SAND2019-8241C

Scalable Triangle Counting on Distributed-Memory Systems

Seher Acer*, Abdurrahman Yaşar†, Sivasankaran Rajamanickam*, Michael Wolf *, and Ümit V. Çatalyürek†
sacer@sandia.gov, ayasar@gatech.edu, srajama@sandia.gov, mmwolf@sandia.gov, and umit@gatech.edu
*Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, U.S.A.
†College of Computing Georgia Institute of Technology, Atlanta, GA, U.S.A.

***Abstract*—Triangle counting is a foundational graph-analysis kernel in network science. It has also been one of the challenge problems for the "Static Graph Challenge". In this work, we propose a novel, hybrid, parallel triangle counting algorithm based on its linear algebra formulation. Our framework uses MPI and Cilk for exploiting the benefits of distributed-memory and shared-memory parallelism. The problem is partitioned among MPI processes using a two-dimensional (2D) cartesian block partitioning. One-dimensional (1D) rowwise partitioning is used within the cartesian blocks for shared-memory parallelism using the Cilk programming model. Besides exhibiting very good strong scaling behavior in almost all tested graphs, our algorithm achieves the fastest time on the 1.4B edge real-world twitter graph, which is 3.217 seconds, on 1,092 cores. In comparison to past distributed-memory parallel winners of the graph challenge, we demonstrate speed up of 2.7 on this twitter graph. This is also the fastest time reported for parallel triangle counting on the twitter graph when the graph is not replicated.**

***Index Terms*—triangle counting, distributed-memory systems, scale-free graphs, two-dimensional partitioning**

I. Introduction

Given an undirected graph G, the triangle counting problem is defined as finding the number of triangles, i.e., the edge triplets $\langle (i,j),(j,k),(k,i)\rangle$ in G, where $i \neq j \neq k$. The triangle counting problem is a canonical graph-analysis problem with applications in computing k-truss decomposition, subgraph isomorphism and malware detection. It is one of the IEEE HPEC Graph Challenge [1] problems and its popularity has been growing in the last few years. Past work in 2017 and 2018 Graph Challenge improved the running times for triangle counting on various architectures and programming paradigms including shared-memory parallelism on CPUs [2]–[5], GPUs [6]–[8] and distributed memory architectures [9]. Despite these improvements, times for some real world graphs such as twitter are still high, especially on distributed-memory architectures. The best reported time for twitter is 8.52 seconds on a distributed-memory system, using 256x24=6,144 cores [9]. The best reported time for twitter on shared-memory architectures is 28.36 seconds using Cilk [3] and 6.5 seconds on eight GPUs with the graph replicated and without including the copy times. This paper focuses on a scalable distributed-memory algorithm to achieve the best times on real-world graphs on distributed-memory architectures.

Traditionally, parallel algorithms for triangle counting have used both graph based and linear-algebra based approaches. Linear-algebra based approaches are becoming more popular due to the improved performance and ability to reuse linear-algebra based kernels. We consider the linear-algebra based formulation $(L \times L).*L$ of the triangle counting problem. In this formulation, L denotes the lower triangular portion of the adjacency matrix A of the given graph G. As pointed out by past work, $(L \times L).*L$ counts the triangles only once and is usually faster than other formulations in practice [2], [3].

We propose a hybrid parallelization for the $(L \times L).*L$ formulation that exploits both distributed- and shared-memory parallelism benefits. We use MPI for inter-process communication. Matrix L is distributed among MPI processes using the 2D block cartesian partitioning scheme [10], which is commonly used for obtaining upper bounds on communication overheads in different computational kernels [11]–[14]. For P MPI processes, the number of processes with which each process communicates is $O(\sqrt{P})$. Within each MPI process, we use Cilk [15], which is a work-stealing, multithreaded runtime, for shared-memory parallelism. We chose Cilk as it has shown to be more efficient than OpenMP in the triangle counting context [3].

Our work differs substantially from the 2017 distributed-memory triangle counting champion [9]. While they leveraged a vertex-centric HavoqGT for their distributed memory coordination, we use MPI-based message passing. Unlike [9], we also use hybrid parallelism, using a MPI+Cilk programming model. Our work also differs in terms of the distribution of the graph. We focus on a 2D distribution of the problem for better strong scaling whereas [9] uses a 1D graph distribution. This previous work used a graph-based approach but we use a linear-algebra based formulation. 2D distribution for triangle counting was inspired by the success of this approach in the shared-memory parallel context [16].

Our main contributions in this paper are:

- A hybrid, parallel, triangle counting algorithm implemented with MPI on distributed-memory compute nodes and Cilk for shared-memory parallelism.
- Demonstration of good strong scaling properties achieved by the 2D cartesian partitioning of the problem that provides nice upper bounds on communication overheads.
- Demonstration of the fastest known time (at time of publication) for the real-world twitter graph (3.217 seconds) on 1,092 cores with no replication of the graph.
- Demonstration of up to 2.7x speed up compared to the best running time that was obtained on 6,144 cores by

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

the state-of-the-art distributed-memory triangle counting algorithm, which was the champion in past graph challenge [9].

II. Approach

We consider the $(L \times L) . * L$ formulation [2], which represents triangle counting as a sparse matrix-matrix multiplication followed by an element-wise multiplication. The sum of the entries in the resulting matrix gives the number of triangles. This formulation checks only one wedge of each triangle, that is, only the $i-j-k$ wedge is checked for $i > j > k$. Typically, the element-wise multiplication "$. * L$" is fused in the matrix-matrix multiply $L \times L$ as a mask in the implementation in order to avoid forming the result of $L \times L$.

As discussed in Wolf *et al.* [2], the performance of this algorithm depends on the ordering of the rows in L, which is also the ordering of the columns. It has been observed that reordering the rows of A in decreasing order of the number of nonzeros before obtaining L helps the execution time of $(L \times L) . * L$. Therefore, we use the L matrices that are obtained as described above.

We partition the L matrix among MPI processes using 2D cartesian partitioning, where L is written in $Q \times Q$ block-structure form:

$$L = \begin{bmatrix} L_{1,1} & & & & \\ L_{2,1} & L_{2,1} & & & \\ L_{3,1} & L_{3,2} & L_{3,3} & & \\ \vdots & \vdots & \vdots & \ddots & \\ L_{Q,1} & L_{Q,2} & L_{Q,3} & \dots & L_{Q,Q} \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ \vdots \\ C_Q \end{bmatrix}. \quad (1)$$

In this form, only the blocks $L_{q,r}$ with $q \geq r$ may have nonzero entries, so we omit those with $q < r$ from our consideration. Here, C_q denotes the qth row chunk and can be written as $C_q = [L_{q,1} \ L_{q,2} \ \dots \ L_{q,q}]$. Note that C_q has q blocks, for $1 \leq q \leq Q$. The rows and columns of L are partitioned conformally, that is, the indices of the rows in the qth chunk C_q correspond to the indices of the columns in the qth column chunk for $1 \leq q \leq Q$. In our parallel algorithm, each MPI process is assigned a single block $L_{q,r}$. So, the number of MPI processes, which we denote with P, should be a number that can be written as

$$P = Q(Q+1)/2,$$

where Q denotes the number of row chunks in the block-structure form.

We divide the overall computation task of $(L \times L) . * L$ among MPI processes so that each process already has its assigned block as the left operand of $(L \times L)$. Let $p_{q,r}$ denote the process to which block $L_{q,r}$ is assigned. The computation to be performed by $p_{q,r}$ is then formulated as

$$(L_{q,r} \times [L_{r,1} \ L_{r,2} \ \dots \ L_{r,r}]) . * \ [L_{q,1} \ L_{q,2} \ \dots \ L_{q,r}],$$

which can also be written as

$$(L_{q,r} \times C_r) . * \ [L_{q,1} \ L_{q,2} \ \dots \ L_{q,r}].$$

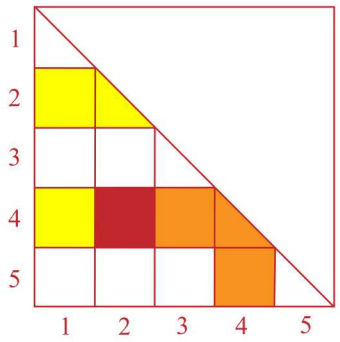

Fig. 1: A sample L matrix partitioned for $(5 \times 6)/2 = 15$ MPI processes. Yellow blocks correspond to the processes $p_{4,2}$ receives its operands for the matrix multiplication (right hand side matrix for the matrix-matrix multiplication or the elementwise multiplication). Orange blocks correspond to the processes to which $p_{4,2}$ will send its nonzeros.

We determine the block-structure form in (1) on the given L matrix in such a way that the number of nonzeros in blocks is balanced. Let nnz_{avg} denote the number of nonzeros in a block for a perfectly-balanced partition, that is, $nnz_{avg} = nnz(L)/P$, where $nnz(L)$ denotes the total number of nonzeros in L. To determine the rows in C_1, we sum the number of nonzeros in the rows of L starting from the top row and include all the considered rows in C_1 until the running sum exceeds nnz_{avg}. For C_2, we start from the row where we left and repeat the same process until the running sum exceeds $2 \cdot nnz_{avg}$ since there are two blocks in C_2. This procedure is repeated for each row chunk C_q so that C_q contains about $q \cdot nnz_{avg}$ nonzeros. Determining row chunks $C_1, C_2, \dots, C_Q$ automatically induces the $Q \times Q$ block-structure in (1) since we use conformal row and column partitions of L.

In order for process $p_{q,r}$ to perform $(L_{q,r} \times C_r) . * [L_{q,1} \ L_{q,2} \ \dots \ L_{q,r}]$, it needs to receive some nonzeros of C_q and C_r from their owners. Recall that C_r has r blocks containing nonzeros, so, communication with r MPI processes is needed for receiving those nonzeros. Hence, $p_{q,r}$ may receive nonzeros from r processes, namely $p_{r,1}, p_{r,2}, \dots, p_{r,r}$ to construct C_r with the needed rows. Note that out of all rows of C_r, only the ones that correspond to the columns of $L_{q,r}$ with at least one nonzero are needed for $(L_{q,r} \times C_r)$. Similar to C_r, $[L_{q,1} \ L_{q,2} \ \dots \ L_{q,r}]$ has also r blocks but $L_{q,r}$ is already assigned to $p_{q,r}$, so it will receive the respective nonzeros of $L_{q,1}, L_{q,2}, \dots, L_{q,r-1}$ from $r-1$ processes, namely $p_{q,1}, p_{q,2}, \dots, p_{q,r-1}$. Note that out of all rows in blocks $L_{q,1}, L_{q,2}, \dots, L_{q,r-1}$, only the ones in which $L_{q,r}$ has nonzeros are needed. Hence, $p_{q,r}$ receives nonzeros from a total of $r + (r-1) = 2r - 1 = O(\sqrt{P})$ processes if $q \neq r$, whereas it receives them from only $r-1 = O(\sqrt{P})$ processors if $q = r$.

In a dual manner, $p_{q,r}$ sends the nonzeros in $L_{q,r}$ to each process $p_{k,q}$ with $q < k \leq Q$ in order for them to construct C_q as the right operand of their respective matrix-matrix

multiplication. For the element-wise multiplication, $p_{q,r}$ sends its nonzeros to each process $p_{q,k}$ with $r < k \leq q$. Hence, $p_{q,r}$ sends nonzeros to a total of $(Q-q)+(q-r) = Q-r = O(\sqrt{P})$ processes if $q \neq r$, whereas it sends them to only $Q-q = O(\sqrt{P})$ processes if $q = r$.

Figure 1 illustrates a sample L matrix partitioned for $(5\times 6)/2 = 15$ MPI processes. Process $p_{4,2}$ owns $L_{4,2}$, which is colored red in the figure. $p_{4,2}$ receives nonzeros from blocks $L_{2,1}$ and $L_{2,3}$, which are colored yellow, to form C_2 as the right operand of matrix-matrix multiplication. It also receives nonzeros from $L_{4,1}$ for the element-wise multiplication. It sends its nonzeros to $p_{4,4}$ and $p_{5,4}$, whose owned blocks are colored orange, for them to construct C_4 for their matrix-matrix multiplication. It also sends its nonzeros to $p_{4,5}$, for its element-wise multiplication.

We simultaneously perform both types of communication: those for the right operand of the matrix multiplication and those for the element-wise multiplication. We used non-blocking receive and blocking send functions for communication.

After receiving the needed rows, each process $p_{q,r}$ performs its overall computation operation $(L_{q,r} \times [L_{r,1}\ L_{r,2}\ \ldots\ L_{r,r}]) .* [L_{q,1}\ L_{q,2}\ \ldots\ L_{q,r}]$ in r small computation tasks, where the kth task is formulated as

$$(L_{q,r} \times L_{r,k}) .* L_{q,k}$$

for $1 \leq k \leq r$. The resulting matrix of each such computation task can be removed after the sum of its entries is accumulated to the triangle count. This approach allows using dense hashmaps since the column range of the right operand of the matrix-matrix multiply is always limited to the block size.

We perform each computation task of $(L_{q,r} \times L_{r,k}) .* L_{q,k}$ using multiple Cilk threads. We use 1D block partitioning on the rows of $L_{q,r}$ and assign the computation operations on different row blocks to different threads. If there are t threads, we divide $L_{q,r}$ into αt row blocks in such a way that the number of nonzeros in these blocks are balanced. Here, α denotes the decomposition rate. It is emprically observed that $\alpha = 4$ yields the smallest running time.

Processes store their corresponding blocks in compressed row storage (CRS) format and the Cilk threads perform their respective computations accordingly. Communication is also performed using the same format, that is, the nonzeros sent in the last communication step are packed in the CRS format.

III. Experiments

We considered 10 test graphs whose basic properties can be seen in the first four columns of Table I. The column headers $|V|$, $|E|$, and $|T|$ denote the numbers of vertices, edges, and triangles, respectively. scale23-25 are synthetic graph500 networks and were obtained from the Graph Challenge website [17]. uk-2005, it-2005, twitter, friendster are real-world graphs and were obtained from SuiteSparse matrix collection [18]. uk-2007 is also a real-world graph and was obtained from [19]. twitter2 and friendster2 were generated using Adapted Block Two-Level Erdos Renyi (A-BTER) scaling and generation methods [20], [21]. Scaling of the degree and clustering coefficient distributions was done such that the generated graphs had 2x the number of vertices and edges while holding fixed the average degree and 'Native Mu'. Native Mu is the ratio of generated edges external to BTER blocks to total generated edges, which are 0.66 and 0.67 for twitter and friendster, respectively.

Our test graphs are undirected graphs and the vertices are in decreasing order of their degrees. The number of edges reported in the third column of Table I corresponds to the number of edges in the undirected graph, which is also equal to the number of nonzeros in matrix L in our linear-algebra-based algorithm.

Our code is implemented in C++ and compiled using Intel compiler (version 19.0.3.199) with optimization flag "-O3" and OpenMPI (version 2.1). We performed our experiments on two different clusters with Skylake and Broadwell architectures. In Skylake cluster, each node has two Intel Xeon Platinum 8160 CPUs with clock frequency 2.10GHz and 196GB memory. In Broadwell cluster, each node has two Intel Xeon E5-2695 CPUs with clock frequency 2.10GHz and 128GB memory. Both clusters have Intel OmniPath interconnect.

We tested our code using $P = 28$, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, and 190 MPI processes. On Skylake cluster, which is small, the maximum number of MPI processes tested is 91 and each node was assigned 4 MPI processes, i.e., two per socket. Each node of this cluster has 48 physical cores, so each MPI process effectively uses 12 Cilk threads. On Broadwell cluster, each node was assigned 2 MPI processes, i.e., one per socket. Each node of this cluster has 36 physical cores, so each MPI process effectively uses 18 Cilk threads. In Table I and Figure 2, labels Skylake and Broadwell are followed by (4x12) and (2x18) to indicate the number of MPI processes per node times the number of Cilk threads per MPI process, respectively.

We evaluate the performance of the proposed algorithm by the overall running time as well as two different rate metrics: edges per second (EPS) and triangles per second (TPS). Table I displays the best running times achieved by our code and the corresponding rates in columns 5–6 and 9–10, respectively. The EPS rate can simply be computed as the number of edges divided by the minimum of the best running times on Skylake and Broadwell. Similarly, the TPS rate can simply be computed as the number triangles divided by the minimum of the best running times on Skylake and Broadwell. The reported running times include both communication and computation times. On Skylake, best running times are achieved using 91 MPI processes except for scale23 and uk-2007, for which using 66 MPI processes results in the best running time. On Broadwell, best running times are achieved using 190 MPI processes except for uk-2005, for which using 120 MPI processes results in the best running time. For the sake of completeness, we also present the computation time for the corresponding best running time in columns 7–8.

Table I displays the most important results highlighted. On

TABLE I: Properties of the test graphs and best running times and rates achieved.

Graph	$\|V\|$	$\|E\|$	$\|T\|$	Total time (s) Skylake (4x12)	Total time (s) Broadwell (2x18)	Computation time (s) Skylake (4x12)	Computation time (s) Broadwell (2x18)	Rate Edges per sec. (EPS)	Rate Triangles per sec. (TPS)
scale23	4,606,314	129,250,705	4,549,133,002	0.440	0.339	0.165	0.123	3.81E+08	1.34E+10
scale24	8,860,450	260,261,843	9,936,161,560	0.845	0.712	0.312	0.242	3.66E+08	1.40E+10
scale25	17,043,780	523,467,448	21,575,375,802	1.678	1.551	0.725	0.666	3.38E+08	1.39E+10
uk-2005	39,459,925	783,027,125	21,779,366,056	1.255	2.019	0.568	1.186	6.24E+08	1.74E+10
it-2004	41,291,594	1,027,474,947	48,374,551,054	1.654	2.374	0.615	1.413	6.21E+08	2.92E+10
twitter	61,578,414	1,202,513,046	34,824,916,864	3.217	3.728	1.361	1.695	3.74E+08	1.08E+10
twitter2	103,809,266	3,107,433,379	151,582,758,659	14.630	9.883	10.030	5.286	3.14E+08	1.53E+10
friendster	65,608,366	1,806,067,135	4,173,724,142	6.997	4.959	4.344	2.326	3.64E+08	8.42E+08
friendster2	131,216,732	3,604,811,068	16,803,555,478	19.710	11.540	14.640	7.282	3.12E+08	1.46E+09
uk-2007	105,896,555	3,301,876,564	286,701,284,103	5.316	6.042	2.871	3.760	6.21E+08	5.39E+10

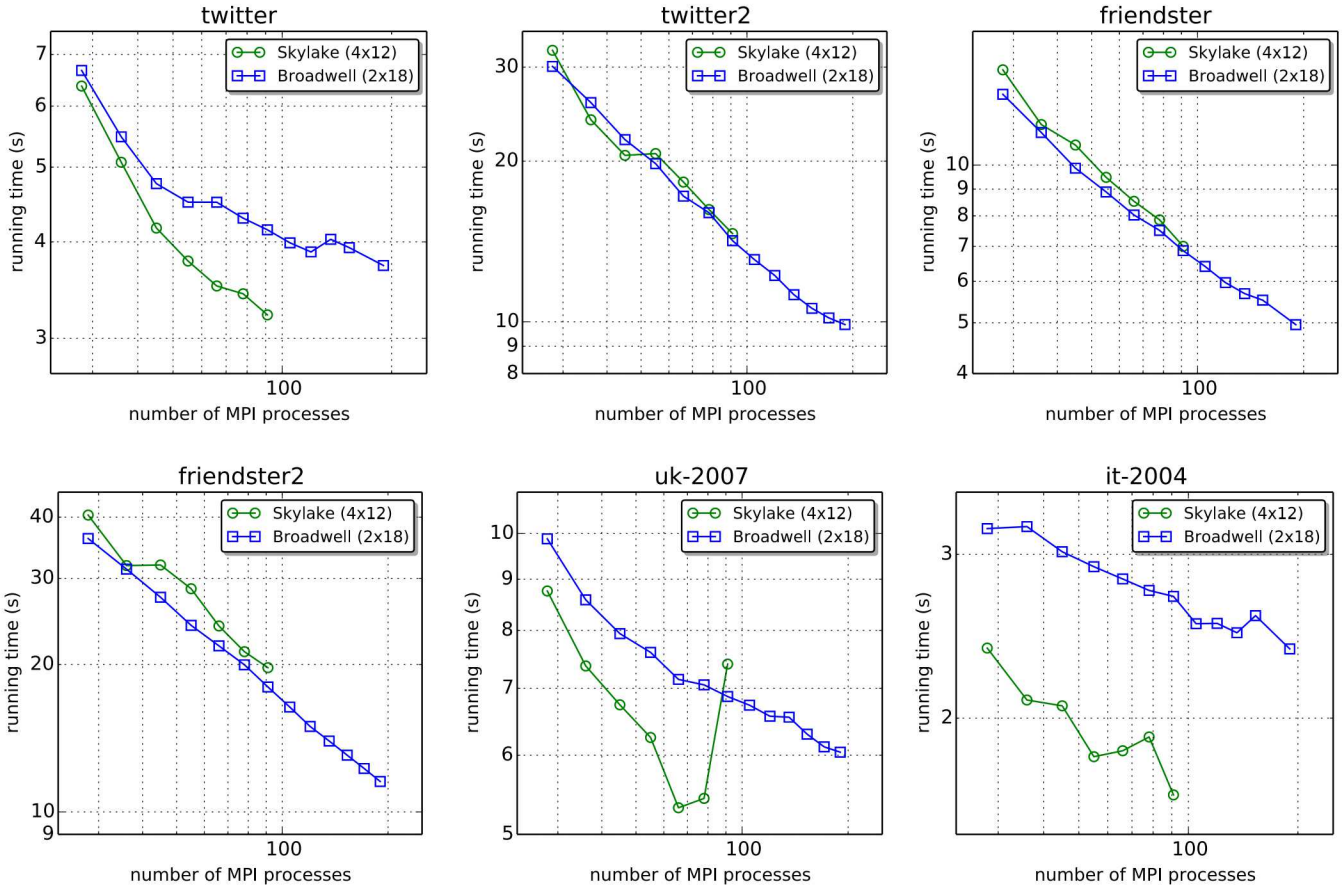

Fig. 2: Strong scaling plots on two different architectures. $(a \times b)$ denotes using a MPI processes per node and b Cilk threads per MPI process.

twitter, our triangle counting algorithm runs in 3.217 seconds on Skylake and in 3.728 seconds on Broadwell. The running time of 3.217 seconds is 2.7x better than 8.52 seconds obtained by the last distributed-memory-parallel champion [9]. On this graph, the best running times reported by the champions of 2018 were 6.5 seconds using GPUs excluding the copy time during data replication [6] and 28.36 seconds using Cilk [3]. Note that our implementation does not involve any replication and the running time that we report includes the communication time as well as the computation time.

The best EPS rates achieved by the proposed algorithm on our test graphs are between 3.12E+08 and 6.24E+08, which is obtained on uk-2005 and followed by 6.21E+08 on it-2004 and uk-2007. The best TPS rates achieved by the proposed algorithm are between 8.42E+08 and 5.39E+10, which is obtained on uk-2007.

Figure 2 displays the strong scaling plots on both architectures for the six largest test graphs. The x-axis denotes the number of MPI processes, whereas y-axis denotes the running time, best of which is given under columns 5 and

6 in Table I. As seen in the plots, the proposed algorithm scales very well in terms of the running time except for uk-2007 on Skylake. Note that the running times on Skylake are generally better than those on Broadwell when the same number of MPI processes are used. However, on friendster and friendster2 graphs, running times on Broadwell are better than those on Skylake.

Good strong scaling property of the proposed parallel algorithm can be attributed to its hybrid nature and using 2D cartesian partitioning scheme. In the distributed-memory level, 2D cartesian partitioning proves to be the best option for reducing communication overheads and in the shared-memory level, Cilk provides the fastest computation.

IV. Conclusion

We proposed a hybrid parallel triangle counting algorithm which uses MPI on distributed-memory compute nodes and Cilk for shared-memory parallelism. Besides exhibiting very good strong scaling behavior in almost all tested graphs, our algorithm achieved the fastest time on the real-world twitter graph, which is 3.217 seconds, on 1,092 cores including both communication and computation times. This runtime is 2.7× smaller than that obtained by the state-of-the-art distributed-memory triangle counting algorithm, which was the champion in the past graph challenge and used 6,144 cores.

Acknowledgment

We thank George Slota for generating twitter2 and frindster2 graphs for us. We thank Jonathan Berry for helpful discussions. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–6.

[2] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–7.

[3] A. Yaşar, S. Rajamanickam, M. Wolf, J. Berry, and U. V. Çatalyürek, "Fast triangle counting using cilk," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[4] H. Kabir and K. Madduri, "Parallel k-truss decomposition on multicore systems," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[5] A. S. Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis, "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[6] Y. Hu, H. Liu, and H. H. Huang, "High-performance triangle counting on GPUs," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–5.

[7] M. Bisson and M. Fatica, "Update on static graph challenge on GPU," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–8.

[8] ——, "Static graph challenge on GPU," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–8.

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

[10] B. Hendrickson, R. Leland, and S. Plimpton, "An efficient parallel algorithm for matrix-vector multiplication," *International Journal of High Speed Computing*, vol. 07, no. 01, pp. 73–88, 1995. [Online]. Available: https://doi.org/10.1142/S0129053395000051

[11] U. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010. [Online]. Available: https://doi.org/10.1137/080737770

[12] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 65:1–65:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063471

[13] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 902–911.

[14] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *Proc. Int'l. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55 – 69, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731596901070

[16] A. Yasar, S. Rajamanickam, J. Berry, M. Wolf, J. Young, and U. Catalyurek, "Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments," in *High Performance Extreme Computing Conference (HPEC), 2019 IEEE (submitted)*, 2019.

[17] "Graphchallenge Data Sets," https://graphchallenge.mit.edu/data-sets.

[18] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[19] P. Boldi and S. Vigna, "WebGraph Datasets: Laboratory for algorithmics," http://law.di.unimi.it/datasets.php, April 2018.

[20] G. M. Slota, J. Berry, C. Phillips, S. Rajamanickam, S. Olivier, and S. Hammond, "Scalable generation of graphs for benchmarking HPC community-detection algorithms," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[21] T. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C424–C452, 2014. [Online]. Available: https://doi.org/10.1137/130914218