SAND2018-8297C

Fast Triangle Counting Using Cilk

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

***Abstract*—Triangle counting is a representative graph analysis algorithm with several applications. It is also one of the three benchmarks used in the graph challenge workshop. Triangle counting can be expressed as a graph algorithm and in a linear algebra formulation using sparse matrix-matrix multiplication (SpGEMM). The linear algebra formulation using the SpGEMM algorithm in the Kokkoskernels library was one of the fastest implementations for the triangle counting problem in last year's graph challenge. This paper improves upon that work by developing an SpGEMM implementation that relies on a highly efficient, work-stealing, multithreaded runtime. We demonstrate that this new implementation results in improving the triangle counting runtime up to $5\times$ to $12\times$ on different architectures. This new implementation also breaks the 10^9 barrier for the rate measure on a single node for the triangle counting problem. We also compare the linear algebra formulation with a traditional graph based formulation. The linear algebra implementation is up to $2.96\times$ to $7\times$ faster on different architectures compared to the graph based implementation. The differences due to the runtime is evident for both triangle counting implementations. Furthermore, we present analysis of the scaling of the triangle counting implementation as the graph sizes increase using both synthetic and real graphs from the graph challenge data set.**

I. Introduction

The triangle counting problem has been one of the three benchmark problems in the graph challenge workshop [1]. Another graph challenge problem - k-truss computation - also depends on the triangle counting problem [2]. Triangle counting also forms the core of number of other kernels such as the triangle enumeration, subgraph isomorphism, dense neighborhood graph discovery [3], and link recommendation [4]. The 2017 graph challenge had several papers related to the triangle counting problem improving the state-of-the-art for a foundational graph analysis kernel (see Section II for a summary). Samsi et al. [5] analyzed the different results presented at the workshop to understand the state-of-the-art for the triangle counting problem. Typical results for some of the fastest submissions in 2017 graph challenge were in the order of 10^8 edges per second for the triangle counting problem.

The primary motivation of this paper is to improve this state-of-the-art for triangle counting using an efficient, task-stealing, multithreaded runtime – Cilk [6]. This paper improves the earlier linear algebra based triangle counting implementation, which was one of the "champions" of the 2017 graph challenge workshop [7], by using the Cilk runtime for the sparse matrix-matrix multiplication (SpGEMM) kernel instead of an OpenMP based implementation. The choice of new runtime was motivated by an analysis of the OpenMP implementation's scalability for larger problems and large number of threads. Section III describes the algorithms that are implemented and the Cilk based implementation. The differences between these two runtimes for the triangle counting problem on irregular graphs are evident based on the results (see Section IV).

The primary contribution of this paper is a Cilk based implementation (KKTri-Cilk) of the KK-SpGEMM algorithm from the Kokkoskernels library [8] and a Cilk-based triangle counting implementation using the new SpGEMM. The focus is on the best performance on a single multicore node. This version of triangle counting surpasses the 10^9 limit for the rate measure on a single node for the first time. Previous work has reported such performance only when some preprocessing steps are not included [9]. Furthermore, detailed experiments comparing an OpenMP (KKTri) and Cilk (KKTri-Cilk) based triangle counting demonstrate the Cilk based triangle counting results in a speedup up to $5\times$ to $12\times$ on different architectures on the wb-edu matrix. KKTri-Cilk is up to $2.96\times$ to $7\times$ faster on different architectures when compared with the graph based implementation (TCM) [10]. We also analyze the scaling triangle counting time for synthetic and real graphs. The analysis show that the triangle counting time per vertex is highly correlated $\frac{4}{3}$-moment of the graph. Finally, we show the correlation between the rates and the conductance of graph.

II. Background

A. KKTri and 2017 Static Graph Challenge

For the 2017 Static Graph Challenge [1], we submitted a linear algebra-based triangle counting implementation KKTri (previously designated TCKK) [7], focused on efficient single node parallelism. KKTri was built upon the performance portable SpGEMM (called KK-MEM) [11] in the Kokkos Kernels library [8].

We described two linear-algebra based formulations of triangle counting that were based on the adjacency matrix of the graph [1]. The first formulation was a slight variant of Azad, et al. [12] (a linear algebra-based formulation of Cohen's algorithm [13]). These formulations represent triangle counting in terms of sparse matrix-matrix multiplication followed by an element-wise matrix multiplication: $D = (L \cdot U) . * L$, where L and U are the lower and upper triangle parts of the adjacency matrix for the graph, respectively (Azad et al. used A for the element-wise multiply). The SpGEMM operation can be fused with the the element-wise multiply by using the rightmost L as a mask for the SpGEMM,

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.

which avoids explicitly storing all the nonzeros resulting from $L \cdot U$. However, even with this optimization, each triangle is "counted" twice (with one being filtered out). This method was not evaluated in the previous work. Instead, we presented a second formulation, $D = (L \cdot L).*L$. This formulation follows the same logic as the previous method with the SpGEMM operation counting wedges and element-wise multiplication filtering out the wedges that are not triangles. However, this formulation resulted in an additional constraint on wedges "visited", which reduced the number of wedges stored after the SpGEMM operation. Typically, we saw a reduction in the number of operations and runtime. Thus, for the 2017 Graph Challenge we used this formulation. Both formulations are used in this paper and implemented using Cilk.

Three optimizations helped in achieving good performance. First, masked SpGEMM (mentioned before) reduced the memory needed for triangle counting. Secondly, our masked SpGEMM operation used data compression on the right hand side matrix Finally, the ordering of the graph before forming the lower and upper triangle matrices is essential for good performance. All three optimizations are used in the Cilk implementation described here as well.

KKTri was one of the fastest triangle counting methods in the 2017 Static Graph Challenge with a top rate (defined to be the number of undirected edges in the graph divided by the runtime) of 637 million edges per second on a single multicore system. We also compared KKTri to the Cilk implementation of parallel merge-based triangle counted method (TCM) [10] (TCM-Cilk). KKTri performed favorably to TCM-Cilk except when compression did not help or when hyperthreads were used. This observation led us to consider developing a Cilk based implementation of our algorithm, the focus of this paper.

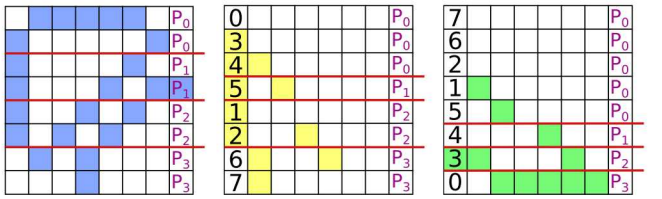
The KK-MEM algorithm in Kokkos Kernels library has been improved with a meta-algorithm KK-SpGEMM to choose between different accumulators, compression scheme and hashmaps [14]. This will be the basis of the Cilk implementation as well. The meta-algorithm has been improved further to support Cilk runtime specific optimizations.

The 2017 Graph Challenge has been a great resource for state-of-the-art in triangle counting. The focus of these work vary from shared-memory parallel implementations [15], [16], [17], single node using accelerators [9], multiple accelerators [18], an operator formulation [19], distributed-memory implementations [20]. We also compared favorably to the Graph BLAS numbers in SuiteSparse [21] on the IBM Power8 architecture using 160 cores(up to XXXX times faster).

III. Approach

A. Cilk Based KKTri Algorithm

KKTri-Cilk algorithm inherits from the KK-SpGEMM algorithm described in [14] and tries to improve load-balancing and efficient hyper-thread usage issues using Cilk based programming model and optimizations. The Cilk programming model supports different parallel patterns such as `cilk_for`, and `cilk_spawn`. Cilk tasks are lightweight with efficient task-stealing allowing an implicit way to load balance. Aside from the runtime system, the parallelization strategy is the main difference between KKTri-Cilk and KKTri. KKTri used a very simple scheme, partitioning the matrix evenly into partitions of a fixed number of rows. KKTri-Cilk also creates row-wise partitions (with each partition being assigned to a Cilk task) but not with a constant number of rows per partition. Instead, a KKTri-Cilk partition, P_i, stores its border in two pointers, one for its first row and one for its last row.



(a) Matrix Form (b) LL Alg: L Matrix (c) LU Alg: L Matrix

Fig. 1. Example problem. (a) is the matrix representation of a graph, each colored cell represents an edge. (b) and (c) are the lower triangular matrices of (a) ordered for LL and LU algorithms, respectively. In (a-c) permutation and partition ids of the rows are written in the first and last column, respectively. Red line is the partition border.

Algorithm 1: KKTri-Cilk(G)

▷ Graph G is given in CSR format
$n_{|T|} \leftarrow 0$ ▷ Initialize number of triangles.
▷ Decide algorithm then order and set A - B matrices
$\langle A, B \rangle \leftarrow$ AnalyzeMatrix(G)
▷ Partitions are balanced based on number of non-zeros.
$P \leftarrow$ CreatePartitions(G)
▷ Run KKTri algorithm in parallel for each P_i
for each $P_i \in P$ **do**
  `cilk_spawn` KKTriSpGEMM-Cilk($A, B, P_i, n_{|T|}$)
`cilk_sync`
return $n_{|T|}$

To balance the work among the tasks, KKTri-Cilk uses an heuristic to find the partitions, creating partitions such that the number of non-zeros within each partition are approximately equal. This is a greedy block (row) partitioning approach to balance the number of non-zeros in a partition. For instance, in our example (Fig. 1(b)), we have four partitions (divided by the red lines), $P = \{P_0, P_1, P_2, P_3\}$, with these partitions have $3, 2, 3, 3$ non-zeros respectively. The orderings used for triangle counting help reduce the number of dense rows and achieve better partitions as well. We partition the L matrix for both LL and LU algorithms.

Algorithm 1 outlines the basic KKTri-Cilk triangle counting algorithm. Each task is responsible for counting the number of triangles in its partition, P_i. These partitions are disjoint therefore tasks can be freely executed in parallel. Since each partition may contain different numbers of rows, `cilk_for` is not used. Instead, `cilk_spawn` is used to initiate these tasks and execute them in parallel. Experimentally, we saw that this approach gave better performance than using `cilk_for` with different grain sizes. Finally, when each task has finished computing the number of triangles within its partition (ensured by the synchronize step), the global triangle count has been

Algorithm 2: KKTRISPGEMM-CILK($A, B, P_x, n_{|T|}$)

```
▷ A is lower triangular matrix.
▷ B is lower or upper triangular matrix; depends on the alg.
▷ P_i is the partition, keeps first and last row's pointers.
▷ n_|T| is shared between tasks.
n_t ← 0  ▷ Count of the local number of triangles
H ← ∅  ▷ Initialize hashmap accumulator
for each i ∈ P_x do
    H[v ∈ B(i)] ← i
    for each j ∈ A(i) do
        for each k ∈ B(j) do
            n_t ← n_t + 1 if H[k] = i else 0
    H ← ∅  ▷ Clean hashmap
▷ Atomically add local number of triangles.
ATOMICADD(n_|T|, n_t)
```

atomically updated to give the final answer.

Each task does a matrix multiplication on its partition. As described in the background, matrix multiplication based triangle counting can be solved using two different algorithms; LL and LU. Algorithm 2 describes these methods. A represents the lower triangular matrix in both algorithms. B is the upper (lower) triangular matrix in LU (LL) algorithm. In both algorithms, we multiply A and B matrices, $C = A \cdot B$, and then mask their result, C, with lower triangular matrix ($C.*L$). KKTri-Cilk implements an in-place masking strategy to reduce number of operations and also memory movement. For a given row, $i \in P_x$, all non-zeros , $v \in B(i)$, are inserted into a hashmap, H, by using their column ids as the key and i as their value. For each non-zero column, j, of $A(i)$ we visit $B(j)$ and check H. If a non-zero column $k \in B(j)$ is set to i, we do in-place masking. If $H[k]$ is equal to i, we have found a triangle. The hashmap needs to be reset for each row in P_x. Finally, we add the local triangle count to the global one using atomic addition.

KKTri-Cilk inherits some of the techniques, and the data structures such as compression, linked-list based hashmap accumulator and dense hashmap accumulator from KKTri [14] and optimizes them for the Cilk implementation.

Hashmap accumulator is critical role for the efficiency of the KKTri algorithm. As previously defined data structures for this problem are highly optimized, we use KKTri's linked list based and dense hash accumulators with minor changes. First, in KKTri-Cilk a graph is partitioned into disjoint sets and, each of these disjoint sets are executed in parallel. Therefore, there are no concurrent inserts, which would require atomic compare and swap instructions. KKTri-Cilk simply defines a hashmap accumulator for each parallel task. Second, as shown in Alg. 2 when execution of a row is completed, we need to reset the hashmap. This can be done in two ways; first, all entries can be reset, second used hashes can be tracked and cleaned. For, the first case using Cilk's array notation to reset all of the elements in a vectorized fashion gives better performance. If the number of used hashes are less than the 50%, cleaning the tracked hashes results in better performance.

Compression of the right hand side matrix can decrease the problem size significantly, and allow using efficient bitwise operations. However, compression is not always successful because of the natural order of the matrices. If there is low locality in the matrix (e.g. RMAT graphs) then compression doesn't improve the execution time [7]. Locality of the partitions which are being processed by tasks has an important effect on efficient usage of the memory bandwidth and the cache. Hence, locality can significantly impact the triangle counting rate obtained by KKTri. In this work, we use conductance as as a way to evaluate locality for a given matrix. In this context, conductance is defined as the ratio between the number of non-zeros in a partition where rows of their column indices appear in different partitions, and total number of non-zeros within that partition. Denoted by C^d, formally conductance of a partition, P, can be defined as follows:

$$C^d(P) = \frac{|\{A(u,v) \neq 0 : |\{u,v\} \cap P| = 1\}|}{|\{A(u,v) \neq 0 : |\{u,v\} \cap P| > 0\}|} \quad (1)$$

For example, in the matrix represented in Figure 1(a), P_0 has 2 rows 7 non-zero elements, 2 of the non-zeros ($A(0,1)$ and $A(1,0)$) can be accessed within this partition. Therefore we can define conductance of this partition as follows; $C^d(P_0) = \frac{5}{7} = 0.71$. We use the average of the conductance of 16-way partition to study the locality of a graph (Table II).

TABLE I
CHARACTERISTICS OF THE ARCHITECTURES USED.

	Server 1	**Server 2**	**Server 3**
Code Name	Skylake	Haswell	KNL
Model	Intel Xeon Platinum 8160	Intel Xeon E7-4850 v3	Intel Xeon Phi 7250
Cores/Freq.	2×24/2.1GHz	4×14/2.2GHz	4×68/1.4GHz
Cache/Mem	33MB/196GB	35MB/16GB	1(MB)/16GB

IV. EXPERIMENTAL EVALUATION

We present several experiments to identify the performance trade-offs of the KKTri-Cilk algorithm. These experiments were carried out on three architectures with multicore processors that are shown in Table I. Intel compiler (`icpc`) version 18.1.163 is used to compile both KKTri and TCM codes. We use 2 hyperthreads for Skylake and Haswell architectures.

A. Dataset and Peak Rate

Table II lists 27 graphs that we used in our experiments along with the number of vertices ($|V|$), number of edges ($|E|$), number of triangles ($|T|$), complement of the conductance ($1 - C^d$), $4/3$ moment, execution time in seconds on Skylake server, and the rate between edges and execution time (Rate) on three different architectures. In addition to 20 graphs on which triangle computing is costly (based on the reference implementation) 7 additional large real-world graphs [22], [23], [24] (highlighted as blue in Table II) are included in our experiments. We used the Graph Challenge procedure of symmetrizing the matrices for the problems we added from public datasets. All experiments were median of five different runs. Table II shows the best achievable rate of 1.86×10^9 for the uk-2007 matrix. We are also able to achieve more than 10^9 rate for uk-2005 and wb-edu matrices. This is about $3\times$ better

the reported peak rate in the 2017 Graph Challenge. All three matrices also have very high locality based on the conductance of the graph. The Skylake architecture results in the best runtimes for all graphs except friendster. Table II also has the times highlighted in green when KKTri-Cilk is the fastest method compared KK-OMP and the TCM implementations. We observe a high correlation (0.88) between the conductance and the rate achieved.

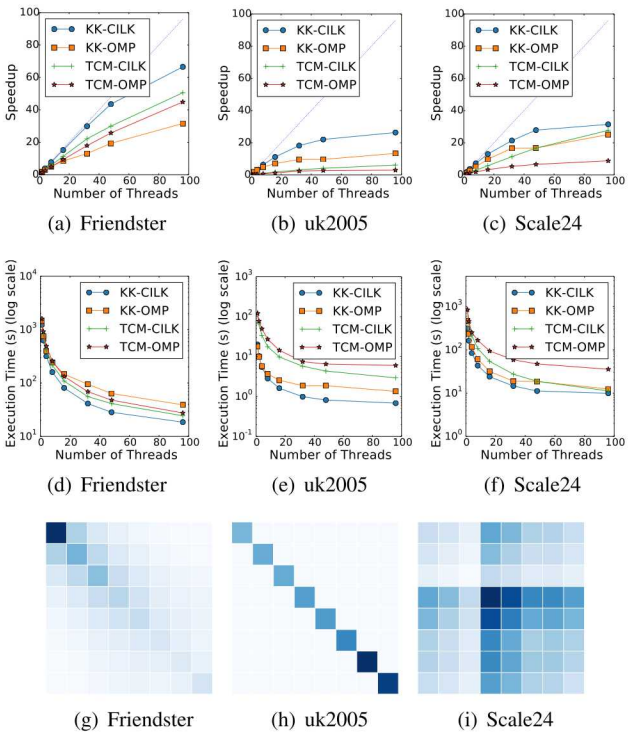
B. Runtime Comparison

Table III presents observed speedups using the Cilk programming model compared to the OpenMP runtime for both KKTri and TCM algorithms, on three different architectures. The geometric mean of the speedups is in the last row of the table. In Table III each cell is highlighted with yellow if the Cilk implementation is faster than the OpenMP implementation. The KKTri-Cilk of the KKTri algorithm outperforms the KKTri-OpenMP in 63 of 78 (81%) cases. For TCM, these numbers are 70 out of 78 (90%) cases. Clearly, Cilk helps to getter better performance for both algorithms.

Although KKTri-Cilk outperforms KKTri-OpenMP in all of the problems on Haswell, the OpenMP version outperforms the Cilk version in smaller instances on Skylake. Since Skylake gives the best performance in almost all problems, we hypothesize this could be due to the tasking overhead in Cilk for these small graph instances. TCM's Cilk implementation outperforms the OpenMP implementation in all problems on both Haswell and Skylake architectures with a geometric mean of 2.6 and 2.3. Part of this difference could be attributed to optimizations used in different implementations. However, there is a clear trend that Cilk helps these algorithms on Haswell and Skylake architectures. However, in KNL architecture OpenMP implementations become competitive with Cilk implementations in many problems and geometric mean of the speedup decreases to $1.07\times$ and $1.65\times$ for KKTri and TCM. This could be due to the relatively smaller caches on the KNL architectures not helping the tasking runtime.

C. Strong Scaling

Figure 2 presents strong scaling speedup of KKTri and TCM for three graphs - Friendster, uk2005 and scale24 - on Skylake architecture. KKTri-Cilk scales the best in all three problems when scaled by the *best* sequential execution time among the different methods. In uk-2005 and and scale24 graphs scaling is better before using hyperthreads. Hyperthreads give a small but measurable benefit. In Friendster all implementations show better scaling with hyperthreads. Figure 2(h) shows the concentration of non-zeroes in the uk-2005 graph. This graph has a very good ordering and most of the non-zeros appear to be near diagonal resulting in highly local computations. Also, we observe from Figure 2(e) that KKTri algorithms' sequential execution time is also less than TCM thanks to these local computations. In contrast to the uk-2005 graph, as can be seen in Figure 2(i) scale24 graph has a very random distribution of the non-zeros therefore during the computation memory access becomes very random. As we can see in



(a) Friendster (b) uk2005 (c) Scale24

(d) Friendster (e) uk2005 (f) Scale24

(g) Friendster (h) uk2005 (i) Scale24

Fig. 2. Strong scaling experiments: (a-c) x-axis presents number of threads, y-axis presents speedup with compare to best sequential algorithm. (d-f) x-axis presents number of threads, y-axis presents the execution time in seconds. (g-i) presents heat map of the graphs when we partition the graph 8 by 8 grids, darker color means more non-zeroes.

Figure 2(g) Friendster graph's distribution is worse than uk-2005 and better than scale24. In this graph there are some small number of highly dense parts and the other parts seem to present a more balanced distribution. This seems to be the best case for scalability even though the best rates are achieved in uk-2005. The poor locality in scale24 seems to hinder both the the rate and scalability measures.

D. Relative Speedup

Figure 3 presents relative speedup between KKTri and TCM codes. Graph sizes grow from left to right. These results use relative speedup from the minimum of the median execution times of 5 runs of the OpenMp and Cilk implementations of the KKTri and TCM algorithms. KKTri outperforms TCM in 23 of 27 cases. In four small instances (3 amazon graphs and emailEU) TCM better depending on the architecture. From Figure 3 we observe that KKTri can achieve up to $7\times$ speedup on graphs that have a good ordering such as wb-edu, uk-2005 and uk-2007. TCM also runs out of memory for uk-2007 on two architectures. As both algorithms use the same runtime the primary difference can be attributed to the algorithms, data structures used and the implementation.

E. Scalability of Triangle Counting

Let n and m be the number of vertices and edges in an undirected graph. If the graph is a clique, the number of triangles is bounded by $\Theta(n^3)$ and $\Theta(m^{3/2})$. These are therefore trivial worst-case bounds on the complexity of triangle

TABLE II

PROPERTIES OF THE DATASET. CONDUCTANCE IS REPORTED FOR LOWER TRIANGULAR MATRICES. BEST OF THE MEDIANS OF THE EXECUTION TIMES IN SECONDS ON SKYLAKE IS REPORTED. MOMENT IS THE 4/3 MOMENT. HIGHLIGHTS: GREEN - KK-CILK IS BETTER THAN TCM, RED - THE FASTEST RATE FOR A GRAPH, BLUE- THE BEST RATE AMONG ALL GRAPHS, PURPLE - PUBLIC DOMAIN GRAPHS.

Data Set	$\|V\|$	$\|E\|$	$\|T\|$	$1 - C^d$	Moment	Time (s)	Rates Skylake	Rates Haswell	Rates KNL
cit-HepTh	27,770	352,285	1,478,735	0.141	98.079	0.003	1.20E+08	8.24E+07	1.54E+07
email-EuAll	265,214	364,481	267,313	0.112	13.521	0.003	1.16E+08	1.10E+08	2.16E+07
soc-Epinions1	75,879	405,740	1,624,481	0.086	48.062	0.004	1.06E+08	6.72E+07	2.44E+07
cit-HepPh	34,546	420,877	1,276,868	0.091	86.684	0.004	1.11E+08	8.77E+07	2.47E+07
soc-Slashdot0811	77,360	469,180	551,724	0.067	50.726	0.004	1.18E+08	7.97E+07	2.71E+07
soc-Slashdot0902	82,168	504,230	602,592	0.069	51.416	0.003	1.57E+08	8.64E+07	2.77E+07
flickrEdges	105,938	2,316,948	107,987,357	0.098	268.872	0.013	1.85E+08	1.15E+08	2.99E+07
amazon0312	400,727	2,349,869	3,686,467	0.229	30.455	0.006	3.87E+08	2.51E+08	9.34E+07
amazon0505	410,236	2,439,437	3,951,063	0.233	31.100	0.006	3.79E+08	2.75E+08	9.36E+07
amazon0601	403,394	2,443,408	3,986,507	0.276	17.434	0.006	4.17E+08	2.87E+08	9.81E+07
scale18	174,147	3,800,348	82,287,285	0.059	347.232	0.031	1.24E+08	1.07E+08	2.88E+07
scale19	335,318	7,729,675	186,288,972	0.058	395.145	0.075	1.04E+08	8.06E+07	2.79E+07
as-Skitter	1,696,415	11,095,298	28,769,868	0.17	73.835	0.026	4.23E+08	3.23E+08	1.23E+08
scale20	645,820	15,680,861	419,349,784	0.059	448.022	0.184	8.53E+07	5.63E+07	2.50E+07
cit-Patents	3,774,768	16,518,947	7,515,023	0.027	21.888	0.028	5.82E+08	4.21E+08	1.22E+08
scale21	1,243,072	31,731,650	935,100,883	0.059	506.130	0.511	6.21E+07	4.78E+07	2.01E+07
soc-LiveJournal1	4,847,571	42,851,237	285,730,264	0.242	53.766	0.137	3.14E+08	2.28E+08	1.07E+08
wb-edu	9,845,725	46,236,105	254,718,147	0.938	16.775	0.042	1.10E+09	6.55E+08	1.48E+08
scale22	2,393,285	64,097,004	2,067,392,370	0.058	569.872	1.581	4.05E+07	3.50E+07	1.71E+07
scale23	4,606,314	129,250,705	4,549,133,002	0.059	640.093	3.786	3.41E+07	2.62E+07	1.45E+07
scale24	8,860,450	260,261,843	9,936,161,560	0.059	717.548	10.282	2.53E+07	2.04E+07	1.21E+07
scale25	17,043,780	523,467,448	21,575,375,802	0.059	802.552	25.652	2.04E+07	1.88E+07	9.11E+06
uk-2005	39,459,925	783,027,125	21,779,366,056	0.925	90.495	0.684	1.14E+09	9.35E+08	2.59E+08
it-2004	41,291,594	1,027,474,947	48,374,551,054	0.942	125.144	1.293	7.95E+08	5.86E+08	1.47E+08
twitter	61,578,414	1,202,513,046	34,824,916,864	0.126	687.587	28.359	4.24E+07	4.46E+07	N/A
friendster	65,608,366	1,806,067,135	4,173,724,142	0.182	344.160	18.552	9.74E+07	7.93E+07	N/A
uk-2007	105,896,555	3,301,876,564	286,701,284,103	0.968	144.436	3.545	1.86E+09	1.50E+09	N/A

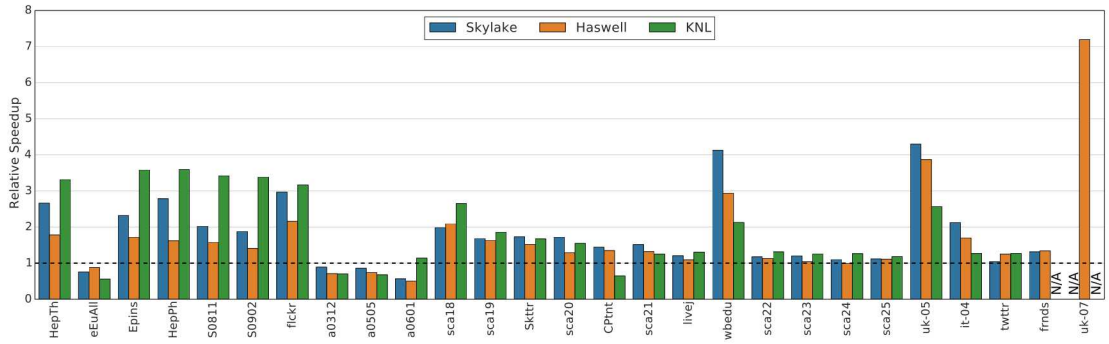


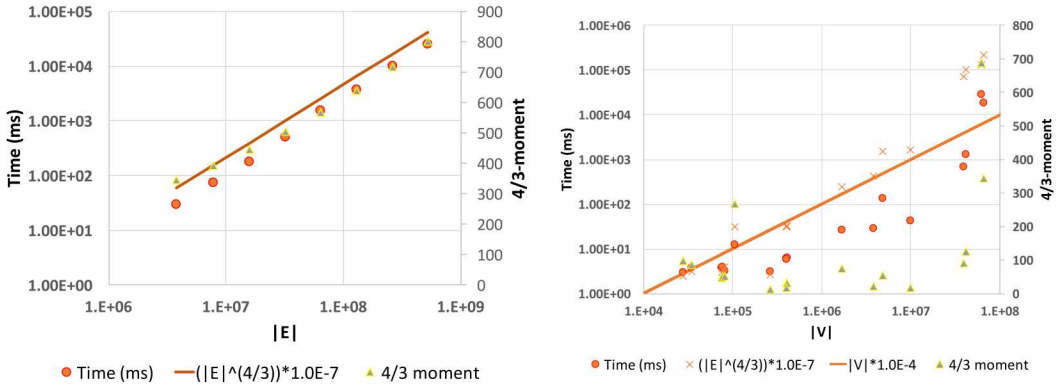
Fig. 3. Relative speedup in Skylake, Haswell and KNL compared to TCM. The KKTri-Cilk time for each graph in every architecture is used as the baseline. In KNL, because of memory limitations, all of the implementations failed on Friendster and uk-2007 graphs. In Skylake TCM fails on uk-2007 graph because of its memory requirement while KKTri doesn't. In the plot these cases are represented with 'N/A'.

enumeration. A trivial worst-case bound on triangle *counting* is $O(n^\omega)$, where ω is the exponent of matrix multiplication. Latapy gives a summary of related complexity results and shows that triangle enumeration is bounded by $\Theta(mn^{\frac{1}{\alpha}})$ if the degree distribution is governed by a power law with exponent α [25]. This is an asymptotic improvement over the worst-case, but Berry, et al. improved upon this further by showing that triangle enumeration complexity can actually be $\Theta(n)$ in realistic circumstances [26]. These circumstances exist in many of the Graph Challenge instances, as we will show.

While the results of [26] are derived from a synthetic graph generation model (the erased configuration model), we find that they help explain empirical results from the Graph Challenge graphs. The essential argument of [26] is that fast triangle enumeration is associated with small values of the $4/3$-moment of the degree distribution. Letting d_v be the degree of vertex v, the $4/3$-moment is defined as: $\mathbf{E}[d_v^{4/3}] = 1/n \sum_v (d_v^{4/3})$. This moment is small for many of the Graph Challenge instances.

The Graph Challenge 2017 organizers have summarized all submitted results, informally fitting their runtimes with a regression line of $O(m^{4/3})$ [5]. In this section, we use the $4/3$-moment to explain the empirical runtime complexity more accurately. Graph Challenge triangle enumeration algorithms can be evaluated this way to ensure that they obtain optimal performance when possible.

The LU algorithm in this paper is asymptotically equivalent to the "MinBucket" algorithm analyzed in [26] and proved to



(a) Scale Graphs (b) Non Scale Graphs

Fig. 4. Scale graphs and nonscale graphs triangle counting time in comparison to the edge count. Trend lines for $|E|^{(4/3)}$ and $|V|$ are also shown. The secondary axis shows the $4/3$-moment.

TABLE III
COMPARISON BETWEEN OPENMP AND CILK RUNTIMES IN THREE SERVERS. FOR EACH SERVER KK REPRESENTS KKTRI/KKTRI-CILK SPEEDUP. TCM REPRESENTS TCM-OPENMP/TCM-CILK SPEEDUP. MEDIANS OF THE 5 RUNS ARE USED.

Data Set	**Haswell**		**Skylake**		**KNL**	
	KK	**TCM**	**KK**	**TCM**	**KK**	**TCM**
cit-HepTh	1.82	1.46	0.52	1.62	0.32	0.82
email-EuAll	3.38	3.05	1.86	1.86	1.04	0.69
soc-Epinions1	1.65	1.18	0.75	1.04	0.70	0.94
cit-HepPh	1.70	1.49	0.44	3.17	0.37	0.89
soc-Slashdot0811	1.93	2.03	0.73	1.23	0.67	0.96
soc-Slashdot0902	1.73	1.54	0.96	1.42	0.68	0.93
flickrEdges	1.98	2.24	1.56	3.39	0.63	2.10
amazon0312	2.04	2.26	2.29	1.89	1.61	0.99
amazon0505	2.18	2.02	2.50	1.95	1.50	1.05
amazon0601	1.60	2.82	1.15	2.20	0.45	0.98
scale18	1.72	2.63	1.17	2.42	0.74	1.70
scale19	1.29	2.65	1.23	2.49	0.92	1.89
as-Skitter	2.46	3.04	2.30	2.56	2.71	2.14
scale20	1.03	3.44	1.33	3.16	1.14	2.08
cit-Patents	2.33	3.16	1.98	1.75	1.10	1.76
scale21	1.17	2.81	1.31	3.13	1.16	1.94
soc-LiveJournal1	2.00	4.06	1.81	2.76	2.25	2.59
wb-edu	5.77	2.11	5.28	1.53	12.66	1.24
scale22	1.20	3.72	1.17	2.95	1.19	2.02
scale23	1.33	2.66	1.46	2.75	1.24	2.82
scale24	1.38	3.56	1.21	3.16	1.27	2.18
scale25	1.50	3.02	1.24	2.41	1.15	2.26
uk-2005	2.85	2.40	1.97	2.05	1.30	2.60
it-2004	2.33	18.33	2.00	16.24	0.90	15.91
twitter	1.91	2.88	1.78	2.80	N/A	N/A
friendster	2.06	1.26	2.11	1.13	N/A	N/A
uk-2007	xxx	11.04	xxx	xxx	N/A	N/A
Geometric Mean	1.87	2.61	1.42	2.33	1.07	1.65

Graph Type	**Time-per-vertex**		**Time-per-edge**	
	$E^{\frac{4}{3}}$	$\frac{4}{3}$-moment	$E^{\frac{4}{3}}$	$\frac{4}{3}$-moment
Scale graphs	0.90	0.98	0.89	0.98
Non-scale graphs	0.26	0.95	0.02	0.76

TABLE IV
CORRELATION COEFFICIENT BETWEEN TRIANGLE COUNTING TIME PER VERTEX/EDGE AND THE $E^{\frac{4}{3}}$ OR $\frac{4}{3}$-MOMENT.

be optimal when the $4/3$-moment is bounded by a constant (MinBucket is alternatively called nodeIterator++ or Cohen's algorithm [27], [13], [28]). Therefore, we expect $O(n)$ performance from LU if $4/3$-moments remain small as n grows.

To test this conjecture, we separate the Graph Challenge instances into two classes which we call *Scale graphs* and *Non Scale graphs*. The former comprise all of the R-MAT instances, and would include other synthetic graphs that have increasing $4/3$-moments. Nonscaled graphs are all others, many of which have small $4/3$-moments despite their size. We compute the $4/3$-moment for all instances and calculate two correlation coefficients: that between runtime per vertex/edge and the $4/3$-moment, and that between runtime per vertex/edge and $m^{4/3}$ (the function identified by the Graph Challenge organizers). Table IV shows the results. Note that the per-vertex/edge runtimes are highly correlated with the $4/3$-moment, as predicted by [26].

Figure 4 depicts the relationship between the $4/3$-moment and runtime on Graph Challenge instances. Note that in the Scaled graphs, $m^{4/3}$ predicts the runtime trends effectively. However, $m^{4/3}$ does not model the runtime on real graphs as effectively as the $4/3$-moment does.

V. CONCLUSIONS

We developed a triangle counting method using the Cilk programming model. This implementation resulted in a rate of of 10^9 on a single multicore node and was able to achieve measurable speedups compared to the OpenMP implementation of the same algorithm. This linear algebra implementation is also faster than graph based implementation (up to $4\times$). We also showed correlation between the high rates achieved and the conductance of the graph. Finally, we were able to show that the scalability of the triangle counting is bounded by O(n) when the 4/3-moment is small.

Acknowledgements: We thank Simon Hammond, Cynthia Phillips, and Stephen Olivier for helpful discussions, and the test bed program at Sandia National Laboratories for supplying the hardware used in this paper. Sandia National Laboratories is a multimission 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 *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, 2017.

[2] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 822–830.

[3] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung, "On triangulation-based dense neighborhood graph discovery," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 58–68, 2010.

[4] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos, "Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation," *Social Network Analysis and Mining*, vol. 1, no. 2, pp. 75–81, 2011.

[5] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, *et al.*, "Graphchallenge. org: Raising the bar on graph analytic performance," *arXiv preprint arXiv:1805.09675*, 2018.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

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

[8] KokkosKernels. [Online]. Available: https://github/kokkos/kokkoskernels

[9] M. Bisson and M. Fatica, "Static graph challenge on gpu," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–8.

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

[11] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 693–702.

[12] A. Azad, A. Buluç, and J. R. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proceedings of the IPDPSW, Workshop on Graph Algorithm Building Blocks (GABB)*, 2015. [Online]. Available: http://gauss.cs.ucsb.edu/~aydin/triangles-gabb.pdf

[13] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[14] M. Deveci, C. Trott, and S. Rajamanickam, "Multi-threaded sparse matrix-matrix multiplication for many-core and gpu architectures," *arXiv preprint arXiv:1801.03065*, 2018.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] T. A. Davis, "Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and K-truss," Texas A&M University, Tech. Rep., 2018.

[22] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2017.

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

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

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

[26] J. W. Berry, L. K. Fostvedt, D. J. Nordman, C. A. Phillips, C. Seshadhri, and A. G. Wilson, "Why do simple algorithms for triangle enumeration work in the real world?" in *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science*, ser. ITCS '14. New York, NY, USA: ACM, 2014, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/2554797.2554819

[27] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Experimental and Efficient Algorithms*, S. E. Nikoletseas, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 606–609.

[28] 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