DistTC: High Performance Distributed Triangle Counting

Loc Hoang*, Vishwesh Jatala*, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali
The University of Texas at Austin
Austin, Texas
{loc, udit, roshan, gill, pingali}@cs.utexas.edu, {vishwesh.jatala, xuhao.chen}@austin.utexas.edu

***Abstract*—We describe a novel multi-machine multi-GPU implementation of triangle counting which exploits a novel *application-agnostic* graph partitioning strategy that eliminates almost all inter-host communication during triangle counting. Experimental results show that this distributed triangle counting implementation can handle very large graphs such as clueweb12, which has almost one billion vertices and 37 billion edges, and it is up to 1.6× faster than TriCore, the 2018 Graph Challenge champion.**

***Index Terms*—triangle counting, distributed-memory, multi-GPUs, clusters, partitioning**

I. Introduction

Triangle counting is a simple example of finding *motifs* (patterns) within undirected graphs, and it is used in applications such as social network analysis [1], graph statistics (e.g. clustering coefficients [2]), and k-truss identification [3]. The problem is to count the number of triangles contained in an undirected graph[1].

Triangle counting algorithms are based on the following observation. Vertices v and w are said to be *neighbors* if they are connected by an edge. Let $neighbors(v)$ denote the set of neighbors of a given vertex v. The number of triangles that contain given vertices v_1 and v_2 is the cardinality of the set $neighbors(v_1) \cap neighbors(v_2)$. This can be computed efficiently if the graph representation permits efficient access to the set of neighbors of a given vertex as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) representations do. Two vertices that are not neighbors in the graph cannot be part of a triangle, so triangle counting is often implemented by iterating over all edges (v_1, v_2) of the graph and, for each edge, intersecting the neighbor lists of v_1 and v_2. The intersection of neighbor lists can be done efficiently if the neighbor list of each vertex is sorted by a key such as the vertex ID.

Although this algorithm is straightforward, implementing it efficiently can be challenging. First, the graphs we deal with today are very large; for example, the largest graph we use in this study is clueweb12, which has almost one billion vertices and 37 billion undirected edges (and 1,995,295,290,765 triangles). Second, the basic triangle counting algorithm described above is not a *vertex program*, which are programs in which vertex labels are updated iteratively using the labels of their neighbors in the graph (it is possible to reformulate the algorithm as a vertex program, but this is tricky and has its own disadvantages). Third, the DRAM of most machines is limited to 64-128 GB. This means execution on shared memory systems may not be possible, so alternatives such as distributed execution or out-of-core execution are necessary. However, systems support for non-vertex-programs is currently very limited. In particular, existing graph partitioning algorithms are designed for vertex programs, and it is not clear how one uses them effectively for triangle counting.

These problems limit the effectiveness of current implementations of triangle counting. Most implementations are either sequential or shared-memory multicore implementations [4], [5] so they cannot deal with very large graphs unless one uses a very expensive machine with many TBs of DRAM or non-volatile memory (NVM) such as Intel's Optane memory. Other implementations are single GPU implementations [5], [6], but these are even more constrained by memory limitations. Section II describes this related work in more detail.

In this paper, we address the triangle counting problem using distributed computing. Our implementation is based on a novel *application-agnostic* graph partitioning strategy, discussed in Section III, that eliminates almost all communication for distributed triangle counting. Section IV describes our triangle counting application which uses this partitioning strategy. Each machine in a distributed cluster performs triangle counting independently in its own partition without communicating with other hosts. At the end, the total triangle count is obtained by aggregating the local triangle counts from all machines. This triangle counting solution can be run in the cloud, which is very cost-effective.

Section V describes the experimental evaluation of our implementation called *DistTC* on a distributed multi-GPU platform. The results show that our solution can handle very large graphs such as clueweb12, which has almost one billion vertices and 37 billion undirected edges, and it is up to $1.6\times$ faster than TriCore, the 2018 Graph Challenge champion.

*Both authors contributed equally.

[1]A set of vertices $\{v_1, v_2, v_3\}$ is said to be form a *triangle* if there is an edge in the graph between each pair of these vertices.

II. Related Work

Triangle counting has been implemented on various platforms including shared-memory CPUs [7], clusters [8], [9], [10], [11] and GPUs [12], [13]. We briefly discuss this prior work below.

Triangle counting on shared memory CPUs: Shun *et al.* [14] detail a cache-oblivious parallel triangle counting on shared-memory multicore CPUs. Tangwongsan *et al.* [7] present parallel algorithms for streaming graphs on the shared memory CPUs. Zhang *et al.* [15] compare the performance of different triangle counting optimizations, such as hashing, merging, binary search, and SIMD.

Optimizations that apply to CPUs may not necessarily be useful for GPUs. For example, binary search of neighbor lists may not give faster performance on CPUs, but it does give better performance on GPUs due to coalesced memory accesses [15], [16].

Triangle counting on single GPU: Green *et al.* [17] and Voegele *et al.* [5] implement triangle counting on a single GPU using a merge-based approach [18]. Date *et al.* [19] present a GPU algorithm that leverages the CPU as well to improve the performance of both triangle counting and k-truss: they use GPU zero-copy memory and unified memory capabilities to decrease CPU-GPU data transfer overhead, and they use the CPU to perform some computations. Wang *et al.* [20] study three techniques for triangle counting on GPU: subgraph matching, programmable graph analytics with a set-intersection approach, and a matrix formulation based on sparse matrix-matrix multiplies. Hu *et al.* [16] presented a distributed implementation of triangle counting, and they noted that even on a single GPU, a binary search based intersection method for finding triangles can speed up computation on GPUs due to improved exploitation of memory bandwidth.

Our implementation leverages the binary search based intersection method [16] to improve performance on a single host which in turn improves overall distributed runtime. We also use the CPUs to do graph partitioning and graph preprocessing.

Triangle counting on distributed CPUs and GPUs: Suri *et al.* [21] implement triangle counting using MapReduce. They rank vertices by degree and distribute them across hosts in a round-robin fashion. Similarly, several other techniques [22], [23], [24], [25] were also proposed to improve the performance of triangle counting using MapReduce framework. PDTL [26] is a distributed CPU triangle counting solver that duplicates the graph across machines and does static load-balancing to split tasks. Pearce [27] presents a CPU distributed triangle counting algorithm implemented in distributed asynchronous graph processing framework HavoqGT. TriCore [28] is a multi-GPU triangle counting implementation which was used in a 2018 Graph Challenge champion's implementation [29]. It partitions the CSR data across multiple GPUs and streams the edgelist from the CPU memory to the GPU memory on the fly. It uses binary search to increase coalesced memory accesses [16] and employs load balancing by dynamically assigning independent units of work (created during preprocessing) to GPUs.

Our implementation is similar to TriCore [28], [29] in that it uses binary search based intersection as well as independent units of work across multiple GPUs. However, our partitioning is static while TriCore dynamically assigns partitions to GPUs on the fly. Additionally, our partitioning policy can be used for multi-machine CPU and GPU implementations.

Miscellaneous: Huang *et al.* [30] implement triangle counting on an FPGA: they use the low-latency capabilities of FPGA to improve energy efficiency over GPU implementations. Approximation techniques for triangle counting [24], [31] have also been studied. Additionally, triangle counting has been explored for streaming graphs (e.g., [32], [33], [34]).

Our implementation is targeted at static graphs since partitioning is done once at the beginning. However, our distributed algorithm is platform-agnostic as described in Section IV: one can use our partitioning policy to distribute computation across multiple platforms and aggregate the independent counts at the end to arrive at a correct solution, regardless of whether the compute units are CPUs, GPUs or FPGAs.

III. Graph Partitioning Policy

The goal of our partitioning is to distribute an undirected graph across multiple machines for triangle counting.

A. Proxy Model for Partitioning

Our graph partitioning is based on the proxy model of partitioning [35] in which edges are distributed among the host machines and cached copies of the endpoints called proxy vertices are created. One proxy for a vertex in the graph will be designated a master proxy which is responsible for the canonical value of the vertex; all other proxies are designated as mirror proxies.

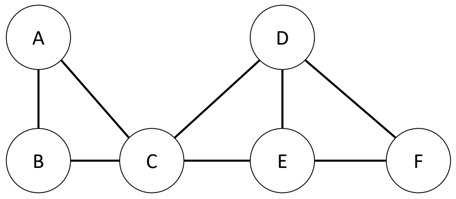
Periodically, the proxy vertices are synchronized to keep computation consistent across hosts. This involves mirror proxies communicating updates to the master proxy which will use them to determine the vertex's canonical value. This value is then communicated to all proxies for use in local computation.

This model can be extended further to support *proxy edges*: the partitioning can duplicate an edge among multiple hosts, and similarly, one edge will be designated the master proxy to reconcile edge data among hosts as necessary.

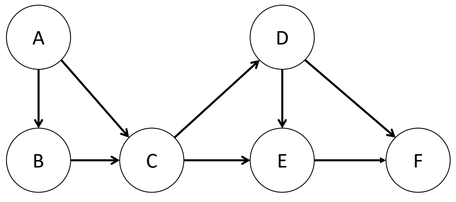
B. Partitioning for Triangle Counting

As detailed in Section I, triangle counting requires knowing the neighbors of vertices in order to compute the intersection of the neighbor lists. In a distributed setting, edges are distributed across machines, and not all the edges of a triangle may exist for a vertex on a host. Therefore, communication across hosts to determine the full set of neighbors would be required if not all edges are present.

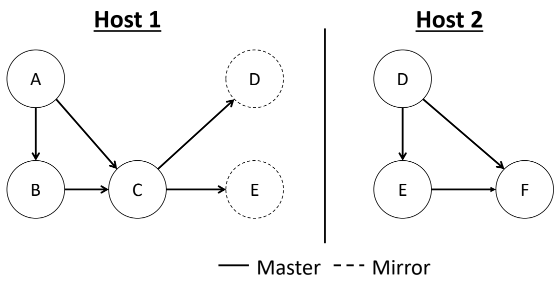
To reduce communication, we partition the graph such that all proxy vertices on a particular host will know which vertices on that host are its neighbors; in other words, *if two vertex*



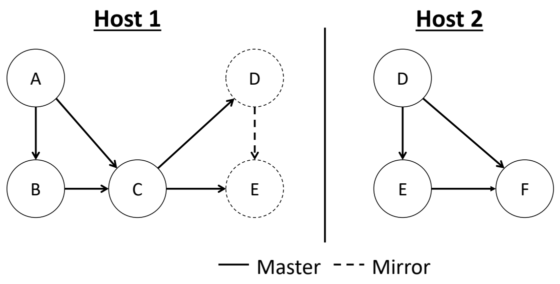
(a) Undirected input.



(b) Directed edges based on total order (alphabetical).



(c) Partitioned graph using outgoing edge-cut.



(d) Creation of edge proxies to obtain induced subgraph from vertex proxies.

Fig. 1: Steps of partitioning for triangle counting.

proxies on a host are neighbors in the original graph, then an edge proxy will exist between the two proxies on that host. By doing so, it is possible to count the triangles that the vertex proxies present on a host create without requiring communication.

We detail this partitioning with an example. First, to avoid counting duplicate triangles, triangle counting algorithms typically establish a total order among the vertices of a given graph. Hence, we convert the given undirected graph (Figure 1a) into a directed acyclic graph (DAG), where the undirected edges become directed edges based on some total ordering of vertices (Figure 1b; the ordering is based on alphabetical order). In this example, the vertex lower in the total order will point to a vertex higher in the total order.

Next, we partition the directed graph across machines. Figure 1c shows an example of partitioned graphs of the original graph shown in Figure 1b. We use an *outgoing edge-cut* policy to distribute edges: all outgoing edges of a vertex are placed on the host which contains the master proxy. In the example, vertices A, B, and C, along with their outgoing edges, are assigned to Host 1 while the vertices D, E, and F, along with their outgoing edges, are assigned to Host 2. For each edge whose endpoints are not assigned to a host, mirror proxies are created. For example, Host 1 creates the mirror proxies for D and E, as they are endpoints for the edges (C, D) and (C, E) which are not present.

Finally, edge proxies are created as necessary to get the proxy-induced subgraph on each host. Figure 1d illustrates this: since the edge (D, E) exists in the original graph, a mirror proxy for the edge is created on Host 1.

The creation of edge proxies allows local triangle counting to proceed without communication. To understand this, consider Figure 1c, which shows the partitioned graph before edge proxies are created. Note that communication is required to count a triangle whose endpoint(s) is a mirror proxy; for instance, triangle CDE cannot be identified on Host 1 without communication with Host 2 to determine if edge (D, E) exists. The creation of the proxy-induced subgraph avoids this problem and allows Host 1 to find the triangle without communication.

C. Discussion

The partitioning strategy described in this section is application-agnostic and is implemented in the Customizable Streaming Partitioner (CuSP) framework [36]. The experimental results in this paper are based on the partitions created by CuSP.

IV. Distributed Triangle Counting

Figure 2 details the overall execution of our distributed triangle counting implementation. The graph is read, partitioned, and processed to sort the neighbor lists based on vertex IDs. The partitioned graph is then marshaled to the GPU for GPU processing. Finally, distributed triangle counting in done in two steps.

1) Each host counts the number of triangles locally.
2) The local counts are aggregated at the end of local computation to get the final triangle count.

Note that during the computation, communication is needed only for the aggregation at the very end.

A. Local Triangle Counting

To count triangles locally on each host, we modify an IrGL-based triangle counting implementation [5], [37] to use binary search to find triangles instead of edgelist intersection to improve GPU locality [28], [29]. As the graph is already preprocessed before it is marshaled to the GPU, there is no need to do graph preprocessing on the GPU: it is only responsible for counting triangles.

Local triangle counting on each host must take into account the fact that the graph is distributed across multiple hosts. To avoid overcounting triangles, a triangle ABC, $A < B < C$ is only counted if the vertex A is a master proxy.

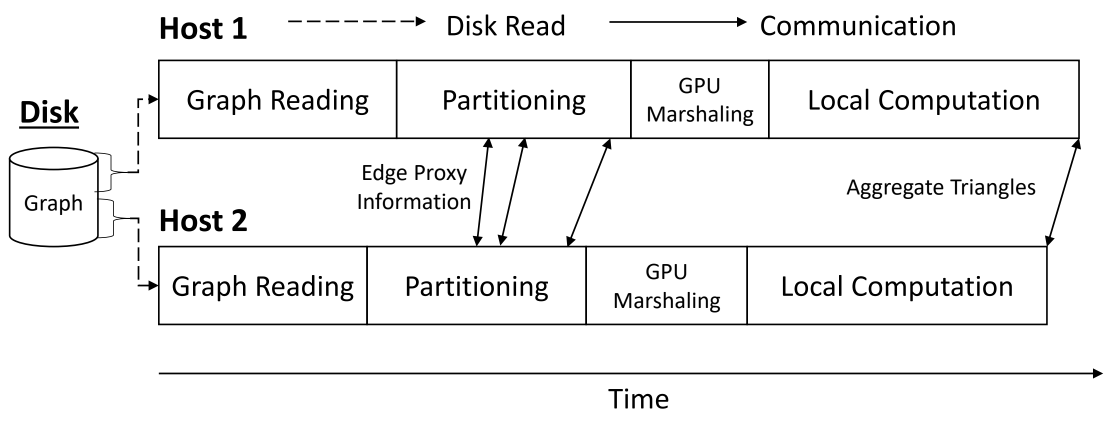


Fig. 2: Execution flow for distributed triangle counting.

B. Correctness of Distributed Triangle Counting

We briefly discuss the correctness of our distributed triangle counting. Let ABC be a triangle such that $A < B < C$. For correctness, it (1) must be counted if it exists and (2) must be counted exactly once. We describe three observations from the partitioning described in Section III that are useful in reasoning about the correctness of the distributed triangle counting algorithm.

Observation 1. *If undirected edges $\{A, B\}$ and $\{A, C\}$ exist, then directed edges (A, B) and (A, C) will be present on the same host that contains the master proxy of A. This holds because the host that has the master proxy has all its outgoing edges. This implies proxies for B and C exist on that host as well.*

Observation 2. *If proxies B and C exist on some host H and undirected edge $\{B, C\}$ exists, then an edge proxy for directed edge (B, C) will exist on H.*

Observation 3. *A triangle ABC can exist on more than one host since proxies for A, B, and C may exist on more than one host.*

Let Host H_a contain the master proxy for A. From Observation 1, H_a contains edges (A, B) and (A, C), and H_a contains the proxies for B and C. Following Observation 2, the edge (B, C) exists on H_a, which means the triangle is counted by H_a since all three edges exist and A is a master proxy. Therefore, if the triangle exists, it is counted by our approach. Secondly, it will be counted only once: even if triangle ABC exists on any other host H_b (Observation 3), it not counted by the host since A's master proxy only exists on H_a. Thus, if a triangle exists, it is counted exactly once.

C. Platform-Agnosticism

The approach described here can be used with any local triangle counting algorithm, provided it counts a triangle ABC only if A is a master proxy. In addition, the platform is not a factor: CPUs, GPUs, and/or FPGAs can all be used as long as they are able to aggregate triangle counts across all platforms at the end of the local computation. In general, the best local triangle counting algorithm will be platform-dependent as explained in Section II.

V. Evaluation

A. Experimental Setup

We implemented the proposed graph partitioning policy using the Customizable Streaming Partitioner (CuSP) [36] framework. We modified CuSP to create edge proxies in addition to vertex proxies. We also modified it to treat undirected edges as a single directed edge based on a total ordering heuristic specified by the user. In this paper, we establish the order among the vertices based on their degree: edges point towards the vertex with a higher degree, and ties are broken so the edge points to the vertex with the larger vertex ID [5].

The machine used for our evaluation is the Bridges cluster [38], [39] at the Pittsburgh Supercomputing Center [40]. We used up to 64 NVIDIA Tesla P100 GPUs located on 32 distributed machines with 128GB DRAM each. Each P100 GPU has 16GB of memory. Each node in the cluster has 2 Intel Broadwell E5-2683 v4 CPUs with 16 cores per CPU. The nodes in the cluster are connected through the Intel Omni-Path Architecture.

We evaluated our DistTC application on the Graph500 inputs provided by the Graph Challenge; the graph properties are shown in Table I. These inputs are relatively small in size, so we used a single GPU for them. In addition, we used the larger input graphs shown in Table II on up to 64 GPUs (32 nodes on Bridges). rmat26 is a synthetic graph generated by an RMAT generator [41]; twitter40 [42], friendster [43] uk-2007, gsh-2015, and clueweb12 [44], [45], [46], [47] are web crawls. Input graphs are symmetric, have no self-loops, and have no duplicated edges. We represent the input graphs in memory in a compressed sparse row (CSR) format. The tables report the number of *undirected edges* after dropping duplicated edges and self-loops; each undirected edge $\{a, b\}$ is represented by two directed edges (a, b) and (b, a). The input graph is partitioned across machines using the strategy in Section III. We also sorted the neighbor lists of source and destination vertices of each edge during this preprocessing

	Vertices	Undirected Edges	# triangles	IrGL (sec)	DistTC (sec)
graph500-scale18-ef16	174,147	3,800,348	82,287,285	0.06	0.06
graph500-scale19-ef16	335,318	7,729,675	186,288,972	0.18	0.15
graph500-scale20-ef16	645,820	15,680,861	419,349,784	0.56	0.35
graph500-scale21-ef16	1,243,072	31,731,650	935,100,883	1.58	0.80
graph500-scale22-ef16	2,393,285	64,097,004	2,067,392,370	4.41	1.91

TABLE I: Graph properties and performance comparison (in seconds) of Graph500 datasets on a single GPU.

	Vertices	Undirected Edges	# triangles
rmat26	67,108,864	1,065,788,093	43,646,321,254
twitter40	41,652,230	1,202,513,046	34,824,916,864
friendster	65,608,366	1,806,067,135	4,173,724,142
uk2007	105,896,435	3,301,876,564	286,701,284,103
gsh-2015	988,490,691	25,690,705,118	910,140,734,636
clueweb12	978,407,686	37,372,179,311	1,995,295,290,765

TABLE II: Large graphs and their properties

	# GPUs	TriCore	DistTC
twitter40	8	6.5	6.7
friendster	8	2.1	4.0
gsh-2015	32	253.4	159.9

TABLE IV: Performance comparison (in seconds) with TriCore [29]: friendster and twitter40 compares compute time; gsh-2015 compares end-to-end time.

Graph	**GPUs**	**Pre-Processing**	**Exec. Time**	**Total Time**
rmat26	16	30.16 (29.15)	5.95	36.11
	32	26.92 (25.99)	3.63	30.55
	64	23.74 (22.89)	2.62	26.36
twitter40	16	24.90 (24.20)	3.92	28.82
	32	20.81 (20.20)	2.83	23.64
	64	18.73 (18.19)	2.35	21.08
friendster	16	52.13 (51.32)	2.49	54.62
	32	41.80 (41.19)	1.64	43.44
	64	36.13 (35.64)	1.50	37.63
uk2007	16	12.16 (11.63)	8.64	20.80
	32	12.06 (11.66)	6.52	18.58
	64	11.41 (11.05)	5.47	16.88
gsh-2015	16	-	-	-
	32	143.43 (142.30)	16.44	159.87
	64	143.72 (142.97)	15.25	158.97
clueweb12	16	-	-	-
	32	-	-	-
	64	162.92 (162.61)	9.49	172.41

TABLE III: Performance (in seconds) for large graphs on multiple GPUs (Partitioning time in parentheses).

step. We compiled our implementation using gcc 6.3.0 and cuda 9.0, and we report the results for a mean of three runs.

B. Experimental Results

In this section, we analyze the performance of distributed triangle counting and compare it with other third-party implementations.

DistTC performance on single GPU: Table I shows the performance of DistTC on the Graph500 inputs on a single GPU. It also shows the runtime of IrGL-generated single-GPU triangle counting code [5], [37] (denoted *IrGL*) that was a Graph Challenge 2017 champion. On average, *DistTC* achieves 1.54$\times$ speedup over *IrGL*. On a single GPU, we do not partition the graph, so the difference in performance is due to the computation phase on the GPU. IrGL uses merge-based intersection for computing the number of triangles on an edge, whereas *DistTC* employs binary search [29]. Binary search has better locality and improved coalesced memory accesses [28], [29], so we observe an increase in performance even on a single GPU.

DistTC performance on multi-machine multi-GPUs: Table III shows the performance on multiple GPUs on multiple hosts using the large graphs shown in Table II. We used 16, 32, and 64 GPUs. In the table, we report the following metrics: (1) preprocessing time, which is the time taken by CuSP to load the graph from disk, partition it, add directions to edges, sort edges, construct the in-memory representation of each partition, and marshal the graph from the CPU to the GPU (in parentheses, we show the graph partitioning time, i.e., preprocessing time excluding the time to marshal the data from the CPU to the GPU), (2) execution time, which is the time taken to compute the number of triangles on the GPUs and aggregate the final result, and (3) total time, which is end-to-end time taken to execute the application.

Most of the preprocessing time is taken up in reading the graph from disk and partitioning it. With the increase in the number of GPUs, the time taken compute the number of triangles decreases since our distributed triangle counting algorithm is free from the synchronization except for the final aggregation. We also observe some scaling for the total time since increasing the number of hosts allows graphs to be read from disk faster and reduces the partitioning and marshalling time.

We also compare the performance of our approach with that of TriCore [28], [29], which is a multi-GPU implementation of triangle counting and a Graph Challenge 2018 champion, in Table IV. As their source code is not available, we use the times reported in their paper which were collected on Tesla P100 GPUs. For friendster and twitter40, we compare execution time, while for gsh-2015, we compare the total end-to-end execution time; these are the only times reported in their paper. For gsh-2015, the number of triangles reported by the TriCore paper seems incorrect since the number in their paper does not match the number reported by other runs (we ran a shared-memory Galois version of triangle counting on a multicore machine with Intel Optane non-volatile memory and obtained the same number of triangles as DistTC does) and the number of edges reported in their paper does not match the gsh-2015 specification [48].

For the small graphs with low runtime, we are competitive with TriCore's execution time. For the larger gsh-2015, our implementation is faster than TriCore's (end-to-end time) by

1.58×.

In summary, the proxy-edge based partitioning policy and the multi-machine, multi-GPU triangle counting implementation perform well as graphs get larger in size due to the elimination of communication except for the aggregation of independent counts across all hosts at the end of the computation.

VI. Conclusion

This paper describes a high-performance distributed GPU implementation of triangle counting using a new partitioning scheme that duplicates edges to avoid communication during triangle counting. Evaluation of triangle counting with this new policy shows execution time scaling and good performance as graphs grow in size due to the removal of communication overhead during triangle counting computation.

Acknowledgment

This research was supported by the NSF grants 1406355, 1618425, 1705092, 1725322, and by the DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. We used XSEDE grant ACI-1548562 through allocation TG-CIE170005. We used the Bridges cluster, supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center.

References

[1] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*, ser. Structural Analysis in the Social Sciences. Cambridge University Press, 1994.

[2] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, Jun. 1998. [Online]. Available: http://dx.doi.org/10.1038/30918

[3] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–6.

[4] E. Donato, M. Ouyang, and C. Peguero-Isalguez, "Triangle Counting with A Multi-Core Computer," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[5] C. Voegele, Y. Lu, S. Pai, and K. Pingali, "Parallel triangle counting and k-truss identification using graph-centric methods," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–7.

[6] M. Bisson and M. Fatica, "Update on Static Graph Challenge on GPU," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–8.

[7] K. Tangwongsan, A. Pavan, and S. Tirthapura, "Parallel Triangle Counting in Massive Streaming Graphs," in *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 781–786. [Online]. Available: http://doi.acm.org/10.1145/2505515.2505741

[8] 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*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 529–538. [Online]. Available: http://doi.acm.org/10.1145/2505515.2505545

[9] S. Arifuzzaman, M. Khan, and M. Marathe, "A fast parallel algorithm for counting triangles in graphs using dynamic load balancing," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 1839–1847.

[10] 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*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 539–548. [Online]. Available: http://doi.acm.org/10.1145/2505515.2505563

[11] W. Wang, Y. Gu, Z. Wang, and G. Yu, "Parallel Triangle Counting over Large Graphs," in *Database Systems for Advanced Applications*, W. Meng, L. Feng, S. Bressan, W. Winiwarter, and W. Song, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 301–308.

[12] A. Polak, "Counting Triangles in Large Graphs on GPU," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 740–746.

[13] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A Comparative Study on Exact Triangle Counting Algorithms on the GPU," in *Proceedings of the ACM Workshop on High Performance Graph Processing*, ser. HPGP '16. New York, NY, USA: ACM, 2016, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2915516.2915521

[14] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *2015 IEEE 31st International Conference on Data Engineering*, April 2015, pp. 149–160.

[15] J. Zhang, D. G. Spampinato, S. McMillan, and F. Franchetti, "Preliminary Exploration on Large-Scale Triangle Counting in Shared-Memory Multicore System," in *IEEE High Performance Extreme Computing (HPEC)*, 2018.

[16] Y. Hu, P. Kumar, G. Swope, and H. H. Huang, "TriX: Triangle counting at extreme scale," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–7.

[17] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast Triangle Counting on the GPU," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IA3.2014.7

[18] O. Green, R. McColl, and D. A. Bader, "GPU Merge Path: A GPU Merging Algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 331–340. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304621

[19] K. Date, K. Feng, R. Nagi, J. Xiong, N. S. Kim, and W. Hwu, "Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition on the Minsky architecture: Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–7.

[20] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A Comparative Study on Exact Triangle Counting Algorithms on the GPU," in *Proceedings of the ACM Workshop on High Performance Graph Processing*, ser. HPGP '16. New York, NY, USA: ACM, 2016, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2915516.2915521

[21] 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*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 607–614. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963491

[22] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science Engineering*, vol. 11, no. 4, pp. 29–41, July 2009.

[23] T. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, "Counting Triangles in Massive Graphs with MapReduce," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. S48–S77, 2014. [Online]. Available: https://doi.org/10.1137/13090729X

[24] R. Pagh and C. E. Tsourakakis, "Colorful Triangle Counting and a MapReduce Implementation," *Inf. Process. Lett.*, vol. 112, no. 7, pp. 277–281, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.ipl.2011.12.007

[25] 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*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 539–548. [Online]. Available: http://doi.acm.org/10.1145/2505515.2505563

[26] I. Giechaskiel, G. Panagopoulos, and E. Yoneki, "PDTL: Parallel and Distributed Triangle Listing for Massive Graphs," in *2015 44th International Conference on Parallel Processing*, Sep. 2015, pp. 370–379.

[27] R. Pearce, "Triangle counting for scale-free graphs at scale in distributed memory," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–4.

[28] Y. Hu, H. Liu, and H. H. Huang, "TriCore: Parallel Triangle Counting on GPUs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2018, pp. 171–182.

[29] Y. Hu, H. Liu, and H. H. Huang, "High-Performance Triangle Counting on GPUs," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–5.

[30] S. Huang, M. El-Hadedy, C. Hao, Q. Li, V. S. Mailthody, K. Date, J. Xiong, D. Chen, R. Nagi, and W. Hwu, "Triangle Counting and Truss Decomposition using FPGA," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[31] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "DOULION: Counting Triangles in Massive Graphs with a Coin," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 837–846. [Online]. Available: http://doi.acm.org/10.1145/1557019.1557111

[32] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 623–632. [Online]. Available: http://dl.acm.org/citation.cfm?id=545381.545464

[33] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting Triangles in Data Streams," in *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '06. New York, NY, USA: ACM, 2006, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1142351.1142388

[34] M. Jha, C. Seshadhri, and A. Pinar, "A Space-Efficient Streaming Algorithm for Estimating Transitivity and Triangle Counts Using the Birthday Paradox," *ACM Trans. Knowl. Discov. Data*, vol. 9, no. 3, pp. 15:1–15:21, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2700395

[35] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '18. New York, NY, USA: ACM, 2018, pp. 752–768. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192404

[36] L. Hoang, R. Dathathri, G. Gill, and K. Pingali, "CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics," in *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2019, 2019.

[37] S. Pai and K. Pingali, "A Compiler for Throughput Optimization of Graph Algorithms on GPUs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984015

[38] N. A. Nystrom, M. J. Levine, R. Z. Roskies, and J. R. Scott, "Bridges: A Uniquely Flexible HPC Resource for New Communities and Data Analytics," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: ACM, 2015, pp. 30:1–30:8. [Online]. Available: http://doi.acm.org/10.1145/2792745.2792775

[39] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "XSEDE: Accelerating Scientific Discovery," *Computing in Science and Engineering*, vol. 16, no. 5, pp. 62–74, Sept-Oct 2014.

[40] "Pittsburgh Supercomputing Center," 2019. [Online]. Available: https://www.psc.edu/

[41] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, pp. 442–446. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43

[42] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772751

[43] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[44] T. L. Project, "The ClueWeb12 Dataset," 2013. [Online]. Available: http://lemurproject.org/clueweb12/

[45] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602. [Online]. Available: http://doi.acm.org/10.1145/988672.988752

[46] 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*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 587–596. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963488

[47] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive Crawling for the Masses," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. New York, NY, USA: ACM, 2014, pp. 227–228. [Online]. Available: http://doi.acm.org/10.1145/2567948.2577304

[48] "gsh-2015," 2019. [Online]. Available: http://law.di.unimi.it/webdata/gsh-2015/