Distributed Triangle Counting
in the Graphulo Matrix Math Library

Dylan Hutchison

University of Washington

Abstract—Triangle counting is a key algorithm for large graph analysis. The Graphulo library provides a framework for implementing graph algorithms on the Apache Accumulo distributed database. In this work we adapt two algorithms for counting triangles, one that uses the adjacency matrix and another that also uses the incidence matrix, to the Graphulo library for server-side processing inside Accumulo. Cloud-based experiments show a similar performance profile for these different approaches on the family of power law Graph500 graphs, for which data skew increasingly bottlenecks. These results motivate the design of skew-aware hybrid algorithms that we propose for future work.

I. INTRODUCTION

Today's data analytics continue to push the envelope in data size and complexity. A class of NoSQL databases based on the Google Bigtable design [1] offers one solution framework: purchase as many commodity machines as needed, and stitch them together into a database cluster that provides performance on top of a bare-bones yet flexible data model that a user can adapt to particular applications. Simple queries such as insert and scan perform well in these frameworks; complex queries such as graph algorithms are difficult to implement in a way that realizes the performance capabilities of the database.

In this work we show a high performance implementation of the Static Graph Challenge [2] on the Apache Accumulo distributed database within the Bigtable family. Specifically we build on the graph processing abstractions provided by Graphulo, a matrix math library for Accumulo tables [3]. Past work on Graphulo has focused on scaling up [4] matrix multiplication [5] as well as the other GraphBLAS matrix math operations [6] and sample algorithms, including the k-Truss algorithm that is part of the Static Graph Challenge [7]. We therefore focus on the remaining algorithm: triangle counting. Triangles are defined as length-3 paths from a node to itself; their frequency has many applications ranging from social network mining and cybersecurity to functional biology and link recommendation [8].

Specifically, we focus on counting triangles in large graphs that exceed main memory. Accumulo stores large graphs on disk (in the Hadoop distributed file system); users expand disk and parallel compute capacity by adding more machines. Burkhardt and Waring demonstrated the performance potential of large scale graph processing with the Accumulo database via a 70 trillion edge (scale 42, 1.1 PB) graph breadth-first search on a cluster of 1200 machines (57.6 TB collective memory) [9]. Main-memory databases, on the other hand, require supercomputer-sized investments in order to process

Input: Unweighted adjacency matrix $\mathbf{A}$
Output: Number of triangles t
1 Split $\mathbf{A}$ into $\mathbf{L} + \mathbf{U}$ *// lower and upper triangle of* $\mathbf{A}$
2 $\mathbf{B} = \mathbf{LU}$ *// matrix multiply*
3 $\mathbf{C} = \mathbf{B} * \mathbf{A}$ *// element-wise multiply (mask) with* $\mathbf{A}$
4 $t = \text{sum}(\mathbf{C})/2$

Algorithm 1: Cohen's triangle counting

large graphs. For example, the highest-scale Graph500 benchmark (based on [10]) submission as of June 2017 conducted breadth-first search on a 32 trillion edge graph (scale 41) with a national supercomputer consisting of 98k machines and over 1.5 PB collective memory.

We assume some familiarity with the Accumulo data model and the Graphulo method for in-database computation. Accumulo's primary computational primitive is the *range scan* over sorted and partitioned ranges of key-value entries that pass them through a series of *server-side iterators*. Graphulo repurposes Accumulo's server-side iterators to read from additional tables and write to tables inside range scans.

The rest of the paper is organized as follows. Section II presents the mathematics and implementation of two methods for counting triangles with Graphulo. Both methods admit a number of interesting optimizations, and so it is unclear which one will scale better a priori. Section III's experiments show that the two methods have similar performance profiles when run on power law data sets. We discuss a possible explanation in that the two methods face a common bottleneck—data skew—which motivates the design of a skew-aware hybrid algorithm we propose for future research. Section IV concludes and comments on the potential of code generation for scaling out graph algorithm programmability to a wider audience.

II. ALGORITHMS

In this section we describe two algorithms for counting the number of triangles in a graph. The first algorithm uses the graph's adjacency matrix as input; the second algorithm uses both the graph's adjacency and incidence matrices as input. Both algorithms require undirected graphs without self-edges.

A. Adjacency-only Triangle Counting

Our first algorithm adapts Cohen's algorithm [11] in order to run in two passes. We briefly review Cohen's algorithm in Algorithm 1. Cohen's algorithm first computes the set of wedges (paths of length 2) by multiplying the lower and upper

Input: Upper triangle of unweighted adjacency matrix $\mathbf{A}$
Output: Number of triangles t
1 $\mathbf{T} = \mathbf{A}$ *// clone $\mathbf{A}$ to $\mathbf{T}$*
2 $\mathbf{T} = \mathbf{T} + \mathrm{triu}(\mathbf{A}^\mathsf{T}\mathbf{A})$ *// upper triangle of matrix multiply*
   // custom multiply: $a \otimes b = 2$ if $a = b = 1$, otherwise 0
3 $\mathbf{T}(\mathbf{T} \% 2 == 0) = 0$ *// filter to odd entries*
4 $t = \mathrm{sum}((\mathbf{T} - 1)/2)$

Algorithm 2: Graphulo Adjacency-only triangle counting

Input: Lower triangle of unweighted adjacency matrix $\mathbf{A}$
Input: Unweighted incidence matrix $\mathbf{E}$
Output: Number of triangles t
1 $\mathbf{T} = \mathrm{triu}(\mathbf{A}^\mathsf{T}\mathbf{E})$ *// upper triangle of matrix multiply*
2 $t = \mathrm{sum}(\mathbf{T} == 2)$ *// count the entries of $\mathbf{T}$ equal to 2*

Algorithm 3: Graphulo Adj.+Incidence triangle counting

triangles of the adjacency matrix $\mathbf{A}$. It then restricts these wedges to the set of wedges that are closed by masking the result with $\mathbf{A}$, which is an element-wise operation. Closed wedges are triangles. The number of triangles is given by counting the number of closed wedges and dividing by two, since each wedge is formed twice in the matrix multiply.

The Graphulo adaptation of Cohen's algorithm is given in Algorithm 2. We made the following changes to Cohen's algorithm in order to reduce the number of intermediate entries produced as well as the number of times each entry is read:

1) Only use the upper triangle of the adjacency matrix by rewriting the matrix multiply $\mathbf{LU}$ as $\mathbf{U}^\mathsf{T}\mathbf{U}$. This form admits the one-pass outer product matrix multiply algorithm [5] and also cuts the input in half. It also cuts the output of $\mathbf{C}$ in half, which is what we would do in line 4 of Cohen's algorithm anyway.

2) Filter the output of the matrix multiply to the upper triangle. Because the lower triangle output of the matrix multiply will be zeroed during the element-wise multiply anyway, the lower triangle can be pruned early, before writing to $\mathbf{T}$.

3) Add the result of the matrix multiply into $\mathbf{A}$ via an Accumulo table clone and a "parity trick" to determine when matrix multiply entries overlap with $\mathbf{A}$. Double each partial product from the matrix multiply, making them all even, which allows entries that overlap with $\mathbf{A}$ to be detected by checking for odd parity. The parity trick eliminates the need to do a further element-wise operation with $\mathbf{A}$.

The parity trick is one way of performing a *masked matrix multiply* in the Accumulo database. In-memory databases can implement masks more directly by holding $\mathbf{A}$ in memory and restricting the output of the matrix multiply to those overlap with $\mathbf{A}$ right away. Because Accumulo is an out-of-core, distributed database, it cannot prune entries outside of $\mathbf{A}$ right away because it cannot hold $\mathbf{A}$ in memory, which is always true for large enough graphs. It could even be the case that partial products are written to separate files, which means that we cannot eagerly check for the presence of the 1 from $\mathbf{A}$. Instead we use the parity trick in a delayed fashion, filtering entries during a scan of $\mathbf{T}$ after all partial products are written.

Given the mathematics in Algorithm 2, we now describe its Graphulo implementation. The row and column of matrix entries are stored in their string encoding in the row and column qualifier of Accumulo entries. With this schema,

Accumulo partitions $\mathbf{A}$ into a set of tablets, each of which contain a sorted, consecutive block of $\mathbf{A}$ at the granularity of rows; every row resides within a single tablet. We choose the particular splits that partition the rows of $\mathbf{A}$ into tablets such that each tablet contains an approximately equal share of $\mathbf{A}$'s entries. Accumulo assigns these tablets evenly among all available tablet servers.[1] Compacting $\mathbf{A}$ ensures these splits take effect.

The table clone of $\mathbf{A}$ to $\mathbf{T}$ is a "copy-on-write" metadata operation that only results in new data files when data is written to $\mathbf{T}$. The cloned table $\mathbf{T}$ has the same splits as $\mathbf{A}$.

The $\mathbf{T} = \mathbf{T} + \mathrm{triu}(\mathbf{A}^\mathsf{T}\mathbf{A})$ is implemented as a fused Graphulo TableMult operation on $\mathbf{A}$ with itself. The TableMult operation uses a custom "row multiply" function that applies the custom multiply function and filters the generated partial products to the upper triangle. These operations run within a scan of each of $\mathbf{A}$'s tablet servers and write their entries to $\mathbf{T}$.

Specifically, the TableMult acts on entries from $\mathbf{A}$ of the form $(r, c, 1)$ and $(r, c', 1)$ where r is a row of $\mathbf{A}$, c and c' are columns of $\mathbf{A}$, and '1' is the value of the entries (since $\mathbf{A}$ is unweighted). The result of the TableMult are entries of the form $(c, c', 2)$ where $c < c'$, and these are written to $\mathbf{T}$. At $\mathbf{T}$, partial products are summed together by standard server-side iterators during flushes (when Accumulo spills entries from memory to disk) and compactions (when Accumulo merges files on disk together).

When the matrix multiplication completes (after all partial products are written to $\mathbf{T}$), a Reduce operation is initiated on $\mathbf{T}$. For each value from the matrix multiply (fully summed together from the partial products), we (1) filter the entries to only accept odd values, (2) transform the value by $v = (v - 1)/2$, and (3) sum together all values. This sum runs independently on each tablet of $\mathbf{T}$, resulting in partial sums which are collected at a client and summed together into a final count of the number of triangles, as per a standard user-defined aggregation pattern [12].

B. Adjacency+Incidence Triangle Counting

Our second algorithm uses both the adjacency and incidence matrix of a graph as input. In contrast to the first algorithm, the second only uses the lower triangle of the adjacency matrix. We define the incidence matrix as the matrix whose rows are vertices, whose columns are edges, and whose values, for vertex v and edge e, are defined as $\mathbf{E}(v, e) = 1$ if e is incident on v and 0 otherwise. This definition requires every column of $\mathbf{E}$ to have exactly 2 nonzero entries.

---

[1] Tablet servers are the worker machines of an Accumulo database.

Our algorithm takes inspiration from Wolf's triangle enumeration algorithm [13]. Algorithm 3 specializes Wolf's algorithm to triangle counting and a Graphulo implementation.

The algorithm identifies triangles by combining two pieces of information: that the presence of a 2 in the $\mathbf{AE}$ indicates that one vertex has a connection to two other vertices, and that these two other vertices because they are connected in the incidence matrix. Our adaptation restricts $\mathbf{A}$ to its lower triangle and the output of the $\mathbf{A}^\mathsf{T}\mathbf{E}$ to its upper triangle in order to eliminate redundant computation.[2]

Because the incidence matrix is not square, the notion of "upper triangle of $\mathbf{A}^\mathsf{T}\mathbf{E}$" requires further explanation. Each vertex is encoded into the rows and columns of $\mathbf{A}$ and the rows of $\mathbf{E}$ as normal. Each edge is encoded into the columns of $\mathbf{E}$ as the concatenation of the vertex labels that the edge is incident on *in ascending order*. Thus, each edge is stored as the pair of vertices $[v_1, v_2]$, where $v_1 < v_2$.[3] We define the upper triangle of $\mathbf{E}$ as the restriction of $\mathbf{E}$ onto only the entries $(v, [v_1, v_2])$ where $v < v_1$.

We now describe Algorithm 3's Graphulo implementation. Like the parity trick in Algorithm 2, we use data format tricks in order to compare vertices to edges and to distinguish counted triangles from lone partial products.

We switched from a string encoding to a fixed 4-byte encoding of the vertex labels[4] in order to facilitate the concatenation and un-concatenation of vertices. Thus, the columns of $\mathbf{E}$ are stored as 8-byte labels composed of two 4-byte vertex labels.

We split $\mathbf{A}$ and $\mathbf{E}$ on their rows into approximately equal sized tablets and compact them. We set the splits of intermediary $\mathbf{T}$ to the same splits as $\mathbf{E}$.

We implemented the matrix multiply $\mathrm{triu}(\mathbf{A}^\mathsf{T}\mathbf{E})$ as another fused Graphulo TableMult. This TableMult eagerly filters its output to the upper triangle. Its output value is an empty (0-byte) value, which serves as a marker for one partial product of the matrix multiply. Because the incidence matrix only has two nonzero values per column, only two partial products per entry are possible. In total, on input $(v, v_1, 1)$ from $\mathbf{A}$ and $(v, [v_2, v_3], 1)$ from $\mathbf{E}$, the TableMult writes the entry $(v_1, [v_2, v_3], '')$ to $\mathbf{T}$ when $v_1 < v_2$.

We run two special aggregation iterators during $\mathbf{T}$'s flushes and compactions in order to pre-sum the result of $\mathrm{sum}(\mathbf{T} == 2)$ while entries are being written to $\mathbf{T}$ in the middle of the matrix multiply. Pre-summing during the matrix multiply is important because it reduces the number of entries written to disk, reducing Accumulo's write bottleneck and speeding up the full sum reduction that takes place once the matrix multiply finishes.

The first aggregation iterator watches for two consecutive empty values that have the same key (in our case, the same row and column). When this condition occurs, it means that we have found an entry in $\mathbf{T}$ whose partial products sum to 2, which indicates a triangle. The first iterator replaces these two entries with empty values with an entry with value 1, to indicate the triangle. This computes $\mathbf{T} == 2$.

The second aggregation iterator sums together values that are numbers (i.e., values that are not empty) irrespective of their keys. Entries with empty values pass through. This iterator effectively performs early aggregation of discovered triangles, which can be done even before the $\mathrm{triu}(\mathbf{A}^\mathsf{T}\mathbf{E})$ matrix multiply finishes.

When the matrix multiply does finish, we initiate a full sum reduction by scanning $\mathbf{T}$. This last scan sums together non-empty values, which may already be partially summed as a result of the second aggregation iterator. A client gathers the local sums from each tablet server and sums those into a final triangle count.

III. Experiments

A. Setup and Results

We ran experiments testing the scalability of the two algorithms on a cloud deployment of Accumulo. We present the experiment details first, followed by results and discussion.

We tested Graphulo's triangle counting algorithms on data from the synthetic Graph500 RMAT unpermuted power law graph generator [14]. We chose to run on power law data because it well models many real world applications [15] while also being experimentally convenient, since the structure of the graph remains the same as graph size increases. The generator creates matrices that range from 2^{10} rows (scale 10) to 2^{20} rows (scale 20), with roughly 16 times that many nonzero entries. In order to create undirected adjacency matrices, we added the resulting matrix to its transpose, eliminated the diagonal, and set all nonzero values to 1.[5]

We deployed Accumulo onto an Amazon EC2 cluster of 12 `m3.xlarge` machines, consisting of 8 tablet servers, 3 coordinators (for Zookeeper, the Hadoop NameNode, and the Accumulo Master), and a monitor machine that tracks the health of the others. Each machine has two 40 GB SSDs and 4 vCPUs on a 2.5 GHz Intel Xeon E5-2670v2. The cost of this cluster is \$3.192/hour, though only the 8 tablet server machines make up a variable cost component in terms of cluster scalability (the 3 coordinators and monitor are essentially fixed costs). For each tablet server, we allocated 8 GB to tablet server Java heap memory, 2 GB to data cache, 1 GB to index cache, and 2 GB to Accumulo's native in-memory maps. Amazon rates the network performance of these machines as "high" but does not guarantee a particular level of throughput or latency. Our experiments are I/O-bound and therefore vulnerable to cloud-based network and disk performance variance [17], but it is unclear how much variance actually affected our experiments.

---

[2]In Wolf's algorithm, every triangle is enumerated three times and counting triangles requires division by 3 after summing entries, just as Cohen's algorithm requires division by 2 after summing entries.

[3]$v_1 \neq v_2$ because there are no self-edges.

[4]Four bytes per vertex is sufficient for this work. The number of bytes per label is not significant in general because Accumulo uses run-length encoding.

[5]To reproduce our power law graphs, download Octave 4.2.1 and D4M [16], set the random seed in Octave as `rand('seed',20160331)`, run the D4M file `KronGraph500NoPerm.m`, eliminate the diagonal, and add the result to its transpose.

We split each table into at most 24 tablets. This number of tablets appeared to provide the best performance during an initial experiment.

Our performance metrics are as follows:

- *runtime* is the best recorded time to count the number of triangles across 2 or 3 runs.
- *nedges* is the number of edges, which is the number of nonzero entries in the upper (or lower) triangle of the adjacency matrix $\mathbf{A}$. The structure of the incidence matrix gives nnz($\mathbf{E}$) = 2*nedges.
- *nppf* is the number of partial products formed as a result of the matrix multiply in each algorithm *after* applying the upper triangle filter.[6] It holds that nppf $\gg$ nedges due to the nature of matrix multiply. The real workload of this task is therefore due to nppf.

Because each partial product is processed at least twice—once by a matrix multiply and once by a reduce—we calculate the processing rate as $2 * nppf/runtime$. This rate more accurately captures the work required to count triangles with these algorithms than nedges / runtime, because the work required is quadratic in each vertex's degree and the input edge count hides this fact. We do not count filtered-out partial products because the computation is I/O-bound and the result of the matrix multiply is far larger than the input, and so filtered-out partial products are not as significant a factor.

Figure 1 plots the runtime of the two triangle counting algorithms on a log-log scale. The two algorithms show very similar performance profiles within an order of magnitude. Table I tabulates the results for closer inspection. Figure 2 plots the processing rate defined above.

B. Comparison to Graph Challenge Baseline

The Graph Challenge baseline implementations are designed for a very different execution environment and problem size than that of Graphulo. When the input graph (and intermediary results) fit into the memory of a single node, we expect that the baseline implementations, all of which are in-memory, will count triangles faster than Graphulo, even without optimizing them with algorithmic tricks or multi-threading. However, the baseline implementations cannot scale to large graph sizes that Graphulo can handle.

We ran the MATLAB baseline implementation[7] on one machine from the Amazon cluster, without any other processes running, and found that the implementation exceeded the machine's 15GB of memory at graphs larger than scale 15 (524k edges).

We plot the baseline runtime up to scale 15 alongside the Graphulo runtime in Figure 1 and in Table I. The baseline runtime include the time to load data from a file into an adjacency and incidence matrix.

While the comparison of Graphulo with a baseline in-memory implementation is a good indicator of triangle counting performance, we caution users on making infrastructure

---

[6]The total number of partial products is a bit more than double nppf.

[7]The MATLAB baseline computes $t = \text{nnz}(\mathbf{AE} == 2)/3$. The intermediary result $\mathbf{AE}$ is the memory bottleneck.

---

decisions solely on the basis of this comparison. Users usually have additional requirements beyond triangle counting, such as indexed access to graph subsets and the ability to apply custom filters. These additional, holistic requirements often favor storage in a database, such as Accumulo or other graph systems, rather than storage in flat files as in the baseline implementation.

C. Discussion: Skew & Hybrid Matrix Multiply

Several observations led us to diagnose skew in the graph's degree distributions (i.e., the presence of high-degree vertices) as a major problem. First we noticed that the choice of row and column encoding—whether to encode the rows as 4-byte fixed-width integers vs. a UTF-8 string encoding—impacted runtime significantly. The adjacency-only algorithm ran 2x faster under the string encoding. We believe that the change of format led to a permutation on the rows and columns that affects the number of partial products computed at each row of $\mathbf{A}$ (due to a permuted ordering) and therefore load balance. Past work on sparse matrix multiplication has likewise noted that permuting the input matrix can reduce runtime [18], and so we expect better performance, to a certain extent, if we explicitly permute the generated input matrices.

We also noticed that the computation's bottleneck shifts as graph size increases. The reduction operation bottlenecks at lower graph sizes, whereas the matrix multiply operation bottlenecks at larger graph sizes (and increasingly more so as graph size increases). A phase transition occurs between scales 15 and 16 (between 520K and 1.05M edges), at which point the time to compute the matrix multiply exceeds the time to compute the reduction. Interestingly, this is also the graph size range at which both algorithms achieve peak processing rate.

We attribute the shift of bottleneck to the power law nature of the input graphs, and in particular a few high-degree vertices. Because matrix multiply is quadratic in the degree of the each vertex, the presence of high-degree vertices leads to skew in the matrix multiply work load among the tablet servers.

We similarly observed these load balancing problems during the matrix multiply (and to a lesser extent during the reduction), in which one tablet server takes far longer to finish than the others. This skew is unavoidable in the sense that, no matter how the rows of $\mathbf{A}$ are partitioned among the tablet servers, some tablet server must have the highest-degree vertex and will take far longer to process that than the other tablet servers. Thus, even though we could have chosen to split $\mathbf{A}$ and $\mathbf{T}$ in a non-uniform manner that is application-specific and possibly even data-dependent (which may preclude their use on general tables), a side experiment showed that these changes skirt the main problem of high-degree nodes.

Both the database literature [19] and the high performance computing literature [20] have studied the problem of skew in detail. We recommend adapting some of their techniques to Graphulo by, for example, inserting the first bits of the column qualifier into the column family and leveraging locality groups in order to implement a 2-D partitioning strategy that spreads
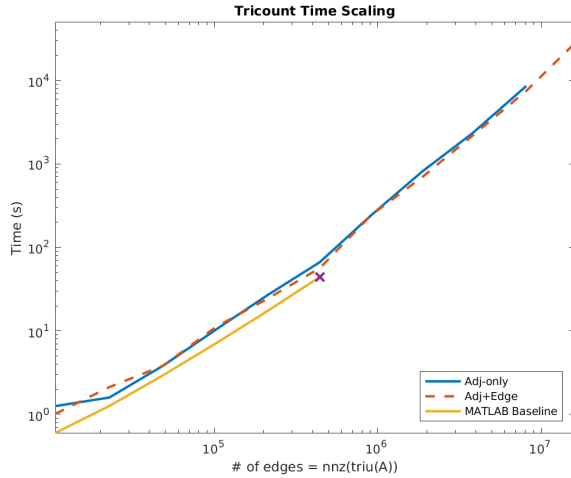
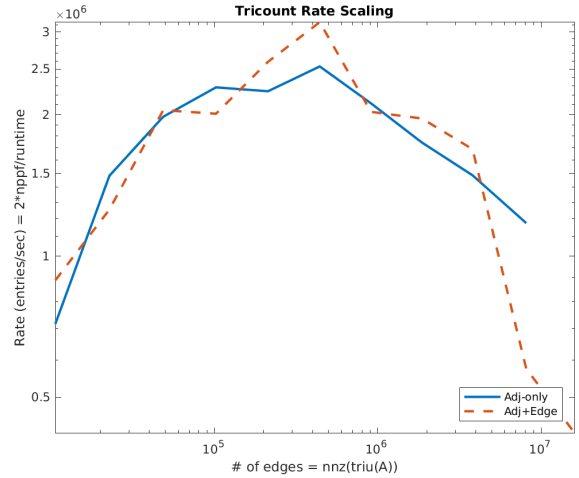Fig. 1: Baseline and Graphulo triangle counting runtime.

Fig. 2: Graphulo processing rate during triangle counting.

| SCALE | # of edges | Adjacency-only Algorithm | | | Adjacency+Incidence Algorithm | | | Baseline |
		nppf (entries)	Time (s)	Rate (entries/s)	nppf (entries)	Time (s)	Rate (entries/s)	Time (s)
10	1.06×10^4	4.50×10^5	1.26	7.17×10^5	4.50×10^5	1.02	8.87×10^5	0.60
11	2.28×10^4	1.18×10^6	1.59	1.48×10^6	1.33×10^6	2.13	1.26×10^6	1.26
12	4.86×10^4	3.80×10^6	3.84	1.98×10^6	3.92×10^6	3.84	2.04×10^6	2.95
13	1.02×10^5	1.19×10^7	1.04×10^1	2.28×10^6	1.12×10^7	1.12×10^1	2.01×10^6	7.11
14	2.13×10^5	3.04×10^7	2.71×10^1	2.24×10^6	3.17×10^7	2.45×10^1	2.59×10^6	1.74×10^1
15	4.42×10^5	8.42×10^7	6.66×10^1	2.53×10^6	8.88×10^7	5.65×10^1	3.15×10^6	4.43×10^1
16	9.10×10^5	2.56×10^8	2.42×10^2	2.11×10^6	2.46×10^8	2.43×10^2	2.03×10^6	
17	1.86×10^6	7.03×10^8	8.06×10^2	1.74×10^6	6.77×10^8	6.90×10^2	1.96×10^6	
18	3.81×10^6	1.73×10^9	2.34×10^3	1.48×10^6	1.84×10^9	2.18×10^3	1.69×10^6	
19	8.13×10^6	5.04×10^9	8.59×10^3	1.17×10^6	2.16×10^9	7.51×10^3	5.75×10^5	
20	1.61×10^7				5.82×10^9	2.77×10^4	4.20×10^5	

TABLE I: Tricount algorithm experiment metrics. nppf = number of partial products after filtering to upper triangle.

high-degree vertices among the tablet servers. This strategy could be applied to every vertex, as in Sparse SUMMA [21], or just to the high-degree vertices in a degree-aware manner, as in Hypercube Join [22].

One way we might solve the problem of high-degree vertices is by reconsidering the 1-D inner product matrix multiply algorithm. In general, inner product matrix multiply is extremely inefficient on the Accumulo database because, for $\mathbf{C} = \mathbf{AB}$, the matrix $\mathbf{B}$ must be read n times in full, where n is the number of rows of $\mathbf{A}$ (or vice versa). This is prohibitive for even moderately sized matrices, since these entail disk reads.

However for this particular matrix multiply, $\mathbf{T} = \mathbf{A}^\mathsf{T}\mathbf{A}$, the inner product algorithm has several advantages. First, only the upper (or lower) triangle is required, which eliminates half the times that $\mathbf{A}$ must be read. More importantly, at the time the entry $\mathbf{T}(r, c)$ would be computed, the corresponding row $\mathbf{A}^\mathsf{T}(r, *)$ and column $\mathbf{A}(*, c)$ are held in memory. The computation can be avoided if $\mathbf{A}(r, c) = 0$ because the resulting entry would be masked by the future element-wise multiply. Last, the inner product algorithm fully sums together the partial products of each entry in the result $\mathbf{T}$. These entries do not need to be materialized because the reduce operation applies immediately; the triangle count can be computed

during the matrix multiply.

The hybrid algorithm we propose is to run inner product on high-degree vertices and outer product on all other vertices. Whereas outer product is expensive on high-degree vertices, inner product handles them efficiently by immediately summing their partial products and avoiding the writing of values that would be zeroed by the element-wise multiply mask. Whereas inner product is too expensive to run on every vertex, running inner product just on the high-degree vertices is reasonable.

In general, our proposed hybrid algorithm exploits the idea that the optimal choice of an operator's implementation depends on its context. In databases, a sort-merge join may outperform an otherwise optimal hash join in databases when followed by a range selection on that same sort order [23]. In high performance computing, the optimal choice of matrix multiplication algorithm changes when the matrix multiply is considered in the context of a loop, rather than in isolation [24]. The same concept may apply here as well, in the sense that an inner product matrix multiply may outperform an otherwise optimal outer product matrix multiply when followed by a mask and full aggregation, as in the triangle counting problem.

IV. Conclusion

In this work we adapted two algorithms for triangle counting, one that uses on the adjacency matrix and another that uses the adjacency and incidence matrix, to the Graphulo library for server-side processing on the Apache Accumulo database. Experiments show a similar performance profile for these different approaches on power law synthetic graphs.

In future work we recommend investigating hybrid algorithms that better handle high-degree vertices, such as the one proposed in Section III-C. We also recommend studying real-world graphs that likely exhibit less skew than the Graph500 power law synthetic graphs whose high levels of skew are unrealistic for many applications.

An alternative strategy is exporting data from Accumulo to an external system and running a specialized algorithm there, such as parallel shared-memory triangle counting [25]. Polystore systems such as Myria [26], BigDAWG [27], and Rheem [28] facilitate using multiple systems together via data movement techniques, as in PipeGen [29] and Portage [30].

User programmability is another important area in the sense of "how easily can users write custom, application-specific graph algorithms without sacrificing performance?" *Code generation* is a modern technique often used (e.g., in SystemML [31]) to bridge the gap between higher-level APIs that are easily programmable and lower-level code that implement efficient data structures and runtime tricks, such as those employed for counting triangles in this work. The LaraDB system [32] prototypes this approach by compiling programs in the high-level Lara algebra to Graphulo iterator code, and we expect further future work to push this approach.

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[2] V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," in *High Performance Extreme Computing (HPEC)*. IEEE, 2017, pp. 1–6.

[3] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for NoSQL databases," in *International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2015.

[4] T. Weale, V. Gadepally, D. Hutchison, and J. Kepner, "Benchmarking the Graphulo processing framework," in *High Performance Extreme Computing (HPEC)*. IEEE, 2016.

[5] D. Hutchison, J. Kepner, V. Gadepally, and A. Fuchs, "Graphulo implementation of server-side sparse matrix multiply in the Accumulo database," in *High Performance Extreme Computing (HPEC)*. IEEE, 2015.

[6] D. Bader, A. Buluç, J. Gilbert, J. Gonzalez, J. Kepner, and T. Mattson, "The graph blas effort and its implications for exascale," in *SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities (EX14)*, 2014.

[7] D. Hutchison, J. Kepner, V. Gadepally, and B. Howe, "From NoSQL Accumulo to NewSQL Graphulo: Design and utility of graph algorithms inside a BigTable database," in *High Performance Extreme Computing (HPEC)*. IEEE, 2016.

[8] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1870–1881, 2013.

[9] P. Burkhardt and C. A. Waring, "A cloud-based approach to big graphs," in *High Performance Extreme Computing (HPEC)*. IEEE, 2015.

[10] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *Cyberinfrastructure Technology Watch*, vol. 2, pp. 1–10, 2006.

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

[12] S. Cohen, "User-defined aggregate functions: bridging theory and practice," in *International Conference on Management of Data (SIGMOD)*. ACM, 2006, pp. 49–60.

[13] M. M. Wolf, J. W. Berry, and D. T. Stark, "A task-based linear algebra building blocks approach for scalable graph analytics," in *High Performance Extreme Computing (HPEC)*. IEEE, 2015.

[14] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *PKDD*, vol. 5. Springer, 2005, pp. 133–145.

[15] V. Gadepally and J. Kepner, "Using a power law distribution to describe big data," in *High Performance Extreme Computing (HPEC)*. IEEE, 2015.

[16] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz *et al.*, "Dynamic distributed dimensional data model (D4M) database and computation system," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012, pp. 5349–5352.

[17] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus in-house cluster: evaluating amazon cluster compute instances for running mpi applications," in *State of the Practice Reports*. ACM, 2011, p. 11.

[18] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[19] P. Beame, P. Koutris, and D. Suciu, "Skew in parallel query processing," in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2014, pp. 212–223.

[20] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, p. 18, 2016.

[21] A. Buluç and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE, 2008, pp. 503–510.

[22] S. Chu, M. Balazinska, and D. Suciu, "From theory to practice: Efficient join query evaluation in a parallel database system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 63–78.

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 1979, pp. 23–34.

[24] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.-Y. Oh, L. Oliker, and K. Yelick, "Communication-avoiding parallel sparse-dense matrix-matrix multiplication," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 842–853.

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

[26] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu, "The Myria big data management and analytics system and cloud service," in *Conference on Innovative Data Systems Research (CIDR)*, 2017. [Online]. Available: https://homes.cs.washington.edu/~magda/papers/wang-cidr17.pdf

[27] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, "The bigdawg polystore system and architecture," in *High Performance Extreme Computing (HPEC)*. IEEE, 2016, pp. 1–6.

[28] D. Agrawal, L. Ba, L. Berti-Equille, S. Chawla, A. Elmagarmid, H. Ham-

mady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse *et al.*, "Rheem: Enabling multi-platform task execution," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 2069–2072.

[29] B. Haynes, A. Cheung, and M. Balazinska, "Pipegen: Data pipe generator for hybrid analytics," in *Proceedings of the Seventh Symposium on Cloud Computing*. ACM, 2016.

[30] A. Dziedzic, A. J. Elmore, and M. Stonebraker, "Data transformation and migration in polystores," in *High Performance Extreme Computing (HPEC)*. IEEE, 2016, pp. 1–6.

[31] M. Boehm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian, "SystemML's optimizer: Plan generation for large-scale machine learning programs." *IEEE Data Eng. Bull.*, vol. 37, no. 3, pp. 52–62, 2014.

[32] D. Hutchison, B. Howe, and D. Suciu, "LaraDB: A minimalist kernel for linear and relational algebra computation," in *SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR)*. ACM, 2017.