A Study of APIs for Graph Analytics Workloads

Hochan Lee*, David Wong†, Loc Hoang*, Roshan Dathathri*, Gurbinder Gill*, Vishwesh Jatala*, David Kuck†, and Keshav Pingali*

*The University of Texas at Austin
{hochan,loc,roshan,gill,pingali}@cs.utexas.edu, vishwesh.jatala@austin.texas.edu

†Intel Corporation
{david.c.wong,david.kuck}@intel.com

***Abstract*—Traditionally, parallel graph analytics workloads have been implemented in systems like Pregel, GraphLab, Galois, and Ligra that support graph data structures and graph operations directly. An alternative approach is to express graph workloads in terms of sparse matrix kernels such as sparse matrix-vector and matrix-matrix multiplication. An API for these kernels has been defined by the GraphBLAS project. The SuiteSparse project has implemented this API on shared-memory platforms, and the LAGraph project is building a library of graph algorithms using this API.**

How does the matrix-based approach perform compared to the graph-based approach? Our experiments on a 56 core CPU show that for representative graph workloads, LAGraph/SuiteSparse solutions are $5\times$ slower on the average than Galois solutions. We argue that this performance gap arises from inherent limitations of a matrix-based API: regardless of which architecture a matrix-based algorithm is run on, it is subject to the same inherent limitations of the matrix-based API.

***Index Terms*—graph analytics, performance study, GraphBLAS, high performance computing**

I. Introduction

Graph analytics algorithms are used in many application areas. For example, betweenness centrality [1] can be used to find key actors in terrorist networks [2], [3] or find important river confluence points in a river system [4], and search engines use the pagerank algorithm to estimate the importance of webpages. Systems like Ligra [5] and Galois [6], [7] give users a graph-based API that provides data structures like graphs and worklists as well as constructs for iterating over vertices or edges of the graph in parallel. The GBBS [8] and Lonestar [9] benchmark suites implement graph algorithms using the Ligra and Galois graph-based API, respectively.

Graphs can also be represented with sparse matrices, and some graph algorithms can be implemented using sparse matrix operations to do bulk updates. GraphBLAS [10], [11] is an API for writing such sparse matrix programs. API functions include sparse matrix-vector and matrix-matrix multiplication in which the addition and multiplication operations are generalized to semiring operations. SuiteSparse [12] is a parallel implementation of this API, and a library of graph algorithms called LAGraph [13] uses the GraphBLAS API.

The main advantage of the matrix-based approach is that it permits a separation of concerns: system programmers focus on efficient implementation of the GraphBLAS API while algorithm developers implement portable code for applications on top of the API. This separation of concerns has been successful in areas of high performance computing like dense numerical linear algebra, but how well does it perform for graph workloads in which the approach has not been compared in-depth with a graph-based approach?

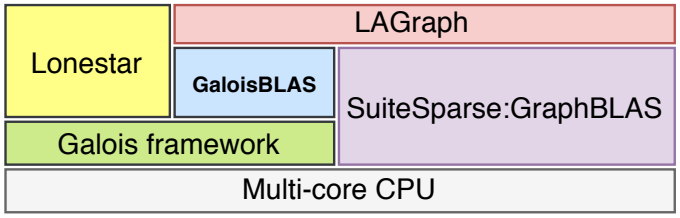


Fig. 1: Systems used in our study.

To address this question, we perform an in-depth study of two systems: LAGraph/SuiteSparse and Lonestar/Galois, which we use as representative systems for the matrix-based and graph-based approaches, respectively. Although expressiveness and portability differ between the matrix and graph-based approaches, in this paper, we focus on the performance differences and reasons behind such differences.

For six widely-used graph problems and nine graphs of various sizes and characteristics, our experimental evaluation on a 56-core CPU shows that Lonestar/Galois programs outperform LAGraph/SuiteSparse programs by $5\times$ on average. To understand how much of this advantage is intrinsic to the graph-based approach and how much of it arises from differences in runtime systems, we also implemented a subset of the GraphBLAS API on top of the Galois runtime to produce a library called GaloisBLAS and studied the performance of LAGraph/GaloisBLAS programs. The relationship among the various systems used in this study is shown in Figure 1.

Our study identifies the following inherent limitations of a matrix-based API and quantifies their impact on performance:

1. **Lightweight loops:** Algorithms written using matrix-based APIs may make several passes over the graph to do very little work in each loop because the matrix-based API does not support composite operations in a single sparse matrix function. With a graph-based API, the application programmer can write a loop that performs the composite operation. For example, Lonestar/Galois is $5\times$ faster than LAGraph/SuiteSparse for breadth-first search on the road-USA graph due to this.

2. **Materialization:** Programs written using matrix-based APIs usually materialize large numbers of intermediate matrices unlike programs written using graph-based APIs. For example, Lonestar/Galois is $3\times$ faster than LAGraph/GaloisBLAS for triangle counting on the uk07 web-crawl primarily because it can avoid materialization.
3. **Bulk operations:** For some problems such as connected components, algorithms like Afforest [14] that perform fine-grained operations on vertices (randomly sampling vertices) are more efficient than algorithms like Shiloach-Vishkin [15] that perform only bulk (or coarse-grained) operations. Such fine-grained operations on vertices cannot be expressed in a matrix-based API. Due to this, Lonestar/Galois is on average $3\times$ faster than LAGraph/-SuiteSparse for connected components on all graphs.
4. **Round-based execution:** The notion of rounds is intrinsic to matrix-based APIs, so asynchronous algorithms in which there is a single worklist and no notion of rounds cannot be expressed. For example, for single-source shortest path (sssp), Lonestar/Galois is more than 300 times faster than LAGraph/SuiteSparse on the eukarya graph primarily due to asynchronous execution.

Although we demonstrate the performance impact of these limitations on a multi-core CPU, these limitations are not restricted to this architecture since they are intrinsic to the matrix-based computation model: any algorithm written with the matrix approach is subject to these limitations regardless of which archictecture the algorithm is run on. Consequently, we believe the performance of matrix-based systems on a GPU or a distributed cluster will likely be worse than the performance on a graph-based system like Galois that supports execution on a GPU [16] and on distributed CPUs or GPUs [17].

In addition to quantifying the limitations, we identify possible avenues of future work to address them. For dense matrix programs, compiler optimizations like loop fusion and scalarization have been used to avoid lightweight loops and materialization. However, this would break the separation of concerns between algorithm developers and system programmers because either the system programmers need to implement architecture-specific code for each possible fused loop or the compiler needs to generate such code automatically. We believe similar restructuring compiler technology is required to improve the performance of GraphBLAS programs.

The rest of the paper is organized as follows. Section II describes graph-based and matrix-based APIs for graph analytics workloads. Section III describes how SuiteSparse and GaloisBLAS are implemented. Sections IV and V present the results of our experimental evaluation using LAGraph/SuiteSparse, LAGraph/GaloisBLAS, and Lonestar/Galois. Section VI discusses related work.

II. APIs for Graph Analytics

This section describes the main features of graph analytics workloads (Section II-A) and shows how these workloads can be expressed in graph-based APIs (Section II-B) and matrix-based APIs (Section II-C).

A. Graph Workloads

In graph analytics workloads, vertices in the graph have labels that are initialized at the start of the computation and then updated repeatedly during the computation until a quiescence condition is reached. Vertex label updates are performed by applying an *operator* to *active vertices* in the graph. In some systems, an operator may read and update an arbitrary portion of the graph surrounding the active vertex [6]. However, most systems support only vertex operators, which read and write the labels of just the active vertex and its immediate neighbors in the graph.

To find active vertices, graph algorithms take one of two approaches. A *topology-driven* algorithm executes in rounds, and in each round, it applies the operator to all graph vertices; the Bellman-Ford algorithm [18] for single-source shortest path (sssp) computation is an example. Topology-driven algorithms are relatively simple to implement since they iterate on the graph topology, but they may not be work-efficient if there are only a few active vertices in each round. *Data-driven* algorithms have no notion of rounds, and they track active vertices explicitly in a worklist. Initially, some vertices are active, and they are placed on the worklist. During execution, threads get active vertices from the worklist and apply the operator to them; if this causes other vertices to become active, they are placed on the worklist. The algorithm terminates when the worklist is empty. Dijkstra [18] and delta-stepping [19] algorithms for sssp are examples. Active vertices can be tracked using a bit-vector of size N if there are N vertices in the graph: we call this a *dense worklist* [20]. Other implementations keep an explicit worklist containing only the active vertices [6]: we call this a *sparse worklist*. Textbook descriptions [18] of data-driven algorithms use sparse worklists.

Some frameworks support *round-based data-driven* algorithms, which can be viewed as a hybrid of the topology-driven and data-driven approaches. They maintain two worklists: *current* and *next*. In each round, the operator is applied only to vertices on the current worklist; if this creates active vertices, they are placed in the next worklist and processed only in the next round.

B. Graph-Based APIs

We use Galois [6], [7] as our example of a graph-API system, although the discussion extends to other graph-API systems as well. The Galois library provides abstract data types (ADTs) for graphs and worklists. The graph ADT contains methods to obtain a random vertex, to obtain the immediate neighbors of a given vertex, to obtain the edges connected to a given vertex, to iterate over all the vertices or edges in a graph, to apply an operator to a given vertex, etc. The worklist ADT contains methods to put and get active vertices. Worklists support soft priorities [21], which permits some vertices to be returned preferentially over others. The library and runtime system inter-operate to ensure that concurrent operations on the ADTs are performed atomically.

Algorithm 1 shows pseudocode for a round-based data-driven breadth-first search (bfs) algorithm written using a

Algorithm 1 BFS with the graph abstraction

```
Require: uint32_t src                          ▷ source vertex number
Require: Graph& graph                          ▷ input graph
 1: Worklist<uint32_t> curr, next;
 2: uint32_t level = 1;
 3: galois::do_all(galois::iterate(graph),
 4: [&](uint32_t node) {    ▷ initialize all vertices in parallel
 5:    graph.getData(node) = BFS::DIST_INFINITY; } );
 6: graph.getData(src) = 1;                    ▷ initialize source
 7: next.emplace(src);                         ▷ initialize worklist
 8: while !next.empty() do
 9:     swap(curr, next);              ▷ switch active worklists
10:     next.clear();
11:     ++level;
12:     galois::do_all(galois::iterate(curr),  ▷ iterate worklist
13:     [&](const uint32_t& frontier_node) {
14:     for (auto& edge : graph.edges(frontier_node)) do
15:         auto dst = graph.getEdgeDst(edge);
16:         uint32_t& dstData = graph.getData(dst);
17:         if dstData == BFS::DIST_INFINITY then
18:             dstData = level;    ▷ mark unvisited with level
19:             next.emplace(dst);    ▷ new vertex to worklist
20:         end if
21:     end for
22:     } );                                  ▷ end parallel do_all loop
23: end while
```

graph-based API. Two empty worklists are created: `curr` and `next` (Line 1). All nodes are initialized to ∞. The source vertex for the `bfs` is an input to the program, and it is added to the worklist after being initialized to an initial value (Line 6-7). The loop between lines 8 and 23 iterates over the vertices in the active worklist until it is empty. The operator is the body of this loop; it iterates over the immediate neighbors of the active vertex and updates their labels if they have not been visited in a prior loop iteration (Line 17-20). Updated nodes are added to the `next` worklist (Line 19), and that worklist is swapped to become the active `curr` worklist for the next iteration of the loop (Line 9). In a parallel execution, threads can get active vertices in parallel from the worklist using a parallel loop construct (Line 12), and each thread runs the loop body in parallel. The graph ADTs in a graph-API system can support parallel accesses and can provide mechanisms for users to synchronize computation among threads.

C. Matrix-Based APIs

A graph $G = (V, E)$ can be represented by its adjacency matrix A of size $|V|\times|V|$ in which entry $A(i, j)$ is 1 if and only if an edge exists from vertex i to vertex j in the graph [22]. If edges have weights on them (like in `sssp` problems) entry $A(i, j)$ is the weight on edge (i, j). For most graphs, the adjacency matrix is very sparse. Vertex labels are represented by a vector of size $|V|$ with one entry for each vertex; multiple labels are represented using multiple vectors.

Algorithm 2 BFS with the matrix abstraction [13], [23]

```
Require: uint32_t src                          ▷ source vertex number
Require: GrB_Matrix A                          ▷ input graph
 1: GrB_Vector frontier = NULL, dist = NULL;
 2: GrB_Index nNodes, nNonZero;
 3: GrB_Matrix_nrows(&nNodes, A);          ▷ #rows=#nodes
 4: GrB_Vector_new(&frontier, bool, nNodes);
 5: GrB_Vector_new(&dist, uint32, nNodes);
 6: GrB_assign(dist, NULL, NULL,
 7:    0, GrB_ALL, n, NULL);
 8: GrB_Vector_setElement(frontier, true, src);    ▷ init wl
 9: int level = 1;
10: for (int64_t level = 1; level ≤ nNodes; level++) do
11:     GrB_assign(dist,        ▷ set distance of frontier nodes
12:        frontier, NULL, level, GrB_ALL, nNodes,NULL);
13:     GrB_Vector_nvals(&nNonZero, frontier);
14:     if nNonZero == 0 then
15:         break;          ▷ terminate if no new work items
16:     end if
17:     GrB_vxm(frontier, dist, NULL,   ▷ find next frontier
18:        LOr_LAnd_Bool_Semiring, frontier, A,
19:        Replace_Complemented_Desc);
20: end for
```

Unlike graph-based APIs which are formulated around operations on individual vertices, matrix-based APIs encourage the use of *bulk operations* on vertices. *The most important of these is matrix-vector product in which the matrix is either the adjacency matrix of the graph or its transpose and the vector is a vector of vertex labels.*

To understand this, consider the operation $y = A^T x$, where A is the adjacency matrix of a graph G, and x and y are vectors of vertex labels. This operation expresses the following vertex operator on the graph: for each vertex j, label $y(j)$ is computed by iterating over the incoming neighbors i and summing up the values of $A(i, j) * x(i)$. Since transposing a vector is easier than transposing a matrix, this operation is usually written as $y^T = x^T A$. We can relate this computation to the concepts in Section II-A as follows.

- A matrix-vector product corresponds to one round of computation.
- If the vector x is dense, it contains entries for all vertices, and the matrix-vector product corresponds to a round of a topology-driven algorithm.
- If the vector x is sparse, only some vertices participate in the computation, and the matrix-vector product corresponds to a round in a round-based data-driven algorithm.
- Implementing the matrix-vector product by iterating over y and computing each element by performing a dot-product of the corresponding row of the matrix and the vector x (this is called SDOT) corresponds to using a pull-style vertex operator. Alternatively, iterating over x, scaling the corresponding column of the matrix with the element of x, and adding the result to y (this is

called SAXPY) corresponds to using a push-style vertex operator. The choice between SDOT or SAXPY depends on how the sparse matrix is represented in memory.
- Redefining the semiring of addition and multiplication enables the implementation of a wide variety of vertex operators; for example, for `sssp` problems, the addition is interpreted as a minimum operation and the multiplication operation is interpreted as an addition operation [22].

Matrix-based APIs also provide sparse general matrix-matrix multiplication (SpGEMM), which is needed for problems like triangle counting. SpGEMM computes $C(i,j) = A(i,:) * B(:,j)$[1]. Implementing this requires knowing how many explicit entries there will be in C since A and B are sparse matrices.

Algorithm 2 shows an LAGraph code snippet for `bfs`. It is a round-based, data-driven, push-style algorithm in which frontier vertices propagate new levels to their out-neighbors in each round. We use this to explain the GraphBLAS API in more detail.

LAGraph `bfs` uses a dense vector *dist* and a sparse vector *frontier*. The vector *dist* keeps track of the distance of each vertex from the source vertex. After the `bfs` computation is finished, it will store the distances of all the vertices which are reachable from the source vertices. The vector *frontier* keeps track of the vertices in the current frontier in each round.

To make the *dist* vector dense, `GrB_assign()` is called (Line 6). The fifth parameter *GrB_ALL* specifies that all entries of *dist* are set to the fourth argument, 0. At line 8, `GrB_Vector_setElement()` adds the source vertex to the empty sparse vector *frontier*.

Line 10 to 20 perform the actual `bfs` computation. Each round consists of three steps or passes.

In the first step (Lines 11-12), `GrB_assign()` assigns new distances to the frontier vertices. The parameters *dist*, *level*, *GrB_ALL*, and *nNodes* specify that the function will assign the current level to the all *nNodes* elements of the *dist* vector. The parameter *frontier* is used as a *mask matrix*. In GraphBLAS, a *mask* is used to filter elements of the output matrix/vector to be updated. For example, if an element *(i, j)* of the mask has a non-zero value, only the corresponding element *(i, j)* of the output can be updated through operations. Therefore, the mask vector *frontier* ensures that only vertices in the current frontier are assigned the new distance.

The next step (Lines 13-16) checks if the current *frontier* is empty. `GrB_Vector_nvals()` returns the number of *explicit* (non-null) entries in the sparse vector. If the frontier is empty, computation terminates as there are no more nodes to visit.

In the last step (Lines 17-19), `GrB_vxm()` constructs the new *frontier* vector. The fourth parameter specifies the semiring for multiply: `bfs` uses *logical-or (LOR)* in place of addition and *logical-and (LAND)* in place of multiplication in the matrix-vector product. Multiplying the *frontier* vector (the fifth parameter) with the adjacency matrix of the graph (the sixth parameter) using the LOR LAND semiring produces a vector that contains all the out-neighbors of the vertices in the current *frontier*; this is the frontier for the next round of execution. To prevent vertices that have already been visited in a previous round from being added to the new frontier, the complement of the *dist* vector is used as a mask (visited entries will have a non-zero value) to prevent the setting of those vertices. The seventh parameter, *Replace_Complemented_Desc*, specifies this behavior for the mask.

[1]The colons used in this paper are MATLAB notation [24]. A(i, :) denotes the ith row of A, and B(:,j) denotes the jth column of B.

D. Discussion

The matrix-based API requires understanding concepts like semirings and mask matrices that are not needed for using the graph-based API. The job of the systems programmer is simplified in the matrix-based API, but the job of the application programmer is arguably more complex. In addition, we make several observations from matrix-based APIs.

1. Programs written in matrix-based APIs require more passes or loops over vertices than those written in graph-based APIs. For example, one round of `bfs` needs 3 loop nests using matrix abstraction (one each for lines 11, 13, and 17 in Algorithm 2), whereas it only needs 1 loop nest using graph abstraction (lines 14-21 in Algorithm 1). Each of these nests also acts as a barrier to parallel execution which limits the benefits of parallelism.
2. Programs for problems like triangle-counting that are written using matrix-based APIs can create a large number of intermediate matrices (This is explained in more detail in Section V).
3. The notion of bulk operations is intrinsic to matrix-based APIs. Algorithms like Afforest [14] that require fine-grained operations on some vertices like a random sample of vertices cannot be expressed.
4. The notion of rounds is intrinsic to matrix-based APIs. Asynchronous data-driven algorithms like Dijkstra [18] or delta-stepping [19], in which there is a single worklist and no notion of rounds, cannot be expressed.
5. Algorithms based on matrix-vector multiplication implement Jacobi iteration: label values generated in one round are available only in the next round. There is no obvious way to allow label values to be read in the same round as the one in which they are written (Gauss-Seidel iteration).

Section V studies the performance impact of these differences experimentally.

III. Implementations of GraphBLAS API

This section describes two implementations of the GraphBLAS API: SuiteSparse [25] and GaloisBLAS, an implementation in the Galois system which we use to control for the effects of using different runtimes in SuiteSparse and Galois.

A. SuiteSparse

Data structures: SuiteSparse represents sparse matrices in Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats. Matrix elements that are manifest

in the compressed format are called *explicit entries*; some may be zeros. Vectors in SuiteSparse are represented as matrices in which one of the dimensions is of length one. This permits economy in the implementation: matrix-vector and vector-matrix product are implemented using matrix-matrix multiplication.

Computation: The core operation in SuiteSparse is sparse matrix-matrix multiplication (SpGEMM). SpGEMM computes $C(i,j) = A(i,:) * B(:,j)$. In relational terms, this can be viewed as a join operation on a row vector of A and a column vector of B. There are two main approaches to implementing SpGEMM.

a) SAXPY: Explicit entries in A are enumerated by row, and for each entry *A(i, k)*, a *SAXPY* operation ($y = ax + y$) is performed with the row vector *B(k, :)* to compute a contribution to *C(i, j)*. The explicit entries in *B(k, :)* can be accessed efficiently since B is stored in CSR format. If the entry *C(i, j)* already exists, the contribution is added to it; otherwise, storage must be allocated for *C(i, j)*.

Parallel implementations of this method handle the creation of new explicit entries in *C* in different ways. The *Gustavson* method uses a dense accumulator [26], [27] whose size is equal to the column dimension of matrix B to store contributions to $C(i,j)$; they are merged into C at the end of computation. *Hash-table-based* matrix multiplication [28] uses a hash table to store intermediate results. It is more memory efficient than Gustavson but requires additional computation for look-ups.

b) SDOT: The dot-product SpGEMM computes C by transposing B to get B^T and taking a dot product of a row of A and a row of B^T: $C(i,j) = A(i,:) \cdot B^T(j,:)$. This requires no intermediate storage if storage for C is allocated before computation. Usually, an inspector is used to determine how much storage to allocate for C [29].

SuiteSparse implements matrix-matrix multiplication using both the SAXPY (Gustavson/hash-table) and SDOT, and it chooses between the two depending on the input matrices. Parallel SAXPY in SuiteSparse divides the rows of CSR A or the columns of CSC B among threads, and the SAXPY operations are done in parallel. SuiteSparse's parallel SDOT also partitions the rows of CSR A or columns of CSC B among threads.

Runtime: SuiteSparse is implemented using OpenMP. Load-balancing is accomplished using SuiteSparse's self-scheduling of work on top of OpenMP static/dynamic balancing.

B. GaloisBLAS

GaloisBLAS is an implementation of a subset of the GraphBLAS API in the Galois system [6] described in Section II.

Data structures: GaloisBLAS uses CSR to represent the graph like SuiteSparse does. Sparse vectors have three representations: ordered map, unordered list, and dense array. The *ordered map* stores explicit entries as key-value pairs in sorted order. The *unordered list* is a thread-safe structure that supports parallel additions/removals. The *dense array* uses the maximum value of the vector type to denote entries that are not explicit. In this study, we chose the best representations for each application, input, and operation. For example, the distance vector of the `bfs` which stores distances for all the nodes uses the *array* type.

Computation: GaloisBLAS supports both parallel SDOT- and SAXPY-based SpGEMM. SAXPY-based SpGEMM uses both the hash-table-based and the Gustavson methods. In this study, we chose the best implementation for each application and input graph. We also implemented an optimized SpGEMM for diagonal matrix times sparse matrix: if a diagonal matrix is detected, then each row of the other matrix is multiplied by the diagonal entry in the row of the diagonal matrix. GaloisBLAS implements custom matrix-vector and vector-matrix products instead of using matrix-matrix multiplication for both.

Runtime: Parallel execution is managed by Galois [6] which provides huge page support, thread binding, and work-stealing.

IV. Experimental Setup

Table I shows the graphs used in our study: they represent a wide variety of graph structures. rmat22 and rmat26 are synthetic power-law graphs [30]. indochina04 and uk07 [31], [32] are web crawls, road-USA-W and road-USA [33] are road networks, twitter40 [34] and friendster [35] are social networks, and eukarya [36] is a protein dataset. The road networks and protein dataset have edge weights. For the other graphs, we generate random edge weights. The size reported is for the CSR representation (including edge weights).

We used six widely-used graph problems as representatives of graph workloads.

bfs: Breadth-first search of a directed graph.
cc: Maximal weakly connected components of a graph.
ktruss: Largest subgraph of an undirected graph in which every edge is in at least (k-2) triangles within the subgraph.
pr: Page-rank computation for a directed graph.
sssp: Single-source shortest-path in a weighted directed graph.
tc: Triangle counting in an undirected graph.

We used LAGraph [13] programs from their repository[2]. We used SuiteSparse:GraphBLAS 3.2.1 [12], [25][3] and Galois 6.0 [6][4]. We used Lonestar programs in the Galois repository. Our GaloisBLAS implementation uses Galois 6.0.

Our experiments were done on Intel Xeon Gold 5120 2.2 GHz CPUs with 56 cores on 4 sockets and 187 GB of DRAM. We reserved huge pages for GaloisBLAS and Lonestar. We do not reserve huge pages for SuiteSparse as we observed that their implementation performs better without them.

For `bfs` and `sssp`, we use the highest degree vertex as the source vertex except for road networks where we use the vertex 0. For `ktruss`, we use a k value of 7 for all graphs except road networks where we use 4. All programs were run to convergence except for `pr`, which we run for 10 iterations.

[2]Commit 896125aa002eab782b44ba745ee6b36b598c35e8 on the LAGraph GitHub at https://github.com/GraphBLAS/LAGraph.

[3]Commit 13f961b5091ee8b74f60a81b30f3987c9628c985 on the GraphBLAS GitHub at https://github.com/DrTimothyAldenDavis/GraphBLAS/

[4]https://github.com/IntelligentSoftwareSystems/Galois/tree/release-6.0

TABLE I: Input graphs and their properties (friendster is an undirected graph while the rest are directed graphs).

	road-USA-W	road-USA	rmat22	indochina04	eukarya	rmat26	twitter40	friendster	uk07
$\|V\|$	6.3M	23.9M	4.2M	7.4M	3.2M	67.1M	41.7M	65.6M	105.9M
$\|E\|$	15.1M	57.7M	67.1M	191.6M	359.7M	1,074M	1,468M	1,806M	3,717M
$\|E\|/\|V\|$	2.4	2.4	16	25.8	110.9	16	35.3	28	35.1
max D_{out}	9	9	98,961	6,984	2,875	0.6M	3M	5,214	15,402
max D_{in}	9	9	0.1M	0.3M	2,875	0.6M	0.8M	5,214	1M
Apprx. Diam.	3,137	6,261	6	2	48	5	12	21	115
CSR Size (GB)	0.2	0.6	0.5	1.5	2.8	8.6	12	28	29

TABLE II: 56-thread execution time in seconds. Fastest runtime is highlighted. TO signifies timeout (2hr), C signifies correctness bug, OOM signifies out of memory. SS: LAGraph on SuiteSparse, GB: LAGraph on GaloisBLAS, LS: Lonestar on Galois.

		road-USA-W	road-USA	rmat22	indochina04	eukarya	rmat26	twitter40	friendster	uk07
bfs	**SS**	1.73	6.06	0.09	0.01	0.18	0.88	1.26	2.61	2.06
	GB	3.23	6.87	0.08	0.01	0.12	0.80	1.06	2.41	1.98
	LS	0.58	1.20	0.04	0.00	0.05	0.59	0.87	2.10	0.50
cc	**SS**	0.33	1.11	0.12	0.36	C	2.00	1.27	2.62	4.95
	GB	0.32	0.82	0.11	0.38	C	1.49	1.22	2.44	4.05
	LS	0.06	0.07	0.09	0.06	0.11	0.82	0.20	1.22	0.45
ktruss	**SS**	0.09	0.33	2449.16	6227.92	891.59	TO	TO	TO	OOM
	GB	0.07	0.31	1681.76	5840.05	847.57	TO	TO	TO	OOM
	LS	0.10	0.21	43.05	497.52	21.63	1722.25	TO	926.15	TO
pr	**SS**	0.15	0.42	0.41	0.65	0.86	9.08	7.23	29.20	9.27
	GB	0.06	0.17	0.16	0.25	0.69	4.64	4.95	19.54	4.38
	LS	0.06	0.17	0.03	0.14	0.30	3.88	4.24	16.54	2.36
sssp	**SS**	15.06	50.32	0.77	0.22	53.05	7.80	12.12	53.41	53.93
	GB	14.92	40.54	0.27	0.08	47.67	2.68	4.89	15.10	33.94
	LS	0.14	0.34	0.17	0.01	0.16	1.66	3.01	11.22	10.15
tc	**SS**	0.05	0.19	9.93	7.58	8.40	400.89	513.80	80.01	OOM
	GB	0.02	0.04	9.05	8.32	7.48	335.29	440.20	96.66	68.09
	LS	0.01	0.06	2.48	6.08	4.03	91.54	42.96	38.17	22.89

We use 56 threads for all inputs unless otherwise mentioned. The reported runtimes for the main experiments do not include graph loading and preprocessing time and are an average of 3 runs. We used a timeout of 2 hours for the runs.

LAGraph provides many variants of each algorithm, so we selected the best performing ones. We use the *basic* variant of `bfs`. We use the FastSV [37] variant for `cc`. We modified LAGraph's `pr` to produce the same answer the Lonestar version does, and GaloisBLAS and SuiteSparse each use the `pr` variant that perform best for that respective system. We use LAGraph's delta-stepping [38] variant 12c for `sssp`. We use SandiaDot variant for `tc`. We use the default variant for `ktruss`. For Lonestar, we use default algorithms with default command line arguments. For `sssp` in all systems, (1) we use the default value of 2^{13} for the delta except for eukarya where we change it to 2^{20}, and (2) we change the 32-bit integer *distance* type to 64-bit only for eukarya. Other than this exception, the algorithm selected or its runtime parameter was not tuned based on the input graph for all systems.

The Intel CapeScripts tool is used to collect performance and energy information dynamically from a running application; we use it to get performance counters in our study. CapeScripts predefines a set of counters (memory traffic, FLOP rates, timing, speculative execution, control unit buffers); the user enables a set of counters, and the script will select specific appropriate counters for each given architecture. After collecting the raw measurement metrics, CapeScripts will combine the results in a machine independent manner.

V. Experimental Results

In this section, we refer to the LAGraph/SuiteSparse, LAGraph/GaloisBLAS, and Lonestar/Galois systems as SuiteSparse (SS), GaloisBLAS (GB), and Lonestar (LS) respectively. First, we give an overview of the main results in Section V-A. We then dive deeper into the reasons for the performance differences by doing a differential analysis of algorithms expressible in matrix and graph-based APIs in Section V-B.

A. Results Overview

Table II shows the main results from our study. The main takeaways are that (i) within the context of matrix-based API systems, GaloisBLAS is competitive to or faster than SuiteSparse in almost all cases and (ii) Lonestar is substantially faster than SuiteSparse and GaloisBLAS.

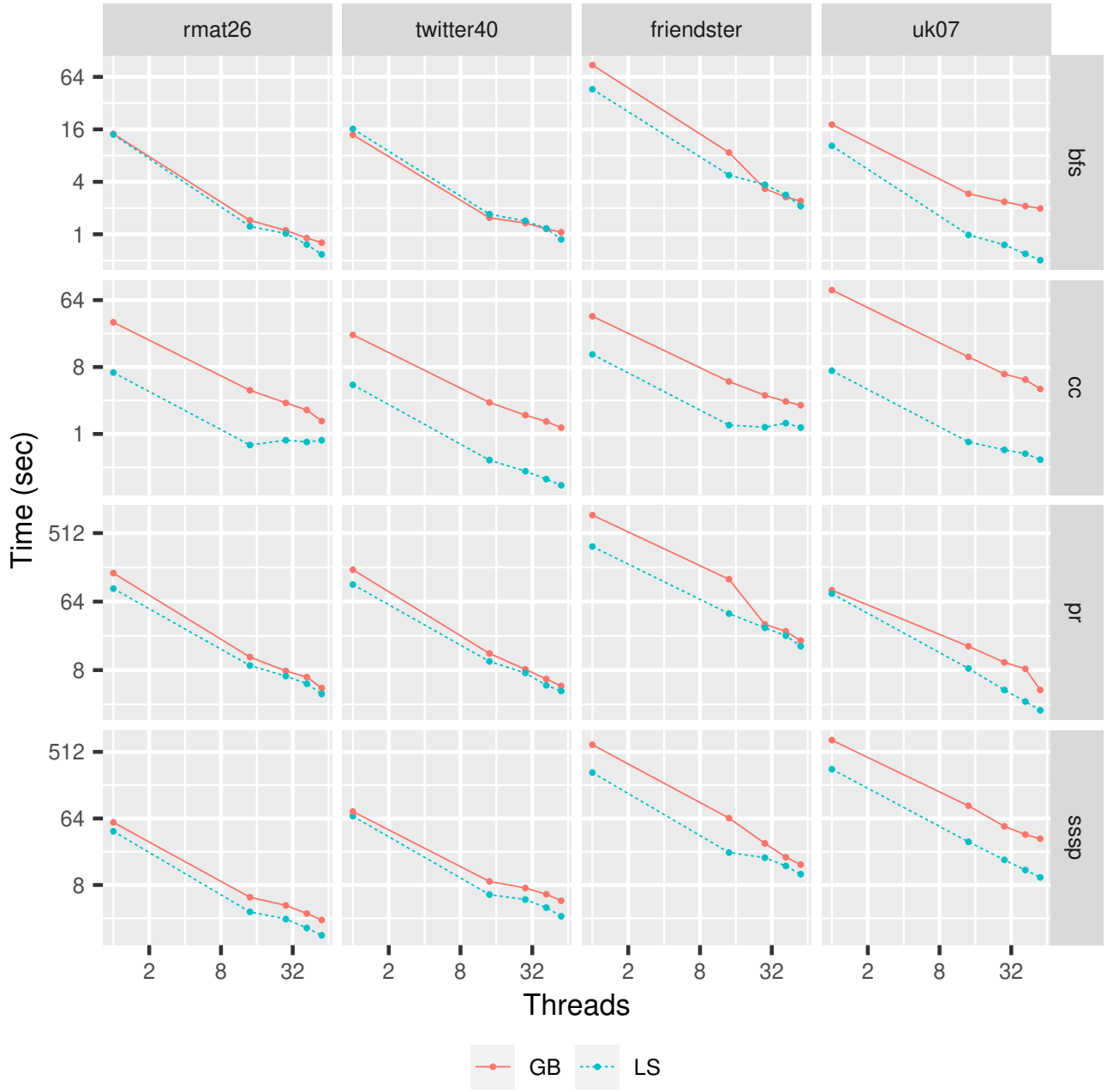
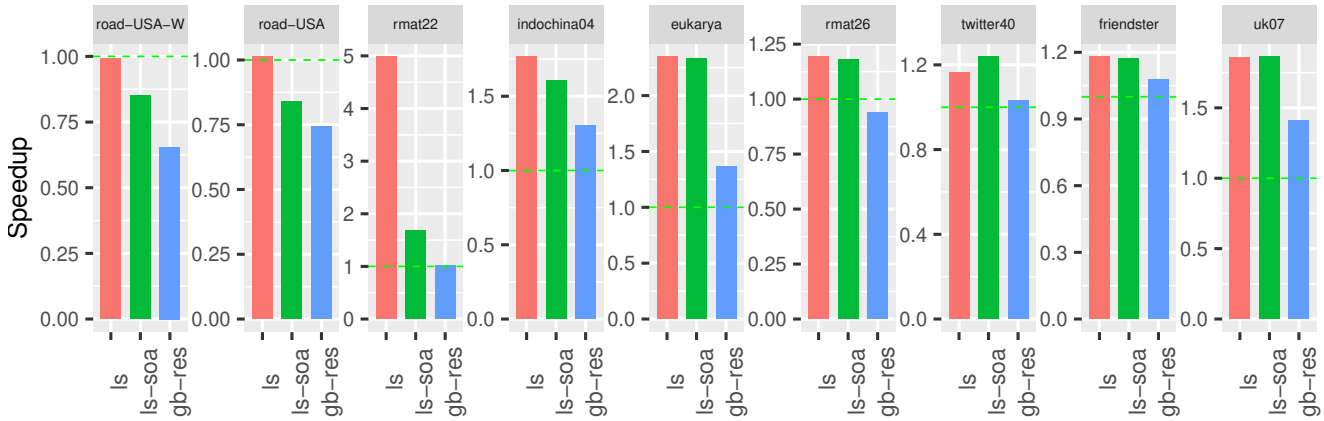


Fig. 2: Strong scaling of GaloisBLAS (GB) and Lonestar (LS) for four applications on the four largest graphs (log-log scale).

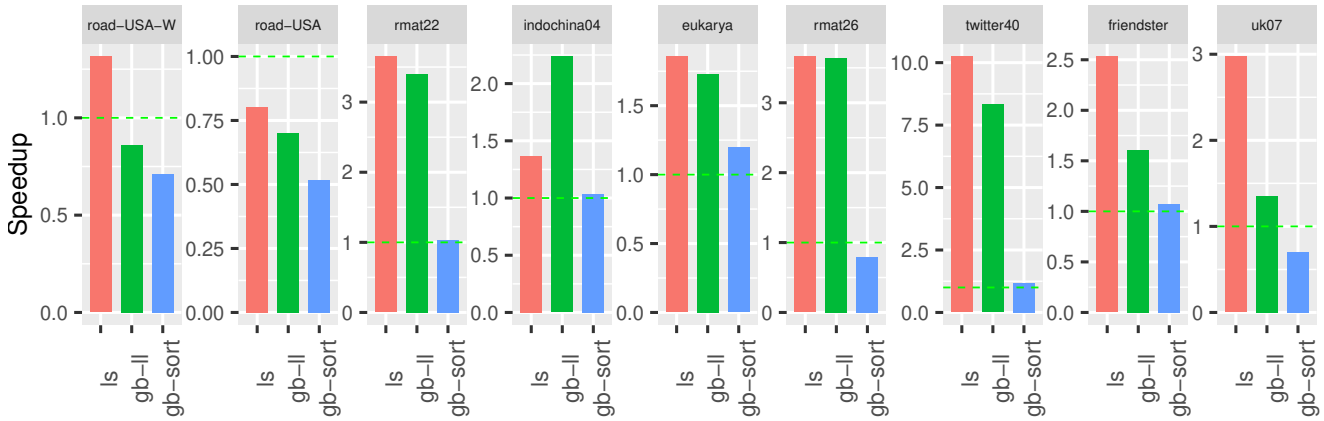
1) SuiteSparse vs. GaloisBLAS: First, we study SuiteSparse and GaloisBLAS to determine if GaloisBLAS is a reasonable implementation of the GraphBLAS API. Table II shows that for most applications and inputs, GaloisBLAS matches or outperforms SuiteSparse: on average, GaloisBLAS is $1.4\times$ faster than SuiteSparse for these experiments. For the remainder of our discussion, we use GaloisBLAS as the GraphBLAS implementation in our comparison with Lonestar, which uses a graph-based API.

2) GaloisBLAS vs. Lonestar: On average, Lonestar is $3.5\times$ faster than GaloisBLAS. Figure 2 illustrates the strong scaling of GaloisBLAS and Lonestar from 1 to 56 threads (the x and y axes are logarithmic) for the four largest graphs (we omit `tc` and `ktruss` because they take a long time to execute on fewer threads). Both systems scale similarly, but there is a performance gap between the systems at all thread counts. The main reasons why Lonestar performs better are closely related to the inherent limitations of a matrix-based API.

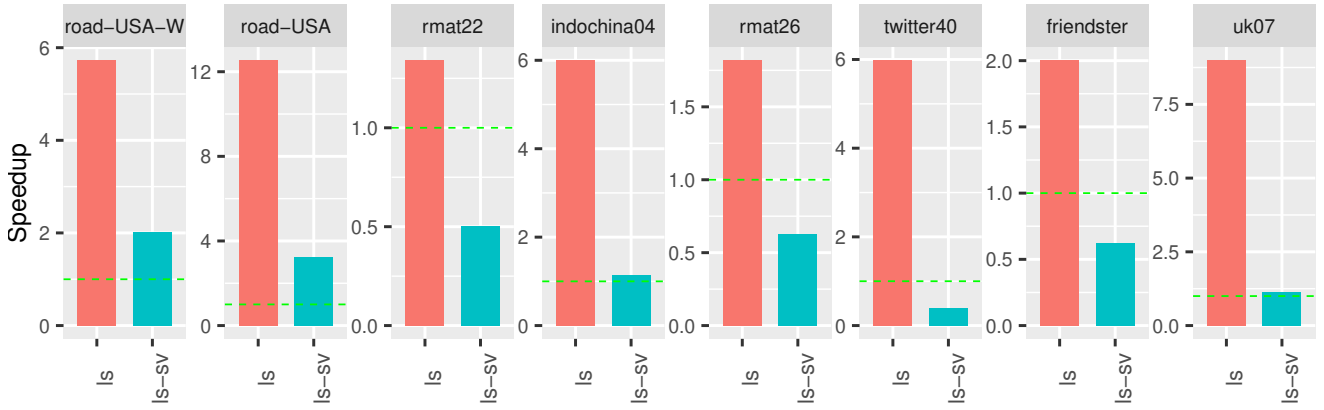
Loop fusion: Lonestar programs typically perform composite operations in a single fused (nested) loop, whereas LAGraph programs require multiple sparse-matrix function calls (each of which is a nested loop) to do the same computation. Loop fusion may lead to better locality as the same fields might be accessed in multiple loops otherwise. Algorithms 1 and 2



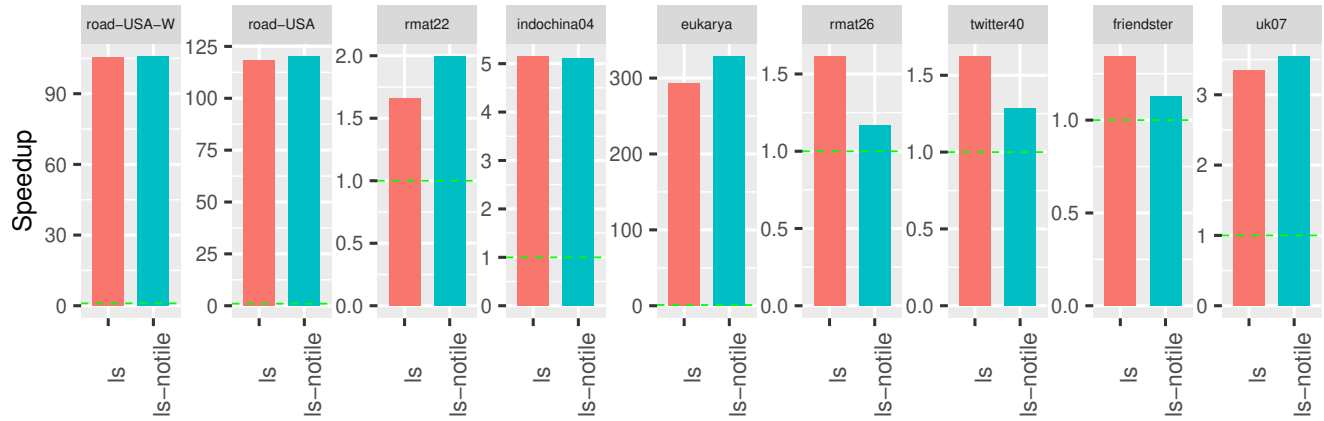
(a) pr variants.



(b) tc variants.



(c) cc variants.



(d) sssp variants.

Fig. 3: Speedups of variants for cc, sssp, pr, and tc. The green line is the baseline "gb" variant for the problem used in Table II.

illustrate this for bfs. As shown in Table IV, GaloisBLAS makes significantly more DRAM accesses than Lonestar for bfs because it must make several passes over the graph to apply the necessary operators. Due to this, Lonestar is 5× faster than LAGraph for bfs on road-USA.

Avoiding materialization: LAGraph programs may need to materialize temporary (sparse) vectors or matrices between sparse-matrix function calls to pass the intermediate values whereas Lonestar programs may instead use scalars for the intermediate values in the fused (nested) loop. Consider tc in Table IV. In GaloisBLAS, tc must be materialized in an output matrix, and the total count is computed by iterating over this matrix: this increases instructions and memory accesses. In Lonestar, no materialization is required: a global counter is incremented for every triangle found during the computation. Lonestar is 3× faster than GaloisBLAS for triangle counting on the uk07 web-crawl because it can avoid the overheads associated with additional materialization.

Vertex operations: Matrix-based programs are constrained to bulk operations, whereas graph-based programs can perform fine-grained vertex operations. For cc, Lonestar implements the Afforest [14] algorithm, which randomly samples vertices and executes operations only on those vertices. Such fine-grained operations on vertices cannot be expressed in a matrix-based API. Due to this, Lonestar executes an order of magnitude fewer instructions and memory accesses than GaloisBLAS as shown in Table IV. Lonestar is on average 3× faster than GaloisBLAS for cc.

Asynchronous execution: The notion of rounds is intrinsic to matrix-based APIs, so asynchronous algorithms cannot be expressed. This can hurt performance drastically when a large number of rounds are executed. For sssp on uk07, Lonestar is 3× faster than GaloisBLAS primarily due to asynchronous execution. Asynchronous execution may execute less redundant work and lead to faster convergence. The hardware metrics for sssp shown in Table IV validate this.

3) Memory Analysis: Table III shows the maximum resident set size (MRSS) at the end of computation for each system in this study. MRSS is the most memory that the program used during execution (including graph loading).

There are a few noteworthy points from these numbers. First, observe that GaloisBLAS and Lonestar have roughly the same MRSS for most graphs and applications since they share the same runtime. For the smaller graphs, the MRSS for GaloisBLAS and Lonestar is higher than that of SuiteSparse. This is because the Galois runtime preallocates memory to avoid dynamic memory allocation during execution, and the preallocation is more than the memory actually needed by the program. As the graphs grow in size, SuiteSparse's MRSS begins to be significantly higher than GaloisBLAS and Lonestar: the preallocation done by the Galois runtime becomes insignificant compared to the size taken by the graph and the data structures used during computation. This suggests that SuiteSparse may be able to improve performance by optimizing memory usage. In some cases, Lonestar has slightly higher MRSS than GaloisBLAS: Lonestar maintains additional data structures such as worklists for algorithms like sssp which increases memory usage for better performance. Lastly, observe that tc for SuiteSparse and GaloisBLAS have noticeably higher MRSS than Lonestar does for the larger graphs (friendster, uk). tc in SuiteSparse and GaloisBLAS must initialize additional matrices derived from the original graph such as upper/lower triangular matrices in order to do triangle counting which increases the RSS, and this increased memory footprint may decrease memory locality during execution.

B. Differential Analysis

To understand the performance gaps between GaloisBLAS and Lonestar, we performed differential analyses. We constrain the Lonestar programs in various ways and implement better algorithms in GraphBLAS where possible to determine the impact of features like asynchronous execution and loop fusion. This provides insight into performance differences between the approaches. We present these results in Figure 3.

pr: We examine four pr variants: (1) (pr-)ls, the residual-based Lonestar pr implementation used in Table II, (2) (pr-)ls-soa, which is ls except with vertex data in a structure of arrays instead of array of structures, (3) (pr-)gb-res, a new GraphBLAS implementation of pr that matches the residual-based computation done by Lonestar, and (4) (pr-)gb, the topology-driven LAGraph pr implementation used in Table II. The results are shown in Figure 3(a).

First, gb-res outperforms gb because gb-res performs fewer data accesses and has better locality: gb uses edge data to store the pagerank contributions (or delta) of node, whereas gb-res uses a separate vector for them.

Next, we compare gb-res and ls-soa. In a residual-based pr, two operations must be performed in each iteration using the residual values: the pagerank is updated with the residual, and the outdegree is multiplied with the residual. In a matrix API, these operations must be done using two calls to the API, so the residual vector is iterated over twice, increasing memory accesses. In the graph API, these operations can be fused into a single iteration over the residual values: this reduces memory accesses and increases efficiency. This is shown in the hardware metrics collected for ls-soa and gb-res in Table V: gb-res has double the instruction count of ls-soa and significantly more L1 accesses.

Finally, we compare ls-soa and ls. In ls, the pagerank and outdegree fields are accessed in the loop over residual values, so they are packed into a structure rather than being stored in separate arrays like in ls-soa. This improves data locality, which improves performance. It is difficult to express operations on such structures in a matrix-based API.

tc: We examine four tc variants: (1) (tc-)ls, the Lonestar tc implementation used in Table II based on triangle listing [39] which sorts vertices by degrees, (2) (tc-)gb-ll, a new GraphBLAS implementation of the triangle listing algorithm, (3) (tc-)gb-sort, the LAGraph algorithm used in Table II using the sorted graph produced by (1) as input, and (4) (tc-)gb, the LAGraph algorithm used in Table II that uses an unsorted graph. The results are shown in Figure 3(b).

TABLE III: Maximum resident set size (in GB) at the end of computation. TO signifies timeout (2hr), C signifies correctness bug, OOM signifies out of memory. SS: LAGraph on SuiteSparse, GB: LAGraph on GaloisBLAS, LS: Lonestar on Galois.

		road-USA-W	road-USA	rmat22	indochina04	eukarya	rmat26	twitter40	friendster	uk07
bfs	**SS**	0.36	1.37	1.20	3.37	6.10	19.04	25.28	61.58	63.93
	GB	3.79	4.24	4.16	5.12	6.34	12.14	14.85	30.60	32.12
	LS	3.79	4.24	4.16	5.11	6.34	12.13	14.88	31.04	32.10
cc	**SS**	0.68	2.48	2.50	6.40	C	38.27	47.00	92.34	125.28
	GB	3.79	4.24	4.16	5.11	C	12.14	14.88	31.04	32.12
	LS	3.73	4.02	4.16	4.80	6.33	12.07	12.89	31.03	28.92
ktruss	**SS**	0.80	2.96	8.71	19.49	16.19	TO	TO	TO	OOM
	GB	3.79	4.24	4.73	5.57	6.34	TO	TO	TO	OOM
	LS	3.73	4.03	4.16	4.81	6.33	12.07	TO	31.03	TO
pr	**SS**	0.54	2.04	1.70	4.76	7.47	27.05	36.27	88.89	91.86
	GB	3.79	4.24	4.16	5.12	6.33	12.13	14.89	31.03	32.11
	LS	3.79	4.24	3.77	5.11	6.33	12.10	14.88	30.57	31.82
sssp	**SS**	0.62	2.33	1.90	5.26	10.86	30.36	39.74	96.51	100.63
	GB	3.79	4.24	4.16	5.12	6.34	12.09	14.86	30.67	31.83
	LS	3.79	4.24	4.16	5.11	6.34	12.13	14.88	30.95	32.12
tc	**SS**	0.63	2.41	3.71	8.89	10.48	57.89	64.72	122.13	OOM
	GB	3.79	4.23	4.16	5.11	6.33	12.13	14.89	31.04	52.73
	LS	3.73	4.02	4.15	4.81	4.99	12.08	12.90	17.58	29.01

TABLE IV: Average ratios (GaloisBLAS over Lonestar) of hardware counters collected during algorithm execution for all inputs. Greater than 1 means that GaloisBLAS had a higher count for that metric.

	Instruction Count	L1 Access	L2 Access	L3 Access	DRAM Access
bfs	4.26	3.32	1.52	1.48	1.41
cc	27.03	28.85	9.99	7.43	9.03
pr	0.98	1.56	1.07	1.16	1.27
sssp	4.91	2.98	5.03	4.09	3.87
tc*	2.22	2.41	2.89	2.26	2.30

* tc ratios do not include uk07 (CAPE ran out of memory for it).

TABLE V: Average ratios (one algorithm variant over another) of hardware counters collected for all inputs.

	Instruction Count	L1 Access	L2 Access	L3 Access	RAM Access
pr-gb-res / pr-ls-soa	1.81	2.56	1.04	1.03	0.97
cc-gb / cc-ls-sv	4.12	4.56	3.18	2.91	3.66
tc-gb-ll / tc-ls	0.95	1.21	1.32	1.15	1.14

First, note that using the sorted graph in `gb-sort` does not necessarily improve performance: in fact, it degrades it in some cases. The `gb` algorithm does not leverage the sorted-by-degree nature of the input, so there is no purpose in using a sorted graph. To close this gap, we implemented the triangle listing algorithm in LAGraph/GaloisBLAS: `gb-ll`. This algorithm leverages the sorted graph to avoid iterating over high degree vertices: as a result, it gains significant improvement over `gb-sort`.

Next, we compare `gb-ll` with `ls`. Even though both use the same triangle listing algorithm that leverages the sorted graph, `ls` outperforms `gb-ll`. `ls` does not need to materialize any additional matrices to do counting and can fuse the counting step into the finding step. The increased memory accesses are reflected in the hardware counter ratio in Table V: `gb-ll` performs more memory accesses. `gb-ll` has less instructions: this is because preprocessing (which is excluded from the time reported) in `gb-ll` has removed the need for runtime symmetry breaking (triangle uvw is only counted if $u > v > w$): `ls` checks this condition during runtime. Even though `gb-ll` executes fewer instructions, it is slower because it makes more memory accesses.

ktruss: `ktruss` uses triangle counting as a subroutine but unlike `tc`, Lonestar (ls) does not use sorted graphs and LAGraph (gb) is not doing preprocessing to avoid symmetry breaking. Nevertheless, like in `tc`, `gb` materializes intermediate matrices while `ls` does not. Both `ls` and `gb` use a round-based algorithm in which edges are incrementally removed from the graph. However, in `ls`, the edge removals are *immediately visible* to all other threads (Gauss-Seidel iteration) unlike `gb` in which edge removals only occur after the end of a round (Jacobi iteration) due to the bulk nature of matrix operations. Due to this, `gb` executes $1.6\times$ more computational rounds than `ls`. More execution rounds results in increased memory accesses, which results in significantly more runtime: note that `gb` did not finish in 2 hours for some inputs (Table II).

cc: Lonestar uses the Afforest algorithm [14] in Table II; we call this `(cc-)ls`. This algorithm cannot be expressed using the GraphBLAS API because it uses sampling and fine-grained vertex operations, so LAGraph implements a *restricted* version of the Shiloach-Vishkin pointer-jumping algorithm [37]. In pointer-jumping algorithms, vertices have parent pointers that point to other vertices in the same connected component; one vertex in each connected component is called the *root*, and its parent pointer points to itself. At the end of the `cc` computation, the parent pointer of each vertex should point to the root of its connected component. When a matrix-based API is used, the parent pointers must be updated by a bulk operation that performs some *fixed* number of pointer-jumping steps everywhere. We call this LAGraph implementation used in Table II `(cc)-gb`. In contrast, the graph-based API permits

unbounded pointer-jumping in which the parent pointer of a vertex can end up getting short-circuited to any of its ancestors in the chain of parent pointers, independently of pointer-jumping at other vertices. We call this Shiloach-Vishkin algorithm [15] in Lonestar `(cc-)ls-sv`.

Figure 3(c) shows that `ls` significantly outperforms `ls-sv`, as expected [14]. Note that on high-diameter graphs like road networks, `ls-sv` performs much better than `gb`: the asynchronous execution model in Galois permits information to propagate faster through the graph via pointer-jumping than a matrix-based API permits. Due to this, `ls-sv` executes fewer instructions and memory accesses as shown in Table V.

sssp: Both Lonestar (ls) and LAGraph's (gb) `sssp` in Table II use delta-stepping. However, Lonestar exploits the asynchrony in the Galois runtime to implement an asynchronous version of delta-stepping, whereas LAGraph implements a bulk-synchronous version of delta-stepping. `ls` also partitions the edges of high-degree nodes among several threads; this is called *edge-tiling*, and it promotes load balancing for power-law graphs. To understand the impact of edge-tiling, we compare with `ls` without edge-tiling, `(sssp-)ls-notile`. Figure 3(d) shows the performance impact of this optimization. We see that for power-law graphs like rmat26 and twitter40, this optimization boosts performance by $1.5\times$ over `gb`. `ls-notile` is still faster than `gb` due to asynchronous execution even for low-diameter graphs. Asynchrony is especially important for high-diameter graphs: both the tiled and no-tiled variants are over $100\times$ faster than `gb` for road-USA as bulk-synchronous algorithms must execute significantly more rounds of computation for high-diameter graphs.

bfs: Both Lonestar (ls) and LAGraph (gb) implementations of `bfs` in Table II use the same level-by-level `bfs` algorithm which propagates the distance value one edge at a time in each round of execution. The runtime difference between the two implementations arises from an extra matrix operation that `gb` does: `gb` uses a vector-matrix multiply to mark newly visited vertices in a computation round. It does an additional vector assign operation after the multiply in order to update the visited vertices' distance values. In other words, it (1) creates a worklist for the next round and (2) updates the distances using two disjoint operations that both iterate over the vertex data structure. In `ls`, the worklist creation and the distance update are in the same loop; this reduces overall accesses to memory as the vertex data structure is only iterated over once. Table IV shows that `gb`'s instruction count and memory accesses are much higher than those of `ls`.

VI. Related Work

Using matrix operations to do graph algorithms uses the idea that the topology of the graph can be represented in a matrix [22]. Many graph algorithms have been defined in terms of linear algebraic operations [22], [38], [40]–[42]. Graph analytics systems that use matrix-based operations [43] include the Combinatorial BLAS [44], PEGASUS [45], GBASE [46], and SystemML [47]. Although these systems used matrix-based operations, they are not necessarily GraphBLAS-compliant.

GraphBLAS is an API defined to make writing graph algorithms using a matrix-based API standard [10], [11] and aims to provide a set of core functions to define all types of graph algorithms. There are implementations of the GraphBLAS API such as SuiteSparse:GraphBLAS [12] and GraphBLAST [48]. SuiteSparse:GraphBLAS is a full implementation of the API on top of OpenMP. The GraphBLAS community has also developed libraries for heterogeneous systems. GraphBLAST is an implementation for a single GPU. It optimizes memory accesses by exploiting direction optimization [49], [50] and contains optimizations to avoid unnecessary computations. However, it only implements a subset of the GraphBLAS API and does not support all the applications evaluated in this paper. There are no implementations of the GraphBLAS API on multiple GPUs or distributed CPU clusters yet. Our work introduces an implementation of the GraphBLAS API called GaloisBLAS built on the Galois [6] system for shared-memory CPUs. LAGraph is a collection of algorithms defined with the GraphBLAS API [13] that we use in our experiments.

Graph-centric graph analytics systems treat the graph as an ADT that can be used to query vertices and edges, update graph state, add/remove elements of the graph, etc. Examples of such systems include Galois [6], [17], Gemini [51], Ligra [5], and Gunrock [50] among many other systems that span both GPUs [16], [50], [52] and distributed clusters [17], [51], [53], [54]. In particular, Galois is the system that we use to implement GaloisBLAS: we use the parallel constructs and data structures it provides to create a parallel implementation of a subset of the GraphBLAS API.

To the best of our knowledge, this study is the first detailed analysis of performance differences between matrix- and graph-centric approaches for graph analytics workloads.

VII. Conclusion

This paper studied the performance of a matrix-based approach for writing graph algorithms (LAGraph running on top of SuiteSparse or GaloisBLAS) relative to a graph-based approach (Lonestar programs running on the Galois system). Our results show that Lonestar/Galois programs are $5\times$ faster than LAGraph/SuiteSparse programs. We argued that the main reasons for these differences were that (i) programs using a matrix-based API need to iterate over a graph multiple times to apply a composite operator that is not supported by the API unlike in a graph-based API where such updates can be combined in a single loop, (ii) programs written using a matrix-based API will materialize a large amount of intermediate data unlike graph-based approaches, (iii) the matrix approach is constrained to bulk operations whereas the graph-based approach can perform fine-grained operations on vertices, and (iv) the matrix approach is constrained to round-based execution whereas the graph-based approach can enjoy the benefits of asynchronous execution. Although our study was conducted on CPUs, the findings extend to all architectures as these limitations are intrinsic to the matrix-based approach to graph computation: these restrictions apply at the *algorithmic* level and not the architectural level. While these

issues are intrinsic to matrix-based approaches irrespective of the architecture, issues (i) and (ii) may be solved using restructuring compiler technology in the future.

ACKNOWLEDGMENT

This research was supported by the NSF grants 1406355, 1618425, 1705092, 1725322, and by the DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563.

REFERENCES

[1] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, 1977.
[2] V. Krebs, "Mapping networks of terrorist cells," *Connections*, 2002.
[3] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Commun. ACM*, vol. 47, 2004.
[4] B. Cui, X. Shao, and Z. Zhang, "Assessment of flow paths and confluences for saltwater intrusion in a deltaic river network," *Hydrological Processes*, 2015.
[5] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," ser. PPoPP, 2013.
[6] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," ser. SOSP, 2013.
[7] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *PVLDB*, vol. 13, no. 8, 2020.
[8] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA, 2018.
[9] "The lonestar benchmark suite," 2018. [Online]. Available: http://iss.oden.utexas.edu/?p=projects/galois/lonestar
[10] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the GraphBLAS," in *HPEC*, 2016.
[11] "Graph blas forum," 2019. [Online]. Available: http://graphblas.org/index.php?title=Graph_BLAS_Forum
[12] "SuiteSparse:GraphBLAS Webpage," 2019. [Online]. Available: http://faculty.cse.tamu.edu/davis/GraphBLAS.html
[13] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS," in *IPDPSW*, 2019.
[14] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling," in *IPDPS*, 2018.
[15] "An O(log n) parallel connectivity algorithm," *Journal of Algorithms*, 1982.
[16] S. Pai and K. Pingali, "A Compiler for Throughput Optimization of Graph Algorithms on GPUs," in *OOPSLA*, 2016.
[17] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics," in *PLDI*, 2018.
[18] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Eds., *Introduction to Algorithms*. MIT Press, 2001.
[19] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, 2003.
[20] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A High-performance Graph DSL," *OOPSLA*, 2018.
[21] D. Nguyen and K. Pingali, "Synthesizing concurrent schedulers for irregular algorithms," in *ASPLOS*, 2011.
[22] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. Society for Industrial and Applied Mathematics, 2011. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719918
[23] T. A. Davis, "User Guide for SuiteSparse: GraphBLAS."
[24] D. J. Higham and N. J. Higham, *MATLAB Guide*. Siam, 2016, vol. 150.
[25] T. Davis, "Algorithm 9xx: Suitesparse: Graphblas: graph algorithms in the language of sparse linear algebra," *Submitted to ACM TOMS*, 2018.
[26] J. J. Elliott and C. M. Siefert, "Low Thread-count Gustavson: A multithreaded algorithm for sparse matrix-matrix multiplication using perfect hashing," in *ScalA*, 2018.
[27] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *TOMS*, 1978.
[28] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors," *Parallel Computing*, 2019.
[29] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime compilation techniques for data partitioning and communication schedule reuse," in *SC*, 1993.
[30] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, 2004.
[31] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *WWW*, 2004.
[32] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks," in *WWW*, 2011.
[33] ""9th DIMACS Implementation Challenge"," 2019. [Online]. Available: http://users.diag.uniroma1.it/challenge9/
[34] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *WWW*, 2010.
[35] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.
[36] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Bulu, "HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks," *Nucleic Acids Research*, 2018.
[37] Y. Zhang, A. Azad, and Z. Hu, "FastSV: a distributed-memory connected component algorithm with fast convergence," in *Parallel Processing for Scientific Computing*, 2020.
[38] U. Sridhar, M. Blanco, R. Mayuranath, D. G. Spampinato, T. M. Low, and S. McMillan, "Delta-Stepping SSSP: From Vertices and Edges to GraphBLAS Implementations," in *IPDPSW*, 2019.
[39] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *KDD*, 2011.
[40] A. Azad and A. Buluc, "LACC: A Linear-Algebraic Algorithm for Finding Connected Components in Distributed Memory," in *IPDPS*, 2019.
[41] T. A. Davis, "Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and K-truss," in *HPEC*, 2018.
[42] A. Azad, A. Bulu, and J. Gilbert, "Parallel Triangle Counting and Enumeration Using Matrix Algebra," in *IPDPSW*, 2015.
[43] D. Yan, Y. Tian, and J. Cheng, "Matrix-based graph systems," *Systems for Big Graph Analytics*, 2017.
[44] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *Journal of High Performance Computing Applications*, 2011.
[45] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *ICDM*, 2009.
[46] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "GBASE: an efficient analysis platform for large graphs," *VLDB Journal*, vol. 21, no. 5, 2012.
[47] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda, "SystemML: Declarative Machine Learning on Spark," vol. 9, no. 13, 2016.
[48] C. Yang, A. Buluc, and J. D. Owens, "GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU," in *arXiv*, 2019.
[49] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing Breadth-first Search," in *SC*, 2012.
[50] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-gpu graph analytics," in *IPDPS*, 2017.
[51] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A Computation-centric Distributed Graph Processing System," in *OSDI*, 2016.
[52] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations," in *PPoPP*, 2017.
[53] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A Distributed multi-GPU System for Fast Graph Processing," *PVLDB*, vol. 11, no. 3, 2017.
[54] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-parallel Computation on Natural Graphs," in *OSDI*, 2012.