Exploring Optimizations on Shared-memory Platforms for Parallel Triangle Counting Algorithms

Ancy Sarah Tom*, Narayanan Sundaram†, Nesreen K. Ahmed†, Shaden Smith*, Stijn Eyerman‡, Midhunchandra Kodiyath‡, Ibrahim Hur‡, Fabrizio Petrini†, George Karypis*
*Dept. of Computer Science & Engineering, University of Minnesota,
†Parallel Computing Lab, Intel Corporation, ‡Intel Corporation
{tomxx030}@umn.edu, {shaden, karypis}@cs.umn.edu,
{narayanan.sundaram, nesreen.k.ahmed, stijn.eyerman, midhunchandra.kodiyath, ibrahim.hur, fabrizio.petrini}@intel.com

Abstract—**The widespread use of graphs to model large scale real-world data brings with it the need for fast graph analytics. In this paper, we explore the problem of triangle counting, a fundamental graph-analytic operation, on shared-memory platforms. Existing triangle counting implementations do not effectively utilize the key characteristics of large sparse graphs for tuning their algorithms for performance. We explore such optimizations and develop faster serial and parallel variants of existing algorithms, which outperform the state-of-the-art on Intel manycore and multicore processors. Our algorithms achieve good strong scaling on many graphs with varying scale and degree distributions. Furthermore, we extend our optimizations to a well-known graph processing framework, GraphMat, and demonstrate their generality.**

1. Introduction

Finding the number of triangles in a graph is an important operation in network analysis and graph mining with extensive applications. Among others, it is used to compute certain graph characteristics (e.g., clustering coefficient and transitivity ratio), detect community structure, study motif occurrences, study and detect spamming activities, and understand the structure of biological networks [1, 2, 3, 4].

In recent years, driven by the size of the graphs that needs to be analyzed, there has been significant research in developing efficient and scalable serial and parallel algorithms for computing the exact and approximate number of triangles [3, 5, 6, 7, 8, 9].

Our work here, motivated by the recent GraphChallenge [10], seeks to further the design space of existing serial and shared-memory parallel exact triangle counting approaches. We present various optimizations that leverage characteristics of modern processor architectures and key properties of sparse, real-world and synthetic graphs to speed up the computations performed by the serial algorithms. We show that our serial algorithms are over 9× faster than competing serial algorithms on a wide range of real and synthetic graphs. Furthermore, we show how these optimizations can be incorporated into GraphMat [11, 12], and achieve comparable performance advantages. We present an OpenMP parallel formulation of our serial algorithms that achieve good strong scaling performance on Intel's Haswell and Intel's Knights Landing architectures.

2. Definitions and notations

We will assume that the graphs that we operate on are *simple* and *undirected* and are represented using the standard $G = (V, E)$ notation. We will use $\mathrm{Adj}(v_i)$ to denote the *adjacency list* of v_i; i.e., the set of vertices that are adjacent to v_i, $\mathrm{Adj}^{>}(v_i)$ to denote the subset of v_i's adjacent vertices that are numbered higher than v_i, and $\mathrm{Adj}^{<}(v_i)$ to denote the subset of v_i's adjacent vertices that are numbered lower than v_i. We will use $d(v_i)$ to denote the *degree* of v_i, i.e., $d(v_i) = |\mathrm{Adj}(v_i)|$. A *triangle* is a set of three vertices $\{v_i, v_j, v_k\}$ if the edges (v_i, v_j), (v_i, v_k), and (v_j, v_k) exist in E. The problem of *triangle counting* is to compute the total number of unique triangles in G.

3. Background and related work

The computations underlying triangle counting involve finding the common vertices in the adjacency lists of two adjacent vertices, since the number of triangles in which an edge (v_i, v_j) belongs to is equal to $|\mathrm{Adj}(v_i) \cap \mathrm{Adj}(v_j)|$. This operation is commonly implemented using either an approach that jointly traverses the two adjacency lists or an approach that relies on a map data structure. In the list-based approach, the adjacency lists of all vertices are initially sorted in increasing order based on vertex id, and then, for each edge (v_i, v_j), the adjacency lists of v_i and v_j are jointly traversed to find the common vertices. The complexity of the joint traversal is $O(d(v_i) + d(v_j))$. A variation of this approach is called *OrderedMerge* by Shun and Tangwongsan [7] and *AdjacencyIntersection* by Parimalarangan et al. [8]. In the map-based approach, for each edge (v_i, v_j), an auxiliary data structure is used to mark the set of vertices that are in $\mathrm{Adj}(v_i)$ (or $\mathrm{Adj}(v_j)$), and $\mathrm{Adj}(v_j)$ (or $\mathrm{Adj}(v_i)$) is then traversed to probe the map and identify the common vertices. When the cost of creating the map is amortized over different pairs of edges, then the complexity of this traversal is $O(d(v_j))$ (or $O(d(v_i))$). Different approaches have been developed for implementing the map. In their *AdjacencyMarking* approach, Parimalarangan et al. [8] use bit vectors for marking the vertices that are present. In their *OrderedHash* approach, Shun and Tangwongsan [7] use a single hash table to store all the adjacency lists in the graph prior to performing any adjacency list intersections. However, this increases the overall memory requirements of the algorithm.

In order to ensure that each triangle is enumerated only once, most algorithms for triangle counting enumerate them

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

by imposing an ordering on their vertices. The two most common approaches [7, 8] are to enumerate the triangles in a $\langle v_i, v_j, v_k \rangle$ or a $\langle v_j, v_i, v_k \rangle$ order where $i < j < k$. In the $\langle v_i, v_j, v_k \rangle$ order, for vertex v_i, all of its higher-numbered adjacent vertices are considered and the intersection of their adjacency lists involving vertices that are numbered higher than v_j are used to identify the possible v_k vertices that form the triangles. A similar approach is used in the $\langle v_j, v_i, v_k \rangle$ ordering but, in this case, the starting vertex is v_j, and the candidate v_is are v_j's adjacent vertices that are numbered lower than itself. A result of the above orderings is that the original undirected graph is treated as a directed graph and the operations are performed by only considering the upper triangular portion of the adjacency matrix. (In the case of $\langle v_j, v_i, v_k \rangle$ ordering, the candidate v_i vertices correspond to the vertices that are incident on v_j.) Finally, when the triangles are enumerated using the above orderings, it was shown that re-ordering the vertices in the graph in non-decreasing degree prior to triangle counting leads to significantly better performance [7, 8]. This re-ordering creates an adjacency matrix whose density progressively increases and since the upper triangular part of that matrix is used, the adjacency lists of the high degree vertices contain a relatively small number of higher-numbered vertices, which leads to efficient computations.

4. Methods

We developed various list- and map-based triangle counting algorithms that incorporate different optimizations for enhancing cache performance and reducing the overall amount of computations. These algorithms share many high-level characteristics and optimizations of the prior research (Section 3) and like them, they utilize a non-decreasing degree ordering of the vertices, operate on the upper-triangular part of the adjacency matrix (we will use indicating that by representing the graph as $G(V, E^U)$), and pre-sort the adjacency lists of each vertex so its adjacent vertices are in increasing order. In addition, we utilize a CSR-based storage scheme for the graph's sparse adjacency structure.

4.1. List-based approach

The pseudo-code of our list-based approach, referred to as *list*, is shown in Algorithm 1. For each vertex v_i and its adjacent vertex v_j, we traverse their respective adjacency lists to find the number of common vertices between them, which is eventually added to the total triangle count. Since the triangles are enumerated using the $\langle v_i, v_j, v_k \rangle$ ordering, we keep track of v_j's position within v_i's advancing list and the scanning of v_i's list for computing the intersection starts from the position after v_j.

4.2. Map-based approaches

The overall structure of the first map-based approach that we developed, referred to as *hmap-ijk*, is shown in Algorithm 2.

Algorithm 1 List-based triangle counting approach

```
1: procedure TC-ListBased( $G(V, E^U)$ )
2:     $tc \leftarrow 0$                          ▷ Initialize triangle count in $G$
3:     for all $v_i \in V$ do
4:         for all $v_j \in \text{Adj}^>(v_i)$ do
5:             $tc \leftarrow tc + |\text{Adj}^>(v_i) \cap \text{Adj}^>(v_j)|$
6:     return $tc$                             ▷ Total triangle count in $G$
```

Algorithm 2 Triangle counting using map-based approach

```
1: procedure TC-MapBased( $G(V, E^U)$ )
2:     $tc \leftarrow 0$                          ▷ Initialize triangle count in $G$
3:     for $i \in \{1, \ldots, msz\}$ do       ▷ msz is the size of Map
4:         Map[$i$] $\leftarrow 0$
5:     for all $v_i \in V$ do
6:         for all $v_j \in \text{Adj}^>(v_i)$ do
7:             Map.hash($v_j$)
8:         for all $v_j \in \text{Adj}^>(v_i)$ do
9:             for all $v_k \in \text{Adj}^>(v_j)$ do
10:                if Map.exists($vk$) then
11:                    $tc \leftarrow tc + 1$
12:        for all $v_j \in \text{Adj}^>(v_i)$ do
13:            Map.delete($v_j$)
14:    return $tc$                             ▷ Total triangle count in $G$
```

This algorithm follows the $\langle v_i, v_j, v_k \rangle$ ordering scheme and uses a hash-map to mark the adjacent vertices of v_i. We adopt a fast hashing scheme based on bitwise & operation, which uses the lower l bits of the vertex id as the hash value. This scheme leads to a hash-map whose size is 2^l. We choose a power of two for the size of the hash-map that is just greater than the maximum number of non-zeros along the rows of the upper-triangular adjacency matrix. Furthermore, in order to reduce the collisions incurred while using linear probing, we increase the hash-map size by a factor of 16. Note that our algorithm just allocates that hash-map array once and re-uses it for all v_is.

Since the vertices are sorted in a non-decreasing degree order, for each edge (v_i, v_j) with $v_i < v_j$, the length of v_i's adjacency list will tend to be smaller than that of v_j's. Consequently, for all pairs of vertices whose adjacency lists need to be intersected, the *hmap-ijk* approach will store into the hash table the shorter adjacency list and use the longer adjacency list to probe it. Therefore, if v_j's adjacency list is considerably larger than that of v_i's, then it is more beneficial to employ the $\langle v_j, v_i, v_k \rangle$ ordering and hash v_j's adjacency list instead. To process vertices in this fashion, for each vertex v_j, we need to find all of its incident vertices v_i. This is achieved by creating a transpose of the directed graph, or, in other terms, the lower-triangular part of the adjacency matrix. In addition to this, we also encode offsets in this data structure to quickly locate the position in v_i's adjacency list that stores vertices that are numbered higher than v_j. This enables us to weed out unnecessary intersection operations, which would otherwise result in no common vertices. We refer to this approach as *hmap-jikv1*. Additionally, certain vertices are connected to all other vertices in the graph. These can be compared to the hub vertices that are found in real world graphs. On detecting such "full" vertices, we can simply take the degree of vertex v_i, $d(v_i)$, and add it to the triangle count. We decreased our operation count considerably with this optimization. This "full" vertex optimization is incorporated into the *list* approach as well.

Since the graph is re-ordered in non-decreasing degree, the high-degree vertices will be numbered last and those will appear as fairly dense columns at the end of the adjacency matrix. In order to speedup the intersection of these columns, we split the hash table into two parts, the *head* and the *tail*. The *tail* is implemented as a direct access array and stores the high-numbered vertices, whereas the *head* is implemented using the *hmap-jikv1* approach and stores the rest of the vertices. This approach allows us, without using too much memory and incurring large TLB costs, to quickly check if the most frequently occurring vertices are present in the map or not. To determine the size of the *tail* part, we perform a parameter search of the size based on the degree of the sorted vertices and capture it in an ad-hoc fashion. We refer to this version as *hmap-jikv2*.

4.3. Parallelization

We developed OpenMP-based shared-memory formulations of the serial algorithms described in the previous sections. Our parallelization effort focused both on the computations performed during triangle counting and those performed during pre-processing. The parallelization of the triangle-counting computations is quite straightforward. The work can be decomposed over the outermost loop as each of the vertices can be processed independently. In the case of the $\langle v_i, v_j, v_k\rangle$ ordering, the outermost loop involves vertex v_i, whereas in the $\langle v_j, v_i, v_k\rangle$ ordering, the outermost loop involves vertex v_j. As the vertices in a real-world graph exhibit skewed degree distributions, a dynamic partitioning of the vertices with a small chunk size handles the load imbalance that a static partitioning would otherwise introduce. The parallelization of the computations performed during pre-processing is considerably more challenging and involves the following: degree-based sorting, creating the re-ordered upper-triangular adjacency matrix, sorting the adjacency lists in increasing vertex id order, and transposing the upper-triangular matrix to obtain the lower-triangular matrix for the $\langle v_j, v_i, v_k\rangle$ enumeration order. We deployed various strategies for performing the above computations concurrently that (i) split the various computations into multiple phases so that subsequent phases can be done concurrently and (ii) utilize OpenMP atomics during the transpose operation.

4.4. Extending the optimizations to GraphMat

GraphMat is a parallel and distributed graph processing framework that uses a vertex programming frontend and a sparse matrix backend to enable both high performance and productivity for graph algorithms [11, 12]. In order to implement the map-based algorithms (Section 4.2) using GraphMat, which as our experiments will show are considerably faster than the list-based approach, we use CSR representation for the graphs instead of GraphMat's default DCSC representation since triangle counting requires the neighbor list of each vertex. We implement the map-based algorithm by implementing a graph program where each vertex sends its in-neighbors its vertex id and a pointer to its neighbor list (from the CSR representation). A vertex, on receiving the message, sets a thread local hash map to include all its neighbor and then compares the map against the incoming neighbor list for matches. Each match corresponds to a triangle. These counts are then reduced per-vertex and then across all vertices. This fits neatly in a vertex programming model. Since GraphMat already implements optimized generalized sparse matrix-sparse vector multiplication routines, it is able to utilize the backend optimization (multithreading, load balancing across threads etc.) without specialized optimizations for triangle counting.

TABLE 1: Datasets used in the experiments.

Graph	#vertices	#edges	#triangles
cit-Patents [10]	3,774,768	16,518,947	7,515,023
soc-orkut [13]	2,997,166	106,349,209	524,643,952
rmat22 [10]	2,393,285	64,097,004	2,067,392,370
rmat23 [10]	4,606,314	129,250,705	4,549,133,002
rmat24 [10]	8,860,450	260,261,843	9,936,161,560
rmat25 [10]	17,043,780	523,467,448	21,575,375,802
twitter [14]	41,652,230	1,202,513,046	34,824,916,864
friendster [10]	119,432,957	1,799,999,986	191,716

5. Experimental methodology

Datasets. We used several large-scale, real-world and synthetic graphs with varying degree distributions to evaluate the performance of our algorithms. Various statistics related to these graphs and the sources from where they were obtained are shown in Table 1. We converted all the graph datasets to undirected, simple graphs.

Experimental setup. Our experiments were conducted on three architectures: (i) A dual socket system with ten-core Intel Xeon E5-2650v3 (Haswell) processors, totaling to 20 cores and 40 hypertheads, with 25MB of last-level cache and 396GB of main memory. (ii) A dual socket twenty-two core Intel Xeon E5-2699v4 (Broadwell) processors, totaling to 44 cores and 88 hyperthreads, with 128GB of main memory. (iii) A 68-core Intel Xeon Phi CPU 7250 processors (KNL) and 272 hyperthreads, with 16GB of MCDRAM and 98GB of DDR4. Our programs, developed using C and OpenMP, were compiled using GCC (v4.9.2) on Haswell and Intel C/C++ compilers (v17.0.3) on KNL with -O3 optimization. GraphMat was compiled with Intel C/C++ compilers (v17.0.2). Additionally, we set environment variables `OMP_PROC_BIND=TRUE` and `KMP_AFFINITY=granularity=fine,compact,1`

TABLE 2: Serial runtimes on Haswell and KNL.

	Haswell								KNL							
	list		hmap-ijk		hmap-jikv1		hmap-jikv2		list		hmap-ijk		hmap-jikv1		hmap-jikv2	
Graph	ppt	tct	ppt	tct	ppt	tct	ppt	tct	ppt	tct	ppt	tct	ppt	tct	ppt	tct
cit-Patents	0.8	1.0	0.8	0.9	1.7	0.9	1.7	0.7	2.5	4.7	2.5	5.1	5.2	5.6	5.2	5.1
soc-orkut	4.6	48.1	4.5	32.3	11.1	17.9	11.1	18.4	14.3	130.0	14.2	105.3	30.1	54.8	30.0	70.9
rmat22	3.0	101.0	2.9	75.0	5.1	16.7	5.2	15.9	9.1	286.8	9.1	253.7	13.9	36.6	13.9	37.7
rmat23	6.2	249.3	6.3	185.0	11.0	41.1	10.8	37.6	18.7	708.5	18.7	627.4	30.2	92.7	30.2	88.2
rmat24	13.0	606.3	13.0	450.9	23.9	100.2	24.1	87.9	38.2	1745.5	38.2	1550.8	65.2	244.3	65.3	207.3
rmat25	27.4	1488.9	27.3	1112.7	53.2	241.2	53.3	205.2	78.2	4289.6	78.2	3854.1	140.8	481.2	139.2	486.2
twitter	56.4	1642.7	56.3	1218.0	123.8	434.5	123.5	404.4	152.1	4342.3	151.9	3993.4	302.9	902.9	301.4	952.3
friendster	100.9	752.2	101.2	469.7	261.4	332.8	261.4	334.0	263.4	2079.6	263.4	1652.6	679.4	967.9	678.4	1242.1

The columns labeled "ppt" show the time (in seconds) required by the pre-processing phase.
The columns labeled "tct" show the time (in seconds) required by the triangle counting phase.

to bind threads to cores in Xeon and Xeon Phi, respectively. In all our experiments we used a chunk size of 16 to dynamically partition the outermost loop's iterations.

Comparison algorithms. We used *OrderedMerge* by Shun and Tangwongsan [7] to compare against our implementations. It was compiled with the same versions of GCC and Intel C/C++ compilers on Haswell and KNL, respectively. In addition to this, we ran the serial baselines provided by GraphChallenge website [10] to compare our runtimes against that of *minitri*, a simple, triangle-based data analytics code [15] on Haswell.

6. Results and discussion

6.1. Serial performance

Table 2 shows the amount of time required by our triangle-counting algorithms on Haswell and KNL for the different graphs. Focusing on the performance on Haswell, we see that for all but the smallest *cit-Patents* graph, the triangle-counting time of *hmap-ijk* is 32%–60% lower than that of *list*. Similar trends are observed on KNL, but *hmap-ijk*'s performance advantage is somewhat lower, ranging from 9% to 25%. These results show that the hash-map approach used by *hmap-ijk* is more efficient in computing the intersection of the adjacency lists over list-traversal and that it benefits from Haswell's larger caches.

Comparing the performance of the $\langle v_j, v_i, v_k \rangle$ enumeration order over $\langle v_i, v_j, v_k \rangle$, we see that the former leads to significantly lower runtimes. With the exception of *cit-Patents*, the triangle-counting time of *hmap-jikv1* is considerably smaller than that of *hmap-ijk*. On Haswell, it is $4.1\times$–$4.4\times$ faster for the rmat graphs and $1.4\times$–$2.8\times$ faster for the other graphs, whereas on KNL the corresponding figures are $6.8\times$–$8.0\times$ and $1.7\times$–$4.4\times$, respectively. KNL's higher gains can be attributed to the fact that *hmap-jikv1* does a better job in utilizing the KNL's smaller per-core caches than *hmap-ijk*.

Comparing the two variants of the $\langle v_j, v_i, v_k \rangle$ enumeration order on Haswell, we see that *hmap-jikv2* leads to a 1%–17% lower triangle-counting times over *hmap-jikv1*. However, *hmap-jikv2*'s performance on KNL is mixed. For the rmat graphs, it is 8%–17% faster than *hmap-jikv1*, whereas, for the real-world graphs, *hmap-jikv1* is 4%–29% faster than *hmap-jikv2*. KNL's mixed performance can be partially attributed to the fact that the denser portions of the real-world graphs are smaller; thus, limiting the potential gains of using the *tail* array in *hmap-jikv2*. This behaviour is also observed with some graphs on Haswell. However, this is something that we are still investigating.

Finally, comparing the pre-processing times of the different schemes, we see that the variants using the $\langle v_j, v_i, v_k \rangle$ enumeration ordering, require more time than the other two. This is due to the additional cost of transposing the upper-triangular adjacency matrix. Also note that as our various optimizations were able to significantly reduce the time required for triangle counting, the pre-processing time ends up requiring a significant fraction of the overall runtime.

TABLE 3: Parallel runtimes and speedups on Haswell and KNL.

	Haswell			KNL		
Graph	best hmap serial times	best hmap parallel times	best hmap parallel speedup	best hmap serial times	best hmap parallel times	best hmap parallel speedup
cit-Patents	2.41^b	0.38^a	6.3×	10.22^b	0.62^a	16.5×
soc-orkut	28.98^a	2.08^a	13.9×	84.90^a	1.91^a	44.5×
rmat22	21.06^b	1.74^b	12.1×	50.53^a	1.43^b	35.3×
rmat23	48.38^b	3.91^b	12.4×	118.38^b	2.65^b	44.7×
rmat24	112.01^b	8.79^b	12.7×	272.65^b	5.60^b	48.7×
rmat25	258.48^b	20.00^b	12.9×	622.05^a	12.25^b	50.7×
twitter	527.83^b	47.16^b	11.2×	1205.83^a	32.95^a	36.6×
friendster	594.24^a	49.62^b	11.9×	1647.33^a	35.24^a	46.8×

The speedups obtained by the best parallel *hmap-jik* version over the best serial *hmap-jik* version on Haswell and KNL.
On Haswell, the best times/speedups were obtained using 40 threads and 136 threads on KNL.
The speedups are computed by taking into account the overall time required by the algorithm, which includes the pre-processing and the triangle-counting times.
The superscript ^{a} denotes that *hmap-jikv1* is the best method, whereas the superscript ^{b} denotes that *hmap-jikv2* is the best method.
The times are in seconds.

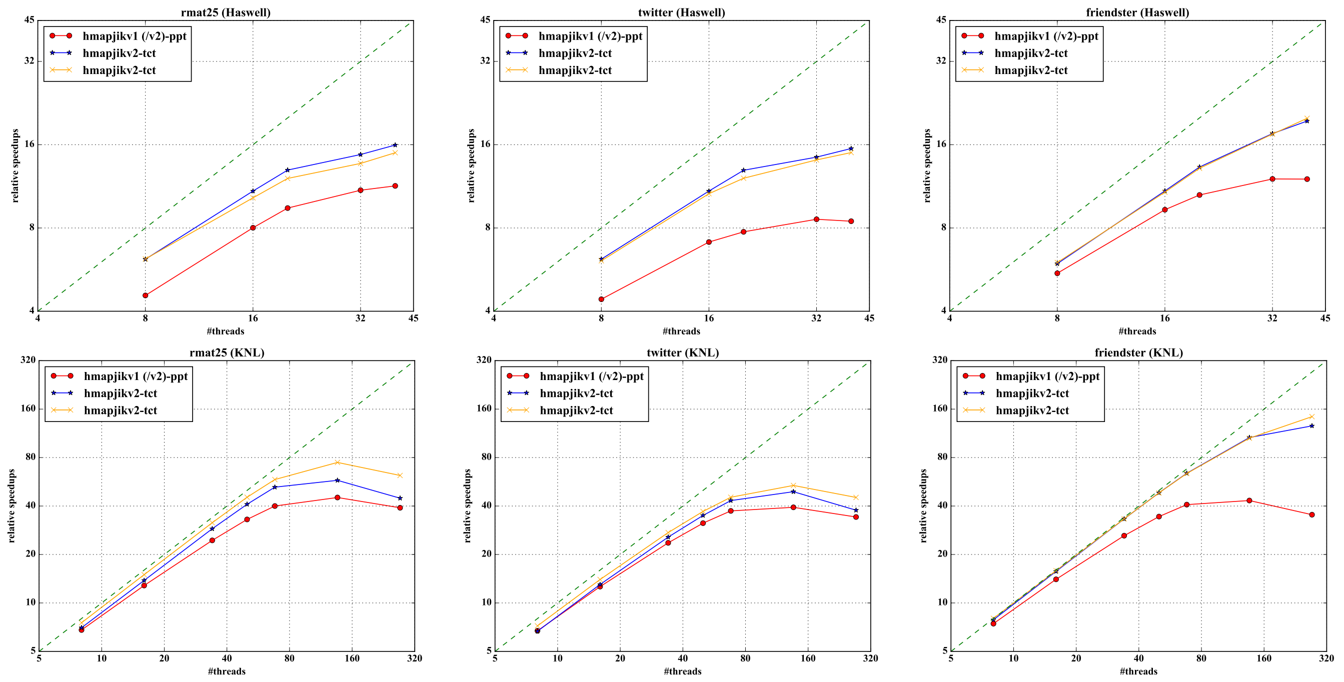

Figure 1: A log-log plot demonstrating the parallel scaling of *hmap-jikv1* and *hmap-jikv2* on Haswell and KNL. The scaling of pre-processing (ppt) and triangle counting (tct) times are shown separately. The scaling of the pre-processing step is plotted only once as it is the same for both *hmap-jikv1* and *hmap-jikv2*. The runtime of the parallel algorithm on one thread was used as a baseline to compute the speedups.

6.2. Parallel performance and scaling

Table 3 shows the best overall speedups obtained by our OpenMP formulations of the *hmap-jikv1* and *hmap-jikv2* methods on Haswell and KNL. Compared to our best performing serial baselines, our parallel formulations are on average 40.48× faster on KNL and 11.68× faster on Haswell.

Figure 1 plots the speedups achieved by *hmap-jikv1* and *hmap-jikv2* for the *twitter*, *friendster* and *rmat25* graphs. Both *hmap-jikv1* and *hmap-jikv2* demonstrate similar scaling capabilities. We also note a slight drop in the speedups on KNL with 272 threads due to the effect of hyperthreading.

6.3. GraphMat performance

Table 4 shows GraphMat's parallel runtimes on Broadwell using 88 threads. For the core triangle counting kernel, GraphMat is only $\sim 20\%$ slower than the corresponding *hmat-jikv1* algorithm for the *rmat25* graph, demonstrating that our algorithm generalizes to other graph frameworks (and GraphBLAS) as well. GraphMat spends more time on pre-processing compared to the native implementation. This is because GraphMat only reads files in mtx format, which makes pre-processing and CSR construction more expensive. This can be rectified in the future by supporting other input format in GraphMat.

TABLE 4: GraphMat triangle counting runtimes on Broadwell.

Graph	ppt	tct	total
cit-Patents	1.24	0.21	1.45
soc-orkut	5.05	0.77	5.82
rmat22	3.86	0.84	4.70
rmat23	7.53	1.82	9.35
rmat24	15.25	4.21	19.46
rmat25	30.97	9.43	40.40
twitter	95.44	24.37	119.81

"ppt" is the pre-processing time in seconds.
"tct" is the triangle-counting time in seconds.
"total" is ppt+tct.

6.4. Comparison with previous approaches

Comparing the serial baseline provided on the GraphChallenge website [10], *minitri* [15] took 229.656 seconds to count the triangles in *cit-Patents* on Haswell. Our best-performing serial algorithm on that dataset requires 1.7 seconds (*hmap-ijk*), which corresponds to a 135× speedup. On the *rmat22* graph, *minitri* required more than two hours (we killed the process after that time), whereas our best performing serial algorithm required just 21.06 seconds (*hmap-jikv2*). We did not run more experiments using *minitri* due to the long running hours.

We compare our work with the *OrderedMerge* algorithm as it achieves the best overall performance [7]. Table 5 shows the results of these comparisons. These results show

TABLE 5: Comparisons with the state of the art on Haswell and KNL.

	Haswell					KNL				
Graph	best hmap serial	best hmap parallel	ordered-merge serial	ordered-merge parallel	hmap parallel speedup	best hmap serial	best hmap parallel	ordered-merge serial	ordered-merge parallel	hmap parallel speedup
cit-Patents	2.41^{b}	0.38^{a}	1.97	0.17		10.22^{b}	0.62^{a}	6.74	0.09	
soc-orkut	28.98^{a}	2.08^{a}	64.45	8.36	4.02×	84.90^{a}	1.91^{a}	126.00	1.75	
rmat22	21.06^{b}	1.74^{b}	153.50	13.30	7.64×	50.53^{a}	1.43^{b}	324.00	4.75	3.32×
rmat23	48.38^{b}	3.91^{b}	380.00	35.87	9.17×	118.38^{b}	2.65^{b}	796.50	10.97	4.14×
rmat24	112.01^{b}	8.79^{b}	916.50	87.23	9.92×	272.65^{b}	5.60^{b}	1950.00	26.83	4.79×
rmat25	258.48^{b}	20.00^{b}	2210.00	211.00	10.55×	622.05^{a}	12.25^{b}	4750.00	66.90	5.46×
twitter	527.83^{b}	47.16^{b}	2050.00	1250.00	26.51×	1205.83^{a}	32.95^{a}	4910.00	69.70	2.12×
friendster	594.24^{a}	49.62^{b}	610.00	57.50	1.16×	1647.33^{a}	35.24^{a}			

Serial and parallel runtimes in seconds of our best-performing map-based algorithms and the best-performing algorithm (ordered-merge) of [7].
The reported times are the overall time required by the algorithms, which include the pre-processing and the triangle-counting times.
The speedup is computed by dividing the parallel runtime of our algorithms against the parallel runtime of ordered-merge.
On Haswell, the parallel results for our approaches and ordered-merge were obtained using 40 threads. On KNL, the parallel results for our approaches were obtained using 136 threads and 272 threads for ordered-merge as that number of threads achieved the best performance.
Underlined entries correspond to the best-performing scheme.
The superscript ^{a} denotes that *hmap-jikv1* is the best method, whereas the superscript ^{b} denotes that *hmap-jikv2* is the best method.

that with the exception of the small graphs, the serial runtimes of our map-based algorithms are considerably faster than the corresponding runtimes of *OrderedMerge*. Moreover, comparing the times of *OrderedMerge* against our list-based approach (Table 2) we can see that *list* is faster than *OrderedMerge* as well. The performance advantage of our approaches also hold with respect to the parallel runtimes. However, we observe that the parallel runtime of *hmap-jikv2* on *friendster* [10] is very close to that of *OrderedMerge*. As *friendster* is an extremely sparse graph, with very few triangles for a graph of its scale, the pre-processing steps performed does not greatly benefit the triangle counting computation. Thus, pre-processing dominates the overall runtime and leads to similar runtimes between *OrderedMerge* and *hmapjik-v2*. Furthermore, because the pre-processing steps of our $\langle v_j, v_i, v_k \rangle$ enumeration approaches have not yet been parallelized as effectively as the triangle-counting step, the relative parallel performance advantage of our algorithms on KNL is somewhat lower than that of the corresponding serial algorithms. This is shown in Figure 1.

Because we did not have access to the code of [8], we were not able to perform direct comparisons. However, according to the results in [8], their triangle counting algorithms outperforms *OrderedMerge* by 2.33× on a dual-socket 8-core Intel Xeon E5 2680 system, and 2.44× (geometric mean of speedups) on Intel Xeon Phi SE10P co-processor [8]. This suggests that our tuned and optimized approaches perform better than the algorithms in [8].

7. Conclusion

In this paper, we presented parallel triangle counting algorithms that are tuned to utilize features of modern processor architectures and characteristics of large sparse graphs, for efficient processing on shared-memory platforms. Our comprehensive experimental evaluations showed that these algorithms achieve efficient performance and strong scaling on Intel Xeon processors. We also demonstrate that these algorithms can be generalized to other graph processing frameworks by porting them to GraphMat.

Acknowledgments

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

References

[1] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[2] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[3] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theoretical Computer Science*, vol. 407, no. 1-3, pp. 458–473, 2008.

[4] N. Shrivastava, A. Majumder, and R. Rastogi, "Mining (social) network graphs to detect random link attacks," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on.* IEEE, 2008, pp. 486–495.

[5] S. Arifuzzaman, M. Khan, and M. Marathe, "Patric: A parallel algorithm for counting triangles in massive networks," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management.* ACM, 2013, pp. 529–538.

[6] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms.* IEEE Press, 2014, pp. 1–8.

[7] 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.

[8] S. Parimalarangan, G. M. Slota, and K. Madduri, "Fast parallel triad census and triangle listing on shared-memory platforms," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2017 IEEE International.* IEEE, 2017.

[9] S. Arifuzzaman, M. Khan, and M. Marathe, “Distributed-memory parallel algorithms for counting and listing triangles in big graphs,” *arXiv preprint arXiv:1706.05151*, 2017.

[10] 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,” *IEEE HPEC*, 2017.

[11] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015. [Online]. Available: http://dx.doi.org/10.14778/2809974.2809983

[12] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, “Graphpad: Optimized graph primitives for parallel and distributed platforms,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 313–322.

[13] R. A. Rossi and N. K. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. [Online]. Available: http://networkrepository.com

[14] H. Kwak, C. Lee, H. Park, and S. Moon, “What is twitter, a social network or a news media?” in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.

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