First Look: Linear Algebra-Based Triangle Counting without Matrix Multiplication

Tze Meng Low, Varun Nagaraj Rao, Matthew Lee, Doru Popovici, Franz Franchetti
Department of Electrical and Computer Engineering
Carnegie Mellon University
Email: lowt@cmu.edu, vnrao@andrew.cmu.edu, {mattlkf, dpopovic, franzf}@cmu.edu

Scott McMillan
Software Engineering Institute
Carnegie Mellon University
Email: smcmillan@sei.cmu.edu

***Abstract*—**
Linear algebra-based approaches to exact triangle counting often require sparse matrix multiplication as a primitive operation. Non-linear algebra approaches to the same problem often assume that the adjacency matrix of the graph is not available. In this paper, we show that both approaches can be unified into a single approach that separates the data format from the algorithm design. By not casting the triangle counting algorithm into matrix multiplication, a different algorithm that counts each triangle exactly once can be identified. In addition, by choosing the appropriate sparse matrix format, we show that the same algorithm is equivalent to the compact-forward algorithm attained assuming that the adjacency matrix of the graph is not available. We show that our approach yields an initial implementation that is between 69 and more than 2000 times faster than the reference implementation. We also show that the initial implementation can be easily parallelized on shared memory systems.

I. Introduction

It is generally known that counting the exact number of triangles in a graph $\mathcal{G}$ can be described using the language of linear algebra as

$$\frac{1}{6}\Gamma(A^3),$$

where A is the adjacency matrix of the graph $\mathcal{G}$, and $\Gamma(X)$ is the trace of the square matrix X [1]. Other linear algebra approaches [2], [3] also require a sparse-matrix multiplication of A or parts of A as part of their computation. Alternative approaches that are not based on linear algebra leverage other formats for describing graphs such as the adjacency list to design their algorithms [4], [5].

In this paper, we show that these approaches are not mutually exclusive. Using the linear algebra approach, we describe an algorithm that computes the number of triangles exactly once. Unlike algorithms described in the language of linear algebra, this algorithm does not require a sparse matrix matrix multiplication, and does not require a subsequent scaling of the number of triangles. This suggests that our algorithm avoids redundant computation and possibly redundant data movement.

We also show that by selecting the appropriate data format when implementing the linear algebra algorithm, the resulting implementation is similar to an algorithm derived using approaches starting with a description of a graph that is not the adjacency matrix.

Finally, we show that our implementation of exact triangle counting algorithm yields sequential performance that is between 69 and more than 2000 times faster than the reference implementation. Initial parallelization efforts yielded an additional factor of 1.2 to 19.7 improvement over the sequential implementation on various architectures.

II. A triangle counting algorithm

Let $\mathcal{G} = (V, E)$ be a simple undirected graph with vertex set V and edge set E. In addition, assume that V has been partitioned into two disjoint sets, V_{TL} and V_{BR}. Under these assumptions, a triangle in G, described using the 3-tuple (u, v, w) where $u, v, w \in V$, can be classified into four categories:

– *Category 1: Triangles in V_{TL}*. Vertices of these triangles are from V_{TL}, i.e. $u, v, w, \in V_{TL}$.

– *Category 2: Triangles mostly in V_{TL}*. Vertices of these triangles are formed with two vertices in V_{TL} and one vertex in V_{BR}, i.e. $u, v \in V_{TL}$ and $w \in V_{BR}$.

– *Category 2: Triangles mostly in V_{BR}*. Vertices of these triangles are formed with one vertex in V_{TL} and two vertices in V_{BR}, i.e. $u \in V_{TL}$ and $v, w \in V_{BR}$.

– *Category 4: Triangles in V_{BR}*. Vertices of these triangles are from V_{BR}, i.e. $u, v, w, \in V_{BR}$.

Figure 1 describes $\mathcal{G}$ and the four categories of triangles.

A. Intuition

Let Δ be the sum of (1) the number of triangles whose vertices are all in V_{TL} (Category 1), and (2) the number of triangles created with two vertices in V_{TL} and a vertex in V_{BR} (Category 2). To compute the number of triangles $\mathcal{G}$, we start with all vertices in V_{BR}. Trivially $\Delta = 0$ since no vertices are in V_{TL}. As we move each vertex from V_{BR} into V_{TL}, we update Δ accordingly. When all vertices in V_{BR} have been moved to V_{TL}, i.e. $V_{TL} = V$, we will have computed all triangles in G since all the triangles in G will fall in the first category, i.e. all three vertices of each triangle are in V_{TL}.This means that when $V_{TL} = V$, Δ will contain the exact number of triangles in $\mathcal{G}$.

Consider an arbitrary vertex v_{11} in V_{BR} that is selected to be moved to V_{TL}. Triangles where one of the vertices is v_{11}

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

must fall into categories 2, 3, or 4. We consider how Δ is updated based on the different categories.

- *Category 2.* Recall that triangles in this category have two vertices in V_{TL} and v_{11} as the third vertex. These triangles will become triangles in Category 1 when v_{11} is added to V_{TL}. Since Δ is the total number of triangles in both categories 1 and 2, no change to Δ is required since these triangles were previously added to Δ.
- *Category 3.* Triangles in this category have one vertex in V_{TL} and two vertices in V_{BR}, where v_{11} is one of those two vertices. This means that when v_{11} is moved into V_{TL}, these triangles will become Category 2 triangles. Therefore, the number of triangles in Category 3 that are connected to v_{11} needs to be added to Δ.
- *Category 4.* When v_{11} is moved into V_{TL}, any triangle in Category 4 will now have two vertices in V_{BR}, and only one vertex, namely v_{11} in V_{TL}. Hence, there is no update to Δ is required.

To summarize, in order to count the number of triangles in $\mathcal{G}$, we start with all vertices in V_{BR}. We move an arbitrary vertex, v_{11}, in V_{BR} to V_{TL}. When v_{11} is moved, we add the number of Category 3 triangles that has v_{11} as a vertex to Δ. When all vertices in V_{BR} are moved to V_{TL}, Δ would hold the number of triangles in $\mathcal{G}$.

B. In the language of linear algebra

In this section, we discuss how the above algorithm can be described in the language of linear algebra. In the subsequent discussion, upper case, lower case and greek letters represent matrices, column vectors and scalar elements respectively.

Let A be the $N \times N$ adjacency matrix of our simple undirected graph $\mathcal{G}$ with N vertices. As $\mathcal{G}$ is simple and undirected, we know that A is symmetric and has zeros along the diagonal.

The total number of triangles in $\mathcal{G}$, denoted as $\Delta_{\mathcal{G}}$, can be computed with the operation:

$$\Delta_{\mathcal{G}} = \frac{1}{6}\Gamma(A^3). \tag{1}$$

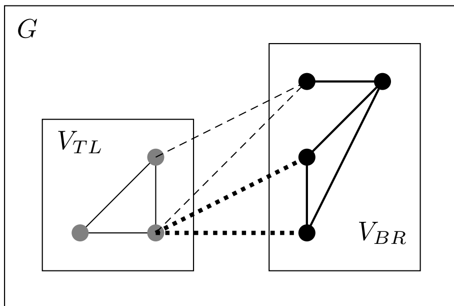

Fig. 1. The graph G and the different components. Vertices are split into two disjoint subsets V_{TL} and V_{BR}, where gray vertices belong to V_{TL} and black vertices belong to V_{BR}. The dashed and dotted lines denote edges in the edge-set E_{BL}. Dashed lines form part of a triangle with two vertices in V_{TL} and one vertex in V_{BR}, while dotted lines form part of a triangle with two vertices in V_{BR} and one vertex in V_{TL}.

Recall that we start by partitioning the vertex set V into two disjoint subsets. This means that matrix A is similarly paritioned into disjoint submatrices in the following manner,

$$A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right),$$

where A_{TL} and A_{BR} are $|V_{TL}| \times |V_{TL}|$ and $(N - |V_{TL}|) \times (N - |V_{TL}|)$ square matrices respectively. A_{TL} and A_{BR} are adjacency matrices for vertices in V_{TL} and V_{BR}. A_{BL} and A_{TR} describe edges (u, v) where$u \in V_{TL}$ and $v \in V_{BR}$.

Equation 1 can then be re-expressed in terms of the different submatrices as follow,

$$\begin{aligned} \Delta_{\mathcal{G}} &= \overbrace{\frac{1}{6}\Gamma(A_{TL}^3) + \frac{1}{2}\Gamma(A_{BL}A_{TL}A_{TR})}^{\Delta} + \\ & \quad \frac{1}{2}\Gamma(A_{TR}A_{BR}A_{BL}) + \frac{1}{6}\Gamma(A_{BR}^3). \end{aligned} \tag{2}$$

The four components of the sum in Equation 2 correspond to the four different category of triangles in $\mathcal{G}$. In addition, the sum of the first two components is Δ.

We next consider moving an arbitrary vertex v_{11} from V_{BR} to V_{TL}. For convenience, we move the $(|V_{TL}| + 1)^{th}$ vertex from V_{BR} to V_{TL}. This can be represented by repartitioning the submatrices of A as follows:

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c} A_{00} & (a_{01}|A_{02}) \\ \hline\hline \left(\begin{array}{c} a_{10}^T \\ \hline A_{20} \end{array}\right) & \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \end{array}\right),$$

where the row and column in A representing the edges of the $(|V_{TL} + 1)^{th}$ vertex are exposed. Here, the double lines indicate the original partitioning of A while the single lines describe how the submatrices have been repartitioned.

Substituting these repartitioned submatrices of A into Equation 2, we know that before v_{11} is moved,

$$\begin{aligned} \Delta &= \frac{1}{6}\Gamma(A_{00}^3) + \frac{1}{2}\Gamma\left(\left(\begin{array}{c} a_{10}^T \\ \hline A_{20} \end{array}\right) A_{00}(a_{01}|A_{02})\right) \\ &= \frac{1}{6}\Gamma(A_{00}^3) + \frac{1}{2}\Gamma(a_{10}^T A_{00} a_{01}) + \frac{1}{2}\Gamma(A_{20}A_{00}A_{02}) \end{aligned} \tag{3}$$

When v_{11} has been moved to V_{TL}, the edges of v_{11} connecting to and from other edges in V_{TL} must be similarly moved to the submatrix A_{TL} as follows:

$$\left(\begin{array}{c||c} \left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline a_{10}^T & \alpha_{11} \end{array}\right) & \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \end{array}\right) \\ \hline\hline (A_{20}|a_{21}) & A_{22} \end{array}\right) \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right),$$

and

$$\begin{aligned} \Delta &= \frac{1}{6}\Gamma\left(\left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline a_{10}^T & \alpha_{11} \end{array}\right)^3\right) + \\ & \quad \frac{1}{2}\Gamma\left((A_{20}|a_{21})\left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline a_{10}^T & \alpha_{11} \end{array}\right)\left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \end{array}\right)\right). \end{aligned}$$

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

which can be expended and simplified into

$$\Delta = \frac{1}{6}\Gamma(A_{00}^3) + \frac{1}{2}\Gamma(a_{10}^T A_{00} a_{01}) + \frac{1}{2}\Gamma(A_{20}^T A_{00} A_{02}) + \Gamma(A_{20} a_{01} a_{12}^T). \quad (4)$$

Comparing Equation 4 with Equation 3, the update to be performed when vertex v_{11} is moved is simply

$$\Delta = \Delta + \Gamma(A_{20} a_{01} a_{12}^T),$$

or

$$\Delta = \Delta + a_{12}^T A_{20} a_{01}, \quad (5)$$

since the trace of a matrix is invariant under rotation, and the result of $a_{12}^T A_{20} a_{01}$ is a scalar element.

III. Implementation

We assume that the adjacency matrix is stored in the compressed sparse row (CSR) format.

A. Updating Δ

The main operation to be performed in each iteration is the computation $a_{12}^T A_{20} a_{01}$. This can easily be computed using two sparse-matrix, sparse-vector multiplications performed sequentially. However, a more efficient implementation is obtained if we make the following observations:

1) $a_{12}^T A_{20}$ is a sum over selected rows of A_{20}, while the second matrix-vector multiplication can be computed as a sequence of (sparse) inner-products between the rows of A_{20} and a_{01}. This means that instead of performing two sparse-matrix, sparse-vector multiplications, we can perform loop fusion to merge the two operations into a single operation resulting in a single pass through A_{20}. We first use a_{12} to select the appropriate row of A_{20}, and then perform the inner-product of that selected row with a_{01}.
2) $a_{01} = a_{10}^T$. a_{01} is currently a column vector which is expensive to access when the data is stored in CSR format. As A is symmetric, we know that $a_{01} = a_{10}^T$. Therefore, by reading a_{10}^T instead of a_{01}, we turn random data access into sequential access.

The final implementation is shown in Figure 2. The main computation in each iteration of the loop is between Lines 34 and 51. The loop at Line 34 iterates over the rows of A_{20} to select them using the non-zero elements in a_{12}^T. The loop at Line 42 performs the inner-product with the selected row and a_{01}. Finally, Line 47 increment the number of triangles when there is an edge in the appropriate row and column of A_{20}.

B. Parallelism

Notice that once the vertex v_{11} is identified, the adjacency matrix A can be partitioned accordingly. This also means that the computation of $a_{12}^T A_{20} a_{01}$ in Equation 5 can be performed independently. However, care must be taken to perform the addition across multiple threads to prevent a race condition.

These properties suggest that parallelism can be easily introduced with the OpenMP [6] `parallel for` construct around the outer-most loop which iterates over the vertices in V_{BR}. To ensure that there is no race condition when updating Δ, we include the reduction clause on the variable `delta` which holds the number of triangles.

The astute reader will also recognize that the shape of A_{20} changes across iterations. When there are few vertices in V_{TL}, A_{20} is tall and skinny. However, near the end of the computation, A_{20} is short and fat. This suggest that the iterations have varying amount of computation (assuming that the non-zeros in A is uniformly distributed). As such, the scheduling of the iterations to threads was set to dynamic scheduling.

C. Triangle Enumeration

The triangle counting algorithms presented can easily be adapted to enumerate triangles. When a triangle is identified, the identified triangle can be printed out immediately or stored for later retrieval. The vertices of the identified triangle (using variables from Figure 2) are `i`, `*A_col`, and `*y_col`. As each triangle is counted exactly once, we are assured that each triangle is never enumerated multiple times. In the case of storing the identified triangles for later retrieval, the storing of the triangle will cause contention for the limited memory bandwidth, and thus triangle enumeration is expected to be slower than triangle counting.

IV. The compact-forward algorithm

The compact-forward algorithm [4] is an exact triangle counting algorithm that was derived using the adjacency array representation of a graph $\mathcal{G}$. In this section, we show that linear algebra based algorithm described in the previous sections is equivalent to the compact-forward algorithm.

The compact-forward algorithm can be described as a two-stage algorithm in the following manner:

1) *Stage 1. Vertex sorting.* Each vertex is first assigned a unique number, and sorted according to that number.
2) *Stage 2. Triangle counting.* For each vertex v taken in increasing order of the unique number, iterate over all elements u with a larger unique number than v and is in the neighborhood of v. For each element u, check if the neighbor of u whose assigned number is less than that for v is also a neighbor of v. If so, increment the triangle count by 1.

We make the following observations to demonstrate the similarities between our algorithm when implemented using the CSR format, and the compact-forward algorithm:

1) Vertex sorting is performed naturally as part of our implementation because the CSR format naturally orders the vertices when it maps each vertex to a particular row and column in the adjacency matrix.
2) The neighborhood of v whose elements have higher unique numbers than v is simply the a_{12}^T vector in our description of the partitioned adjacency matrix A.
3) The neighbors of v with smaller assigned number than v is simply a_{01}.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

```
1  uint64_t count_triangles(uint64_t *IA, // row indices
2                           uint64_t *JA   )// column indices
3  {
4          uint64_t delta = 0; //number of triangles
5
6          //for every vertex in V_{BR}
7          #pragma omp parallel for schedule (dynamic) reduction(+:delta)
8          for(uint64_t i = 1; i<N-1; i++)
9          {
10                 uint64_t *curr_row_x = IA+i;
11                 uint64_t *curr_row_A = IA+i+1;
12                 uint64_t num_nnz_curr_row_x = *curr_row_A - *curr_row_x;
13                 uint64_t *x_col_begin = (JA + *curr_row_x);
14                 uint64_t *x_col_end = x_col_begin;
15                 uint64_t *row_bound = x_col_begin + num_nnz_curr_row_x;
16                 uint64_t col_x_min = 0;
17                 uint64_t col_x_max = i-1;
18
19                 //partition the current row into x and y, where x == a01^T == a10t and y == a12t
20                 while(x_col_end < row_bound && *x_col_end < col_x_max)
21                         ++x_col_end;
22                 x_col_end -= (*x_col_end > col_x_max || x_col_end == row_bound);
23
24                 uint64_t *y_col_begin = x_col_end + 1;
25                 uint64_t *y_col_end = row_bound-1;
26                 uint64_t num_nnz_y = (y_col_end - y_col_begin) + 1;
27                 uint64_t num_nnz_x = (x_col_end - x_col_begin) + 1;
28
29                 uint64_t y_col_first = i + 1;
30                 uint64_t x_col_first = 0;
31                 uint64_t *y_col = y_col_begin;
32
33                 //compute y*A20*x (Equation 5)
34                 for(uint64_t j = 0; j< num_nnz_y; ++j,++y_col)
35                 {
36                         uint64_t row_index_A = *y_col - y_col_first;
37                         uint64_t *x_col = x_col_begin;
38                         uint64_t num_nnz_A = *(curr_row_A + row_index_A + 1) - *(curr_row_A + row_index_A);
39                         uint64_t *A_col = (JA + *(curr_row_A + row_index_A));
40                         uint64_t *A_col_max = A_col + num_nnz_A;
41
42                         for(uint64_t k = 0; k < num_nnz_x && *A_col <= col_x_max; ++k)
43                         {
44                                 uint64_t row_index_x = *x_col - x_col_first;
45                                 while((*A_col < *x_col) && (A_col < A_col_max))
46                                   ++A_col;
47                                 delta += (*A_col == row_index_x);
48
49                                 ++x_col;
50                         }
51                 }
52         }
53         return delta;
54 }
```

Fig. 2. Implementation of a triangle counting algorithm without matrix multiplication.

4) Counting the number of neighbors of u who are also neighbors of v is simply a inner-product of the appropriate row in A_{20} with the a_{01}.
5) The last two observations allows us to conclude that $a_{12}^T A_{20} a_{01}$ computes exactly the same operation as Stage 2 of the compact-forward algorithm.

The key result arising from this similarity comparison is that the theoretical results for the compact-forward algorithm naturally applies to our linear algebra-based algorithm. Specifically, we know that our algorithm lists (counts) all the triangles in $\mathcal{G}$ in time $\Theta(m^{\frac{3}{2}})$, which demonstrates its optimality based on the bounds shown in [5].

V. Results

In this section, we compare our performance against the serial C++ reference miniTri implementation [3] provided as part of the Graph Challenge [7]. We use the Stanford Network Analysis Project (SNAP) [8] datasets found on the Graph Challenge website.

A. Experiment Setup

We report performance attained on the following systems:

1) Intel Core i3 M380, 2.53GHz, 4 cores,
2) Intel i7 E5-2667 v 3 Haswell , 3.2GHz, 2 x 8 cores.
3) ARM big.LITTLE Junoboard, 2 x Cortex-A57, 4 x Cortex-A53

In all experiments, we assume that the input data resides in main memory and is stored in the compressed-sparse-row (CSR) format. Timings reported are for computing the number of triangles. All implementations were compiled with `gcc` with the `-O2` flag.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

1) Accuracy: We verified the number of triangles via two methods. Firstly, we compared the number of triangles provided by the miniTri reference implementation and our implementation whenever possible. Secondly, we compared the number of triangles with the information from the SNAP website. In all cases apart from one, the number of triangles computed by miniTri and our implementation were the same as those presented on the SNAP website.

We highlight a peculiar observation with the Friendster dataset. We are unable to verify with miniTri as there is no appropriate input data format for this particular dataset. In addition, we are unable to verify with the number of triangles reported on the SNAP website as the dataset from the Graph Challenge had a different number of nodes (119,432,957 unique nodes) than what is reported on the SNAP website (65,608,366 unique nodes) . This suggests that the datasets are different. Nonetheless, we counted 191,716 triangles in the Graph Challenge dataset and 4,173,724,142 triangles in the dataset from SNAP.

2) Performance: Time to compute (in milliseconds) are reported for all SNAP datasets found on the Graph Challenge website are reported in Table I. Timings with the reference implementation is only performed on the M380 and the Haswell architecture. We have no comparison timings for the reference implementation on the M380 as the reference implementation computing on the Cit-Patents dataset did not complete after running for more than five hours.

On the whole, our implementation achieves performance that is between 69 and 2699 times faster than the reference implementation. While there are no distinct patterns in the improvements over the reference implementation, the improvements over the reference implementation are similar across both the M380 and the Intel Haswell for the same dataset.

We also report parallel performance for the different architectures. The number of threads used for each architecture is the maximum number of hardware threads available. This means that for the Intel architectures, hyper-threading was enabled. On most datasets, we obtained an additional factor of 1.2 to 19.7 improvement over our sequential implementation. For some of the smaller datasets, we see a drop in performance when we parallelize our implementation on the Haswell architecture.

We highlight the results obtained with the Friendster datasets. As it is the largest dataset, we only obtained timings with the Haswell architecture. With the dataset from the Graph Challenge (Friendster), we counted the number of triangles in 16.5s with 32 threads. This is a speed up of 12.5 times over our sequential implementation. With the Friendster dataset from SNAP (Friendster-SNAP), we obtained an execution time of 68.9s with 32 threads, and a speed up of 23.1 above our sequential implementation. The large difference in execution time is most likely related to the vastly different number of triangles in the two datasets.

VI. Future Work & Conclusion

In this paper, we present a linear algebra-based approach to identify an exact triangle counting algorithm that does not include matrix multiplication as a primitive operation. We also showed that our linear algebra based algorithm is similar to an algorithm derived assuming that the graph is stored as an adjacency list. We show that our sequential implementation obtains between 69 to more than 2000 times improvement over the reference implementation. Initial parallelization effort yielded an additional factor of 1.2 to 19.8 improvement over our sequential implementation on most datasets.

As the title of the paper suggests, this is a first look at triangle counting algorithms in the language of linear algebra that do not require matrix-multiplication as a primitive operation. Opportunities for optimization remains. An obvious improvement is a deeper look into the parallelization of the code. While decent speedup is obtained with our simple OpenMP parallelization, a better distribution of the data across the threads will reduce redundant data movement, prevent cache thrashing and should yield even better scaling behavior.

Another avenue of improvement we are pursuing is in the out-of-core approach to graph algorithms. While we obtain decent performance once the data is in memory, reading the data from disk is the main bottleneck of the entire graph processing pipeline. Often, the time for reading from disk is longer than the compute time time. This, we believe, is where improvement to the overall graph processing pipeline can be made.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM17-0413].

References

[1] P. Burkhardt, “Graphing trillions of triangles,” *Information Visualization*, p. 1473871616666393, 2016.

[2] A. Azad, A. Buluc, and J. Gilbert, “Parallel triangle counting and enumeration using matrix algebra,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 804–811.

[3] M. M. Wolf, J. W. Berry, and D. T. Stark, “A task-based linear algebra building blocks approach for scalable graph analytics,” in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2015, pp. 1–6.

[4] M. Latapy, “Main-memory triangle computations for very large (sparse (power-law)) graphs,” *Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 458–473, November 2008. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2008.07.017

[5] A. Itai and M. Rodeh, “Finding a minimum circuit in a graph,” *SIAM Journal on Computing*, vol. 7, no. 4, pp. 413–423, 1978.

[6] OpenMP Architecture Review Board, “OpenMP application program interface,” November 2015. [Online]. Available: \url{http://www.openmp.org/#}

[7] 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,” http://graphchallenge.mit.edu/, 2017.

[8] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, June 2014.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

Dataset	Sequential Performance							Parallel Performance					
	Haswell			**M380**			**ARM**	**Haswell**		**M380**		**ARM**	
	Ours (ms)	miniTri (ms)	SpeedUp	Ours (ms)	miniTri (ms)	SpeedUp	(ms)	32 threads (ms)	SpeedUp	4 threads (ms)	SpeedUp	6 threads (ms)	SpeedUp
amazon0302	34.02	4,600	135	58.426	7,338	126	207.04	2.29	14.8	23.41	2.5	46.21	4.5
amazon0312	140.73	35,860	255	284.471	58,590	206	914.82	10.64	13.2	106.41	2.7	190.36	4.8
amazon0505	147.45	38,923	264	290.673	61,639	212	949.15	11.81	12.5	123.29	2.4	202.85	4.7
amazon0601	146.58	40,945	279	288.728	60,055	208	932.37	11.72	12.5	117.88	2.4	200.09	4.7
as20000102	0.49	968	1,996	0.731	1,629	2,229	2.81	0.27	1.8	0.49	1.5	0.95	3.0
As-caida20071105	5.20	7,752	1,492	7.759	13,783	1,776	59.19	1.28	4.0	4.14	1.9	10.1	5.9
Ca-AstroPh	31.44	10,871	346	40.623	18,770	462	119.	2.15	14.6	16.85	2.4	40.3	3.0
Ca-CondMat	5.41	1,010	187	7.16	1,616	226	19.96	0.51	10.5	6.09	1.2	5.71	3.5
Ca-GrQc	0.52	102	195	0.709	170	240	4.44	0.23	2.2	0.37	1.9	0.61	7.2
Ca-HepPh	28.14	12,001	427	35.875	32,906	917	141.45	2.39	11.8	17.14	2.1	33.12	4.3
Ca-HepTh	0.95	142	150	1.308	230	176	3.68	0.92	1.0	0.59	2.2	1.03	3.6
Cit-HepPh	58.05	20,802	358	77.366	35,341	457	233.24	3.48	16.7	30.29	2.6	61.6	3.8
Cit-HepTh	62.76	33,989	542	86.124	59,207	687	279.89	7.44	8.4	36.1	2.4	96.25	2.9
Cit-Patents	1,267.07	195,096	154	2832.069	-	-	7865.62	130.17	9.7	1854.43	1.5	2745.75	2.9
Email-Enron	31.06	18,343	591	42.503	33,108	779	143.76	6.34	4.9	23.98	1.8	58.01	2.5
Email-EuAll	42.96	101,705	2,367	64.04	172,854	2,699	278.48	6.56	6.5	29.76	2.2	122.54	2.3
facebook	17.83	6,247	350	22.153	12,129	548	87.55	2.19	8.2	12.52	1.8	32.02	2.7
flickrEdges	1,756.72	1,089,880	620	2343.625	2,548,270	1,087	7701.95	144.7	12.1	1055.68	2.2	1815.27	4.2
Friendster	205,575	-	-	-	-	-	-	16,481	12.5	-	-	-	-
Friendster-SNAP	1,591,835	-	-	-	-	-	-	68,914	23.1	-	-	-	-
Loc-brightkite	20.67	8,525	412	29.019	14,356	495	102.72	2.89	7.1	14.09	2.1	24.88	4.1
Loc-gowalla	196.12	188,176	959	307.689	455,236	1,480	1438.38	50.22	3.9	243.59	1.3	931.91	1.5
Oregon1_010331	2.03	2,788	1,373	3.141	4,377	1,393	27.79	0.71	2.8	2.24	1.4	12.62	2.2
Oregon1_010407	2.03	2,670	1,316	3.146	4,443	1,412	22.43	0.73	2.8	2.1	1.5	6.57	3.4
Oregon1_010414	2.16	2,957	1,369	3.34	4,616	1,382	19.14	0.79	2.8	2.26	1.5	9.37	2.0
Oregon1_010421	2.20	2,989	1,359	3.436	4,765	1,387	19.56	0.94	2.3	2.58	1.3	9.68	2.0
Oregon1_010428	2.34	2,975	1,273	3.452	4,893	1,418	19.43	0.75	3.1	2.34	1.5	16.82	1.2
Oregon1_010505	2.23	3,031	1,357	3.391	4,784	1,411	19.72	0.96	2.3	2.23	1.5	16.91	1.2
Oregon1_010512	2.23	3,063	1,371	3.402	4,753	1,397	19.56	0.7	3.2	2.24	1.5	8.22	2.4
Oregon1_010519	2.18	3,079	1,411	3.394	4,880	1,438	26.78	0.76	2.9	2.22	1.5	10.25	2.6
Oregon1_010526	2.36	3,137	1,330	3.568	5,197	1,457	24.22	0.8	2.9	2.3	1.6	16.87	1.4
Oregon2_010331	3.91	3,382	865	5.483	5,425	989	23.94	1.15	3.4	3.49	1.6	17.02	1.4
Oregon2_010407	3.79	3,455	911	5.384	5,582	1,037	23.72	1.37	2.8	3.51	1.5	16.59	1.4
Oregon2_010414	4.11	3,554	866	5.784	5,765	997	26.6	1.05	3.9	3.65	1.6	17.08	1.6
Oregon2_010421	3.92	3,558	908	5.89	5,760	978	27.67	1.53	2.6	3.49	1.7	13.89	2.0
Oregon2_010428	4.02	3,557	886	5.776	5,814	1,007	27.6	3.82	1.1	3.62	1.6	14.98	1.8
Oregon2_010505	3.81	3,552	932	5.379	5,684	1,057	26.81	0.92	4.2	3.43	1.6	21.91	1.2
Oregon2_010512	3.83	3,586	936	5.523	5,814	1,053	24.72	1.07	3.6	3.41	1.6	18.71	1.3
Oregon2_010519	4.06	3,756	926	5.946	6,134	1,032	26.12	0.97	4.2	3.5	1.7	17.65	1.5
Oregon2_010526	4.21	3,867	918	6.232	6,285	1,009	35.12	1.31	3.2	3.54	1.8	20.34	1.7
P2p-Gnutella04	1.53	251	164	2.256	404	179	9.1	0.19	8.2	0.97	2.3	2.54	3.6
P2p-Gnutella05	1.28	213	166	1.819	346	190	10.4	0.24	5.3	0.79	2.3	2.43	4.3
P2p-Gnutella06	1.25	208	167	1.678	334	199	6.74	0.44	2.9	0.75	2.2	1.95	3.5
p2p-Gnutella08	0.86	169	197	1.175	268	228	4.67	1.1	0.8	0.55	2.1	2.37	2.0
p2p-Gnutella09	1.07	205	192	1.436	326	227	6.43	0.16	6.8	0.62	2.3	2.03	3.2
p2p-Gnutella24	2.40	347	144	3.261	555	170	12.86	0.31	7.8	1.28	2.6	2.7	4.8
p2p-Gnutella25	1.86	255	137	2.541	413	162	8.29	0.22	8.5	1.14	2.2	1.91	4.4
p2p-Gnutella30	3.22	446	139	4.383	714	163	14.13	0.32	10.1	1.95	2.3	2.74	5.2
p2p-Gnutella31	5.64	754	134	7.956	1,219	153	29.63	0.59	9.6	3.14	2.5	6.66	4.5
roadNet-CA	56.97	3,949	69	96.787	6,456	67	277.62	5.4	10.5	33.43	2.9	56.01	5.0
roadNet-PA	26.74	2,204	82	52.73	3,655	69	156.34	1.94	13.8	18.64	2.8	37.16	4.2
roadNet-TX	34.35	2,725	79	65.781	4,549	69	200.72	1.74	19.8	23.07	2.9	45.05	4.5
soc-Epinions1	114.78	73,388	639	164.447	143,871	875	542.35	9.99	11.5	74.03	2.2	159.92	3.4
soc-Slashdot0811	105.49	53,269	505	157.276	91,200	580	526.09	9.85	10.7	66.5	2.4	136.74	3.8
soc-Slashdot0902	116.37	58,531	503	166.35	97,610	587	577.47	11.25	10.3	71.89	2.3	155.66	3.7

TABLE I

SEQUENTIAL AND PARALLEL PERFORMANCE OF TRIANGLE COUNTING IMPLEMENTATIONS ATTAINED ON VARIOUS ARCHITECTURES.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE