Parallel Triangle Counting and Enumeration using Matrix Algebra

Ariful Azad[1], Aydın Buluç[1], John Gilbert[2]
E-mail: azad@lbl.gov, abuluc@lbl.gov, and gilbert@cs.ucsb.edu
[1]Lawrence Berkeley National Laboratory, [2]University of California, Santa Barbara

***Abstract*—Triangle counting and enumeration are important kernels that are used to characterize graphs. They are also used to compute important statistics such as clustering coefficients. We provide a simple exact algorithm that is based on operations on sparse adjacency matrices. By parallelizing the individual sparse matrix operations, we achieve a parallel algorithm for triangle counting. The algorithm is generalizable to triangle enumeration by modifying the semiring that underlies the matrix algebra. We present a new primitive, *masked matrix multiplication*, that can be beneficial especially for the enumeration case. We provide results from an initial implementation for the counting case along with various optimizations for communication reduction and load balance.**

I. Introduction

A triangle is a special type of a subgraph that is commonly used for computing important measures in a graph. The examples include clustering coefficients and transitivity. Finding all triangles is a computationally expensive task, especially for large graphs. The simpler version of the problem reports the counts of triangles (*triangle counting*). Various specialized algorithms, that are either based on the MapReduce framework [1], or based on the wedge (a path of length two) sampling idea [2], has been proposed for triangle counting.

In addition to counting, *triangle enumeration* (sometimes called *triangle listing*) also outputs the actual triangles. Once triangles are enumerated, the output can be used to calculate local (per-vertex) clustering coefficients. The state-of-the-art algorithm for dense neighborhood discovery uses triangles enumeration as a subroutine [3]. Ortmann and Brandes [4] presented a framework unifying all triangle enumeration algorithms published to date. According to them, all subsequently published algorithms are variations of the algorithm presented by Chiba and Nishizeki [5], which, in our view, can be seen as a special case of the row-by-row implementation of sparse matrix-matrix multiplication itself.

In this work, we present algorithms for exact triangle counting and enumeration, which are based on matrix algebra. Many graph algorithms have been previously presented in the language of linear algebra [6]. Many others such as network alignment [7] and maximal independent set [8], are constantly being discovered. Avron [9] previously described a matrix-based approximation algorithm for triangle counting that uses Monte-Carlo simulation and the trace of the cube of the adjacency matrix. Satish et al. [10] counted triangles by explicitly computing the cube of the adjacency matrix in their evaluation of graph frameworks, which is unnecessarily expensive compared to the algorithm we present in this paper.

The complexity of our algorithm is bounded by the time to compute the product of the lower triangular and upper triangular parts of the sparse adjacency matrix. We parallelize our algorithm using a 1D decomposition of sparse matrices, due to its ease of analysis. To minimize communication, we introduce a new primitive, *masked matrix multiplication* on sparse matrices (MaskedSpGEMM), that can avoid communicating edges of a triad (an open wedge) that is known not to form a triangle (a closed wedge), at the expense of communicating information about the output structure. While MaskedSpGEMM is a general operation, in our restricted context of triangle counting and enumeration, the nonzero structure of the adjacency matrix provides the output mask. We use Bloom filters [11] to minimize the overhead of communicating this output mask with other processors.

II. Serial Algorithm for Triangle Counting

A. The Baseline Algorithm

The serial matrix-based algorithm is based on Cohen's MapReduce algorithm for counting triangles [12]. Given an adjacency matrix $\mathbf{A}$, the algorithm first orders rows with increasing number of non zeros they contain. It then splits the matrix into a lower triangular and an upper triangular pieces via $\mathbf{A} = \mathbf{L} + \mathbf{U}$. The lower triangular piece holds edges (i, j) where $i > j$, and the upper triangular piece holds edges (i, j) where $i < j$. In graph terms, the multiplication of $\mathbf{L}$ by $\mathbf{U}$ counts all the wedges of (i, j, k) form where j is the smallest numbered vertex (in our case, is the vertex with lowest degree). More precisely, $\mathbf{B}(i, k)$ captures the count for (i, j, k) wedges where $\mathbf{B} = \mathbf{L} \cdot \mathbf{U}$. The final step is to find if the wedges close by doing element-wise multiplication with the original matrix, i.e. $\mathbf{C} = \mathbf{A} .* \mathbf{B}$.

The $\mathbf{A} = \mathbf{L} + \mathbf{U}$ splitting need not be into lower triangular $\mathbf{L}$ and upper triangular $\mathbf{U}$. An alternative is to not order rows of $\mathbf{A}$ by vertex degree, but rather to identify a total ordering $perm$ of the vertices that is consistent with vertex degree. For each symmetric nonzero pair $\mathbf{A}(i, j) = \mathbf{A}(j, i) = 1$ one can put one of them into $\mathbf{L}$ and the other into $\mathbf{U}$ depending on the ordering of $perm(i)$ and $perm(j)$. In this paper, we do not explore this alternative formulation.

B. Complexity

For an undirected graph with m edges and n vertices, $nnz(\mathbf{L}) = nnz(\mathbf{U}) = m$. We analyze the complexity of our algorithm for an Erdős-Rényi graph with average degree d. The multiplication $\mathbf{B} = \mathbf{L} \cdot \mathbf{U}$ costs d^2n operations. However, the ultimate matrix $\mathbf{C}$ has at most $2dn$ nonzeros. There is a factor of $O(n/d)$ redundancy in the number of arithmetic operations because not all wedges computed end up being closed by an edge. For operations on sparse graphs, it is typical that arithmetic operations are free and the time is dominated by memory references. Consequently, the naïve approach to eliminate flops by skipping scalar multiplications that can not contribute to $\mathbf{C}$ is ineffective because the moment $\mathbf{L}(i, j)$ and $\mathbf{U}(j, k)$ is accessed, the penalty has been paid in terms of memory bandwidth costs.

C. Algorithm using Masked Multiplication

The sparse matrix-matrix multiplication, SpGEMM$(\mathbf{L}, \mathbf{U})$, has a special structure that we can exploit. In particular, the nonzero structure of the output has to be contained in another matrix $\mathbf{A}$. To take full advantage of this property and avoid communicating data for nonzeros that fall outside this nonzero structure, we introduce the "masked" SpGEMM of the form $\mathbf{C} = $ MaskedSpGEMM$(\mathbf{L}, \mathbf{U}, \mathbf{A})$. Here, the operands of the multiplication are $\mathbf{L}$ and $\mathbf{U}$, whereas $\mathbf{A}$ serves as the output mask. Mathematically, $\mathbf{C} = $ MaskedSpGEMM$(\mathbf{L}, \mathbf{U}, \mathbf{A})$ is equivalent to performing $\mathbf{B} = \mathbf{L} \cdot \mathbf{U}$ followed by $\mathbf{C} = \mathbf{A} .* \mathbf{B}$. Computationally, it can be made faster, by avoiding communication.

Using existing terminology, we say that $\mathbf{A}$ provides a "one-at-a-time" structure prediction on $\mathbf{C} = $ MaskedSpGEMM$(\mathbf{L}, \mathbf{U}, \mathbf{A})$ because each position in $\mathbf{C}$ can be made nonzero but there is no guarantee that all can be made nonzero at the same time (which would depend on the exact nonzero structures of the operands) [13].

III. Parallel Algorithm for Triangle Counting

Our parallel algorithm is based on the 1D decomposition of matrices. Here, we describe the algorithm column-wise but the row-wise version is symmetrical. The algorithm is an extension of the so-called "Improved 1D Algorithm" described previously [14], [15]. For SpGEMM of the form $\mathbf{B} = \mathbf{L} \cdot \mathbf{U}$, where matrices are distributed in block columns, the local computation performed by processor P_i is $\mathbf{B}_i = \mathbf{L} \cdot \mathbf{U}_i$. Hence, each processor needs only its own piece of $\mathbf{U}$ to compute its piece of the output $\mathbf{B}$, but it might need all of $\mathbf{L}$ in the worst (dense) case.

More precisely, the basic algorithm is provided in Figure 1. Each processor stores a block of n/p columns. $\mathbf{A}_i$ denotes the $n \times n/p$ slice of $\mathbf{A}$ owned by processor P_i. Given vectors I and J of row and column indices, SpRef extracts the submatrix $\mathbf{A}(I, J)$. We use the MATLAB colon notation to describe whole rows or columns. For example, SpRef$(\mathbf{A}, i, :)$ extract the whole ith row of $\mathbf{A}$

```
BasicTriangleCount(A)
1  (L, U) ← Split(A)
2  for all processors P_i in parallel
3    do Urowsum ← Sum(U_i, rows) ▷ Sum along rows
4       J ← NzIndices(Urowsum)
5       JS ← MPI_Allgatherv(J)
6       for j ← 1 to p
7         do LSpack(j) ← SpRef(L_i, :, JS(j))
8       LSrecv ← MPI_Alltoall(LSpack)
9       Lrecv ← Concatenate(LSrecv)
10      B_i ← SpGEMM(Lrecv, U_i)
11      C_i ← Mask(A_i, B_i) ▷ C = A .* B
12      localcnt ← Sum(Sum(C_i, cols), rows)
13 return MPI_Reduce(localcnt) ▷ Return global sum
```

Fig. 1. The basic parallel algorithm for the exact triangle counting problem using improved 1D SpGEMM on p processors. The input $\mathbf{A}$ is a sparse boolean adjacency matrix that represents an undirected graph without self loops. The set of indices J computed in line 4 is the list of nonzero rows in $\mathbf{U}_i$. All functions that require communication are prefixed with MPI_.

A. Communication Complexity of the Parallel Algorithm

For our parallel complexity analysis, we assume that edges are independently and identically distributed, meaning that the subrow's adjacencies can be accurately modeled as a sequence of Bernoulli trials. This assumption is only for ease of analysis and does not limit the applicability of our algorithm to other graphs with more skewed degree distributions.

The main observation behind the improved algorithm was that not all of $\mathbf{L}$ is needed for the sparse case. Specifically, whether or not $\mathbf{L}(:, k)$ (the kth column of $\mathbf{L}$) is needed by processor P_i depends on whether $\mathbf{U}_i(k, :)$ (the kth subrow owned by P_i) has at least one nonzero. The probability of such a subrow being empty is

$$Pr\{nnz(\mathbf{U}_i(k, :)) = 0\} = (1 - \frac{d}{n})^{n/p},$$

as already analyzed before [14]. The Binomial expansion is:

$$(1 - \frac{d}{n})^{P/n} = 1 - \frac{n}{p}\frac{d}{n} + \frac{O((n/p)^2(d/n)^2)}{2!} - O((d/p)^3)$$

As long as $d < p$, the linear approximation is accurate and yields

$$Pr\{nnz(\mathbf{U}_i(k, :)) = 0\} = 1 - \frac{d}{p}.$$

Hence the number of columns of $\mathbf{L}$ needed by P_i is

$$n \cdot Pr\{nnz(\mathbf{U}_i(k, :)) \neq 0\} = \frac{nd}{p}.$$

MaskedSpGEMM provides further opportunities to reduce this communication volume. In our case, the mask has the same structure as the union of two operands, so it has expected $O(dn)$ nonzeros as well. Since semantically

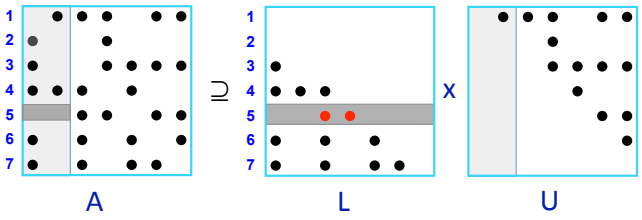

Fig. 2. The effect of one-at-a-time structure prediction on communication costs of SpGEMM. $\mathbf{A} \supseteq \mathbf{L} \cdot \mathbf{U}$ notation means that the nonzero structure of $\mathbf{L} \cdot \mathbf{U}$ has to be contained within the nonzero structure of $\mathbf{A}$. In this example, the nonzeros in the fifth row of $\mathbf{L}$ (marked with red color) are not needed by the first processor (who owns the first two columns) because the fifth subrow of the output is not allowed to contain any nonzeros.

$$\mathbf{C}(i,j) = \mathbf{A}(i,j) .* \Big(\sum_k \mathbf{L}(i,k)\mathbf{U}(k,j) \Big),$$

an empty subrow in the mask (i.e. $nnz(\mathbf{A}_i(l,:)) = 0$) means the corresponding (full) row in the first operand, $\mathbf{L}(l,:)$, is also not needed by P_i. This is illustrated in Figure 2. Exploiting this observation, we achieve the first version of our "masked" algorithm, shown in Figure 3.

Following a similar probabilistic analysis, the number of rows of $\mathbf{L}$ needed by P_i also turns out to be nd/p. Put together, each processor needs an nd/p-by-nd/p submatrix of $\mathbf{L}$ to compute its own piece of the output. In expectation, this would mean a communication volume of

$$\text{Vol}(P_i) = \frac{nd}{p} \cdot \frac{nd}{p} \cdot \frac{d}{n} = \frac{nd^3}{p^2}$$

nonzeros. Since each processor is assumed to have enough memory to hold its piece of inputs, which contains nd/p nonzeros, this data volume is small enough to gather all at once at the target processor (assuming $d < p$). Furthermore, MaskedSpGEMM provides a communication volume reduction of a factor of p/d over the general improved 1D SpGEMM algorithm, which is significant for large p.

Unfortunately, this reduction in the cost of communicating submatrices $\mathbf{L}_i$ comes at the expense of index traffic for requesting a specific subset of rows. In particular, $len(I) = nd/p$. The version of the algorithm that uses Bloom filters to reduce this communication cost is identical to the masked algorithm listed in Figure 3, except the set of row indices I computed in line 6 is further compressed into a Bloom filter before communication. Instead of sending explicit indices, which are typically 64-bit integers each, the Bloom filter only sends a lossy sketch, which is approximately 4 bits per entry in our case. The loss in a Bloom filter is one sided, in the sense that it only has false positives but no false negatives. Consequently, no entry of $\mathbf{L}$ that would be needed in SpGEMM in line 13 can be missing, only a small percentage of redundant entries can be potentially sent (up to the Bloom filter error rate).

B. Computational Costs

The challenging part of making the parallel algorithm scalable is to design the data structures and primitives accordingly

MaskedTriangleCount($\mathbf{A}$)

```
1  (L, U) ← Split(A)
2  for all processors P_i in parallel
3    do Urowsum ← Sum(U_i, rows) ▷ Sum along rows
4       Arowsum ← Sum(A_i, rows) ▷ Sum along rows
5       J ← NzIndices(Urowsum)
6       I ← NzIndices(Arowsum)
7       IS ← MPI_Allgatherv(I)
8       JS ← MPI_Allgatherv(J)
9       for j ← 1 to p
10        do LSpack(j) ← SpRef(L_i, JS(j), IS(j))
11      LSrecv ← MPI_Alltoall(LSpack)
12      Lrecv ← Concatenate(LSrecv)
13      B_i ← SpGEMM(Lrecv, U_i)
14      C_i ← Mask(A_i, B_i) ▷ C = A .* B
15      localcnt ← Sum(Sum(C_i, cols), rows)
16   return MPI_Reduce(localcnt) ▷ Return global sum
```

Fig. 3. The masked parallel algorithm for the exact triangle counting problem using improved 1D SpGEMM on p processors. The input $\mathbf{A}$ is a sparse boolean adjacency matrix that represents an undirected graph without self loops. All functions that require communication are prefixed with MPI_. The code in lines 2–14 is a special case of MaskedSpGEMM. The Mask is still necessary as the communication reduction is only an upper bound.

so that sparse matrix indexing does not become a bottleneck. Each processor on average has to send a nd/p-by-nd/p^2 submatrix to every other processor. Packaging these submatrices for their destinations require p SpRef computations per processor. A naïve implementation of SpRef$(\mathbf{A}, I, J)$ takes time $O(nnz(\mathbf{A}))$ [16] so that would cost

$$\frac{nd}{p} \cdot p = nd$$

per processor, which is clearly unscalable. Instead, we take advantage of the data structures used to store sparse matrices.

We use Compressed Sparse Columns (CSC) to hold submatrices assigned to each processor, a clear choice given the column-wise partitioning of our data. Extracting k columns from a CSC matrix requires touching only those columns. This statement also holds asymptotically for cache complexity, assuming each column has more than a cache line's worth of data (i.e., $d \geq B/8$ where integers are assumed to be 64-bits) on average.

Consequently, we implement our SpRef by first fetching the relevant columns then choosing the appropriate nonzeros with matching row entries within that subset. Fetching nd/p^2 columns of $\mathbf{L}_i$ creates an implicit (because it is not necessarily instantiated) matrix with

$$nnz(\mathbf{L}_i(:,J)) = n \cdot \frac{nd}{p^2} \cdot \frac{d}{n} = \frac{nd^2}{p^2}$$

nonzeros. Serving all other processors by scanning a matrix of this size each take

$$\sum_J nnz(\mathbf{L}_i(:,J)) = p\frac{nd^2}{p^2} = \frac{nd^2}{p}$$

time, which follows the ideal scaling of general SpGEMM.

Similar to the communication analysis, the index vector I could potentially become the bottleneck here. Since $len(I) = nd/p$, we can not afford to touch every entry of IS (the set of I that are received), or the parallel algorithm would run in time $\Omega(n)$ due to scanning all of IS (line 7 in Figure 3). Hence, each $IS(i)$ is represented locally either as a Bloom filter or as a hash table, both of which has $O(1)$ amortized cost for accessing an element.

IV. Extension to Triangle Enumeration

Extending our algorithm to enumerate, as opposed to merely count, triangles is accomplished by changing the underlying semiring. SpGEMM$(\mathbf{L}, \mathbf{U})$ now maps sets of edges to the set of central vertices of wedges. We redefine the binary multiplication, ScalarMult as follows:

$$k \leftarrow \textsc{ScalarMult}(\mathbf{L}(i,k), \mathbf{U}(k,j))$$

In this formulation, ScalarMult either needs to be able to access the indices of its operands, or we simply cast the inputs $\mathbf{L}, \mathbf{U}$ to be of type "pairs of integer indices" as opposed to just booleans. The binary addition, ScalarAdd, is simply defined as list concatenation, creating an output matrix $\mathbf{B}$ whose entries are of type *list*

The masking operator, regardless of it being performed after SpGEMM or being fused into MaskedSpGEMM, is simply defined as:

```
ScalarMask(A(i,j), B(i,j))
1  triangles = {} ▷ Empty list
2  for all entries k in B(i,j)
3    do if A(i,j) ≠ 0
4         do Append(triangles, tuple(i,j,k))
5  return triangles
```

V. Experimental Setup

A. Platform

We evaluate the performance of parallel triangle counting algorithms on Edison, a Cray XC30 supercomputer at NERSC. In Edison, nodes are interconnected with the Cray Aries network using a Dragonfly topology. Each compute node is equipped with 64 GB RAM and two 12-core 2.4 GHz Intel Ivy Bridge processors, each with 30 MB L3 cache. We used Intel C++ Compiler (icc 15.0.1) to compile the code, and Cray's MPI implementation on Edison is based on MPICH2.

B. Implementations

We briefly discuss the implementations of three major functions NzIndices, SpRef, and SpGEMM used by both BasicTriangleCount (Fig. 1) and MaskedTriangleCount (Fig. 3).

Given a matrix $\mathbf{A}$, NzIndices identifies row indices I such that $\mathbf{A}(I,:)$ has at least one nonzero in each row. We implement this function by using a boolean vector v of size n, which is reset to 0 initially. The function scans each nonzero entry of $\mathbf{A}$, and if a nonzero entry is found at row r for the first time, it sets $v(r)$ to 1 and inserts r into I. For efficiency, we created another version of NzIndices that stores I in a Bloom filter. Hence, we have three versions of triangle counting algorithms: (a) Triangle-basic (b) Triangle-masked-bloom and (c) Triangle-masked-list, where the first algorithm is described in Fig. 1 and the second and third algorithms are described in Fig. 3. The differences between Triangle-masked-bloom and Triangle-masked-list lie in the representation of row indices I and the implementation of the SpRef function discussed next.

Assume that the ith processor P_i received the row and column indices I and J of $\mathbf{L}_i$ from another processor. P_i then uses the SpRef function to extract the submatrix $\mathbf{L}_i(I,J)$ and sends it back to the requesting processor. We implemented three different versions of SpRef for three triangle counting algorithms. In Triangle-basic, P_i only receives J and sends back $\mathbf{L}_i(:,J)$. Since we stores $\mathbf{L}_i$ in CSC format, it is trivial to combine the columns indexed by J and create $\mathbf{L}_i(:,J)$. In Triangle-masked-bloom, P_i receives J as an array of indices and I as a Bloom filter. Then, SpRef creates $\mathbf{L}_i(I,J)$ from nonzero row indices of $\mathbf{L}_i(:,J)$ if the indices are present in the received Bloom filter. Hence, the complexity of SpRef used by Triangle-masked-bloom is $O(nnz(\mathbf{L}_i(:,J)))$. Finally, P_i receives both I and J as arrays of indices in Triangle-masked-list algorithm. In this case, each nonzero entry of $\mathbf{L}_i(:,J)$ is searched in I by binary search, hence the complexity $O(nnz(\mathbf{L}_i(:,J)) \log(len(I)))$.

Consider that P_i received necessary submatrices of $\mathbf{L}$ from all other processors and merged the pieces into $Lrecv$. Then, SpGEMM performs $Lrecv \cdot \mathbf{U}_i$. We implemented SpGEMM by using a sparse accumulator (SPA) -based algorithm [17].

C. Dataset

Table I describes a representative set of graphs from the University of Florida sparse matrix collection [18]. In the preprocessing step, we symmetrize each input matrix and remove diagonal entries (self loops) from them. We also randomly permute rows and columns of the matrices for load balancing. The preprocessed matrix $\mathbf{A}$ is considered as the adjacency matrix of an undirected graph and passed to triangle counting algorithm as input. Table II shows the number of nonzeros in matrices generated in different steps of algorithms along with the number of triads and triangles present in the input graph.

VI. Results

A. Relative performance of algorithms

We compare the performance of three parallel algorithms for counting triangles on graphs. The Triangle-basic algorithm performs a distributed SpGEMM to compute $\mathbf{L} \cdot \mathbf{U}$, where the ith processor sends all rows of the requested columns of $\mathbf{L}_i$ to the requesting processors. Both Triangle-masked-bloom and Triangle-masked-list algorithms perform a distributed masked SpGEMM, where the ith processor sends requested rows and columns of $\mathbf{L}_i$ to the requesting processors. In the former

Graph	#Vertices	#Edges	Clustering Coeff.	Description
mouse_gene	45,101	28,967,291	0.4207	Mouse gene regulatory network
coPaperDBLP	540,486	30,491,458	0.6562	Citation networks in DBLP
soc-LiveJournal1	4,847,571	68,993,773	0.1179	LiveJournal online social network
wb-edu	9,845,725	57,156,537	0.0626	Web crawl on .edu domain
cage15	5,154,859	99,199,551	0.1209	DNA electrophoresis, 15 monomers in polymer
europe_osm	50,912,018	108,109,320	0.0028	Europe street networks
hollywood-2009	1,139,905	113,891,327	0.3096	Hollywood movie actor network

TABLE I
TEST PROBLEMS FOR EVALUATING THE TRIANGLE COUNTING ALGORITHMS.

Graph	nnz(A)	$nnz(\mathbf{B} = \mathbf{L} \cdot \mathbf{U})$	$nnz(\mathbf{A} .* \mathbf{B})$	Triads	Triangles
mouse_gene	28,967,291	138,568,561	26,959,674	25,808,000,000	3,619,100,000
coPaperDBLP	30,491,458	48,778,658	28,719,580	2,030,500,000	444,095,058
soc-LiveJournal1	68,993,773	480,215,359	40,229,140	7,269,500,000	285,730,264
wb-edu	57,156,537	57,151,007	32,441,494	12,204,000,000	254,718,147
cage15	99,199,551	405,169,608	48,315,646	895,671,340	36,106,416
europe_osm	108,109,320	57,983,978	121,816	66,584,124	61,710
hollywood-2009	113,891,327	1,010,100,000	104,151,320	47,645,000,000	4,916,400,000

TABLE II
STATISTICS OF TRIANGLE COUNTING ALGORITHM FOR DIFFERENT INPUT GRAPHS.

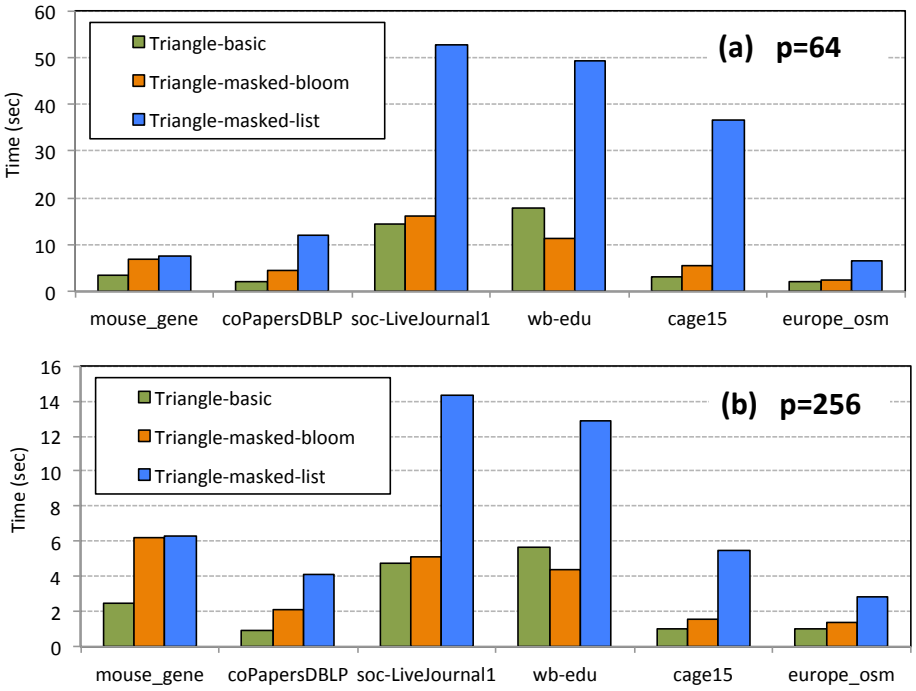

Fig. 4. Runtimes of three triangle counting algorithms for six input graphs on (a) 64 and (b) 256 cores of Edison.

algorithm, a processor sends the indices of necessary rows in a Bloom filter. By contrast, a processor sends an array of necessary row indices in the latter algorithm.

Fig. 4 shows runtimes of three triangle counting algorithms for six input graphs on (a) 64 and (b) 256 cores of Edison. In both subfigures, Triangle-masked-list is the slowest algorithm for all problem instances. On average, over all problem instances, Triangle-masked-bloom runs 2.3x (stdev=0.86) faster on 64 cores and 3.4x (stdev=1.9) faster on 256 cores than Triangle-masked-list. This is due to the fact that sending row indices by Bloom filters saves communication, and SpRef runs faster with the Bloom filter. Since Triangle-masked-list is always slower than Triangle-masked-bloom, we will not discuss the former algorithm in the rest of the paper.

For all graphs except `wb-edu`, Triangle-basic runs faster than the Triangle-masked-bloom algorithm. On average, over all problem instances, Triangle-basic runs 1.6x (stdev=0.7) faster on 64 cores and 1.5x (stdev=0.6) faster on 256 cores than Triangle-masked-bloom. We noticed that the performance difference between these two algorithms is reduced as we

increase p because the communication time to transfer submatrices of $\mathbf{L}$ is expected to increase as we increase p.

B. Scalability

Fig. 5 shows the strong scaling of (a) Triangle-basic and (b) Triangle-masked-bloom algorithms on Edison for different graphs. On 512 cores, the average speedup of parallel Triangle-masked-bloom algorithm is 40.2 (stdev=31.4, min=12.4, max=81.5) relative to the same algorithm on single core. By contrast, the average speedup attained by the parallel Triangle-basic algorithm is 56.5 (stdev=51.4, min=12.5, max=130.2). For both algorithms, the high standard deviations of the speedups reflect huge performance differences across different graphs. Such performance variability is originated from the properties of the graph shown in Tables II and I. The impact of different properties of the input graph is discussed in Section VI-D.

C. Breakdown of runtime

Fig. 6 shows the breakdown of runtimes of (a) Triangle-basic and (b) Triangle-masked-bloom algorithms for different input graphs on 512 cores of Edison. Here, the "Comp-Indices" denotes the time to compute the row and column indices of $\mathbf{L}$ needed by a processor, "Comm-Indices" denotes the communication time to distribute the indices, "SpRef" denotes the computation time to create submatrices of $\mathbf{L}$ requested by other processors, "Comm-L" denotes the communication time to distributed submatrices of $\mathbf{L}$, and "SpGEMM" denotes the computation time to multiply the received submatrices of $\mathbf{L}$ by local submatrix of $\mathbf{U}$. Other steps in the algorithm, such as combining received submatrices of $\mathbf{L}$ into a single matrix, take insignificant amount of time and are not shown here. For all graphs, Triangle-basic spends at least 80% of the time on "Comm-L" and local "SpGEMM". The high fraction of time spent on these two steps takes place because this algorithm does not send row indices when requesting columns of $\mathbf{L}$. Therefore, the serving processor has to send all rows of the requesting columns, which increases the communication volume and the time to perform local SpGEMM. By contrast, Triangle-masked-bloom spends more time on "Comm-Indices" and "SpRef". Distributing indices is more expensive in the Triangle-masked-bloom algorithm because it sends row indices of the necessary columns of $\mathbf{L}$ in a Bloom filter. To serve p processors, the ith processor needs to run p SpRefs with different sets of row and column indices received from different processors. Hence, "Comm-Indices" and "SpRef" become the most expensive steps in the Triangle-masked-bloom algorithm. Since the expected number of nonzeros of $\mathbf{L}_i$ sent from ith processor to every other processor reduces significantly, "Comm-L" and local "SpGEMM" steps become relatively cheap in Triangle-masked-bloom.

We now show how different steps of Triangle-basic and Triangle-masked-bloom scale as we increase the number of processors. Fig. 7 shows the breakdown of time spent by (a) Triangle-basic and (b) Triangle-masked-bloom algorithms on different number of cores for the `coPaperDBLP` graph. As expected, SpGEMM dominates the total runtime on small number of cores for both algorithms. As we increase p, communication time to distribute submatrices of $\mathbf{L}$ becomes dominant in the Triangle-basic algorithm. In fact, this is the primary motivation of designing masked SpGEMM algorithm because the communication time is expected to be the performance bottleneck on large number of processors. By contrast, SpRef becomes the most dominating term on large number of processors for the Triangle-masked-bloom algorithm. Therefore, we reduce the communication time of "Comm-L" at the cost of increasing the computation time of "SpRef". Implementing an efficient SpRef remains a future work.

D. Effect of the clustering coefficient

The clustering coefficient of a graph significantly affects the performance of the Triangle-basic and Triangle-masked-bloom algorithms. Table I shows the clustering coefficient of different input graphs. `europe_osm` has the lowest clustering coefficient mostly because it has only a small number of triangles. Therefore, we consider it an exception and will not consider in the subsequent discussion. On graphs where the clustering coefficient is relatively small, Triangle-masked-bloom benefits from not distributing unnecessary parts of $\mathbf{L}$ and not multiplying them with $\mathbf{U}_i$. Hence, on these graphs, Triangle-masked-bloom is expected to perform better. Indeed, in Fig. 5, we observe that Triangle-masked-bloom outperforms Triangle-basic on `web-edu` graph with the second smallest clustering coefficient (.0626) in our problem set. On two other graphs that have small values of clustering coefficient (`soc-LiveJournal1` and `cage15`), Triangle-masked-bloom performs closely to Triangle-basic. For these two graphs, Triangle-masked-bloom is expected to outperform Triangle-basic with the increased number of processors. By similar argument, on graphs with large clustering coefficient such as `mouse-gene` and `coPaperDBLP`, Triangle-basic runs at least twice as fast as Triangle-masked-bloom.

VII. Future Work

Our implementation should be considered as preliminary and can benefit from various optimizations. In particular, the local data structures can be improved for SpRef. In-node threading would have a quadratic positive effect on reducing memory requirements. Since the requested index vectors I and J are gathered in every process, multithreading would reduce the number of such requests by a factor of t (the number of threads used). Furthermore, each process would now have a factor of t more shared memory. 2D decomposition of matrices can also provide similar improvements. The requested index vectors would be of shorter length (by up to a factor of $\sqrt{p}$) and only $\sqrt{p}$ such vectors (as opposed to p) would need to be gathered in each process. A detailed complexity analysis using the 2D decomposition and a high-performance implementation using the CombBLAS [19] is part of our future work. We also plan to report on comparisons with other parallel algorithms,

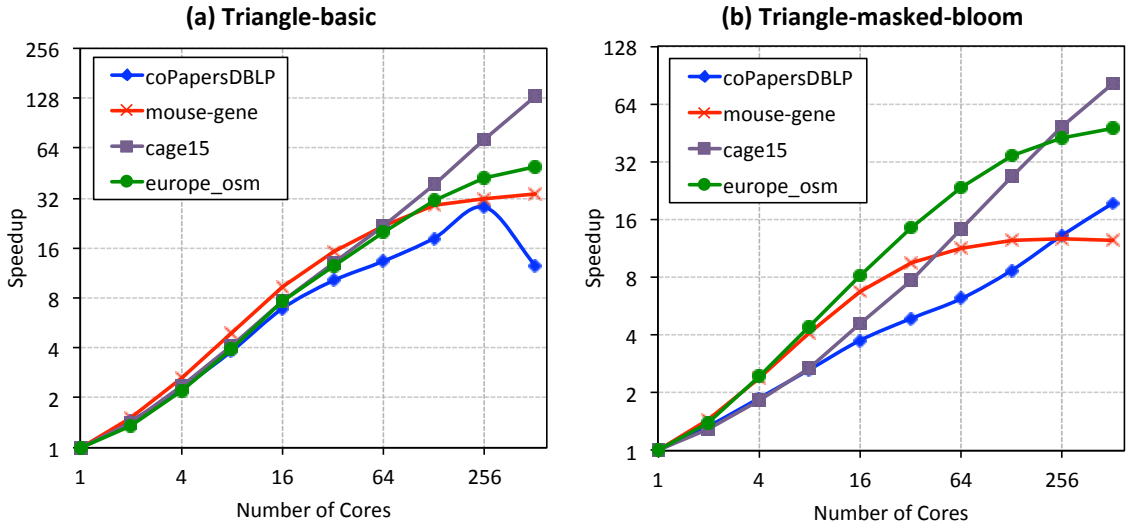

Fig. 5. Strong scaling of (a) Triangle-basic and (b) Triangle-masked-bloom algorithms for four input graphs.

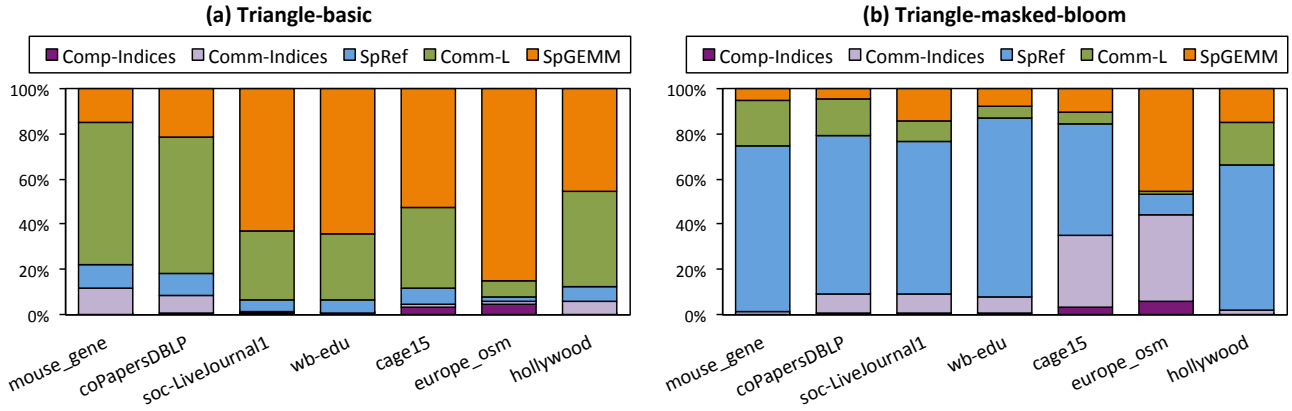

Fig. 6. Breakdown of time spent by (a) Triangle-basic and (b) Triangle-masked-bloom algorithms for different input graphs on 512 cores of Edison.

such as PATRIC [20], which uses MPI, and the multicore implementation of Shun and Tangwongsan [21].

We plan to implement the triangle elimination algorithm presented here as well. For that, we need a high-performance method to pack/unpacks/serialize lists of pairs of indices. Several approaches exists [22] for efficient serialization.

ACKNOWLEDGMENTS

We thank Jon Berry of Sandia for urging us to think about triangle enumeration as well as triangle counting. We also thank Lucas Bang and Adam Lugowski for helpful discussions. Authors from Lawrence Berkeley National Laboratory were supported by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. J. Gilbert was supported by Contract #618442525-57661 from Intel Corp., Contract #8-482526701 from the DOE Office of Science, and by a gift from Microsoft Corp. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with MapReduce," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. S44–S77, 2014.

[2] C. Seshadhri, A. Pinar, and T. G. Kolda, "Wedge sampling for computing clustering coefficients and triangle counts on large graphs," *Statistical Analysis and Data Mining*, vol. 7, no. 4, pp. 294–307, August 2014.

[3] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung, "On triangulation-based dense neighborhood graph discovery," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 58–68, 2010.

[4] M. Ortmann and U. Brandes, "Triangle listing algorithms: Back from the diversion," in *Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX'14)*, 2014, pp. 1–8.

[5] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM Journal on Computing*, vol. 14, no. 1, pp. 210–223, 1985.

[6] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.

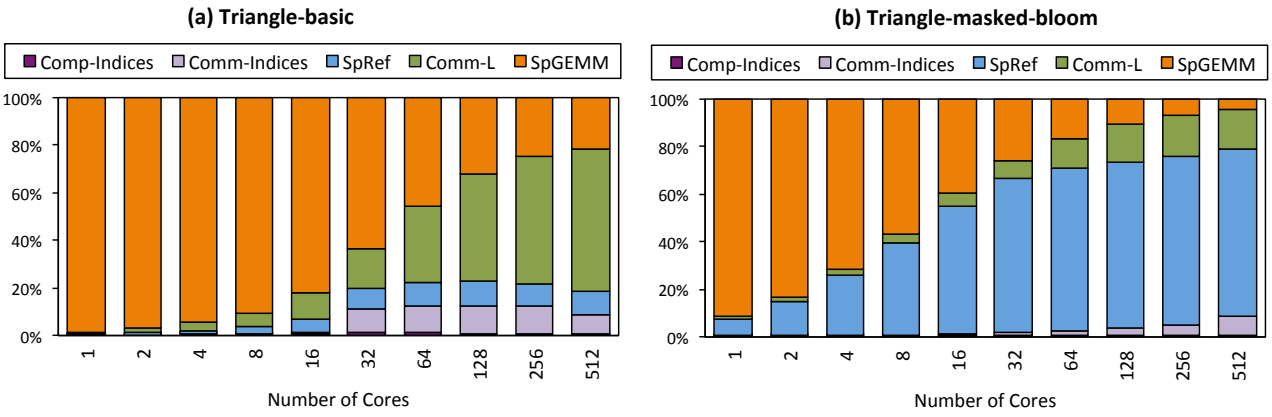

Fig. 7. Breakdown of time spent by (a) Triangle-basic and (b) Triangle-masked-bloom algorithms on different number of cores for the `coPaperDBLP` graph.

[7] M. Bayati, D. F. Gleich, A. Saberi, and Y. Wang, "Message-passing algorithms for sparse network alignment," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 7, no. 1, p. 3, 2013.

[8] A. Buluç, E. Duriakova, A. Fox, J. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams, "High-productivity and high-performance analysis of filtered semantic graphs," in *Proceedings of the IPDPS*. IEEE Computer Society, 2013.

[9] H. Avron, "Counting triangles in large graphs using randomized matrix trace estimation," in *Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

[10] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD'14)*. New York, NY, USA: ACM, 2014, pp. 979–990.

[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

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

[13] J. R. Gilbert, "Predicting structure in sparse matrix computations," *SIAM Journal on Matrix Analysis and Applications*, vol. 15, no. 1, pp. 62–79, 1994.

[14] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *SPAA 2013: The 25th ACM Symposium on Parallelism in Algorithms and Architectures*, Montreal, Canada, 2013.

[15] M. Challacombe, "A general parallel sparse-blocked matrix multiply for linear scaling SCF theory," *Computer physics communications*, vol. 128, no. 1, pp. 93–107, 2000.

[16] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[17] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[18] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 1, 2011.

[19] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496 – 509, 2011.

[20] S. Arifuzzaman, M. Khan, and M. Marathe, "PATRIC: A parallel algorithm for counting triangles in massive networks," in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, 2013, pp. 529–538.

[21] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2015.

[22] W. Tansey and E. Tilevich, "Efficient automated marshaling of C++ data structures for MPI applications," in *Proceedings of the IPDPS*. IEEE, 2008, pp. 1–12.