Preliminary Exploration of Large-Scale Triangle Counting on Shared-Memory Multicore System

Jiyuan Zhang[1], Daniele G. Spampinato [1], Scott McMillan[2], Franz Franchetti[1]
[1]*Electrical and Computer Engineering Department*
[2]*Software Engineering Institute*
Carnegie Mellon University, Pittsburgh, USA
jiyuanz, spampinato, franzf@andrew.cmu.edu, smcmillam@sei.cmu.edu

***Abstract*—As triangle counting is becoming a widely-used building block for a large amount of graph analytics applications, there is a growing need to make it run fast and scalable on large parallel systems. In this work we conduct a preliminary exploration on the optimizations of triangle counting algorithms on shared-memory system with large dataset.**

I. Introduction

Graph analytics is becoming increasingly important in a growing number of domains and there is a need to build fast and scalable systems for graph analytics. Nowadays, there is a growing trend for the design of graph analytics engines for shared-memory systems [1]–[3]. Shared-memory system has lower communication costs and lower data access latency. This can potentially lead to better performance compared with distributed memory systems. On top of that, a state-of-art high-end multi-core machine can integrate hundreds of cores, and terabytes of memory [4]. This further enables shared memory systems to process large-scale datasets in memory.

In this work, we explore one of the basic applications in graph analytics: triangle counting. Triangle counting is one of the frequently computed building block operation in graph analytics, therefore it is important to make it run fast and scalable to large systems. There are rich amount of graph frameworks that can target a general set of graph applications. Those frameworks have different frontend designs [1]–[3], [5]–[7]. For example, [5] is based on vertex-iterator programming model where a utility function is supplied and executed on each vertex, [6] uses linear algebra language as primitives. And others are based on domain-specific languages, etc. However, those graph framework's target are the broad set of graph applications, their backend optimizations may be too general to be effective for triangle countings.

In this work, we plan to study the optimizations specific for triangle counting on the shared-memory systems. There are different algorithms for triangle counting. And for these algorithms, one could have different implementations, including a variety of hashing and merging tweaks. In order to get optimal performance for triangle counting, it is essential to consider system characteristics, as well as input graph's properties to find the effective optimization method. This paper conducts a preliminary explorations on whether such optimizations can be effective for large-scale triangle counting on shared-memory multi-core systems. The structure of the paper is as follows: the discussion of the triangle counting algorithm is presented in section II. Then the baseline parallel implementation is given in section III. Section IV discusses a variety of optimization techniques and analyzes whether they can be effective in practice. The performance results are presented in section V.

II. Algorithm

The straight-forward algorithm for triangle counting using the linear algebra language can be illustrated as [8], [9]:

$$A^2 \circ A \tag{1}$$

where A is the adjacency matrix of the input graph. The square of A gives all the wedges (connected two edges) and the element-wise multiplication with A closes the wedge and forms a triangle. Afterwards, the algorithm sums over each element in the resultant matrix. However, in this algorithm each triangle will be counted repetitively for six times. An improvement to reduce the double counting in the baseline algorithm can use half the adjacency matrix. In this way, each triangle will only be counted once.

There is an algorithm that can further reduce the triangle counting complexity. The compact-forward algorithm in [10] can outperform the above two baseline algorithms by greatly reducing the number of false positives. The algorithm consists of two steps. The first step is direction assignment: it assigns a direction to each undirected edge such that the edge points from the low-degree vertex to the high-degree vertex. Then the second step counts the number of triangles based on the directed graph. There are two benefits of introducing a direction to the edge: First of all, it can avoid double counting. Second, it reduces the (out-)degree of nodes, especially those with high degrees. As the counting has the quadratic complexity to the vertex's (out-)degree and the high (out-)degree vertex dominates the computation among all the vertices, this can greatly reduce the complexity.

III. Baseline Parallel triangle counting

The most time consuming part of the compact forward algorithm is the second step—counting triangles from the directed graph. The peudocode for this step is shown in Algorithm 1. In this step, the algorithm iterates every vertex (v), and for each one of its (directed) neighbors u, sums up the number of common neighbors. We first run a baseline parallel triangle counting algorithm on the shared memory systems. The baseline implementation is based on [11]. The implementation simply parallelizes the loop among vertex v (line 3) and the subsequent loop among the neighborhood of v (line 4).

Algorithm 1 count triangles in directed graph

```
1: procedure TriCount($G$)    ▷ G is the directed graph
2:     $count \leftarrow 0$
3:     for $v \in G.vertices$ do
4:         for $u \in v.neighbors$ do
5:             $count+ =$common neighbors of $u$ and $v$
6:         end for
7:     end for
8:     return $count$
9: end procedure
```

In terms of the counting phase in line 5, the implementation has explored two basic ways to implement counting — hashing-based method and sort-merge based method. The author finds the sort-merge based implementation is faster than the hash-based method.

IV. Preliminary Exploration on Optimizations

Counting the common elements of two vertex's neighbor list is the most time consuming computation. In this section, we will mostly focus on the optimizations for the counting phase in line 5. Counting the common elements of two neighbor lists is essentially founding out the intersecting elements between two sets. Therefore optimizations explored previously in set-intersection problems may be applied here.

Algorithm 2 count common elements in two sorted lists

```
1: procedure CountCommon($A, B$)    ▷ The A, B are two sorted lists
2:     $i \leftarrow 0, j \leftarrow 0, counts \leftarrow 0$
3:     while $i < A.size$ AND $j < B.size$ do
4:         if $A[i] < B[j]$ then
5:             $i{++}$
6:         else if $A[i] > B[j]$ then
7:             $j{++}$
8:         else
9:             $i{++}$, $j{++}$, $count{++}$
10:        end if
11:    end while
12:    return $count$
13: end procedure
```

*a) **Hashing**:* One technique to optimize the set intersection operation is to use hashing. The hashing method maps each vertex's neighborhood into a corresponding hash table. To count the common elements in two vertex's neighborhood, we can iterate through the elements in the smaller neighborhood list and probes into the hash table of the bigger neighborhood. The cost of the hashing based set intersection method is $min(n_1, n_2)$, where n_1 is the size of small list and n_2 is the size of the large list.

*b) **Merging**:* is another technique to count the common elements in two lists. It first sorts each vertex's neighborhood list in increasing order and counts the common elements of two sorted list by linear scan through them. The baseline code to count the intersection of two sorted lists via merging method is shown in Algorithm 2. The cost for the merging based set intersection is $max(n_1, n_2)$.

*c) **Exploiting binary search to accelerate merging**:* When the two list size is quite different, we can binary search the smaller list in the large one. The cost of the counting phase can be reduced to $n_1 log(n_2)$.

*d) **Exploiting SIMD to accelerate merging**:* There has been plenty of work exploring how to utilize the SIMD unit to accelerate set intersection computation [12]–[14]. We implemented a baseline SIMD algorithm based on [14]. Assume the system SIMD width is four-elements, the SIMD intersection algorithm is illustrated with an example in Figure. 1. In this example, during the first iteration the first four-element of list A and B will be compared (the instructions used will be explained subsequently). Based on comparison of the tail elements (i.e. the 4th element), one of the pointers is decided to move. In the example case, pointer of the list B will advance to the next 4-element block because its 4th element is smaller than list A. In second iteration, the first four elements of A are compared with the second 4-element block of B and finally the second 4-element blocks of A and B are compared.

On CPUs with AVX instruction set architecture, the algorithm exemplified in Fig. 1 can be implemented using _mm_cmpeq and _mm_shuffle intrinsics. _mm_cmpeq can compare multiple elements at once and produce a resulting bit mask. In order to conduct all-to-all comparisons between A and B, only one _mm_cmpeq is not enough — the SIMD block of B needs to left-shift using _mm_shuffle intrinsics and then compare with A again till the right-most element in B's element block is compared with the left-most element of A. The process is illustrated in Fig. 2.

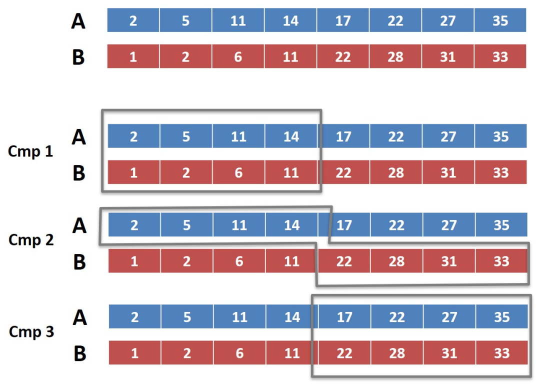


Fig. 1: SIMD algorithm assuming simd width is 4 elements.

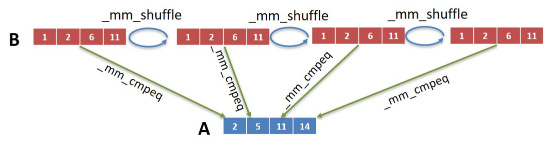


Fig. 2: SIMD algorithm illusion.

*e) **hybrid hashing and merging**:* A hybrid method represents a node's neighborhood with a hybrid structure between sparse list and hash table. The hybrid method partitions the data range into fix-sized (K) blocks. Each block is indexed by its base value and has an associated bit-vector of length K. Each bit of the bit-vector indicates the existence of an element on that position. For example, given a list {0, 1, 4, 52, 102, 493, 534} and K as 64, the list can be compressed as {0, 64, 448, 512}. We can see the size of the list reduces from 8 to 4. In this way, the number of comparisons can be reduced subsequently. To count the common elements in the compressed block list, it firstly scan through the block lists to find blocks with common base values via the merge-sort methods and then count the common elements in the two blocks by comparing the two corresponding indicator vectors.

A. Profiling and Analysis

Whether the above algorithms will work largely depends on the property of the dataset. In this section, we start with the profiling results of some representative graphs datasets in order to exam the effectiveness of the above techniques.

Cit-Patents. This dataset has 3,774,768 vertices and 16,518,947 edges. The original maximum degree 793. The maximum degree after sorting is 73. The degree distribution is shown in Fig. 5. We can see that after direction assignment, the out-degree distribution still follows a power-law distribution. Majority of vertices has small out-degree.

Friendster. This dataset has 65,608,366 vertices and 1,806,067,135 edges. The original maximum degree among all the nodes is 5214. The maximum out-degree in the directed graph is 2389. And their distribution is in Fig. 6.

Graph500-scale23. This dataset has 4,606,314 vertices and 129,250,705 edges. The original maximum degree 272176. The maximum degree after sorting is 1376. The degree distribution is shown in Fig. 7.

We highlight the following observations based on the profilings:

Observation 1. In the directed graph, the number of nodes with high out-degrees is small. However, intersections between those high out-degree nodes' neighbor list dominates the total run-time. And this is not difficult to verify. If we assume that after reordering the rows from highest degree to lowest degree in the adjacent matrix of the graph, the number of nonzero element(nnz, equivalently out-degree) follows a power-law distribution over the rows, then the comparing cost per row will also be skewed and resemble a power-law distribution. As plotted out in Figure 3, we can see that if the nnz element follows a power-law distribution of x^{-1}, then in all the two list(a.k.a. rows) comparisons, the top 40% rows takes 50% of the total comparison cost. If the nnz element distribution is x^{-2}, then the top 10% rows will take up over 50% of the total comparison cost. In general, more skewed the nnz element distribution is in the graph's adjacent matrix, the more skewed the computation cost will be towards the dense part of the matrix. Figure 4 shows the real time and computation cost distribution on some given datasets. Consistent with the theoretical results, the computation cost is also mostly centered among the beginning rows with larger out degrees.

Observation 2. The density of the high-degree nodes' neighbor lists is still very sparse. Each node's neighbor list is a sequence of numbers whose value lies in the range from 0 to $N-1$. We introduce the metric *Density* to measure the sparsity of a vertex's neighborhood vector. Density is the number of elements divided by their maximum value. The smaller the density, the sparser the neighborhood list is. Knowing the density is helpful to determine the hashing parameters as for a sparser neighborhood list, it may require a larger hash table. In the directed graph, usually the vertices with higher degree have higher density. For the scale23 graph, the highest density neighborhood is around 0.763. The top 10% high out-degree vertex has density between $0.1-0.7$. For the friendster graph, the highest density 0.01. The top 10 percent high out-degree vertex has density between $0.001 - 0.01$. We can see that as a graph scales, the density of the vertex's neighbor can be sparser.

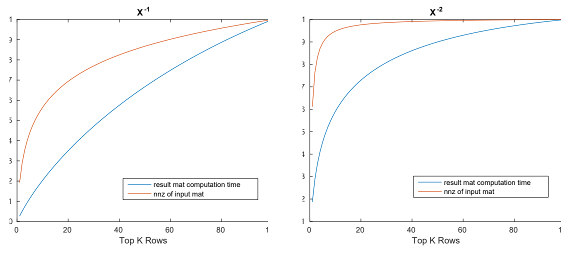
B. Understanding the optimizations

*a) **Hashing**:* Using hashing method, we didn't see any improvement on the speed. Although the complexity of hashing method is $min(n1, n2)$. However, this is most effective when there is at least an order of magnitude difference between nA and nB (i.e, $max(n1, n1) \gg min(n1, n2)$). But profilng shows that the comparison cases where $max(n1, n2) > 10 * min(n1, n2)$ only makes up a small part in all the two-pair list comparisons, according to our observation 1. In other words, majority of time is spent to compare lists $n1 \approx n2$ and both $n1$ and $n2$ are large. Therefore, hashing technique that is helpful to accelerate a dense list versus a sparse list is unable to observe performance improvement.

*b) **Exploiting binary search to accelerate merging**:* Similar to the hashing method, the binary search method does not accelerate the computation, because the binary search also requires the size of two sorted lists to be very different in order to have observable speedup.

*c) **Exploiting SIMD to accelerate merging**:* This method can be applied regardless of the relative size of the two lists. Therefore, it can be effective to accelerate the CountCommon process of two big lists. And we will show the results in the next section.

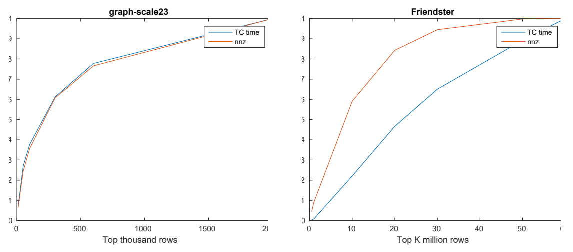
*d) **Hybrid hashing and merging**:* Whether the Hybrid hashing and merging can be effective highly depends on the density of the neighbor list: whether the list has elements condensed together that can be compressed. Otherwise the size reduction in the compressed block list is not sufficient.



(a) Power-law parameter of 1.

(b) Power-law parameter of 1.

Fig. 3: Theoretical computation distribution and non-zero element distribution on graphs with different skewness.



(a) Scale23 graph dataset.

(b) friendster dataset.

Fig. 4: Number of Non-zero element distribution and computation time distribution across rows.

We studied the Friendster dataset and the result suggests the original neighbor list is not friendly for such compressing. As shown in Table I, the base list size after 256bit compression only reduces from 1,806,067,135 elements to 1,690,408,139 elements (about 7% reduction). Moreover, the comparison operation then becomes more expensive when there is a match in the base value. One needs very delicate design in order to draw benefits from such methods.

TABLE I: Compression savings on Friendster dataset. Its original edge list size is 1,806,067,135.

	64bit	128bit	256bit
Cmpr. list size	1,723,833,467	1,711,082,186	1,690,408,139
Cmpr. Saving	5%	6%	7%

V. Performance Results

Hardware The systems used in our experiments were supported by Pittsburgh Supercomputing Center [4], [15], [16]. The experiment was tested on three types of hardware settings:

- *Single Core*: A single core out of the small multicore system, used to study the scalar performance.
- *Small multi-core*: A small multicore system which has two Intel Haswell (E5-2695 v3) CPUs(sockets) and each CPU has 14 cores.
- *Large multi-core*:A large shared-memory multicore system from HPE Integrity Superdome X. It has 16 Intel Xeon E7-8880 v4 CPUs(sockets) with 22 cores per CPU socket, 55MB Last Level Cache.

Results We present the performance results on the above three systems in Table II-IV.

Single Core: Table II shows the speedup of our SIMD-based set intersection over a scalar implementation on a single core (Intel Haswell (E5-2695 v3)) using the small dataset. We can see that SIMD can bring about 1.2x to over 3x times speedups over scalar implementation.

Small multi-core: Table III shows the SIMD speedup on the small multicore system with media-size dataset. Similar to single-core performance, the result shows that our SIMD implementation is about 1.4x to 3.6x speedup on this multicore system.

Large multi-core: Table IV shows the performance on the large shared-memory multicore system with large dataset. Those large datasets are generated using the kronecker graph generator [17]. The kron35 dataset runs over 2-days and unfinished by the given submission time frame. We are unable give the execution time. We estimate the time will be between 60 hours to 100 hours.

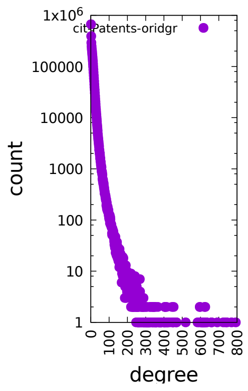
Performance comparison with distributed system. Last year graph challenge champion Pearce, et al. [18] presented their triangle counting performance on distributed systems. They used up to 256 nodes where each node has 24 cores. Their total number of cores is higher than ours. Therefore their overall execution time is shorter. When it comes to performance per-core. The *triangle processed per second per core* they achieved is ***1.9MTPS*** at highest (for WDC dataset, they counted 9.65T triangles in 808.7s on 256 node where each node has 24 cores). Our implementation can get ***3.1MTPS*** per core.

TABLE II: Single-core performance scalar vs. SIMD

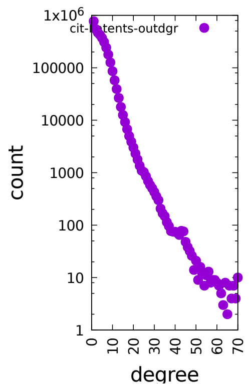
Dataset	V	E	T	Scalar time(s)	SIMD time(s)	SIMD speedup
cit-HepTh	27,770	352,285	1,478,735	0.08	0.03	1.22
cit-Patents	3,774,768	16.518,947	7,515,023	1.23	1.01	2.66
flickrEdges	105,938	2,316,948	107,987,357	1.57	0.569	2.76
graph500-scale18	174,147	3,800,348	82,287,285	3.35	1.17	2.86
graph500-scale19	335,318	7,729,675	186,288,972	8.51	2.84	3.01
graph500-scale20	645,820	15,680,861	419,349,784	21.9	7.29	3.00
graph500-scale21	1,243,072	31,731,650	935,100,883	55.6	19.8	2.80
graph500-scale22	2,393,285	64,097,004	2,067,392,370	142	52.9	2.68

TABLE III: Small 28-core system performance

Dataset	V	E	T	Scalar time(s)	SIMD time(s)	SIMD speedup
Friendster	65,608,366	1,806,067,135	4,173,724,142	96.7	67.1	1.44
graph500-scale23	4,606,314	129,250,705	4,549,133,002	63	17.7	3.56
graph500-scale25	17,043,780	523,467,448	21,575,375,802	259	72	3.60

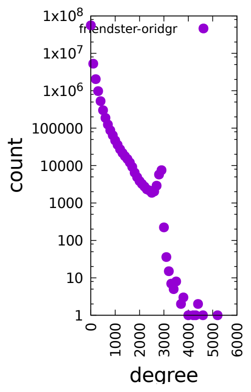


(a) Original degree distribution.

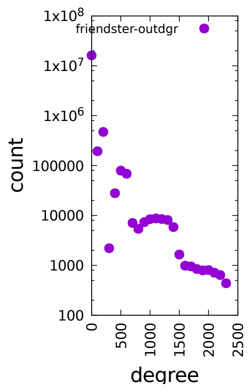


(b) Out-degree distribution after direction assignment

Fig. 5: Degree distribution of cit-Patents.

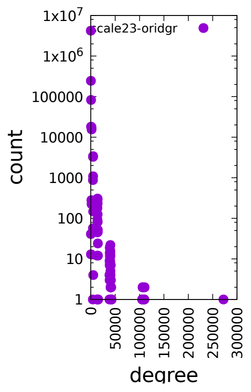


(a) Original degree distribution.

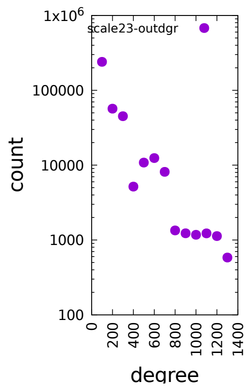


(b) Out-degree distribution after direction assignment.

Fig. 6: Degree distribution of friendster.



(a) Original degree distribution.



(b) Out-degree distribution after direction assignment.

Fig. 7: Degree distribution of graph500-scale23.

TABLE IV: Performance on a HP Superdome X system with 16 sockets, 352 cores

Dataset	V	E	T	Time(s)	Triangles/ (sec*core)
kron26	68,175,120	6,281,609,376	222,966,186,844	202	3.1M
kron31	1,090,801,920	33,501,916,672	1,380,824,051,328	2572	1.5M
kron35	1,380,546,180	1,256,845,342,648		>50h	

Acknowledgment

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC). We thank John Urbanic for his assistance with the system maintenance, which was made possible through the XSEDE Extended Collaborative Support Service (ECSS) program. This material is also 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 [DM18-1039].

References

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[2] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[3] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[4] Nicholas A. Nystrom, Michael J. Levine, Ralph Z. Roskies, and J. Ray Scott. Bridges: A uniquely flexible hpc resource for new communities and data analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 30:1–30:8, New York, NY, USA, 2015. ACM.

[5] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[6] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[7] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 5771–5780, 2018.

[8] Siddharth Samsi, Vijay Gadepally, Michael B. Hurley, Michael Jones, Edward K. Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Jeremy Kepner. Graphchallenge.org: Raising the bar on graph analytic performance. *CoRR*, abs/1805.09675, 2018.

[9] Matthew Lee Thom Popovici Franz Franchetti Tze-Meng Low, Varun Rao and S. McMillan. First look: Linear algebra-based triangle counting without matrix multiplication.

[10] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer, 2005.

[11] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 149–160. IEEE, 2015.

[12] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, 2016.

[13] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. Fast sorted-set intersection using SIMD instructions. In *ADMS@ VLDB*, pages 1–8, 2011.

[14] Fast intersection of sorted lists using SSE instructions. "https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/".

[15] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science and Engineering*, 16(5):62–74, Sept.-Oct. 2014.

[16] Wilkins-Diehr N, S Sanielevici, J Alameda, J Cazes, L Crosby, M Pierce, and R Roskies. In *High Performance Computer Applications 6th International Conference, ISUM 2015, Revised Selected Papers Gitler, Isidoro, Klapp, Jaime (Eds.)*, pages 3–13. Springer International Publishing, 2015.

[17] Jeremy Kepner, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Tim Davis, Vijay Gadepally, Michael Houle, Matthew Hubbell, Hayden Jananthan, et al. Design, generation, and validation of extreme scale power-law graphs. *arXiv preprint arXiv:1803.01281*, 2018.

[18] Roger Pearce. Triangle counting for scale-free graphs at scale in distributed memory.