertices via edges. Algorithm 1 is a fine-grained asynchronous message version of the triangle counting algorithm written in the SPMD fashion enabled by the previously proposed actor runtime. As there have not been many triangle-counting implementations that leverage this programming model, in this poster, we explore implementing a set-intersection-based distributed triangle counting algorithm atop the PGAS runtime.

Algorithm 1 Triangle counting using actors (SPMD, PGAS).

```
1:  function countTrianglesActor(L)
2:      Notation: Let r_p be the local rank, L̂ be the local rows of L owned
        by r_p, c be the local counter owned by r_p, and Actor_p denote the
        actor instance that is running on r_p.
3:      for {l_ij, l_ik ∈ L̂ | l_ij = l_ik = 1, k < j < i} do
4:          Let pe ← FindOwner(L_jk)        ▷ Find a rank that owns L_jk
5:          Actor_p.send(pe, j, k)          ▷ Send an active message (non-blocking)
6:      Wait()                              ▷ Wait for the completion of local send/recv
7:      return AllReduce(c)
8:  function ActorProcess(j, k) ▷ The message handler: j is row number,
    and k is col number
9:      if {l_jk ∈ L̂ | l_jk = 1} then c += 1
10: function FindOwner(row) ▷ Returns a rank that is responsible for row
11:     return row % P                      ▷ 1-D Cyclic distribution
```

2 METHODOLOGY

As our baseline, we use an actor-based distributed triangle counting algorithm (Algorithm 1 [11, 12]), where each rank (actor) iterates over the neighbors of each local vertex (vertex i) that resides on the actor and finds two different neighbors (vertices j and k) and sends a message to a (possibly) remote rank (actor) that owns j to check if there is an edge $j \rightarrow k$. Notice that, since the adjacency matrix is distributed across multiple ranks, the information on j's neighbors may not be available locally in the sender rank.

Shortcomings. We identify two shortcomings that this approach suffers from: (i) network bandwidth is underutilized as each packet sent is small in size (j and k, 16 bytes) though the runtime leverages the Conveyor library [9] for message aggregation, and (ii) opportunities to exploit data parallelism on the receiver side are missed as the receiver sequentially searches the edge $j \rightarrow k$ every time

These authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

a packet is delivered. To mitigate these bottlenecks, we explore the possibility of using a set-intersection-based implementation of triangle counting, where we (i) send the neighbor list of vertex i to its neighbors and (ii) perform a set-intersection operation on the receiver side. We observe that taking the intersection-based approach towards triangle counting improves network utilization and enables vectorization.

Table 1: R-MAT Generated Graph Parameters

Dataset	Small.mtx	Medium.mtx
Scale Parameter	N = 16	N = 20
Edges	909,917 (2^{19} to 2^{20})	15,701,646 (2^{24})
Edges per Vertex	16	16
Vertices	65,536 (216)	1,048,576 (220)

Set-intersection-based implementation. As mentioned above, this version sends the neighbor list of each vertex to its neighbors. While this approach would also have each rank (actor) send O(N^2) messages (similar to the baseline approach), we anticipate that this version would be better, as it would send larger packages, which enables higher network bandwidth utilization.

Message Passing and Communication Optimizations. The baseline message-passing structure only passes minimum information, which is not flexible to support set-intersection-based triangle counting. Instead of sending just two vertices (j, k), the set-intersection-based version sends the entire neighbor list of each vertex to a remote rank as a part of the message-passing structure. One challenging problem is that the size of the neighbor list varies by vertex, which can underutilize network bandwidth. Thus, in our current implementation, we decided to divide the neighbor list into multiple fixed-size chunks and empirically determine an optimal chunk size. Additionally, we optimize for the case where both neighbors' neighbor lists are present on the same rank, in which case only the offset bounds (start and end of a region where an intersection is computed) are sent.

Table 2: Hardware Specifications

System	PACE	Perlmutter
Processor	Intel Xeon Gold 6226	AMD EPYC 7763
Interconnect	Infiniband 100HDR	HPE Slingshot-10
PEs per Node	24	128
Vector ISA	AVX512	AVX2

Vectorization. We first experiment using a simple two-pointer approach to set intersection, which we call the scalar implementation. Set operations have been proven to be far more performant when vectorized, which also have sample reference vectorized implementations [6, 17]. All these implementations operate on 32-bit integers. However, the baseline uses 64-bit integers to cater to trillion-node-scale graphs, which we accommodate. Rather than performing a vectorized intersection on 8 × 32-bit integers in a 256-bit register, we extend the idea and approach to work with 4 × 64-bit integers in a 256-bit register. This is not trivial as vector instructions are primarily designed for 32-bit integers while there is also primitive support for 64-bit integers.

3 EVALUATION & RESULTS

Evaluation. We evaluate our optimizations against the baseline discussed in §2, on benchmark graphs constructed using the Recursive Matrix (R-MAT) generator from Graph-500 [2] with parameters of A=57 and B=C=19. We use graphs of two sizes, "small.mtx" (~50k vertices, ~1M edges) and "medium.mtx" (~1M vertices, 15M edges). The R-MAT parameters used in generating the two graphs are provided in Table 1. In the future, we also plan to evaluate our method on the datasets provided by the Stanford Network Analysis Project (SNAP) [8], and on synthetic Kronecker graphs [7]. We run the algorithms on the PACE cluster [10] at Georgia Tech and the Perlmutter supercomputer at NERSC [5, 15]. Hardware specifications of PACE and Permutter are provided in Table 2.

Results. Our scalar implementation of the set-intersection-based approach outperforms the baseline approach significantly, with up to a 10× speedup. The figures summarizing the experiments are provided in the poster. The speedup is observed on every experiment run, for both small.mtx and medium.mtx on both PACE and Perlmutter. The vectorized implementation showed significant performance improvements in some scenarios but took a hit in scenarios where there was a skew in list sizes being intersected.

4 CONCLUSIONS AND FUTURE WORK

This work serves as a first look into optimizing actor-based triangle counting on distributed memory. We propose using a set-operation based implementation, which can further be optimized by taking advantage of vector instructions, to exploit data-level parallelism.

We observe that imbalance in message sizes is a key bottleneck in the scalability of our approach. It could be improved by accomadating variable length messages to hide communication latency. This could be achieved by introducing load-imbalance mitigation or by using a conveyor library [9] to potentially unlock higher levels of performance.

Finally, our initial analysis shows that scalar and vector implementations of set-intersection operations for the purpose of triangle counting are beneficial in different scenarios. We believe that for best performance, a combination of both versions could be used.

ACKNOWLEDGEMENT

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. We are grateful to Jiawei Yang for their contribution to this work.

REFERENCES

[1] Mohammad Almasri, Neo Vasudeva, Rakesh Nagi, Jinjun Xiong, and Wen-Mei Hwu. 2021. Hykernel: A hybrid selection of one/two-phase kernels for triangle counting on gpus. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[2] James Alfred Ang, Brian W Barrett, Kyle Bruce Wheeler, and Richard C Murphy. 2010. *Introducing the graph 500.* https://www.osti.gov/biblio/1014641

[3] Youssef Elmougy, Akihiro Hayashi, and Vivek Sarkar. 2023. Highly Scalable Large-Scale Asynchronous Graph Processing using Actors. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*. 242–248. https://doi.org/10.1109/CCGridW59191.2023.00049

[4] Sayan Ghosh. 2022. Improved distributed-memory triangle counting by exploiting the graph structure. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[5] Junmin Gu, Greg Eisenhauer, Scott Klasky, Norbert Podhorszki, Ruonan Wang, and Kesheng Wu. 2022. Exploring Large All-Flash Storage System with Scientific Simulation. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management* (Copenhagen, Denmark) *(SSDBM '22)*. Association for Computing Machinery, New York, NY, USA, Article 23, 4 pages. https://doi.org/10.1145/3538712.3538734

[6] Hiroshi Inoue and Kenjiro Taura. 2015. SIMD-and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1274–1285.

[7] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research* 11, 2 (2010).

[8] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[9] F Miller Maley and Jason G DeVinney. 2019. Conveyors for streaming many-to-many communication. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 1–8.

[10] PACE. 2017. *Partnership for an Advanced Computing Environment (PACE).* http://www.pace.gatech.edu

[11] Sri Raj Paul, Akihiro Hayashi, Kun Chen, Youssef Elmougy, and Vivek Sarkar. 2023. A Fine-grained Asynchronous Bulk Synchronous parallelism model for PGAS applications. *Journal of Computational Science* 69 (2023), 102014. https://doi.org/10.1016/j.jocs.2023.102014

[12] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. 2022. A Productive and Scalable Actor-Based Programming System for PGAS Applications. 13350 (2022), 233–247. https://doi.org/10.1007/978-3-031-08751-6_17

[13] Kaushik Ravichandran, Akshara Subramaniasivam, PS Aishwarya, and NS Kumar. 2023. Fast exact triangle counting in large graphs using SIMD acceleration. In *Advances in Computers*. Vol. 128. Elsevier, 233–250.

[14] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, et al. 2017. Static graph challenge: Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[15] Charlene Yang and Jack Deslippe. 2020. Accelerate Science on Perlmutter with NERSC. *Bulletin of the American Physical Society* 65 (2020).

[16] Li Zeng, Kang Yang, Haoran Cai, Jinhua Zhou, Rongqian Zhao, and Xin Chen. 2022. HTC: Hybrid vertex-parallel and edge-parallel Triangle Counting. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[17] Jiyuan Zhang, Yi Lu, Daniele G Spampinato, and Franz Franchetti. 2020. Fesia: A fast and simd-efficient set intersection approach on modern cpus. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1465–1476.