LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS

Tim Mattson[‡], Timothy A. Davis[◇], Manoj Kumar[¶], Aydın Buluç[†], Scott McMillan[§], José Moreira[¶], Carl Yang[*,†]

[‡]*Intel Corporation* [†]*Computational Research Division, Lawrence Berkeley National Laboratory*
[◇]*Texas A&M University* [¶]*IBM Corporation* [§]*Software Engineering Institute, Carnegie Mellon University*
[*]*Electrical and Computer Engineering Department, University of California, Davis*

***Abstract*—In 2013, we released a position paper to launch a community effort to define a common set of building blocks for constructing graph algorithms in the language of linear algebra. This led to the GraphBLAS. We released a specification for the C programming language binding to the GraphBLAS in 2017. Since that release, multiple libraries that conform to the GraphBLAS C specification have been produced.**

In this position paper, we launch the next phase of this ongoing community effort: a project to assemble a set of high level graph algorithms built on top of the GraphBLAS. While many of these algorithms are well-known with high quality implementations available, they have not been assembled in one place and integrated with the GraphBLAS. We call this project the *LAGraph graph algorithms project* and with this position paper, we put out a call for collaborators to join us. While the initial goal is to just assemble these algorithms into a single framework, the long term goal is a library of production-worthy code, with the LAGraph library serving as an open source repository of verified graph algorithms that use the GraphBLAS.

***Index Terms*—Graph Algorithms, Linear Algebra, GraphBLAS**

I. Introduction

Graphs are an essential abstraction for a wide range of problems. There are many ways to represent graphs in graph analytics. One class of methods defines graphs in terms of adjacency matrices. Expressing graph algorithms as linear algebra expressions [1] is a mature subject. Multiple high performance graph libraries based on sparse linear algebra [2], [3], [4], [5], [6] have been developed. With this "Graphs as Linear Algebra" approach established, a community of researchers came together to define common building blocks for graphs expressed in the language of linear algebra. They launched this effort with a position paper [7] in 2013 and formed the GraphBLAS Forum [8]. After three years of steady work, the GraphBLAS forum completed the mathematical formalization of the GraphBLAS [9]. With another one and a half years of work, the group completed the C language binding to the GraphBLAS [10].

Currently there are multiple implementations of the GraphBLAS C specification. We have learned a great deal about how to define the operations within the GraphBLAS and how to implement them efficiently. We are now ready to launch the next phase of the project: to produce a library of high-level graph algorithms implemented on top of the GraphBLAS. We call this library of algorithms *LAGraph*. Just as we launched the GraphBLAS project with a position paper, we are launching this next phase of the project with a position paper.

The goals of the LAGraph effort include, first and foremost, bringing together the full range of known graph algorithms that can be constructed with the GraphBLAS. From this collection we will be able to systematically assess the coverage of graph algorithms based on linear algebra. This will also serve as raw material in ongoing studies of the fundamental design patterns exploited by linear algebra-based graph algorithms.

The basic outline of the LAGraph project is summarized in Figure 1. The library of algorithms is the single box towards the middle of the figure. These algorithms use the GraphBLAS C API which exists as a separate implementation for a wide range of hardware targets at the bottom of the figure. To motivate our work and stay aligned with the way data scientists use graph algorithms, we need to appreciate that people will use a wide range of languages to access the graph algorithms. A good library must be able to work whether called directly from C or indirectly through a wrapper in Python. Furthermore, the developers of the graph algorithm libraries will need a test harness to validate the algorithms, I/O utilities, and a build system. All of these components must be addressed as part of the LAGraph project.

LAGraph may not be a production-worthy library in its first incarnation. However, we expect the LAGraph effort to produce such a library over time. Anticipating that goal, we are constructing the library "as if" it will be used by data analytics end-users; that is, by people who need the results from graph algorithms with little concern for how they are implemented. This requirement of building a library for *end-users* as opposed to a library for *graph algorithm researchers* has far reaching implications for the design of this software.

We start the paper with a brief summary of the objects and operations defined in the C specification of the GraphBLAS. We then summarize some of the early libraries that implement the GraphBLAS C specification. Next, we describe the repository where we will build LAGraph. This is important since the purpose of a position paper is to attract a community of researchers to join the effort, which in turn means we want people to understand how to work with and perhaps contribute algorithms to the repository. We then discuss the challenges

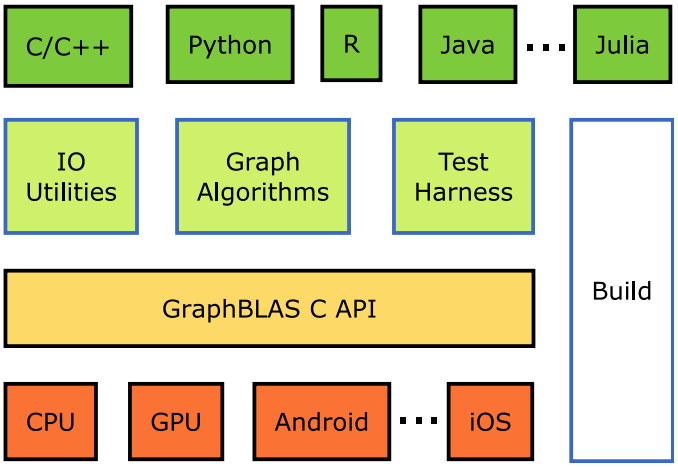

Fig. 1: **LAGraph Project Overview.** The project consists of a library of graph algorithms and assorted components to support algorithm development and validation (a test harness and I/O utilities). It must interface to a wide range of languages. Underneath these components is a build system, implementations of the GraphBLAS, and a variety of hardware targets.

we faced in writing the early version of LAGraph and what it suggests about future developments needed in the GraphBLAS themselves. We close with concluding remarks.

II. The GraphBLAS

Consider a graph represented as an n-by-n adjacency matrix $\mathbf{A}$, where A_{ij} is the weight of the edge from vertex i to vertex j, and a second k-by-n matrix $\mathbf{B}$ representing a subset (of size k) of the vertices in the graph, such that B_{ji} is 1 if the jth element of the subset is vertex i (and all other elements of $\mathbf{B}$ are 0). The traditional matrix product $\mathbf{B} \times \mathbf{A}$ over real arithmetic of these two matrices returns the cost based on the edge weights of reaching the set of vertices adjacent to the vertices in $\mathbf{B}$. This fundamental operation can be used to construct a wide range of graph algorithms.

We extend the range of graph operations by keeping the basic pattern of a matrix-matrix multiplication, but varying the operators and the interpretation of the values in the matrices (the *domain*). By carefully choosing operators and the domain, we control the relation between matrix operations familiar in linear algebra and graph operations, thereby enabling composable graph algorithms.

In addition to matrix multiplication, the GraphBLAS math specification defines a range of additional operations over matrices and vectors. These are summarized in Table I.

In mapping the GraphBLAS as a set of mathematical operators onto the C programming language we made a number of fundamental choices [10]. First, the core data structures required to represent the objects defined by the GraphBLAS are opaque. The GraphBLAS API defines a contract with the programmer for how these objects will be used, but the implementations and underlying data structures are left to the implementation. This opaqueness is critical if the API is to serve

TABLE I: A mathematical overview of the fundamental GraphBLAS operations supported in the specification. $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are GraphBLAS matrices; $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are GraphBLAS vectors; i and j are single indices; $\mathbf{i}$ and $\mathbf{j}$ are arrays of indices; $\oplus$ and $\otimes$ are arbitrary element-wise operators; the element-wise $\odot$ operator is used for the optional accumulation with the output GraphBLAS object where $x \odot\!= y$ implies $x = x \odot y$; and $F_u()$ is a unary function. Although not shown here, the input matrices $\mathbf{A}$ and $\mathbf{B}$ may be selected for transposition prior to the operation, and masks can be used to control which values are written to the output GraphBLAS object.

Operation name	Mathematical description
mxm	$\mathbf{C} \odot\!= \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w} \odot\!= \mathbf{A} \oplus . \otimes \mathbf{v}$
vxm	$\mathbf{w}^T \odot\!= \mathbf{v}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C} \odot\!= \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w} \odot\!= \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C} \odot\!= \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w} \odot\!= \mathbf{u} \oplus \mathbf{v}$
reduce (row)	$\mathbf{w} \odot\!= \bigoplus_j \mathbf{A}(:, j)$
apply	$\mathbf{C} \odot\!= F_u(\mathbf{A})$
	$\mathbf{w} \odot\!= F_u(\mathbf{u})$
transpose	$\mathbf{C} \odot\!= \mathbf{A}^T$
extract	$\mathbf{C} \odot\!= \mathbf{A}(\mathbf{i}, \mathbf{j})$
	$\mathbf{w} \odot\!= \mathbf{u}(\mathbf{i})$
assign	$\mathbf{C}(\mathbf{i}, \mathbf{j}) \odot\!= \mathbf{A}$
	$\mathbf{w}(\mathbf{i}) \odot\!= \mathbf{u}$

diverse hardware ranging from CPUs to GPUs to specialized graph hardware. Second, we defined a non-blocking execution model that allows lazy evaluation. Ultimately, to optimize sparse linear algebra software we need to aggressively fuse operations and even restructure algorithms. This requirement meant that we had to carefully define when results from a sequence of GraphBLAS operations must be materialized.

Since the release of the GraphBLAS specification, several implementations of the GraphBLAS have been developed. These are briefly described below.

A. *SuiteSparse:GraphBLAS*

SuiteSparse:GraphBLAS is the first full implementation of the GraphBLAS standard, first released in November of 2017. It is available at http://suitesparse.com [11].

The design of a GraphBLAS library is flexible, because its data structures are opaque to the user. SuiteSparse:GraphBLAS uses a compressed-sparse vector data structure, in four different forms. A matrix can be stored in row-major order (CSR), or column-major order (CSC). Each sparse vector consists of a sorted list of indices, and the corresponding numerical values. The sparse vectors are packed together into two arrays, and another "pointer array" (of size equal to the dimension of the matrix, say n) keeps track of where each row (or column) starts. The memory taken is $O(n + e)$ for a CSR matrix with n rows or a CSC matrix with n columns, and with e entries. Most graphs have $e = O(n)$ entries, but some graphs (and in particular, subgraphs) can be *hypersparse* [12], with $e \ll n$. In the hypersparse form, the pointer array itself becomes sparse, and empty vectors take no space at all. The space is reduced to $O(e)$, so that matrices with enormous dimensions can be

created, as long as $e \ll n$. SuiteSparse:GraphBLAS exploits hypersparsity automatically, and all methods can operate on all four matrix formats in any combination.

The ability to incrementally modify a graph is critical in many applications. GraphBLAS includes two operations that can make small incremental changes to a graph/matrix: namely `GrB_setElement` and `GrB_assign`. It would be exceedingly slow to insert or delete a single entry in a CSR or CSC format, taking $O(n + e)$ time **per entry** inserted or deleted. Instead, the non-blocking aspect of GraphBLAS is exploited. Fast deletion of entries is handled by creating *zombies*, which are entries tagged for later deletion. Fast insertion is handled with *pending tuples*, which is a separate unordered list of (i, j, a_{ij}) for each new entry. When a matrix operation occurs (such as matrix multiply), all zombies are killed and all pending tuples are assembled, in a single $O(n + e + p \log p)$ step (for p pending tuples), or $O(e + p \log p)$ in the hypersparse case. As a result, it is just as fast to use a sequence of e `GrB_Matrix_setElement` operations to build a matrix, as it is to create an array of e tuples and use `GrB_Matrix_build`. Internally, SuiteSparse:GraphBLAS is building the list itself, for the user, and then does a `GrB_Matrix_build` when the matrix is completed.

To enable high-performance matrix-matrix multiply, a code generation mechanism is used to build functions for each semiring that can be created with built-in operators. The functions can rely on Gustavson's method [13], a dot product method, and a heap-based method [14], all with masked variants. With this code generation mechanism, 6 functions containing 2 versions of Gustavson's method (no mask / with mask), three versions of the dot product (no mask / with mask / with complemented mask) and one version of the heap method, automatically expand into the 960 unique semirings supported by the built-in operators in GraphBLAS SuiteSparse:GraphBLAS adds a few extensions to the set of operators; Using the built-in types and operators from the GraphBLAS C API, 600 unique semirings can be constructed. All of them are as fast, or much faster, than `C=A*B` in MATLAB. Submatrix assignment (`C(I,J)=A`) can be 100× faster than in MATLAB, even when non-blocking mode is not exploited.

A current prototype of the package adds an early exit mechanism for the MIN, MAX, OR, and AND monoids, where a dot product can terminate as soon as a terminal value is found in the result (`true` for OR, for example). This will enable a fast direction-optimizing BFS [15] to be written in GraphBLAS. The "pull" is a dot product, and the "push" a saxpy-based operation (Gustavson's or the heap method).

Since its creation was commissioned as the GraphBLAS reference implementation, testing is a vital component to the package. In SuiteSparse:GraphBLAS, each GraphBLAS operation was written twice: once in high-performance algorithms in C, and again in a very simple and short MATLAB script, using dense matrices with the required type. The pattern in the MATLAB version is held as a separate Boolean matrix. For example, `GrB_assign` requires about 3,908 lines of C (not counting comments), but only 161 lines in MATLAB. Of those 161 lines, 33 are for error-checking that do not need to be considered when determining conformance to the spec. The MATLAB functions are not intended to be fast. Instead, they exactly mimic the GraphBLAS API Specification, line by line, so they can be visually inspected for conformance to the spec. For example, matrix multiply is written with a brute-force triply-nested `for` loop. Then, to test the package, each computation is done both in SuiteSparse:GraphBLAS (via a MATLAB interface) and in the MATLAB mimic. The tests pass only if the results are identical in both value and pattern (even with identical floating-point roundoff error, in most cases).

The package is extremely robust and production-ready. It is fully compliant with the GraphBLAS C API. Excluding SuiteSparse-specific extensions and beta releases, there have been only 3 bugs in the entire package since its first release, two of which would be triggered in only rare cases. All three bugs are fixed, and the current version has no known bugs in any part of the code.

The current release is single-threaded, but an OpenMP implementation is in progress. SuiteSparse:GraphBLAS appears in Debian and Ubuntu Linux distros, and has been released as part of the RedisGraph database module of the Redis database systems, by RedisLabs, Inc. [16].

B. IBM GraphBLAS

The IBM GraphBLAS implementation was announced at IPDPS 2018 and made available at `https://github.com/IBM/ibmgraphblas`, fulfilling the GraphBLAS C API requirement of two conforming implementations, and promoting that specification from *provisional* to *definitive*. The approach adopted by the development team was heavily influenced by their experience with IBM's Graph Programming Interface [4], [17].

Among the various objectives of the IBM implementation, we note the desire to experiment with an implementation that allows multiple data representations and a layered approach to algorithms, keeping a GraphBLAS API layer on top of a more fundamental *back-end* layer that performs the heavy computation.

Use and operation of the IBM GraphBLAS is straightforward. The application programmer has access to a C11-compliant include file (`GraphBLAS.h`) that defines the API according to the specification. This include file exposes nothing of the internals of the run-time. The run-time itself is written in C++ and packaged as a library (`libibmgraphblas.so`) with C language bindings. The choice of C++ as the implementation language has led to a simple and concise specification-compliant implementation.

One of the jobs of the `GraphBLAS.h` include file is to convert the polymorphic version of the API into the nonpolymorphic one. The nonpolymorphic methods are then directly supported by the library. Conversion of the polymorphic interface is accomplished through standard C preprocessor features, primarily in supporting number of arguments polymorphism,

in combination with the standard C11 language `_Generic` construct to support type polymorphism.

As previously mentioned, the IBM GraphBLAS library is implemented in C++. The API methods are declared to have a C interface, so that C user programs can bind to them as specified. Objects internal to the library are declared as C++ classes, with member methods doing the actual work. We want to emphasize that this is a C++ implementation of a C API, and not an API for GraphBLAS that exploits C++ features, as other efforts are pursuing [18].

The contents of a GraphBLAS vector object are implemented using standard C++ containers. An unordered set of indices (`uset<GrB_Index>`) is used to represent the structure of the vector, while an unordered map of indices to pointers (`umap<GrB_Index,void*>`) represents the elements. (Currently, `uset` and `umap` are just renames for the `std::unordered_set` and `std::unordered_map` standard containers of C++, but one could use more specialized implementations.)

Similarly, the contents of a matrix are represented both as a standard C++ container vector of rows and a standard C++ container vector of columns. Accessor methods enforce consistency of both representations.

In the current IBM GraphBLAS, all methods are fully blocking. That is, the methods return only after all computations are fully performed (or an error is detected). Since the GraphBLAS nonblocking mode allows for deferred execution but does not require it, this behavior complies with the specification.

We finish this brief overview of the IBM GraphBLAS with a discussion of error handling, as we believe it reflects an important aspect of the interoperability between a C API and a C++ implementation. The GraphBLAS C API defines two kinds of errors: API errors and execution errors. API errors reflect incorrect usage of the API (for example, passing parameters that are not valid or consistent). Execution errors indicate that something went wrong during the actual execution of a method, and can be caused either by programmer error or by environment issues (for example, not enough memory).

In the IBM GraphBLAS run-time, API errors are detected through explicit checks in the implementation of each method in the API layer (the *front-end*). Execution errors, on the other hand, occur inside the member methods of the various objects internal to the library (the *back-end*). Those methods, following standard C++ practice, use exceptions to signal an error. To transform those exceptions into proper GraphBLAS C API return codes, the body of each GraphBLAS API method is wrapped by a `try/catch` block, which then returns the GraphBLAS execution error code corresponding to the caught exception.

C. GBTL: GraphBLAS Template Library

The first version of the GraphBLAS Template Library (GBTL) was written in C++ when the GraphBLAS C API project was just beginning. It was used, in part, to study early ideas under discussion in the specification process and was released as a proof of concept prior to the finalization of the GraphBLAS API Specification [19], [20]. With the release of the GraphBLAS C API Specification [10] in 2017, GBTL was updated to conform to the mathematical behavior defined by the specification and released as version 2.0 [18]. Unlike the C API Specification, GBTL is written in C++ and makes judicious use of templates to make the generic aspects of the GraphBLAS specification easier to implement and more natural for the C++ programming language. When the GraphBLAS language committee starts its work on the C++ language binding to the GraphBLAS, GBTL will be submitted as a proposed starting point for the discussion.

Central to GBTL's design is the concept of a separation of concerns between implementation of algorithms written in terms of the GraphBLAS primitives and the implementation of those primitives on a targeted hardware platform. This separation of concerns is defined by the GraphBLAS API Specification as illustrated in Figure 1. Above this API, GBTL has developed and includes a collection of graph algorithms written against its C++ API and has already been shown to be easily translated to the C API (compare Figures 2(c) and 2(d)). Below the separation/API, different implementations of the GraphBLAS library can be supported for different hardware architectures (referred to as "backends"). In this way we verify that algorithms written once against the API can run on different targeted hardware. One backend that is provided with Version 2.0 of GBTL implements a mathematically correct version of the C API specification and serves as a reference implementation for verifying correctness. It runs in a single thread on a CPU (an earlier version of GBTL also had a GPU implementation). Other backend implementations are currently under development to optimize performance, use multiple threads, and to target specialized computer architectures.

D. PyGB: python DSL for GraphBLAS

Another development effort closely related to GBTL is a DSL (domain-specific language) in Python called PyGB [21]. The goal for PyGB is to closely resemble the GraphBLAS mathematical notation found in the GraphBLAS math spec [9]. PyGB is a framework designed and implemented to dispatch dynamically generated and compiled templated classes that make calls into native GBTL code. It demonstrates how Python's syntax and dynamic execution provides a high-level abstraction with minimal performance penalty. While we leave a detailed discussion of PyGB to elsewhere [21] we provide a level-BFS function using PyGB in Figure 2.

Notice how the meaning of the code is straightforward since the DSL closely tracks the notation from the GraphBLAS math spec. We believe in the long run, the future of graph algorithms will depend heavily on such DSLs.

E. GraphBLAST GraphBLAS

GraphBLAST [22] is the first high-performance GPU (graphics processing unit) implementation of GraphBLAS. Inspired by the design of GBTL, the architecture of GraphBLAST is also C++ based and maintains a separation of concerns

```
Input: graph, frontier, levels
depth ← 0
while nvals(frontier) > 0:
  depth ← depth + 1
  levels[frontier] ← depth
  frontier<¬levels,replace> ← graph^T ⊕.⊗ frontier
    where ⊕.⊗ = ⨁.⨂(LogicalSemiring)
```

(a) Pseudocode

```
def bfs(graph, frontier, levels):
  depth = 0
  while frontier.nvals > 0:
    depth += 1
    levels[frontier][:] = depth
    with gb.LogicalSemiring, gb.Replace:
      frontier[~levels] = graph.T @ frontier
```

(b) PyGB

```
template<class Mat, class Frontier, class Levels>
void bfs(Mat &graph, Frontier frontier, Levels &levels)
{
  GrB::IndexType depth = 0;
  while (frontier.nvals() > 0) {
    ++depth;
    GrB::assign(levels, frontier, GrB::NoAccumulate(),
                depth, GrB::AllIndices(), false);
    GrB::mxv(frontier, GrB::complement(levels),
             GrB::NoAccumulate(),
             GrB::LogicalSemiring<GrB::IndexType>(),
             GrB::transpose(graph), frontier, true);
  }
}
```

(c) GBTL C++

```
void bfs(GrB_Matrix  graph,
         GrB_Vector  frontier,
         GrB_Vector *levels)
{
  GrB_Index n, nvals;
  GrB_Matrix_nrows(&n, graph);
  GrB_Vector_nvals(&nvals, frontier);
  GrB_Semiring LogicalSemiring;
  GrB_Descriptor Desc_TranA_ScmpM_Replace;
  //...
  GrB_Index depth = 0;
  while (nvals > 0) {
    ++depth;
    GrB_assign(*levels, frontier, GrB_NULL,
               depth, GrB_ALL, n, GrB_NULL);
    GrB_mxv(frontier, *levels, GrB_NULL,
            LogicalSemiring, graph, frontier
            Desc_TranA_ScmpM_Replace);
    GrB_Vector_nvals(&nvals, frontier);
  }
}
```

(d) GraphBLAS C API

Fig. 2: Level-based BFS traversal in math pseudocode, PyGB, GBTL C++, and using the GraphBLAS C API.

between a top-level interface defined by the GraphBLAS C API specification and the low-level backend.

One novel aspect about GraphBLAST is that it supports performance-oriented optimizations such as direction-optimization (also known as push-pull traversal), which was discovered by Beamer, Asanovic and Patterson [15] and generalized by Shun and Blelloch [23] to other graph algorithms. Yang, Buluç and Owens [24] show that this optimization is key for a GraphBLAS implementation to meet the performance of state-of-the-art graph frameworks on the GPU like Gunrock [25]. In each iteration of an `GrB_mxv`, the GraphBLAST backend checks whether the vector sparsity has crossed a threshold k. If it has gone above the threshold, then the traversal will switch from push to pull. If it has gone below the threshold, then the traversal will switch from pull to push. If neither outcome has occurred, then it will use the traversal it used in the previous iteration.

To support direction-optimization, the GraphBLAST backend maintains a SparseVector and DenseVector object as shown in Figure 3. The push traversal is performed using Gustavson's method as a sparse-matrix sparse-vector multiply (SpMSpV) between the SparseVector and the adjacency matrix transpose in CSC format. The pull traversal is performed in a dot-product manner as a sparse-matrix dense-vector multiply (SpMV) between the DenseVector and the adjacency matrix in CSR format. This raises the issue of having to keep around two copies of each `GrB_Matrix` object when it is not symmetric. An environment variable is used to control whether the user wants this performance-oriented storage, or whether they want a more memory-inexpensive storage of only CSR or only CSC in which case the direction-optimization feature is disabled.

Direction-optimization is a good example where the power of abstracting away implementation details shines through, and the linear algebraic approach to graph analytics is at its most powerful. The end user has two conflicting desires:

1) They want to communicate their request *abstractly* enough that they do not have to decide whether the matrix-vector multiply is implemented as push, pull or any of the myriad possible ways.
2) They want their computation to be described *specifically* enough that the computer can optimize for the best approach for doing the computation.

This desire is met by `GrB_mxv`, which is at the same time *abstract* enough to not limit the computer in choosing push traversal when it should be choosing pull traversal, yet *specific* enough that the computer has enough information to cut corners and pick the best algorithm.

Table II shows a comparison of how many lines of code it takes an implementation of GraphBLAS to express a given algorithm in C++. It is compared with two state-of-the-art graph frameworks in shared memory, the aforementioned Ligra [23] and GraphIt [26], which is a DSL designed for expressing graph algorithms.

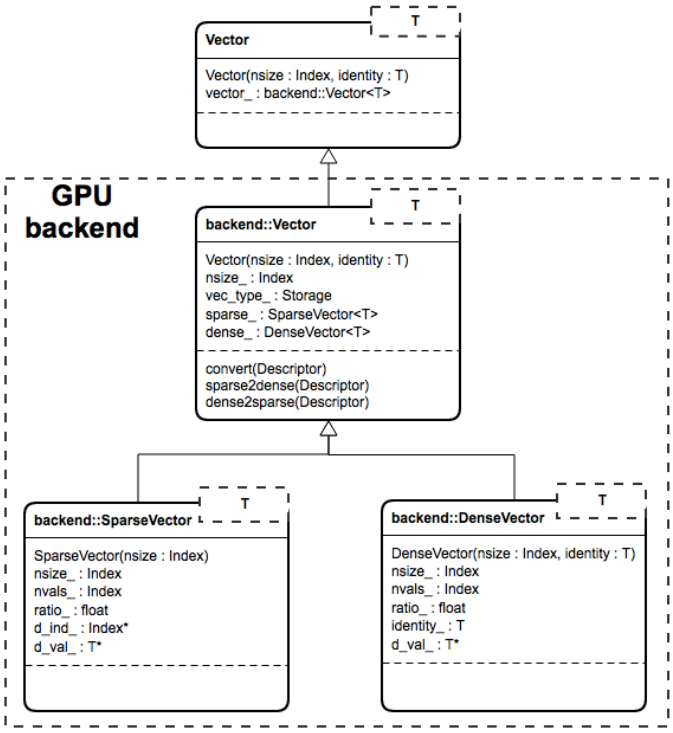

Fig. 3: **GraphBLAST Vector UML diagram.**

Algorithm	Ligra [23]	GraphIt [26]	GraphBLAS
Breadth-first-search	29	22	25
Single-source shortest-path	55	25	25
Local graph clustering	84	N/A	45

TABLE II: Comparison of lines of C++ application code counted by 'cloc' except for numbers of GraphIt, which come from the paper [26]. N/A means not implemented. The specific GraphBLAS implementation for this comparison is GraphBLAST [22].

III. LAGraph Repository

The hypothesis underlying the GraphBLAS is that algorithm designers can focus on expressing their algorithms in terms of the high level linear algebra operations defined in the GraphBLAS while leaving low level optimizations to any particular hardware platform to the implementation of the GraphBLAS. Ultimately, we want hardware vendors to be responsible for creating highly tuned versions of the GraphBLAS specialized to the features of their systems. Kumar et al. [27] show that mitigating the adverse impact of memory latency on performance of algorithms for large graph can lead to significant improvements. Such optimizations require detailed knowledge of the underlying system, and hence the low-level optimizations are best left to hardware vendors. Furthermore, a tuned linear algebra library delivers better performance than straight-forward textbook implementation on many basic graph algorithms [28].

Algorithm designers will naturally wonder how much performance is lost due to the use of a high level API such as the GraphBLAS. As shown in [28], a linear algebra implementation brings inherent efficiency advantages to graph algorithms due to the more structured access to data afforded by the linear algebra formulation [28]. The GraphBLAS API is more general, but we expect its implementations to retain or improve upon the efficiency advantages. This is an untested hypothesis, since until now, we have not had multiple implementations of the GraphBLAS API tuned to the features of a range of platforms.

Testing this hypothesis of the performance potential afforded by the GraphBLAS is a major outcome we anticipate from the LAGraph project. By collecting high level graph algorithms and validating them across an engaged community, we will produce the library of algorithms needed to evaluate the effectiveness of the GraphBLAS approach.

Before we can conduct such experiments, however, we need to collect graph algorithms implemented on top of the GraphBLAS. We have created a GitHub repository at https://github.com/GraphBLAS/LAGraph for members of the LAGraph community to use to contribute GraphBLAS algorithms. The basic elements of the repository include:

- A Build System for creating the LAGraph library and the test routines.
- A library of utilities including loading matrices from disk in Matrix Market format [29], evaluating results, and creating random test matrices.
- A directory of graph algorithms.
- A directory holding a test harness for each algorithm.

We will write documentation, a programmer's reference guide and define procedures for how people can add new algorithms.

IV. Discussion

We are early in the LAGraph project. At this point, we've defined the basic structure of the repository and the overall goals of the project. We have built an early framework for testing and core utility routines to support software development. Finally, we assembled a few algorithms which we are using to test the basic structure of the software system.

Even at this early phase of the project, we have learned a great deal about how the GraphBLAS will interact with end-users. The objects manipulated by the GraphBLAS are opaque. A GraphBLAS implementation is given complete freedom in how data structures underlying the GraphBLAS are implemented. A graph algorithm, however, uses GraphBLAS as part of a processing pipeline. For example, data may exist in data frames. A subset of the data is collected and filtered to produce relationships represented by a graph. Properties of the graph are computed and based on the result a new branch in the processing pipeline may be accessed.

The key here is that graphs inside the GraphBLAS are opaque, but externally they are anything but opaque. This suggests that we need to define functions to import and export data in standard sparse array formats into LAGraph. The initial thinking was that this import functionality would be part of LAGraph and not the GraphBLAS. The only way to do that, however, is if we repacked the input sparse format into separate arrays for column indices, row indices, and values

and then use GrB_Matrix_build to construct the GraphBLAS matrix object. This is extremely inefficient. We need a way to directly import arrays in standard sparse formats, such as CSR/CSC (Compressed Sparse Row/Column) formats, into the GraphBLAS and since the GraphBLAS data types are opaque, this can only be done as a GraphBLAS routine.

Graph algorithms do not occur in isolation. The LAGraph library, therefore, needs to return a handle to an opaque GraphBLAS object so it can be used without incurring copy overhead in subsequent graph operations. Given the nonblocking execution model, this raises interesting design questions about how memory consistency between the library and the application is managed.

Graphs can be quite large. Hence, the default mode should avoid copying sparse arrays input to LAGraph into a separate memory region to hold the opaque GraphBLAS object. We believe it is important that the memory for the input array be reused to hold the GraphBLAS object as much as possible. This means the input array is often "destroyed" (from the perspective of LAGraph, another external library, or the user application) and "realloc"ed for the GraphBLAS opaque object. In the interest of performance and efficient use of memory resources, the above violates the separation of concerns between application and library code expected in well engineered software. There is also the question of communicating to the library routine how the input sparse array was allocated in the first place so the right deallocator can be used.

A draft of SuiteSparse:GraphBLAS includes a working and fully-tested implementation of the import/export feature, using a strategy much like the "move constructor" of C++. For the export of a CSC matrix, for example, three arrays are removed from the GraphBLAS matrix A: a pointer array Ap of size n+1, an index array Ai of size e, and a values array Ax. The row indices of the jth column of the matrix appear as the list Ai[Ap[j]...Ap[j+1]], and the values are in the same locations in Ax. This format is identical to the simple CSC sparse matrix data structure in CSparse [30], except that GraphBLAS allows for many built-in types and arbitrary user-defined types.

The remains of the GraphBLAS object A are then deleted, but all of its content is now "owned" by the external library (LAGraph, say), which is then responsible for freeing these three arrays. Assuming that the opaque GraphBLAS object A is already in the CSC format, the export takes just $O(1)$ time, and no new memory is allocated. The external library (LAGraph, in particular) now has access to the graph. If the GraphBLAS implementation does not support the CSC format in its internal opaque data structure, it can allocate these arrays, populate them, and then free A. The effect is the same; only the performance differs. Opacity is maintained, while at the same time reducing time required for the export from $\Omega(e)$ (for GrB_extractTuples) to as little as $O(1)$.

The import is symmetric with the export: LAGraph (or any other external library) passes in the three arrays Ap, Ai, and Ax, which are then either incorporated as-is into the GrB_Matrix A (taking $O(1)$ time), or copied and freed (taking $O(e)$ time and memory). Either way, the three arrays are now owned by GraphBLAS, not the external library, and would be freed at some point no later than GrB_free(&A). Since the matrix A is opaque, the GraphBLAS library can select whatever method it chooses to take ownership of Ap, Ai, and Ax: a copy (in $O(e)$ time, or a move construction in $O(1)$ time. It may choose later to realloc these arrays if the number of entries needs to grow.

After an export of A, and then an import of the same arrays, the GraphBLAS matrix A is perfectly reconstructed, ideally in a total of $O(1)$ time. SuiteSparse:GraphBLAS supports the import/export of all four of its formats: CSR, CSC, and their hypersparse variants.

A malloc of these Ap, Ai, and Ax arrays by an external library followed by freeing the same space inside GraphBLAS with GrB_free(&A) requires both libraries to agree on using the same malloc/free routines. To do this, the GraphBLAS API would need to be augmented to allow an external library to select which malloc/free routines should be used. This is essential for a MATLAB interface, since a MATLAB mexFunction must use mxMalloc and mxFree. With the import/export feature, sparse matrices can be passed between MATLAB and GraphBLAS in $O(1)$ time, unless typecasting is required. MATLAB supports all of the built-in types of GraphBLAS, plus double complex, but only for dense matrices. For sparse matrices in MATLAB, only double and double complex are available. A MATLAB interface to the GraphBLAS is in progress. If the import/export feature is added to the GraphBLAS C API (as is being considered), this MATLAB interface would interoperate with any GraphBLAS library that is compliant with the spec.

The point of these issues is that in designing an effective library, there are a host of complicated issues to resolve. A major part of the research contribution of this project will be how we solve these issues.

V. Algorithms targeted by LAGraph

Many graph algorithms have been successfully implemented in the language of linear algebra. The following is a list of key algorithms and representative implementations; emphasizing that this list is not exhaustive.

- Breadth-first search (BFS) [31], [19], [11], including the direction optimizing BFS [24],
- Shortest-path (both single-source [22], [32], [19] and all-pairs [33]) calculation,
- Centrality measures, such as Betweenness centrality [2],
- Triangle counting and enumeration [34], [35] as well as k-truss enumeration [36], [37],
- Connected components [38],
- PageRank [39],
- Graph coloring [40],
- Subgraph counting [41],
- Maximal [42] and maximum [43] cardinality matching on bipartite graphs,
- Maximal independent set [44], [22].

Machine learning algorithms are also implemented using libraries that are in the spirit of GraphBLAS:

- Clustering, such as Markov clustering [45] and peer-pressure clustering [46],
- Deep neural network inference [47],
- Collaborative filtering using Stochastic Gradient Descent [39].

Finally, there are algorithms we consider to be important but has so far not been implemented using a GraphBLAS-like library:

- A* search,
- Graph neural network training and inference,
- Branch and bound,
- Graph kernels for supervised learning.

VI. Conclusion

The GraphBLAS forum started its work in 2013 to standardize the building blocks for graph algorithms formulated from linear algebra expressions [7]. We now have a C specification for the GraphBLAS [10] and multiple implementations [11], [17], [18]. The next step in this journey is to define a library of high level graph algorithms that are based on the GraphBLAS.

The GraphBLAS was a community effort launched by a position paper. This is a position paper to launch LAGraph project, our community effort to collect and validate 1) a set of high quality basic graph algorithms that run on top of the GraphBLAS, and 2) support libraries for development of graph analytics applications. Example of support libraries are I/O, generation of scale-free graphs, basic measurements on graphs, and changing representation of graphs. We urge readers interested in joining us as we work on LAGraph to contact any of the authors of this position paper.

Acknowledgments

Tim Davis would like to acknowledge the support of SuiteSparse:GraphBLAS by MIT Lincoln Laboratory, Intel, and the National Science Foundation (NSF CNS-1514406). Scott McMillan was supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM19-0234]. Aydın Buluç and Carl Yang are supported by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231. Carl Yang is also supported by funding support from the Defense Advanced Research Projects Agency (Awards # FA8650-18-2-7835 and HR0011-18-3-0007) and the National Science Foundation (Awards # OAC-1740333 and CCF-1629657).

References

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

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

[3] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for NoSQL databases," in *Intl. Parallel & Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2015, pp. 822–830.

[4] K. Ekanadham, W. P. Horn, M. Kumar, J. Jann, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu, "Graph Programming Interface (GPI): A linear algebra programming model for large scale graph computations," in *Proc. ACM Intl. Conference on Computing Frontiers*, ser. CF '16. New York, NY, USA: ACM, 2016, pp. 72–81.

[5] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[6] S. Che, B. M. Beckmann, and S. K. Reinhardt, "Programming GPGPU graph applications with linear algebra building blocks," *Intl. Journal of Parallel Programming*, pp. 1–23, 2016.

[7] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conf. (HPEC)*. IEEE, 2013.

[8] "The GraphBLAS Forum," http://graphblas.org/.

[9] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson, "Mathematical foundations of the GraphBLAS," in *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[10] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *Graph Algorithms Building Blocks workshop at IPDPS (GABB)*. IEEE, 2017.

[11] T. A. Davis, "Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Software*, 2019 (under submission), see http://suitesparse.com.

[12] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11.

[13] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, 1978.

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

[15] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–10.

[16] "RedisGraph module for the Redis database system," https://oss.redislabs.com/redisgraph/.

[17] K. Ekanadham, B. Horn, J. Jann, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and Y. H., "Graph programming interface: Rationale and specification," Yorktown Heights, NY, Tech. Rep. RC25508 (WAT1411-052), Nov. 2014.

[18] "GraphBLAS Template Library (GBTL)," https://github.com/cmu-sei/gbtl.

[19] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "GBTL-CUDA: Graph algorithms and primitives for GPUs," in *Intl. Parallel & Distributed Processing Symposium Workshop (IPDPSW)*. IEEE, 2016, pp. 912–920.

[20] S. McMillan, "Design and implementation of the GraphBLAS Template Library (GBTL)," in *GraphBLAS: Graph Algorithms in the Language of Linear Algebra Minisymposium at SIAM Annual Meeting (AN16)*, July 2016.

[21] J. Chamberlin, M. Zalewksi, S. McMillan, and A. Lumsdaine, "PyGB: GraphBLAS DSL in Python with dynamic compilation into efficient C++," in *Graph Algorithms Building Blocks (GABB) Workshop at IEEE Intl. Parallel and Distributed Processing Symposium*, May 2018.

[22] C. Yang, "GraphBLAST library," http://github.com/gunrock/graphblast, 2015.

[23] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 135–146.

[24] C. Yang, A. Buluç, and J. D. Owens, "Implementing push-pull efficiently in GraphBLAS," in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP 2018, Aug. 2018, pp. 89:1–89:11.

[25] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017.

[26] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance DSL for graph analytics," *arXiv preprint arXiv:1805.00923*, 2018.

[27] M. Kumar, M. Serrano, J. Moreira, P. Pattnaik, W. P. Horn, J. Jann, and G. Tanase, "Efficient implementation of scatter-gather operations for large scale graph analytics," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–7.

[28] M. Kumar, W. P. Horn, J. Kepner, J. E. Moreira, and P. Pattnaik, "IBM POWER9 and cognitive computing," *IBM Journal of Research and Development*, vol. PP, pp. 1–1, 06 2018.

[29] R. F. Boisvert, R. Pozo, and K. A. Remington, "The Matrix Market exchange formats: initial design," Tech. Rep. NISTIR 5935, 1996.

[30] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2006.

[31] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 65.

[32] 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.

[33] E. Solomonik, A. Buluç, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *Proceedings of the IPDPS*. IEEE Computer Society, 2013.

[34] A. Azad, A. Buluç, and J. R. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proceedings of the IPDPSW, Workshop on Graph Algorithm Building Blocks (GABB)*, 2015, pp. 804 – 811.

[35] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A comparative study on exact triangle counting algorithms on the GPU," in *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 2016, pp. 1–8.

[36] T. A. Davis, "Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss," in *IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

[37] T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan, "Linear algebraic formulation of edge-centric k-truss algorithms with adjacency matrices," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[38] A. Azad and A. Buluç, "LACC: a linear-algebraic algorithm for finding connected components in distributed memory," in *Proceedings of the IPDPS*, 2019.

[39] 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 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 979–990.

[40] M. Osama, M. Truong, C. Yang, A. Buluc, and J. D. Owens, "Graph coloring on the GPU," in *IPDPSW*, 2019.

[41] L. Chen, J. Li, A. Azad, L. Jiang, M. Marathe, A. Vullikanti, A. Nikolaev, E. Smirnov, R. Israfilov, and J. Qiu, "A GraphBLAS approach for subgraph counting," *arXiv preprint arXiv:1903.04395*, 2019.

[42] A. Azad and A. Buluç, "A matrix-algebraic formulation of distributed-memory maximal cardinality matching algorithms in bipartite graphs," *Parallel Computing*, 2016.

[43] ——, "Distributed-memory algorithms for maximum cardinality matching in bipartite graphs," in *Proceedings of the IPDPS*. IEEE, 2016.

[44] A. Lugowski, S. Kamil, A. Buluç, S. Williams, E. Duriakova, L. Oliker, A. Fox, and J. Gilbert, "Parallel processing of filtered queries in attributed semantic graphs," *Journal of Parallel and Distributed Computing (JPDC))*, vol. 79-80, pp. 115–131, 2015.

[45] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks," *Nucleic acids research*, vol. 46, no. 6, pp. e33–e33, 2018.

[46] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *International Workshop on Applied Parallel Computing*. Springer, 2006, pp. 260–269.

[47] J. Kepner, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, and H. Tufo, "Enabling massive deep neural networks with the GraphBLAS," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–10.