Distributed, Shared-Memory Parallel Triangle Counting

Thejaka Amila Kanewala
thejkane@indiana.edu
Indiana University
School of Informatics, Computing, & Engineering
IN, USA

Marcin Zalewski
marcin.zalewski@pnnl.gov
Pacific Northwest National Laboratory
and University of Washington
Seattle, WA, USA

Andrew Lumsdaine
andrew.lumsdaine@pnnl.gov
Pacific Northwest National Laboratory
and University of Washington
Seattle, WA, USA

ABSTRACT

Triangles are the most basic non-trivial subgraphs. Triangle counting is used in a number of different applications, including social network mining, cyber security, and spam detection. In general, triangle counting algorithms are readily parallelizable, but when implemented in distributed, shared-memory, their performance is poor due to high communication, imbalance of work, and the difficulty of exploiting locality available in shared memory. In this paper, we discuss four different (but related) triangle counting algorithms and how their performance can be improved in distributed, shared-memory by reducing in-node load imbalance, improving cache utilization, minimizing network overhead, and minimizing algorithmic work. We generalize the four different triangle counting algorithms into a common framework and show that for all four algorithms the in-node load imbalance can be minimized while utilizing caches by partitioning work into blocks of vertices, the network overhead can be minimized by aggregation of blocks of work, and algorithm work can be reduced by partitioning vertex neighbors by degree.

We experimentally evaluate the weak and the strong scaling performance of the proposed algorithms with two types of synthetic graph inputs and three real-world graph inputs. We also compare the performance of our implementations with the distributed, shared-memory triangle counting algorithms available in PowerGraph-GraphLab and show that our proposed algorithms outperform those algorithms, both in terms of space and time.

KEYWORDS

triangle counting, distributed, shared-memory graph algorithms, parallel graph algorithms

ACM Reference Format:
Thejaka Amila Kanewala, Marcin Zalewski, and Andrew Lumsdaine. 2018. Distributed, Shared-Memory Parallel Triangle Counting. In *PASC '18: Platform for Advanced Scientific Computing Conference, July 2–4, 2018, Basel, Switzerland*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3218176.3218229

1 INTRODUCTION

For a given graph $G = (V, E)$, triangle counting involves finding structures that have three vertices connected to each other by an edge. Triangle counting is used in many applications. For example, triangles are used to assess the content quality of social networks[29], to detect web spam[3] and to uncover thematic structures of the web[7]. With the increasing size of data sets, the graphs generated for those applications are growing larger and may not fit into a single machine memory.

Hybrid, Single Program, Multiple Data (SPMD) approaches that represent distributed, shared-memory runtimes are a popular way to deal with these large graphs. *MPI + OpenMP* or *MPI + PThreads* are two such examples. When extending triangle counting for such distributed, shared-memory runtimes we face several challenges: 1. in-node load imbalance, 2. poor cache utilization, and, 3. higher message communication. Further, certain preprocessing optimizations applicable for shared-memory parallel triangle counting algorithms become cumbersome and inefficient to apply when the processing graph is distributed. In this paper, we demonstrate step-by-step how we developed four related triangle counting algorithms that are scalable in distributed, shared-memory runtimes.

Most of the existing parallel triangle counting algorithms are variations of the sequential triangle counting algorithm: *Node-Iterator* [25]. Node-Iterator partitions neighbors of a vertex into two sets. The set $pred(v)$ is defined as the set of neighbors of v that are less than v and set $succ(v)$ is defined as the set of neighbors of v that are greater than v. The set $pred(v)$ is called *predecessors* of v and the set $succ(v)$ is called *successors* of v. The parallel version of this algorithm (Listed in Algorithm 1) iterates over vertices and checks whether a vertex in predecessor set and a vertex in successor set makes an edge; if they do, the number of triangles is incremented.

A straightforward extension of this algorithm for distributed, shared-memory parallel execution, is to distribute vertices among different nodes and process every vertex in a separate parallel thread. This approach creates load imbalance between parallel threads when processing a skewed graph as threads that process hub vertices take more time compared to the threads that process vertices with fewer edges. In distributed execution, the algorithm needs to send each (v, u) pair to the node that owns vertex w to check whether u is in the predecessor set of w. When the number of distributed nodes is increased, the number of messages the algorithm needs to send also increases, as the successor vertex set of w

Algorithm 1: Parallel Node Iterator Triangle Counting Algorithm

NodeIterator $G^{\text{local}} = (V, E)$:
1: **for** each v in V *parallel* **do**
2: **for** each $u \in pred(v)$ **do**
3: **for** each $w \in succ(v)$ **do**
4: **if** $u \in pred(w)$ **then**
5: $TC = TC + 1$

PASC '18, July 2–4, 2018, Basel, Switzerland

ACM ISBN 978-1-4503-5891-0/18/07…$15.00
https://doi.org/10.1145/3218176.3218229

can be distributed among the nodes. Therefore, when the number of nodes is increased algorithm spends more time on communication and less time on computation.

By processing every wedge (a possible triangle) in a separate parallel thread, algorithm can balance the load. However, processing every wedge in a separate parallel thread reduces the cache utilization and also increases the number of messages in distributed execution. A common optimization to reduce the number of processing wedges is to order vertices by their degree (e.g., [21]). Applying this optimization as it is for distributed, shared-memory triangle counting is challenging. Permuting vertex ids in a distributed setting requires sorting vertices by the degree, assigning new ids and exchanging those new ids with all other adjacencies in remote ranks and then, rebuilding the graph. This preprocessing step consumes time and is cumbersome to implement in a distributed setting.

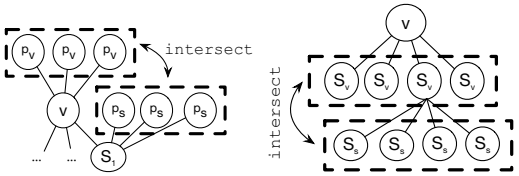
In this paper, we show how we handled each of the above discussed challenges. First, we show that Node-Iterator is one of four ways to do triangle counting and we generalize those four algorithms using two common abstractions (Section 2). All the previously discussed challenges are applicable to all four algorithms. To balance the overhead of load imbalance, cache utilization and to reduce the latency of each of the small messages, we propose blocking neighbors – this is described in Section 4. Further, many real world graphs tend to have a small number of vertices with a high degree and a large number of vertices with a small degree. Therefore, we saw that latency incurred during distributed execution was still high with vertex blocks. Section 4.3 shows how latency of small blocks can be further reduced using block aggregation. Finally, we propose four triangle counting algorithms with two super-steps in each. In the first super-step, the algorithms partition neighbors by their degree, and in the second super-step the algorithms calculate the number of triangles. Partitioning neighbors by degree avoids the need for distributed sorting and graph re-construction.

Much of the existing work focuses on shared-memory, distributed-memory, or external memory. In this paper we consider hybrid runtime models and the problems that one encounters when implementing triangle counting in such hybrid models. Some of the techniques we use are not new in separation (e.g., message aggregation), we use them in novel combinations to achieve better scaling. Selecting features and showing how to use them in combination to achieve better performing distributed, shared-memory parallel triangle counting algorithms is the main contribution of this paper.

The performances of our proposed algorithms are evaluated on weak-scaling and strong-scaling (Section 6). For weak scaling we use two types of synthetic graphs and for strong scaling we use two types of synthetic graphs and three large real-world networks from [15]. Results show that the proposed algorithms scale well. In addition, we also compare our results with another distributed, shared-memory triangle counting implementation: PowerGraph-GraphLab[8] triangle counting algorithms and show that our algorithms outperform that triangle counting algorithms.

2 TRIANGLE COUNTING

The parallel Node-Iterator algorithm discussed in Section 1 partitions neighbors of every vertex into two. To partition the neighbors, the algorithm uses vertex ids. For every vertex, v, the algorithm



(a) **(b)**

Figure 1: Set intersection in PSP and SSS algorithms.

Algorithm	set_1	set_2	set_3
PSP	*succ*	*pred*	*pred*
SPS	*pred*	*succ*	*succ*
SSS	*succ*	*succ*	*succ*
PPP	*pred*	*pred*	*pred*

Table 1: set_1, set_2 and set_3 parameter values for each triangle counting algorithm.

intersects the predecessor set of v with v's successor's predecessors (See Figure 1a). The size of the intersection is accumulated into the number of triangles. We call this algorithm PSP. Mathematically, this triangle counting algorithm is expressed in Equation (1). In Equation (1), TC is the total number of triangles counted, and, as explained above, every vertex goes through its successors and intersects the vertex's predecessors with each successor's predecessors.

$$TC = \Sigma_{v \in V} \Sigma_{s \in succ(v)} |pred(v) \cap pred(s)| \quad (1)$$

The same number of triangles can be counted by switching *pred* with *succ* and *succ* with *pred* in Equation (1). In other words, instead of intersecting a vertex's predecessor set with successor's predecessor set we can intersect the vertex's successors with the predecessor's successor sets. We call this algorithm *successor, predecessor's successor* (SPS) and it is expressed in Equation (2).

$$TC = \Sigma_{v \in V} \Sigma_{p \in pred(v)} |succ(v) \cap succ(p)| \quad (2)$$

Another approach to count triangles is to intersect the successor set of a vertex with its successor's successors. Figure 1b depicts the algorithm. We call this algorithm SSS, and the mathematical equation for this algorithm is given in Equation (3).

$$TC = \Sigma_{v \in V} \Sigma_{s \in succ(v)} |succ(v) \cap succ(s)| \quad (3)$$

In Equation (4), we switch succ with pred, as we did in Equation (1). The resulting equation is given in Equation (4). The algorithm represented in Equation (4) goes through predecessors of every vertex and intersects with predecessor's predecessor set (Named as *predecessor, predecessor's predecessor* (PPP)).

$$TC = \Sigma_{v \in V} \Sigma_{s \in pred(v)} |pred(v) \cap pred(s)| \quad (4)$$

An important observation of these algorithms is that they all involve three sets per each vertex. A generalized equation for triangle counting is given in Equation (5). The general form has a predecessor set or a successor set of the iterating vertex (set_1) to iterate, a predecessor set or a successor set of the iterating vertex (set_2), and a predecessor set or a successor set corresponding to a vertex in the iterating set (set_3). Table 1 summarizes the algorithms with appropriate values for set_1, set_2, and set_3.

Algorithm 2: PSP Triangle Counting Algorithm

PSP $G^{\text{local}} = (V, E)$:
1: **epoch** {
2: **for** each v in V in a *parallel thread* **do**
3: **for** each $s \in succ(v)$ **do**
4: Send $pred(v)$ & s to *owner* of s
5: }
Receive $pred(v)$, s:
1: TC += $pred(v) \cap pred(s)$

Algorithm 3: Generalized Triangle Counting Algorithm

TC $G^{\text{local}} = (V, E)$:
1: **epoch** {
2: **for** each v in V in a *parallel thread* **do**
3: **for** each $s \in set_1(v)$ **do**
4: Send $set_2(v)$ & s to *owner* of s
5: }
Receive $set_2(v)$, s:
1: TC += $set_2(v) \cap set_3(s)$

$$TC = \Sigma_{v \in V} \Sigma_{s \in set_1(v)} |set_2(v) \cap set_3(s)| \quad (5)$$

3 DISTRIBUTED, SHARED-MEMORY TRIANGLE COUNTING

As we saw in the previous section, the main operation in triangle counting is the intersection between two sets. In a distributed environment all the elements of these two sets may not reside in the same locality. In such situations we need to collect all the elements in both sets into one locality and perform the set intersection.

3.1 Graph Distribution & Data Structures

In our implementations, vertices are distributed equally among the participating nodes (*1D distribution*). A vertex id is represented using 64 bit integers. The first 48 bits represent the local vertex id and the second 16 bits represent the node id a vertex belongs to. Every rank contains a set of vertices and a set of edges corresponding to the vertices in the vertex set. Vertices and edges local to a rank are stored in *compressed sparse row* (CSR) format. This is shown in Figure 2. In Figure 2, every rank is separated using a vertical dash line and "Rn" represents rank n. We find the rank which a vertex belongs to by looking at the second 16 bits of the global vertex id. The CSR format consists of two arrays: 1. Indices array – this array stores the edges; and, 2. Indices pointer array – this array stores edge ranges belonging to every vertex. As an example, adjacencies for vertex v_k in rank R0 are found in the indices pointer array from position "a" to position "b".

Assuming the indices range for every vertex is sorted, it is straightforward to calculate the predecessor set and successor set for every vertex. For every vertex, v_k, we need to find the position in the indices array that divides neighbors compared to v_k. Suppose this position is m, then range [a, m) denotes the predecessor set of v_k and the range [m, b) represents the successor set of v_k. In other words, vertices in range (a, m] in the indices array are less than v_k, and vertices in the range (m, b] in the indices array are greater than v_k (See R0 CSR in Figure 2). This way we limit the amount of additional space needed to store predecessor sets and successor sets to $O(|V|)$ (we only need to store the position m for every vertex).

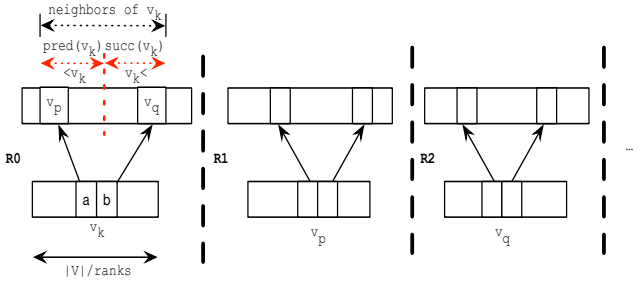


Figure 2: Separating successors and predecessors in CSR structure.

However, many real-world graphs are unsorted. Still, when building the CSR structure we need to sort them by the source vertex (e.g., *Histogram Sort*). For unsorted graphs, this sorting step is extended and vertices are sorted by the source and then, by the destination. Note that this sorting step is local to a node and does not require any form of distributed communication; it takes place during the graph construction.

3.2 Challenges in Distributed Shared-Memory Parallel Triangle Counting

To construct an effective distributed, shared-memory parallel triangle counting algorithm, there were three primary problems we needed to solve: in our initial attempt to solve these challenges we came up with the PSP triangle counting algorithm listed in Algorithm 2. In this algorithm, every vertex processes in a separate parallel thread. An *epoch* (Line 1) is a code region that indicates explicit communication is taking place. Every vertex sends its predecessor set to each owner rank of its successors. Then, the rank that receives the predecessor set performs the intersection in a single thread.

By switching "pred" set with "succ" set and "succ" set with "pred" set in the above algorithm we get the SPS algorithm. In a similar way we can get the other algorithms. Algorithm 3 lists a generalized triangle counting algorithm and by replacing set_1, set_2 and set_3 according to values in Table 1, we get PSP, SPS, SSS and PPP. The performance of these initial implementations are poor because of:

(1) load imbalance between processing elements,
(2) redundant messages to the same rank.

For algorithms derived from Algorithm 3 (including Algorithm 2) we measured the work done by each parallel thread by counting the number of set comparisons performed by each thread. Figure 3 shows the set comparisons performed by each thread when processing LiveJournal social network graph[17] on 16 shared-memory parallel threads with the SSS algorithm. As can be seen in Figure 3, some threads are more utilized than others. Thread number seven performs the fewest set comparisons (i.e., 533529383) and thread number eight performs the most set comparisons.

Load imbalance is mainly caused by the uneven distribution of degrees on vertices. In Algorithm 3 every vertex is processed by a separated parallel thread. Figure 3 shows the maximum degree and maximum number of successors of all the vertices processed

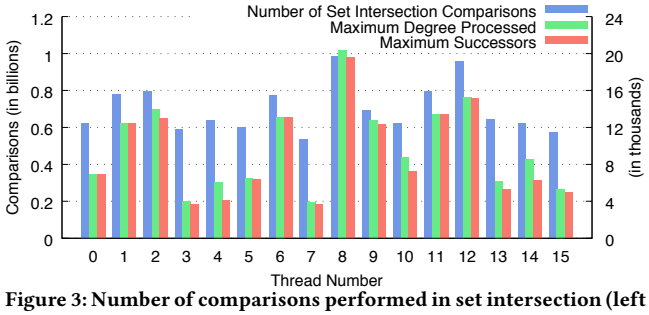


Figure 3: Number of comparisons performed in set intersection (left axis), and maximum vertex degree and number of successors (right axis) in each thread on LiveJournal social network graph with SSS triangle counting algorithm.

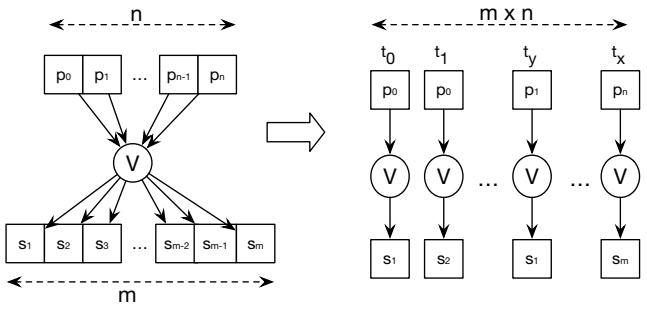


Figure 4: Every thread processes an open wedge

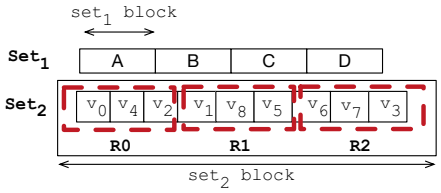


Figure 5: Blocking vertices in sets.

by a thread in SSS algorithm. As can be seen in Figure 3 thread 8 has processed the vertex with highest degree (also the maximum number of successors) and thread seven has processed smaller degree vertices. Most of the real-world graphs are power-low graphs (i.e., they have few vertices that have a high number of neighbors and high number of vertices with fewer number of neighbors). It takes more time to perform set intersections on vertices with higher numbers of neighbors than vertices with fewer neighbors. Therefore, threads that process vertices with higher numbers of neighbors take more time than threads that process vertices with fewer numbers of neighbors.

One approach to overcome the load imbalance is to process each *open wedge* (a possible triangle) in a separate parallel thread. An open wedge is a triple : a predecessor of a vertex, a successor of a vertex and the vertex (See Figure 4). Now each thread processes an open wedge and the "Receive" function of Algorithm 2 needs to be modified to do a binary search on the successor's predecessor set instead of the set intersection. While this approach balances the work among parallel threads, it generates a lot of of messages. If a vertex has "n" number of predecessors and "m" number of successors, then this algorithm creates $n \times m$ number of messages per vertex. If we assume the average degree of a vertex is d, then this approach creates approximately $O(|V| \times d^2)$ messages. The algorithm also loses the ability to take advantage of better cache utilization due to binary search at the receiver's end. The accumulated runtime overhead and binary search to process the significant number of generated messages is quite high. Therefore, this approach does not show the advantage of load balance because the overhead of message communication and binary search is greater than the advantage achieved by load balancing.

4 BLOCKING AND GROUPING VERTICES

To reduce the number of messages while balancing the load on processing threads we came up with a strategy to block vertices in predecessor sets and successor sets (the PSP algorithm). A block is a set of vertices. The block size is configurable. In the following we explain the blocking process in general for all the algorithms.

All the triangle counting algorithms discussed in Section 2 go through all the vertices in parallel and send one set of neighbors (set_1) to the owner of a vertex in an another neighbor set (set_2). For some algorithms (e.g., SSS and PPP) these two neighbor sets are the same. With blocking, instead of sending a complete chunk of neighbors to the owner of an another set, we send a subset, and this subset of vertices is called a "block". This approach reduces the number of messages relative to the approach discussed in Section 3.2.

Similar to set_1 we also block vertices in set_2. However, vertices in a set_2 block may not belong to the same rank. Therefore, in a set_2 block, vertices are *grouped* by their rank (See Figure 5 - suppose set_2 block size is 10). A distributed message will have a block from set_1 vertices, and subset of a block in set_2 vertices.

Every distributed message received by a rank is processed in a separate parallel shared-memory thread. Therefore, every distributed message will cause a thread to execute set intersections equal to the number of vertices in the group. The objective of blocking set_2 is to control the number of set intersections per thread.

The PSP algorithm with message blocking and grouping for one rank is listed in Algorithm 4. The code listed from Line 2 to Line 23 is executed for each shared-memory parallel thread. As discussed earlier, an *epoch* represents a region where the program can send/receive messages. Earlier algorithms (Algorithm 2 and Algorithm 3) process every vertex in a separate shared-memory parallel thread, but in this algorithm each set_1 block (*preblock* in Line 14) is processed in a separate parallel thread. For the PSP, algorithm, set_1 is the predecessor set of a vertex (shown in *pred(v)*) and set_2 is the successor set of a vertex (*succ(v)*). Every thread iterates over vertices (Line 3) and calculates an *offset* (Line 7). However, if a vertex does not have any predecessors or successors we do not need to proceed executing the rest of the algorithm; it will not generate any triangles (See condition in Line 4). The number of predecessor blocks and successor blocks is calculated based on predecessor block size and successor block size (Line 8 and Line 9). Then, the loop in Line 11 iterates over predecessor blocks starting from the offset. The next predecessor block in this "for" loop is selected after *nthreads* number of threads. Therefore, each thread

Algorithm	D1 Hit/Miss(%)	D2 Hit/Miss(%)
Basic SSS	95.0/5.0	55.0/45.0
Blocked SSS, block = 100	96.0/4.0	69.8/30.2

Table 2: Cache utilization of SSS vs. blocked SSS.

processes a single predecessor block at a time. Once a thread selects a predecessor block, it extracts the vertices relevant to that block from pred(v) (Line 12–Line 14). The extracted predecessor block needs to be sent to every successor's rank. Line 15 iterates over successor blocks and extracts an appropriate successor block and stores in the variable *sucblock*. Instead of iterating over every vertex in this successor block, the algorithm, groups vertices in the successor block based on the vertex ownership. That is, vertices belonging to the same rank are grouped together. The function *group-by-rank* is responsible for grouping vertices based on their rank (See Line 19). The output of this function is an array where "i th" position in that array has the successor vertices for rank *i*. Then, the algorithm iterates over all rank ids and sends predecessor block and the respective successor block encapsulated in a message to the respective rank (Line 20 – Line 22).

The receiving rank gets a predecessor block and a set of successor vertices (See *Receive*, Line 2–Line 4). The receiving rank then goes through all vertices in the successor sets and for each successor set it extracts the set of vertices that are greater than or equal to the first element of the predecessor set (*Receive*, Line 3). The first element in the predecessor set is a lower bound for this new set. Then, the algorithm performs an intersection between the predecessors and the calculated lower bound set to determine the number of triangles (*Receive*, Line 4).

Calculating the lower bound set allows us to save some comparison operations in set intersection. Calculating an upper bound is not necessary since the set intersection comparison starts from the beginning of the sets; when set intersection reaches the end of the smaller set, it terminates the set intersection operation.

The logic for the generalized blocked triangle counting algorithm is quite similar to Algorithm 4, and, presented in Appendix A. In the generalized version of the algorithm (given in Algorithm 6), the predecessor set of a vertex is replaced with set_1 and the successor set of a vertex is replaced with set_2. For some algorithms both set_1 and set_2 are the same (e.g., SSS). For those algorithms the logic is simpler because we iterate and send blocks to owners of vertices in the same set. For example, for SSS we can remove the first condition from Algorithm 4, Line 4, and we do not need to calculate *nsetbblks* as its value is the same as *nsetablks*.

4.1 Block Size in Shared-Memory

In shared-memory execution, smaller block sizes (set_1 block size and also set_2 block size) reduce the load imbalance between parallel threads. Figure 6 shows the number of comparisons performed by each thread for set intersection in the SSS algorithm with LiveJournel graph input. Compared to Figure 3, Figure 6 shows much better balancing of work among different threads. For example, in SSS implementation (Algorithm 3, with $set_1 = set_2 = set_3 = succ$) a thread processes a vertex irrespective of the number of neighbors that vertex has. In the blocked implementation (Algorithm 6), a vertex with a higher number of neighbors is processed by more than one thread

Algorithm 4: PSP Triangle Counting Algorithm with Blocking & Grouping

PSP $G^{\text{local}} = (V, E)$, *predblksz*, *sucblksz*, *nthreads*:

```
1:  for each parallel thread : tid in nthreads do
2:    epoch {
3:      for each v in V do
4:        if (|pred(v)| == 0) or (|succ(v)| == 0) then
5:          continue;
6:
7:        offset = (v + tid) % nthreads;
8:        npredblks = (|pred(v)| + predblksz − 1) / predblksz
9:        nsucblks = (|succ(v)| + sucblksz − 1) / sucblksz
10:
11:       for (pos = offset; pos < npredblks; (pos += nthreads)) do
12:         predstart = pos * predblksz
13:         predend = min(predblksz, (|pred(v)|-predstart)) +
    predstart
14:         preblock =
    {x|x ∈ pred(v) & (predstart ≤ indexof(x) < predend)}
15:         for (sucpos = 0; sucpos < nsucblks; sucpos++) do
16:           sucstart = sucpos * sucblksz
17:           sucend = min(sucblksz, (|succ(v)|-sucstart)) + sucstart
18:           sucblock =
    {x|x ∈ succ(v) & (sucstart ≤ indexof(x) < sucend)}
19:           rankgroups = group-by-rank(sucblock)
20:           for each r in ranks do
21:             sucblckforrank = rankgroups[r]
22:             Send(r, preblock, sucblckforrank)
23:     }
```

Receive *preblock*, *sucblckforrank*:

```
1: p = preblock[0]
2: for each s in sucblckforrank do
3:   lowerbound = {x|x ∈ pred(s) and p ≤ x}
4:   TC += |preblock ∩ lowerbound|
```

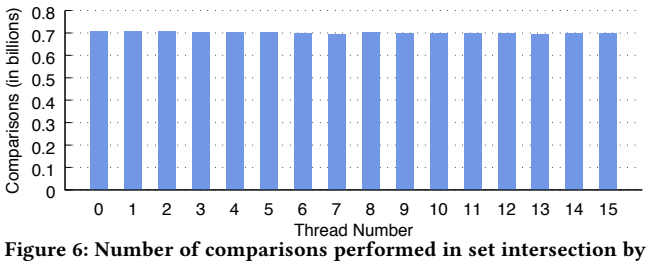


Figure 6: Number of comparisons performed in set intersection by each shared memory parallel thread, on LiveJournal social network graph with SS triangle counting algorithm. set_1 **block size =** set_2 **block size = 100**

(depending upon the size of the block). Therefore, with blocking work can be equally distributed among threads irrespective of the degree distribution of the input graph.

The proposed algorithms (Algorithm 6), block the predecessor set (set_1) of a vertex and the successor set (set_2) of a vertex. In the basic PSP triangle counting algorithm, a predecessor set of a vertex is sent to each of its successors. By blocking and grouping

Block Size	D1 Hit/Miss(%)	D2 Hit/Miss(%)
10	87.1/12.9	53.7/46.3
80	93.7/6.3	54.9/45.1
340	96.2/3.8	59.1/40.9

Table 3: Cache utilization of SSS algorithms with increasing block size. Values are average across the number of threads.

Block Size	Bytes Communicated
10000	16842970024
1000	17927870248
100	29910155304

Table 4: Block size vs. Number of bytes communicated. SPS algorithm with Graph500, Scale 23 graph input and experiments run on four nodes.

successors of a vertex, (set_2), we send a predecessor block together with multiple successor vertices to the same rank (Using PSP as an example). Then, the receiving rank can reuse the predecessor block to do set intersection with several predecessor sets of the successor vertices. This approach achieves better cache utilization because the predecessor block is re-used. This applies to all algorithms. Table 2 compares the cache utilization of the basic SSS algorithm with the blocked SSS algorithm. As can be seen in Table 2, the blocked SSS algorithm gets better utilization of level 2 data (D2) cache (same behaviour is seen in other algorithms). This is mainly because of the for loop in "Receive" functions. However, increasing successor block size would cause load imbalance as the number of vertices loop in increases. Therefore, by adjusting the successor (or set_2) block size we can achieve better cache utilization while minimizing the load imbalance.

Smaller block sizes achieve better balancing of work among parallel threads. However, smaller block sizes tend to reduce the cache utilization. Table 3 shows the cache utilization of the SSS algorithm with increasing block sizes (both set_1 and set_2 block sizes are set to value in the table). As can be seen in Table 3, when we increase the block size we see better cache utilization.

4.2 Block Size in Distributed-Memory

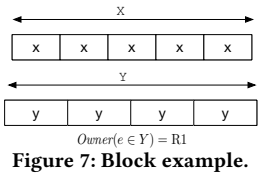


Figure 7: Block example.

In distributed execution, smaller block sizes generate more messages. To explain this better, consider the example in Figure 7. In this example, we are sending set X to the owner of Y. Suppose all the elements of Y belong to the same owner. In a basic algorithm (without blocking) we would send $X+Y$ number of vertices, but in the blocked version we need to send each x size block with each y size block. Not only we are sending $\lceil X/x \rceil * \lceil Y/y \rceil$ number of messages, but also smaller block sizes increase the number of bytes transferred over the network. Therefore, as per Figure 7, the algorithm needs to send each x block with all y blocks; this needs to repeat for each y block. Therefore, the total number vertices transferred is $(Y + x \times Y/y) \times X/x = XY(1/x + 1/y)$. As per the equation, when we decrease the block size, (x or y), the number of vertices transferred increases. Table 4 experimentally shows how the number of bytes communicated over the network increases with the small block sizes.

While vertex blocking and grouping helps to alleviate the load imbalance in shared memory and reduces the number of messages in distributed memory we still see quite a significant number of messages transferred over the network. In Power-law graphs we see fewer high-degree vertices and a larger number of low-degree vertices. There will be at least one message send call per each of the lower-degree vertices. Therefore, algorithm execution still generates a large number of messages when processing a power-low graph. To further reduce the latency overhead incurred by small messages, our next step was to introduce *block aggregation.*

4.3 Block Aggregation

The overhead of sending each of the small messages (less than the size of a block) to a destination is high compared to sending a large message with several of those smaller messages. To reduce the network overhead of sending small messages we use block aggregation.

The size of a block is not fixed and depends on the block size as well as the degree distribution of vertices. To aggregate blocks, we maintain a buffer of bytes per each destination rank. In the send call of an algorithm (e.g., Line 22 in Algorithm 6), block is not directly sent to the destination rank, rather it is stored in a buffer relevant for the intended destination rank. The maximum block size is configurable and specified in the number of bytes.

When a buffer receives a message that cannot fit into the buffer without exceeding the maximum buffer size, the algorithm sends the buffer to the destination and then the buffer is flushed. Parallel shared memory threads store blocks in destination rank buffers. Concurrent access to the buffer is achieved using atomic operations. When a thread attempts to add a block to a destination buffer while the buffer is being sent to the destination, the thread is suspended until a send operation is called. Further, we use non-blocking send-receive operations in our code (i.e., MPI_Isend & MPI_Irecv). Therefore, the time a thread has to wait until an aggregated buffer is transferred is minimized. Figure 8 shows the block aggregation of a triangle counting algorithm running in four ranks. The picture shows the block aggregation in fourth rank and it maintains three outgoing buffers: one per each remote rank. Threads write to buffers as blocks are generated.

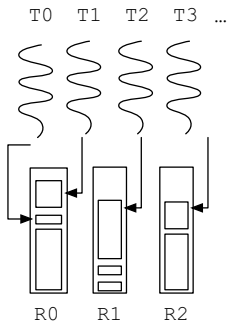


Figure 8: Block aggregation for four destination ranks. Threads add blocks to different destination buffers.

Block aggregation reduces the latency overhead of sending large numbers of small messages. We saw a performance benefit with block aggregation even at very small graph scales. For example, Graph500 scale 13 graph on four ranks takes 7.01 seconds to run "blocked PSP" algorithm and takes 29 seconds to run "blocked SSS" algorithm. With block aggregation both algorithms run in a less than a second.

5 DEGREE BASED PARTITIONING

The performance of distributed, shared-memory parallel triangle counting algorithms can be further improved by reducing the number of bytes transferred over the network. For example, in PSP algorithm, the number of bytes transferred over the network can be reduced by reducing the size of the predecessor set, because the algorithm sends the same predecessor set to multiple ranks (where successor groups are located). To reduce the size of the *duplicating set* (e.g., predecessor set in PSP) to every rank, we use a degree based neighbor partitioning scheme.

Traditionally, triangle counting algorithms partition neighbors using lexicographical comparison. In the proposed approach we partitioned neighbors based on the degree of the vertices. For the PSP algorithm, if a vertex neighbor's degree is higher than the vertex's degree, then we add the neighbor to predecessor set. If the neighbor's degree is less than the vertex's degree, we add that to the vertex's successor set. If the neighbor's degree is equal to the vertex's degree we perform a lexicographical comparison on vertex ids and add the neighbor to either to the successor set or the predecessor set.

The degree partitioned PSP algorithm is listed in Algorithm 5. This algorithm consists of two super-steps. In the first super-step (Line 1–Line 6), the algorithm iterates over all the vertices and exchanges the degrees of the vertices. The *SendDegree* (Line 5) function will call the *ReceiveDegree* (Algorithm 5) function in the same rank or in a different rank. The SendDegree call sends local vertex v and v's degree to its neighbor u. When the neighbor(u) receives the degree and the vertex, it first checks whether the received degree is less than its degree; if so, the received vertex is added as a successor of u (See Line 2 in *ReceiveDegree*); otherwise v is added as a predecessor of u (*ReceiveDegree*, Line 4). If degrees are equal, then we compare vertex ids to decide whether a neighbor is a successor or predecessor (*ReceiveDegree*, Line 6–Line 9).

With degree partitioning we cannot use the CSR data structure to extract the predecessor set and successor set for a vertex (as in Figure 2, Section 3). Therefore, to collect the predecessor set and successor set for each vertex a concurrent data structure (*an append buffer*) is used. Two instances of the concurrent data structure is maintained per every local vertex. The first instance stores the predecessors and the second instance stores the successors. In the first super-step of the algorithm, append buffers for a vertex maybe modified by more than one parallel thread. However, after first super-step we do not need to modify the append buffer concurrently. Even though we partition vertex neighbors based on degrees, our set intersection comparison is based on vertex identifiers (an additional conversion function in set intersection reduces the performance as it tends to reduce cache utilization). Therefore, after calculating predecessors and successors per each vertex, they are locally sorted (See Line 7–Line 9).

With the predecessor-successor relationship we can depict the whole graph as a Directed Acyclic Graph (DAG) (Figure 9). Algorithm 5 makes sure that a predecessor of a given vertex has a higher degree than its own degree. This way, Algorithm 5 makes the highest degree vertices sources of the induced DAG. Also, a vertex has fewer number of predecessors than its successor. In other words,

Algorithm 5: Degree Partitioned PSP Algorithm

PSP $G^{\text{local}} = (V, E)$, *predblksz*, *sucblksz*, *nthreads*:

```
1:  epoch {
2:    for each v ∈ V in a parallel thread do
3:      for each u ∈ neighbors(v) do
4:        d_v = degree(v)
5:        SendDegree(u, v, d_v)
6:  }
7:  for each v ∈ V in a parallel thread do
8:    sort(pred(v))
9:    sort(succ(v))
10: ...
11: Rest of code is same as Algorithm 4, PSP (Line 1–Line 23).
```

ReceiveDegree u, v, d_v:

```
1: if d_v < degree(u) then
2:   succ(u).insert(v)
3: else if d_v > degree(u) then
4:   pred(u).insert(v)
5: else
6:   if u > v then
7:     succ(u).insert(v)
8:   else
9:     pred(u).insert(v)
```

Receive *preblock*, *sucblckforrank*:

```
1: Same as code in Algorithm 4, Receive (Line 1–Line 4).
```

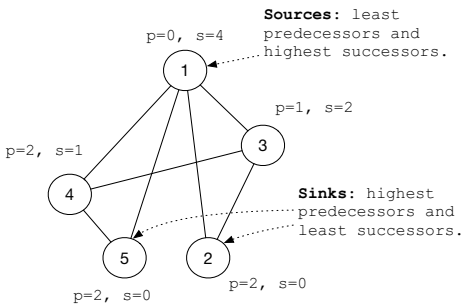


Figure 9: An example DAG and predecessors and successors counts.

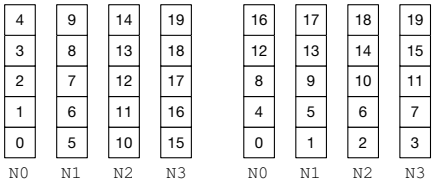
sources have higher number successors, but they have zero predecessors, and sinks of the induced DAG have the least number of successors and highest number of predecessors. Since the partitioning scheme minimizes the number of predecessors, it is favourable for the PSP algorithm. For SPS and SSS algorithms, comparisons in the "ReceiveDegree" function must be switched.

6 RESULTS

The proposed changes (vertex blocking, block aggregration and neighbor partitioning based on degree) to distributed-memory parallel triangle counting algorithms (PSP, SPS, SSS, PPP) improves the performance of distributed memory parallel triangle counting and minimizes the pre-processing overhead. The remainder of this section will present and explain experimental results to demonstrate this claim.

Graph	Vertices	Edges
Friendster	6.83E+07	2.59E+09
Twitter	4.17E+07	1.47E+09
Orkut	3.07E+06	1.17E+08
RMAT-1(25)(rmat1)	3.36E+07	5.37E+08
RMAT-2(25)(rmat2)	3.36E+07	5.37E+08

Table 5: Graph inputs and their attributes used in strong scaling experiments



(a) **(b)**

Figure 10: 1D block distribution and 1D cyclic distribution. "N*i*" is rank id.

We ran our experiments on a Cray XC system that has 2 Broadwell 22-core Intel Xeon processors. Our experiments only used up to 16 cores to uniformly to double the problem size and to double the number of processors in weak scaling. Each node consists of 128 GB DDR4-2400 memory. We use a MPI+PThread, distributed shared-memory runtime. The MPI implementation is Cray MPICH (version 7.4.4).

We evaluate the triangle counting algorithms in terms of *strong scaling* and *weak scaling*. For weak scaling experiments, we use *R-MAT*[5] synthetic graphs. Two types of RMAT synthetic graphs are used. They are: 1.RMAT-1: Graphs based on the current Graph500[20] Breadth First Search benchmark specification with R-MAT parameters $A = 0.57$, $B = C = 0.19$ and $D = 0.05$, and, 2.RMAT-2: Graphs generated based on the proposed Graph500[9] SSSP benchmark specification with R-MAT parameters $A = 0.50$, $B = C = 0.1$ and $D = 0.3$.

In the following sections, the algorithm names starting with "Opt-" are the algorithms that partition neighbors based on vertex degrees (similar to Algorithm 5). Algorithms that do not have the "Opt-" prefix partition neighbors by vertex ids (similar to Algorithm 4).

6.1 Graph Data Distribution Selection

As stated in Section 3.1 we used 1D distribution to distribute vertices among processes. In 1D distribution vertex identifiers can be permuted in several ways. We first experimented with two types of permutations: 1. assign contiguous numbers of vertex identifiers to each node (*1D block distribution*); and, 2. assign vertex identifiers to nodes in round-robin fashion (*1D cyclic distribution*). These two approaches are depicted in Figure 10.

Our initial results showed that the distributed load imbalance is higher when we use the 1D block distribution (Figure 10a) compared to 1D cyclic distribution (Figure 10b). Figure 11 shows the number of comparisons performed in set intersection on a rank when running PPP and Opt-PPP algorithms (on eight ranks). When PPP algorithm is run with block distribution, the number of comparisons performed by each rank is significantly different. However,

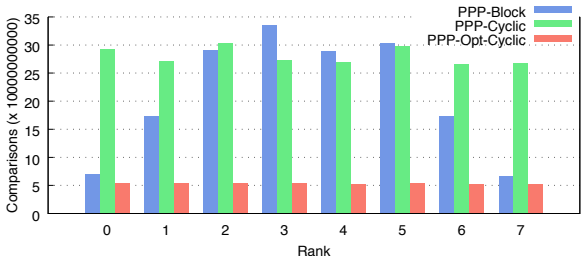


Figure 11: Total set comparisons performed on ranks for PPP algorithm with block and cyclic distributions, Opt-PPP algorithm with cyclic distribution. Input graph : RMAT-1, Scale 24. Threads per rank is 16.

Execution	Algorithm	Time (sec.)	Set Comparisons	Bytes
In-node	PPP	72.68	6.43E+11	–
	Opt-PPP	31.66	1.70E+11	–
Distributed	PPP	21.03	8.11E+11	9.01E+09
	Opt-PPP	8.20	1.70E+11	3.86E+09

Table 6: Number of set-comparisons and network bytes transferred in PPP and Opt-PPP for Scale 23 Graph500 input.

when we use cyclic distribution, the number of comparisons are not skewed as with block distribution.

The triangle counting algorithms discussed in this paper use vertex ids to partition neighbors into successors and predecessors, and one of these sets is sent to the owner nodes of the vertices in the other set. For example, PSP (Algorithm 4) sends the predecessor set of a vertex to its successor's owner's nodes. Since block distribution stores a contiguous number of vertices in a node, more messages are sent to nodes that store greater vertex ids. If we use the block cyclic distribution, messages are not centered on a particular node, but rather, work is divided among participating nodes.

The distributed load imbalance is minimized in optimized algorithms (like Algorithm 5), when they are used with the cyclic distribution. These optimized algorithms push higher-degree vertices to a corner of the DAG, and distribute the set intersections equally, significantly reducing the load imbalance. Figure 11 shows the set comparisons performed on each rank for Opt-PPP algorithm and as can be seen in the plot, the number of comparisons are almost evenly distributed. Therefore, we used 1D cyclic distribution in all our experiments.

6.2 Weak Scaling Results

Weak scaling results for triangle counting algorithms are shown in Figure 12. Triangle counting is *not* a linear time ($O(N)$) algorithm (See [16] for details). Therefore, we do not see constant time scaling, independent of the number of processors for algorithms that partition neighbors by vertex ids (i.e., PSP, PPP, SPS, SSS). However, degree partitioned algorithms show much better scaling behavior.

Table 6 shows the total number of set comparisons in PPP and Opt-PPP algorithms for scale 23, Graph500 input. Shared-memory data is for execution with 16 parallel threads, and distributed execution is for four rank each executing 16 parallel threads. As can be seen in the table, the number of set comparisons for normal PPP algorithms is nearly six times higher than the number of set

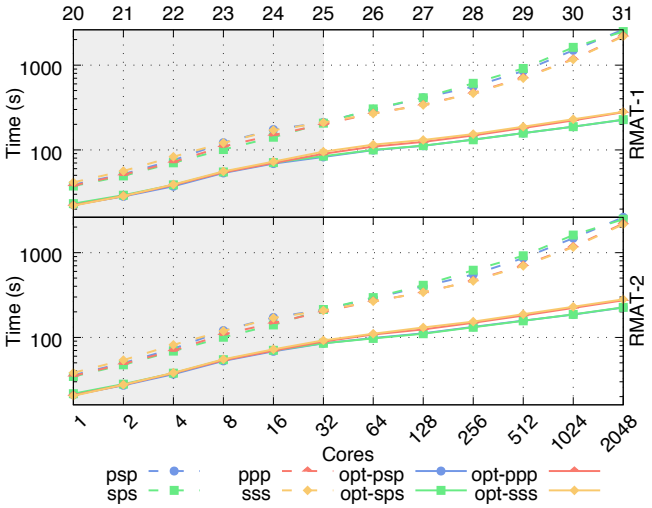


Figure 12: Optimized and non-optimized triangle counting algorithms weak scaling results for RMAT-1 and RMAT-2 graphs. Shaded region shows the shared memory execution.

comparisons for Opt-PPP algorithm. For distributed execution, the number of bytes transferred by the normal PPP algorithm is nearly 2.5 times higher than degree based neighbor partitioned PPP. Also, note that there is not much difference in the number of set comparisons between shared memory execution and distributed memory execution for algorithm Opt-PPP. On the other hand, in PPP algorithms the distributed, shared-memory total set comparisons is higher than the total set comparisons for shared memory.

The degree-based neighbor partitioning algorithm pushes high degree vertices into a corner of the DAG (See Figure 9), minimizes the number of set intersections, and minimizes the amount of data to be transferred. Also, we do not see much difference in the number of set comparisons for shared memory and distributed memory because the sets (set_1 and set_2, e.g., *pred* for PPP algorithm) are small. Sets are small for Opt-PPP because of the degree partitioning. When small set sizes are closer to the block size(s) a minimum amount of data are duplicated (See the discussion in Section 4.2). The PPP algorithm predecessor set sizes are greater than block size, and because of that the same block processed multiple times. Therefore, in distributed settings the total number of set comparisons increases for normal algorithms (i.e., algorithms without "Opt-").

6.3 Strong Scaling Results

For strong scaling experiments, we ran degree partitioned triangle counting algorithms on graphs listed in Table 5 over 1–1024 cores. To gain a better understanding about how algorithms scale relative to each other, we measured Relative Speedup, $= \frac{T_{ref\_1}}{T_n}$ i.e., the ratio of the execution time of the fastest sequential algorithm, T_{ref_1} and the parallel execution time on n processing elements, T_n.

Figure 13 shows strong scaling results for graphs listed in Table 5. As shown in Figure 13 all the graphs show better strong scaling behaviours for degree partitioned algorithms. We see that for the Twitter graph the Opt-PSP and Opt-SPS performs better than Opt-PPP and Opt-SSS. For the Twitter graph, both Opt-PPP and Opt-SSS

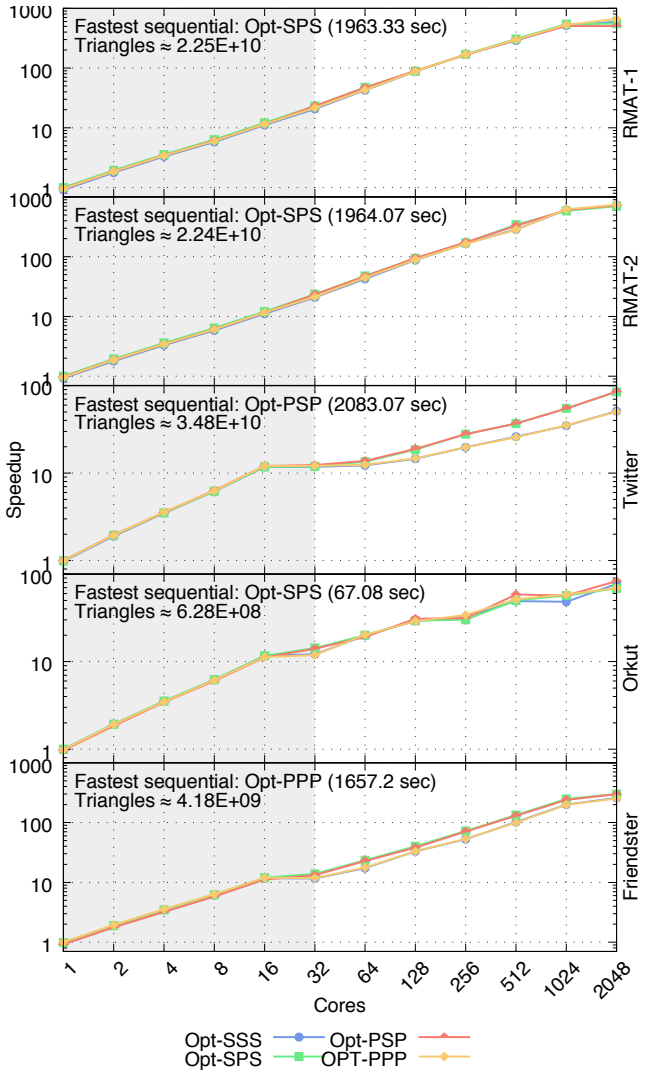


Figure 13: Strong scaling results.

algorithms send more bytes over the network than Opt-PSP and Opt-SPS (e.g., on 64 nodes Opt-PPP sends 3.07E+11 bytes and Opt-PSP sends 2.45E+11 bytes). Therefore, we see that for Twitter graph Opt-SPS and Opt-PSP, algorithms are slightly better than Opt-PPP and Opt-SSS.

6.4 A Comparison with PowerGraph

PowerGraph[8] implements two undirected triangle counting algorithms: *simple_undirected_triangle_counting* (Power-Simple UTC), and, and *undirected_triangle_counting* (Power UTC). Our initial intention was to perform weak scaling for these two triangle counting programs with Graph500 input. However, we observed that both of the triangle counting applications fail with an out of memory error at Graph500 scale 27 and higher in distributed execution. Therefore, we compare the performance of PowerGraph triangle counting algorithms with degree partitioned triangle counting algorithms with

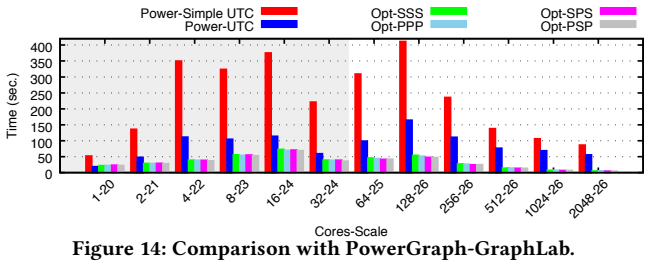


Figure 14: Comparison with PowerGraph-GraphLab.

the maximum scale PowerGraph can run in distributed execution for a given number of nodes.

Figure 14 shows the comparison results. Overall, the PowerGraph undirected triangle counting algorithm performs better than the PowerGraph simple undirected triangle counting algorithm. We see that the undirected triangle counting algorithm shows better performance than the degree partitioned triangle counting algorithm in sequential execution, but in all other executions the degree partitioned algorithms outperform PowerGraph triangle counting algorithms. The performance difference between PowerGraph triangle counting algorithms and degree partitioned triangle counting algorithms is higher at larger scales.

PowerGraph uses Gather-Apply-Scatter (GAS) primitive to perform computations. The undirected triangle counting algorithm (Power UTC) performs better than "Power-Simple UTC" algorithm. According to the logic implemented, the Power-UTC is a hash set version of the triangle counting algorithm in[25]. The implementation maintains a list of all of its neighbors in a hashed set. For each edge (u, v) in the graph algorithm counts the number of set intersections of the neighbor set on u and neighbor set on v and stores the number of intersections on each edge. This algorithm counts each triangle three times.

Algorithms proposed in this paper count triangles only once. Therefore, the number of set intersections are low compared to Power-UTC implementation. Further, Power-UTC shows poor weak-scaling behavior. Also, Power-UTC implementation does not perform operations to reduce the message latency (e.g., aggregation or blocking) and load imbalance.

7 RELATED WORK

Triangle counting has been a well-studied graph problem. There are a number of parallel distributed-memory, parallel shared-memory, and external memory (e.g., [19]) algorithms and implementations. Another classification of triangle counting is whether an algorithm is counting an exact number of triangles or making an approximation (e.g., [14]). In this paper our focus is distributed, shared-memory implementation of triangle counting algorithms that counts the exact number of triangles. Therefore, we limit our review to work that is more closely related to parallel triangle counting algorithms that count exact numbers of triangles.

Shared Memory : *Node-Iterator*[25] is an algorithm that iterates over all vertices and intersects adjacency lists of each pair of vertex neighbors to find the number of triangles. Green et al. [10] optimized this algorithm using vertex-covers. Green et al. also presented a multi-core implementation of Node-iterator in [11] and a GPU implementation in [12]. [26] mainly discuss two parallel cache-oblivious exact triangle counting algorithms for shared memory. The first algorithm merges sorted-directed adjacency lists of a vertex for intersection. The second algorithm uses a hash table to store edges and perform intersections. The paper shows that these algorithms can achieve better cache utilization. Madduri et al. [21] presented variations of triangle counting algorithms and how they related to performance in shared-memory platforms. [28] also discuss number of optimizations to speedup sequential and shared-memory parallel triangle counting algorithms.

Distributed Memory : [1] discuss a Message Passing Interface (MPI) based distributed-memory parallel triangle counting algorithm called *PATRIC*. PATRIC is similar to the PSP algorithm described above. PATRIC uses vertex ids to partition neighbors of a vertex and performs ordering based on degree of a vertex as a pre-processing step.

A number of triangle counting algorithm implementations are available as part of graph processing run-times. PowerGraph[8] discusses a distributed-memory parallel triangle counting algorithm implemented based on gather-apply-scatter primitives. [24] discusses a triangle counting algorithm for distributed (external) memory. The algorithm is based on the asynchronous visitor queue approach presented in the paper and the algorithm is similar to PSP algorithm we discussed above. An extension of this work is presented at the static graph challenge[23].

[2] presents a Matrix based parallel triangle counting algorithm. To minimize the communication the paper also introduces a new matrix algebra primitive: *masked matrix multiplication* and uses Bloom filters[4] to minimize the overhead of communication. Several other linear algebra based triangle counting algorithms are discussed in [30], [13] and [18] (shared-memory). [6] and [27] discuss how triangle counting can be implemented on top of Map-Reduce frameworks. [27], Algorithm 3, finds triangles from the least degree vertex. The underlying idea behind [27], Algorithm 3 is similar to degree based vertex partitioning. However, our method is based on set intersection between adjacencies while [27] is based on a map reduce paradigm that transforms sets of tuples in multiple steps, including generating "special" sets with markers ("$"). Further, algorithms in [27] assumes the vertex degree of an adjacent vertex can be accessed as an attribute of the vertex. This requires additional space to store degree (or adjacencies) as a attribute. The algorithms presented in this paper does not assume remote vertex degree is readily available. [22] discuss an another Map-Reduce triangle counting algorithm that avoids the redundant calculations using a classification method.

Many of the shared-memory triangle counting approaches discussed above are not directly applicable for hybrid memory systems due to high communication overhead. Further, the distributed triangle counting versions does not explore the optimizations that can be achieved using techniques such as vertex blocking, aggregation and involves heavy pre-processing methods.

8 CONCLUSIONS

Triangle counting algorithm performance on hybrid runtimes tends to suffer due to in-node load imbalance, high message communication and poor cache utilization.

In this paper, we showed that different triangle counting algorithms can reduce in-node load imbalance and improve cache utilization by blocking vertices. For distributed execution the block aggregation alleviates the overhead of sending many small messages to a destination. To further reduce the number of set comparisons in set intersection and to reduce the amount of remote communication we presented degree-based vertex neighbor partitioning approach for triangle counting algorithms that consists of two super-steps. In Section 2, we showed that different triangle counting algorithms can be modeled using a generic framework, and showed how these techniques are applicable to the generic model and hence to all triangle counting algorithms presented in Section 2.

The performance results show that the presented algorithms scale well with the problem size as well as with the number of parallel processors. Further, the comparison with PowerGraph triangle counting algorithms demonstrate that presented algorithms outperform PowerGraph triangle counting algorithms both in terms of time and space.

9 ACKNOWLEDGMENTS

This research was partially supported by the National Science Foundation, #ACI-1716828 and by by the Defense Advanced Research Projects Agency (DARPA) Hierarchical Identify Verify Exploit program at the DOE Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. Access to computational resources was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU was also supported in part by Lilly Endowment, Inc. Authors would like to thank *Laura Fowler Hopkins* for assistance with reviewing the paper and for useful feedback that greatly improved the manuscript.

REFERENCES

[1] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. 2013. PATRIC: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 529–538.

[2] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 804–811.

[3] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 16–24.

[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining.. In *SDM*, Vol. 4. SIAM, 442–446.

[6] Jonathan Cohen. 2009. Graph twiddling in a mapreduce world. *Computing in Science & Engineering* 11, 4 (2009), 29–41.

[7] Jean-Pierre Eckmann and Elisha Moses. 2002. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the national academy of sciences* 99, 9 (2002), 5825–5829.

[8] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[9] Graph500Contributors. 2016. Graph 500 Benchmark 1 ("Search"). (2016). http://www.cc.gatech.edu/~jriedy/tmp/graph500/

[10] Oded Green and David A Bader. 2013. Faster clustering coefficient using vertex covers. In *Social Computing (SocialCom), 2013 International Conference on*. IEEE, 321–330.

[11] Oded Green, Lluís-Miquel Munguía, and David A Bader. 2014. Load balanced clustering coefficients. In *Proceedings of the first workshop on Parallel programming for analytics applications*. ACM, 3–10.

[12] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 1–8.

[13] Dylan Hutchison. 2017. Distributed triangle counting in the Graphulo matrix math library. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 1–7.

[14] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis. 2012. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics* 8, 1-2 (2012), 161–185.

[15] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 1343–1350.

[16] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* 407, 1-3 (2008), 458–473.

[17] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[18] Tze Meng Low, Varun Nagaraj Rao, Matthew Lee, Doru Popovici, Franz Franchetti, and Scott McMillan. 2017. First look: Linear algebra-based triangle counting without matrix multiplication. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 1–6.

[19] Bruno Menegola. 2010. An external memory algorithm for listing triangles. (2010).

[20] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the Graph 500 Benchmark. *Cray User's Group (CUG)* (2010).

[21] Sindhuja Parimalarangan, George M Slota, and Kamesh Madduri. 2017. Fast parallel graph triad census and triangle counting on shared-memory platforms. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 1500–1509.

[22] Ha-Myung Park and Chin-Wan Chung. 2013. An efficient MapReduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 539–548.

[23] Roger Pearce. 2017. Triangle counting for scale-free graphs at scale in distributed memory. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 1–4.

[24] Roger Pearce, Maya Gokhale, and Nancy M Amato. 2013. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 825–836.

[25] Thomas Schank. 2007. Algorithmic aspects of triangle-based network analysis. (2007), 26–37.

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

[27] Siddharth Suri and Sergei Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*. ACM, 607–614.

[28] Ancy Sarah Tom, Narayanan Sundaram, Nesreen K Ahmed, Shaden Smith, Stijn Eyerman, Midhunchandra Kodiyath, Ibrahim Hur, Fabrizio Petrini, and George Karypis. 2017. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 1–7.

[29] Howard T Welser, Eric Gleave, Danyel Fisher, and Marc Smith. 2007. Visualizing the signatures of social roles in online discussion groups. *Journal of social structure* 8, 2 (2007), 1–32.

[30] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. 2017. Fast linear algebra-based triangle counting with KokkosKernels. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 1–7.

A GENERALIZED ALGORITHM

Algorithm 6: Generalized Triangle Counting Algorithm with Blocking & Grouping

TC $G^{\text{local}} = (V, E)$, *setablksz*, *setbblksz*, *nthreads*:

1: **for** each parallel thread : *tid* in nthreads **do**
2: **epoch** {
3: **for** each v in V **do**
4: **if** $(|set_1(v)| == 0)$ *or* $(|set_2(v)| == 0)$ **then**
5: continue;
6:
7: offset = (v + tid) % nthreads;
8: nsetablks = $(|set_1(v)|$ + setablksz − 1) / setablksz
9: nsetbblks = $(|set_2(v)|$ + setbblksz − 1) / setbblksz
10:
11: **for** (pos = offset; pos < nsetablks; (pos += nthreads)) **do**
12: setastart = pos * setablksz
13: setaend = *min*(setablksz, ($|set_1(v)|$-setastart)) + setastart
14: preblock = $\{x | x \in set_1(v)$ & (setastart ≤ indexof(x) < setaend)$\}$
15: **for** (setbpos = 0; setbpos < nsetbblks; setbpos++) **do**
16: setbstart = setbpos * setbblksz
17: setbend = *min*(setbblksz, ($|set_2(v)|$-setbstart)) + setbstart
18: setbblock = $\{x | x \in set_2(v)$ & (setbstart ≤ indexof(x) < setbend)$\}$
19: rankgroups = *group-by-rank*(setbblock)
20: **for** each r in ranks **do**
21: setbblckforrank = rankgroups[r]
22: *Send*(r, preblock, setbblckforrank)
23: }

Receive *setablock*, *setbblckforrank*:

1: p = setablock[0]
2: **for** each s in setbblckforrank **do**
3: lowerbound = $\{x | x \in set_1(s)$ and $p \leq x\}$
4: TC += |setablock ∩ lowerbound|