# Triangle Counting with Cyclic Distributions

Andrew Lumsdaine*†, Luke Dalessandro‡, Kevin Deweese*, Jesun Firoz†, Scott McMillan§
*University of Washington, †Pacific Northwest National Lab, ‡Indiana University, §CMU/Software Engineering Institute
*{al75, deweeskg}@uw.edu, †{andrew.lumsdaine, jesun.firoz}@pnnl.gov, ‡ldalessa@iu.edu, §smcmillan@sei.cmu.edu

***Abstract*—Triangles are the simplest non-trivial subgraphs and triangle counting is used in a number of different applications. The order in which vertices are processed in triangle counting strongly effects the amount of work that needs to be done (and thus the overall performance). Ordering vertices by degree has been shown to be one particularly effective ordering approach. However, for graphs with skewed degree distributions (such as power-law graphs), ordering by degree effects the distribution of work; parallelization must account for this distribution in order to balance work among workers. In this paper we provide an in-depth analysis of the ramifications of degree-based ordering on parallel triangle counting. We present approach for partitioning work in triangle counting, based on cyclic distribution and some surprisingly simple C++ implementations. Experimental results demonstrate the effectiveness of our approach, particularly for power-law (and social network) graphs.**

## I. Introduction

At the 2017 HPEC conference, one of this paper's authors (AL) presented a simple C++ implementation of parallel triangle counting to demonstrate the capabilities of the C++ standard library and its parallelization mechanisms (e.g., asynchronous tasking). A challenge was issued to the community to develop a better performing, but similarly concise, implementation. The presented algorithm incorporated features that, when combined with degree-based ordering, results in a very fast simple implementation that is quite competitive when compared to highly optimized shared memory implementations (notably for graphs with highly skewed degree distributions), as will be shown in our contribution (Team NWGraph) to an upcoming paper in IISWC [1].

With the challenge algorithm as a jumping-off point, this paper takes a deeper look at what algorithmic and implementation details are necessary to obtain high performance in parallel triangle counting. We focus on the following: the triangle counting algorithm, the graph representation, ordering the graph by degree, efficient set intersection, and effective parallel load balancing. Although triangle counting is a fairly well-studied algorithm, much of the development and rigorous study has focused on sequential complexity. In addition, achieving high levels of performance in parallel also requires an approach to load balancing that interacts harmoniously with all of the other aspects of the algorithm.

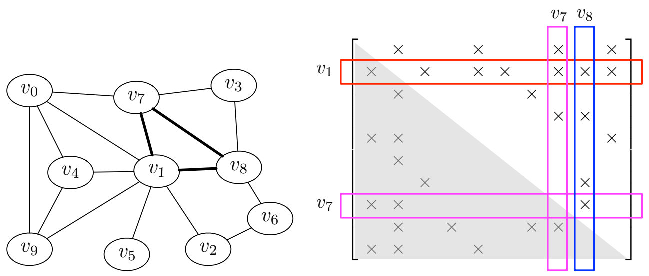


Fig. 1: Graph with triangle $\{v_1, v_7, v_8\}$ highlighted and corresponding representation as a sparse matrix.

## II. Background

### A. Triangle Counting

Given an undirected graph $G = \{V, E\}$, a *triangle* is a triple of vertices $v_i$, $v_j$, $v_k$, each in the vertex set $V$ such that the three edges $\{v_i, v_j\}$, $\{v_j, v_k\}$ and $\{v_k, v_i\}$ are in the edge set $E$. If we further require that $i < j < k$, each such triple is unique. Algorithms for *triangle counting* seek to find the cardinality of the set of unique triangles.

An exhaustive (and inefficient) algorithm is simply to examine all triples of edges:

```
$\alpha \leftarrow 0$
for each edge $\{v_i, v_j\} \in E, i < j$ do
    if $\exists k < i$ s.t. $\{v_k, v_i\} \in E$ and $\{v_k, v_j\} \in E$ then
        $\alpha \leftarrow \alpha + 1$
```

There are some important aspects of the problem structure that can be exploited to improve computational complexity of triangle counting; a thorough exposition of triangle counting algorithms can be found in [2]. Despite AL's (exaggerated) reputation for antipathy towards graph algorithms based on linear algebra, sparse matrix interpretations of triangle counting provide valuable intuition about these exploits.

In sparse matrix terms, a triangle exists if there are entries $(i, j)$, $(j, k)$, and $(k, i)$ in the sparse matrix representation of the graph. Since we have $i < j < k$, we only need to consider one triangle of the sparse matrix (we consider the upper triangle as shown in Figure 1, in which case we look for entries $(i, j)$, $(j, k)$, and $(i, k)$). That is, given a vertex $i$, we consider all of the edges $(i, j)$ to its neighbors with $i < j$. For each such neighbor $j$, we look at all of its neighbors $k$ as well as the remaining neighbors of $i$. If the same neighbor $k$ appears for both $i$ and and $j$, then there is a triangle $\{i, j, k\}$.

*Partially supported by NSF SI2-SSE Award 1716828 and DOE PSAAP II Award DE-NA0002377
†Funding for this work was partially provided by the High Performance Data Analytics (HPDA) program at Pacific Northwest National Laboratory. This research was also supported by PNNL's Segmented Global Address Space (SGAS) LDRD Initiative.
‡Partially supported by DOE PSAAP II Award DE-NA0002377
§This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [DM20-0658]

We thus have the following (the count is stored in $\alpha$):

```
$\alpha \leftarrow 0$
for $i = 0, 1 \dots |V| - 1$ do
    for Each neighbor $j$ of $i$, $j > i$ do
        $\alpha \leftarrow \alpha + |Nb(i,j) \bigcup Nb(j)|$
```

Here, $|Nb(i,j) \bigcup Nb(j)|$ means the number of intersections between the neighbors of $i$ (that are greater than $j$) and the neighbors of $j$.

## B. Implementation

This triangle counting algorithm can be realized in C++ as shown in Figure 2. The details are explained in Section IV but of note here is that the implementation is essentially a direct transliteration of the algorithm. Moreover, the implementation (as does the algorithm) makes no assumptions about the specific storage format of the graph – only that neighbor lists are stored as "forward ranges", that all of the neighbor lists are stored as a "random-access range", and that within each neighbor list, the vertex ids are stored in sorted order.

## C. Shared-memory Triangle Counting Algorithms

A plethora of works have been done in the context of triangle counting algorithms. For example, Shun et al. [3] presented cache-oblivious shared-memory triangle counting algorithms based on Latapy's sequential algorithms [2]. Subsequent work by Parimalarangan et al. [4] classified triangle counting algorithms into adjacency intersection (AI)-based and adjacency marking (AM)-based methods. They applied a canonical representation of the vertex triple $(u, v, w)$, based on degree ordering, to avoid counting triangles more than once. They enumerate possible triangle counting algorithms based on degree ordering. Similar approaches have been taken in [5] too. However, partitioning of work across different threads was left to the OpenMP scheduler (dynamic scheduling) in previous works. In this paper, we show that not only preprocessing such as vertex ordering is helpful for optimal work scheduling, but also distributing the workload in a cyclic manner results in the best performance of a triangle counting algorithm for an interesting set of input graphs.

In addition to the vertex-centric approaches to triangle counting, efficient linear algebra-based triangle counting ($(L \times L). * L$) have also been proposed ( [6]–[9]). However, linear algebra-based triangle counting imposes an additional memory requirement due to the temporary storage to compute intermediate sparse matrix-sparse matrix multiplication. Masking and specialized fusion operation are required too. Furthermore, in [8], distribution of work in terms of greedy block partitioning, similar to our balanced block partitioning, has been considered only with Cilk.

## III. High Performance Triangle Counting

While many triangle counting implementations rely on the basic idea shown in Figure 2, high performance triangle counting on large graphs will need additional improvements for reasonable performance. We aim to incorporate these improvements while not straying too far from the simplicity of Figure 2.

One such improvement is to triangularize the adjacency matrix. The information required to count all triangles is available in either the upper or lower triangular sections of the adjacency matrix so using the full adjacency matrix will require more work than necessary. Instead one can preprocess a graph to contain only successor edges, edges from vertices to neighbor vertices with a higher index. This corresponds to the upper triangular portion of the adjacency matrix. Alternatively one can use predecessor edges which correspond to the lower triangular portion of the adjacency matrix.

In order to minimize the total amount of work, it is often important to relabel vertices according to vertex degree. This is equivalent to permuting the rows and columns of the matrix (aka "diagonal pivoting"). While this can reduce the total amount of sequential work, the net effect is to group all of the high-degree vertices together, which can interfere with partition-based load balancing (the highest-degree nodes end up in the same partition). However, different workload distributions can be used to improve load balancing.

In the rest of this section we explore these ideas and illustrate their benefits on some smaller test graphs from the SuiteSparse matrix collection [10], whose sizes are shown in Figure 4. We analyze these approaches using the number of element comparisons in the set intersection step of triangle counting as a proxy for total work. Full timing experiments with large-scale graphs are given in Section Section V.

## A. Degree Reordering

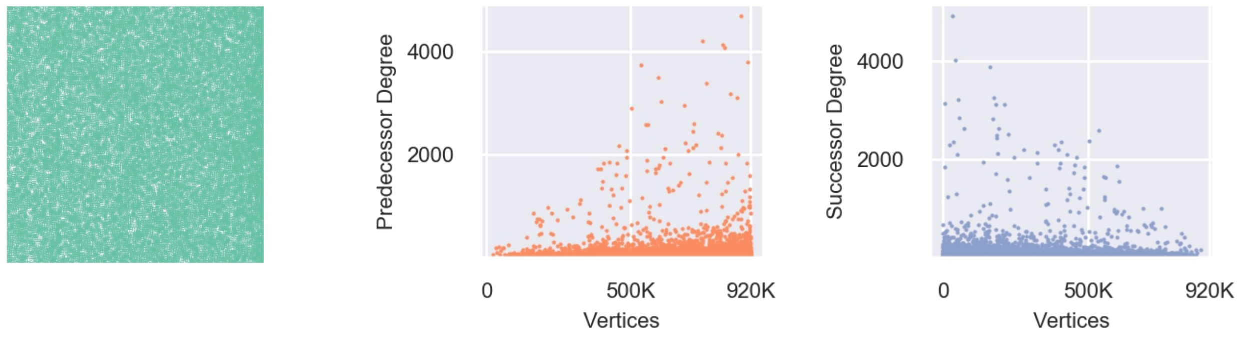
The goal of ordering graphs for efficient triangle counting is closely related to determining orderings for minimizing fill in sparse direct methods for LU factorization [11]. Popular heuristic algorithms for this include Reverse Cuthill-McKee [12], Modified Minimum Degree [13], and Approximate Minimum Degree [14]. Beyond fill minimization, ordering techniques have been applied to improve the performance of some graph kernels (such as BFS, Single-source shortest Paths (SSSP), PageRank and subgraph counting) in a distributed setting [15]. The effects of degree-based ordering on the performance of triangle counting have been noted by numerous authors (e.g., [2]–[4], [16]).

Reordering vertices can affect how many vertex comparisons are performed when intersecting two neighbor lists. The number of comparisons is a good proxy for the total amount of work required, thus reducing it can have a great affect on (sequential) total solution time. When processing successor lists (upper triangle shape), we relabel vertices so that they are in ascending order by degree (the highest-degree vertices go at the end). When processing predecessor lists (lower triangle shape), we relabel vertices so that the are in descending order by degree (the highest-degree vertices go at the beginning).

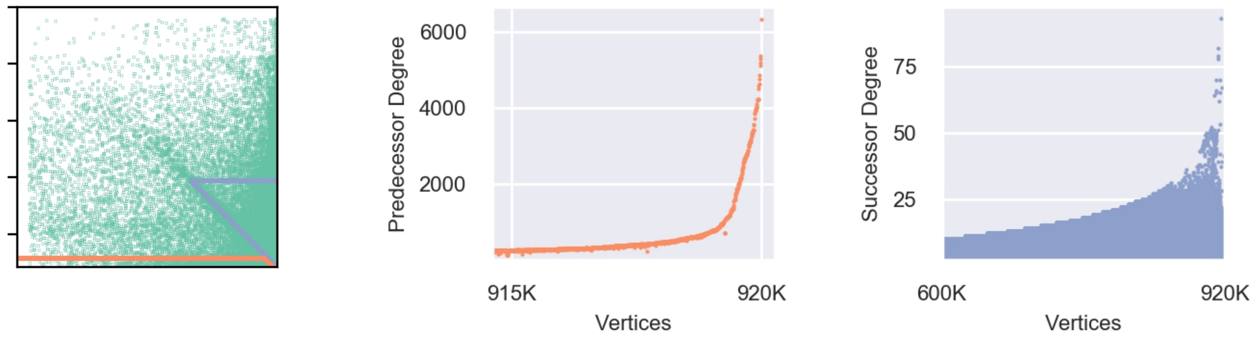
To illustrate the effect of reordering, Figure 4 shows the amount of work saved by degree reordering as a fraction of the work without reordering for an assortment of several test graphs. Note that these are results from triangle counting on

```
template <typename RandomAccessRange>
size_t triangle_count(RandomAccessRange G)
{
  size_t alpha = 0;                                          // α ← 0
  for (auto i = G.begin(); i != G.end(); ++i)                // for i = 0, 1, ... V − 1
    for (auto j = (*i).begin(); j != (*i).end(); ++j)        //   for each neighbor j of i
      alpha += intersection_size(*i, G[*j]);                 //     α ← α + Nb(i, j) ⋃ Nb(j)
  return alpha;
}
```

Fig. 2: Triangle counting algorithm in C++.



(a) No Reordering



(b) Reorder Ascending Degree

Fig. 3: Web-Google Spyplots and Degree Distributions: Spyplots of the web-Google graph are shown (a) before and (b) after reordering by ascending degree along with their corresponding predecessor and successor degree distributions. Predecessor/Successor degrees count number of entries per row of the upper/lower triangular matrix respectively. The degree distributions of (b) are only shown for the region on interest indicated in the spyplot.

| Graph | $\lvert V \rvert$ | $\lvert E \rvert$ | $\frac{\text{Work With Reorder}}{\text{Work Without Reorder}}$ |
|---|---|---|---|
| as-Skitter | 1.70M | 11.1M | 81.8% |
| com_Amazon | 335K | 926K | 8.09% |
| com_Youtube | 1.13M | 2.99M | 83.5% |
| kron_g500-logn19 | 524K | 21.8M | 0.00% (2.75e-11) |
| soc-LiveJournal1 | 4.85M | 34.5M | 100% |
| web-Google | 916K | 2.55M | 75.4% |
| wiki-Talk | 2.39M | 2.51M | 89.7% |

Fig. 4: Successor Degree Reorder Improvement of Upper Triangle: The ratio of total work with degree reordering compared to no reordering for several SuiteSparse graphs.

successor lists, and use ascending degree reordering. Several graphs require 80%-90% of the original work, though sometimes the benefit is much greater in the case of Kronecker graphs, and sometimes there is little benefit as with soc-LiveJournal1. In general, the more skewed the degree distribution of a graph is (the more like a power-law graph it is), the more benefit reordering will provide.

To more fully illustrate the effect of reordering on graph structure, we show spyplots of the same graph before and after ascending degree reordering (Figure 3(a) and Figure 3(b), respectively. Spyplots on the left indicate the sparsity patterns, with the reordered graph having most of its entries shifted to the lower right. Predecessor and successor degree distributions are shown before and after reordering. These degree plots only show the window of interest and are scaled differently for visual clarity. Note that the average degree is the same regardless of which degree list we use or whether we reorder or not. This makes sense as the information needed for triangle counting is preserved whether or not we reorder and whether we use successor or predecessor information. However the several very high degree vertices in both predecessor and successor degree distributions of the unlabeled graph can hurt performance and is why reordering is often useful. Reordering in the wrong direction can be even more harmful. If we try to use the predecessor list to count triangles after relabeling by ascending degree as seen in Figure 3(b), the maximum degree is every higher. There is a clear benefit from using the successor list after reordering by ascending degree, as its max degree is much lower compared to both the unlabeled degree distributions and the predecessor list of the reordered graph. If we had instead reordered by descending degree, the predecessor distribution would have a smaller max degree and a smoother distribution. The rest of the paper assumes the use of successor lists unless otherwise stated.

While degree ordering has the effect of reducing the total amount of sequential work, it also increases the difficulty of load balancing in parallel. This is due to the very skewed tail seen in the successor distribution of Figure 3(b) and the very dense lower right region of the associated spyplot. If some work threads are assigned vertices at the tail of this distribution, they will take much longer to process their vertices' neighbor lists than other threads. We discuss potential ways to avoid this below.

## *B. Parallelization*

In this paper we consider very simple parallel extensions of Figure 2, and we are primarily interested in how to divide the vertex neighbor lists (matrix rows) across multiple threads. It is well known that there are multiple ways to assign rows to threads (or rows to processors in a distributed setting) for better load balancing, yet we have found few results of different row distribution strategies presented in isolation. The most natural neighbor list distribution is a simple block partition, assigning the first $n/threads$ rows to the first thread and so on. This often leads to poor load balancing, with the smaller but much denser rows near the tip of the triangular adjacency matrix containing a disproportionately large amount of work. The poor load balancing of the simple block distribution across 64 threads for the web-Google graph can be seen in Figure 5. The maximum work assigned to a thread is several orders of magnitude larger than the minimum work done by a thread.

To avoid this, the rows could be partitioned non-uniformly depending on their number of entries. In this balanced block distribution, the rows owned by a thread would still be contiguous but the number of rows per thread would vary. The effect of this improvement on the web-Google graph can be seen in Figure 5. This distribution is an improvement over the simple block distribution, but an order of magnitude load balance remains between the minimum and maximum work threads, due to a remaining uneven distribution of high successor degree vertices. To further improve load balance we utilize a cyclic row distribution, where each thread $t$ operates on rows $t$, $threads + t$, $(2 \times threads) + t$, and so on. This cyclic strategy has similarities to a 2D distributed memory implementation [17], where it can be inferred but is not shown explicitly that cyclic load balancing improves performance. For the web-Google graph, this leads to near optimal load balancing as indicated by the flat line in Figure 5.

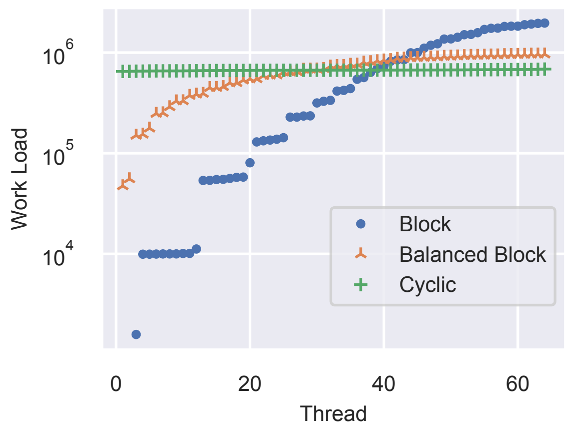


Fig. 5: Web-Google Workload Distributions: The amount of work required for each of 64 threads under three different workload distributions. The flat line for cyclic distribution indicates near optimal workload balancing.

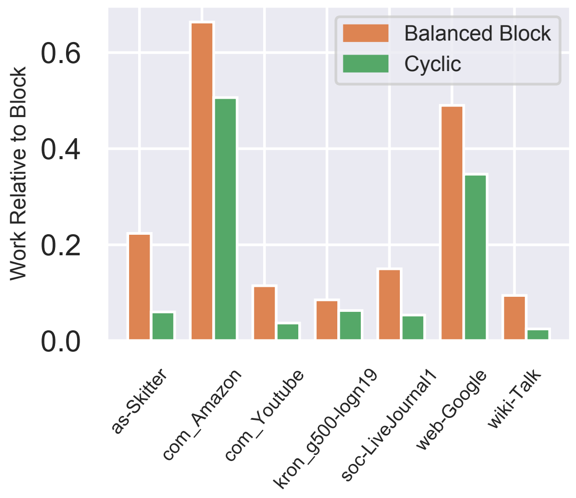


Fig. 6: Maximum Workload Over Threads: The benefit of balanced block and cyclic neighbor list distributions relative to a simple block workload distribution for several SuiteSparse graphs. Shorter bars indicate less work required by the bottleneck thread.

As the thread with the most assigned work becomes the performance bottleneck, we track the max thread work over 64 threads while triangle counting on several smaller graphs and present the results in Figure 6. This bar plot uses the simple block distribution as a baseline and indicates the improvement of balanced block and cyclic methods. Note that using a cyclic distribution is often a small fraction of the max work over threads compared to block distributions.

## IV. Implementations

In this section we present descriptions of different triangle counting implementations based on three different C++ parallelization strategies, `std::async`, `tbb::parallel_for`, and `std::for_each` with parallel execution policies. We apply these strategies to the outer level of parallelism, splitting up the work of processing different neighbor lists. In all the implementations, we try to rely on generic, reusable programming as much as possible, including STL algorithms and data structures. For instance we choose to use std::vector instead of writing our own, even if there is some performance to be gained. There is a potential extra level of parallelism to be gained by using parallel execution policies inside the set intersections of all the implementations discussed below. While we hope to develop a fast triangle counting implementation, the main goal is to understand the performance differences of different algorithmic techniques and parallelization strategies. See Appendix Figures 12-14 for example NWGraph code.

### A. *Implementations based on `std::async`*

The first parallelization strategy we consider is to distribute work among threads by passing lambda functions to `async`. The lambda function uses the size of the graph and the number of threads to determine the neighbor lists owned by the current thread, based on the different neighborlist distribution strategies discussed in Section III. After threads return their triangle count, the sum is reduced with `std::future`.

### B. *Implementations based on `tbb::parallel_reduce`*

Our next triangle counting implementation uses Threading Building Blocks (TBB)'s `parallel_reduce` to split the neighbor lists among threads and to accumulate the number of triangles. This parallel reduce takes as an argument a range of vertices for each thread to operate over. One could pass in a simplistic block range, but again we use a cyclic range for better load balancing. Since the TBB implementation has less explicit thread control, we develop a novel cyclic range adapter that TBB uses to cyclically assign work to threads.

### C. *Implementations based on `std::for_each` and C++ Parallel Execution Policies*

The last implementation we consider uses the C++ built-in `std::for_each` function to process the neighbor lists of every vertex. Instead of explicitly dividing these neighbor lists among threads, `for_each` takes an optional execution policy for processing the lists in parallel. Note that using this strategy is a more hands off approach as a programmer has no control how `for_each` ultimately splits up the workload.

## V. Experimental Setup and Results

To see the benefits of the techniques discussed in the previous section, we count the number of triangles in the graphs described in the GAP benchmark suite [16] and collect timing results. The GAP benchmark is a set of five test graphs and six graph kernels, including triangle counting, designed to test the performance of different graph processing frameworks. The smaller graphs used for the work counting experiments in Section III were chosen to have similar properties to the GAP graphs. GAP also includes a reference implementation, included in our performance results for comparison, which parallelizes the processing of neighbor lists with OpenMP and dynamic scheduling. We choose to omit the GAP-road network as it is an outlier for this study in terms of size and the number of triangles. The included graphs and their sizes are shown in Figure 8. Our testing architecture is an Intel Xeon Gold 6130 CPU with 64 logical cores. For these experiments we utilize all 64 cores. We compiled all implementations, including the GAP reference, with GNU GCC 9.2.0.

Not all graphs will benefit equally from graph reordering and the overhead of reordering might be costly compared to the actual triangle counting. In practice one might employ cheap heuristics to decide whether reordering overhead is worthwhile. To demonstrate the effectiveness of reordering on the GAP graphs, we measure triangle counting time before and after reordering and present the reordering speedup in Figure 7. These results are based on the cyclic `async` implementation. Kron sees nearly an order of magnitude improvement in solve time while the skewed degree web

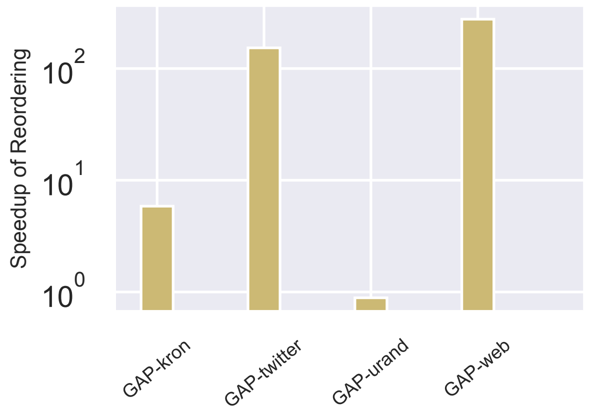


Fig. 7: Reordering on GAP Graphs. Triangle counting speedup by reordering is shown relative to unordered on a log scale. All graphs shown benefit except for GAP-urand.

| Gap Graphs | $\|V\|$ | $\|E\|$ | Triangle Count | NWGraph Relabel(s) | GAP Relabel(s) |
|---|---|---|---|---|---|
| web | 50.6M | 965M | 84.91B | 1.06 | 11.1 |
| twitter | 61.6M | 734M | 34.83B | 1.53 | 15.4 |
| kron | 134M | 2.11B | 106.9B | 2.97 | 24.8 |
| urand | 134M | 2.15B | 5378 | 3.79 | 26.5 |

Fig. 8: Triangle and Relabel Information for GAP Graphs

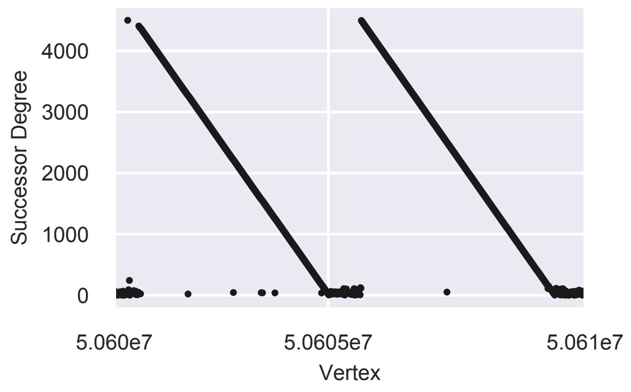
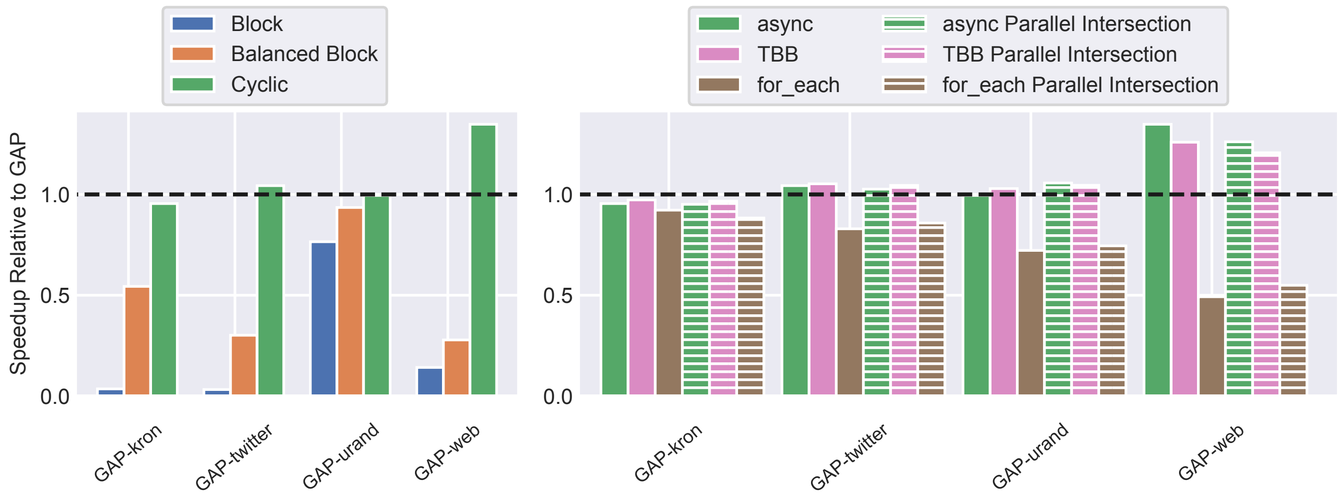


Fig. 9: GAP-Web Partial Successor Degree Distribution. A characteristic double spike of large degree neighbor lists is a challenge to load balance.



(a) Row Distribution (All Async)

(b) Parallelization Strategy

Fig. 10: GAP Comparison: The speedup (or slowdown) of several implementations is shown relative to the GAP reference implementation on the GAP graphs. Implementations above the dotted line run faster than the GAP reference implementation. (a) compares row distribution strategies of different `async` implementations. (b) compares different C++ parallelization strategies.

and twitter graphs see two orders of magnitude improvement. Urand sees a slight slowdown due to reordering since the random nature of the graph already had a smooth degree distribution to begin with.

However, to provide a level playing field for the rest of the algorithmic modifications we relabel all graphs by ascending degree and report relabel time along with the number of triangles in Figure 8. The relabeling time of the GAP reference implementations typically an order of magnitude slower than our implementation because we relabel an edge list before compressing to compressed sparse row (CSR) format.

In order to demonstrate the effect each of the algorithmic techniques and C++ parallelization strategies, we tested multiple implementations and summarize results in Figure 10. Each bar in this figure indicates the speedup over the GAP reference implementation (GAP time / implementation time), with bars above the dotted line indicating improvement over GAP. Figure 10(a) compares the first three implementations which are based on `std::async` and demonstrate the different neighbor list distribution strategies in Section III. Figure 10(b) compares the three outer level parallelization strategies described in Section IV. This figure also includes results from adding an inner level of parallelism with a parallel execution policy inside the set intersection. Note that Figure 10(a) only includes sequential STL intersections and Figure 10(c) only includes cyclic distribution for `async` and TBB. We include the timing numbers behind these plots in Appendix Figure 11.

## VI. Experimental Analysis and Discussion

Figure 10(a) indicates that cyclic distributions often lead to improved triangle solve time compared to block distributions due to better load balancing. GAP-urand is an outlier as it has a more balanced degree distribution to begin with so simple block partitioning has decent load balance. While it might be possible, knowing the cyclic stride in advance, to create an artificial graph which causes poor load balancing for a cyclic distribution, such graphs are unlikely to occur naturally. A cyclic distribution will almost never lead to poor load balancing and will often significantly improve performance, assuming the number of compute resources is not significantly larger than the number of high degree vertices.

Figure 10(b) suggests that cyclic implementations based on std::async and tbb::parallel_reduce are quite competitive, with async slightly outperforming on GAP-web. The implementation based on `for_each` performs worse, significantly worse in the case of GAP-web. We avoid speculating on how `for_each` assigns neighbor lists to threads "behind the curtain". Performance is typically better than for the block and balanced block async implementations shown in Figure 10(a), so it is doing something reasonable. As the parallel `for_each` implementation is perhaps the most ideal in terms of relying on STL features, we would like to better understand what is required to make its performance more comparable to the other parallelization strategies.

The cyclic distribution strategy closely matches the results of the GAP reference implementation on three of the graphs but not GAP-web where the cyclic distribution seems to outperform GAP's block distribution. This led us to further investigate the successor degree distribution of GAP-web after reordering and we found two interesting spikes near the tail end of the distribution shown in Figure 9

None of the outer parallelization strategies benefit from parallel set intersections as evidenced by the dashed bars in Figure 10(b). Perhaps with good load balancing of all 64 threads, we should not expect much benefit of using a parallel execution policy inside the set intersection.

## References

[1] A. Azad et al., *Evaluation of graph analytic frameworks using the GAP benchmark suite*, (to appear in) Proc. IEEE Int. Symp. on Workload Characterization, 2020.

[2] M. Latapy *Main-memory triangle computations for very large (sparse (power-law)) graphs*, Theoretical Computer Science 407.1-3 (2008), pp. 458–473.

[3] J. Shun and K. Tangwongsan, *Multicore triangle computations without tuning*, in Proc. IEEE Int. Conf. on Data Engineering, Seoul, KOR, 2015, pp. 149–160.

[4] S. Parimalarangan, G. M. Slota, and K. Madduri, *Fast parallel graph triad census and triangle counting on shared-memory platforms*, in Proc. IEEE Int. Parallel and Distributed Processing Symp. Workshops, Lake Buena Vista, FL, USA, 2017, pp. 1500–1509.

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

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

[7] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, *Fast linear algebra-based triangle counting with Kokkos kernels*, in Proc. IEEE High Performance Extreme Computing Conf., Waltham, MA, USA, 2017, pp. 1–7.

[8] A. Yaşar, S. Rajamanickam, M. Wolf, J. Berry, and Ü. V. Çatalyürek, *Fast triangle counting using Cilk*, in Proc. IEEE High Performance Extreme Computing Conf., Waltham, MA, USA, 2018, pp. 1–7.

[9] A. Azad, A. Buluç, and J. Gilbert, *Parallel triangle counting and enumeration using matrix algebra*, in Proc. IEEE Int. Parallel and Distributed Processing Symp. Workshop, Hyderabad, IND, 2015, pp. 804–811.

[10] T. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software 38.1 (2011), pp. 1–25.

[11] A. George and J. W. Liu, *Computer solution of large sparse positive definite*, Pretince Hall, 1981.

[12] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proc. of ACM National Conf., New York, NY, USA, 1969, pp. 157–172.

[13] J. W. H. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Transactions on Mathematical Software 11.2 (1985), pp. 141–153.

[14] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. on Matrix Analysis and Applications 17.4 (1996), pp. 886–905.

[15] G. M. Slota, S. Rajamanickam, and K. Madduri, *Order or shuffle: Empirically evaluating vertex order impact on parallel graph computations*, in Proc. IEEE Int. Parallel and Distributed Processing Symp. Workshops, Lake Buena Vista, FL, USA, 2017, pp. 588–597.

[16] S. Beamer, K. Asanović, and D. Patterson, *The GAP benchmark suite*, 2015, arXiv:1508.03619.

[17] A. S. Tom and G. Karypis, *A 2D parallel triangle counting algorithm for distributed-memory architectures*, in Proc. ACM Int. Conf. Parallel Processing, Kyoto, JPN, 2019, pp. 1–10.

## VII. Appendix

| GAP Graphs | GAP Reference | NWGraph | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Block | Balanced Block | Cyclic | | | | | |
| | | Sequential Intersection | Sequential Intersection | Sequential Intersection | | | Parallel Intersection | | |
| | | `async` | `async` | `async` | TBB | `for_each` | `async` | TBB | `for_each` |
| web | 7.71 | 54.3 | 27.7 | 5.72 | 6.13 | 15.7 | 6.11 | 6.39 | 14.0 |
| twitter | 45.4 | 1400 | 151 | 43.5 | 43.2 | 54.8 | 44.3 | 43.5 | 53.0 |
| kron | 277 | 7750 | 509 | 290 | 285 | 301 | 291 | 287 | 314 |
| urand | 15.0 | 19.6 | 16.1 | 15.1 | 14.6 | 20.8 | 14.2 | 14.4 | 20.2 |

Fig. 11: Triangle Counting Timing in (s) for All Implementations Shown in Figure 10.

```
template <class Op>
size_t async_helper(size_t threads, Op&& op)
{
  vector<future<size_t>> futures(threads);
  for (size_t tid = 0; tid < threads; ++tid)
    futures[tid] = async(launch::async, op, tid);

  size_t t = 0;
  for (auto&& f : futures)
    t += f.get();
  return t;
}

template <typename RandomAccessRange>
size_t triangle_count(RandomAccessRange G)
{
  return async_helper(threads, [&](size_t tid) {
    size_t alpha = 0;
    for (auto i = G.begin(); i != G.end(); ++i)
      for (auto j = (*i).begin(); j != (*i).end(); ++j)
        alpha += intersection_size(*i, G[*j]);
    return alpha;
  });
}
```

Fig. 12: Parallel Triangle Counting with `std::async`.

```
template <typename RandomAccessRange>
size_t triangle_count(RandomAccessRange G, size_t stride)
{
  return nwgraph::parallel_for(nwgraph::cyclic(G, stride), [&](auto&& i) {
    size_t alpha = 0;
    for (auto j = (*i).begin(); j != (*i).end(); ++j)
      alpha += intersection_size(*i, G[*j]);
    return alpha;
  }, plus{}, 0.0);
}
```

Fig. 13: Parallel Triangle Counting with TBB.

```
template <typename RandomAccessRange>
size_t triangle_count(RandomAccessRange G)
{
  atomic<size_t> t = 0;
  for_each(execution::par, G.begin(), G.end(), [&](auto&& i)
  {
    size_t alpha = 0;
    for (auto j = (*i).begin(); j != (*i).end(); ++j)
      alpha += intersection_size(*i, G[*j]);
    t += alpha;
  });
  return t;
}
```

Fig. 14: Parallel Triangle Counting with `std::for_each`.