Design of the GraphBLAS API for C

Aydın Buluç†, Tim Mattson‡, Scott McMillan§, José Moreira¶, Carl Yang∗†

†Computational Research Division, Lawrence Berkeley National Laboratory
‡Intel Corporation §Software Engineering Institute, Carnegie Mellon University ¶IBM Corporation
∗Electrical and Computer Engineering Department, University of California, Davis, USA

Abstract—The purpose of the GraphBLAS Forum is to standardize linear-algebraic building blocks for graph computations. An important part of this standardization effort is to translate the mathematical specification into an actual Application Programming Interface (API) that (i) is faithful to the mathematics and (ii) enables efficient implementations on modern hardware. This paper documents the approach taken by the C language specification subcommittee and presents the main concepts, constructs, and objects within the GraphBLAS API. Use of the API is illustrated by showing an implementation of the betweenness centrality algorithm.

I. INTRODUCTION

Graphs are a fundamental abstraction in computer science. They represent relationships among objects in a finite collection. The objects, or vertices, in a graph are connected by edges, the relationships. This leads to the common view of a graph as two sets: a set of vertices and a set of edges.

Graphs can also be represented as matrices. For example, the adjacency matrix for a graph is constructed by labeling rows and columns of the matrix by the vertices of the graph. The elements of the matrix denote the edges in the graph with matrix element $A_{ij}$ defining the edge from vertex $i$ to vertex $j$. Most vertices in a large graph, such as those arising in social networks, are not connected to each other, so the matrices used for graphs tend to be very sparse.

Many graph algorithms have been defined in the “language of linear algebra” [1]. Mapping sparse linear algebra algorithms onto modern architectures is well understood, and several groups have built high performance graph libraries based on sparse linear algebra [2], [3], [4], [5], [6]. A group of graph algorithm researchers formed the GraphBLAS Forum [7] to standardize the low-level building blocks used in these graph algorithms. The Forum completed the mathematical formalizations of GraphBLAS [8] and has undertaken the task to define the binding of the C programming language onto the mathematical definition of the GraphBLAS—the so-called GraphBLAS C API.

This work has been undertaken by a subcommittee of the GraphBLAS Forum, comprised of the authors of this paper. The challenge in formulating the GraphBLAS C API was to balance conflicting objectives: (i) simplicity and ease of use, (ii) enabling high-performance implementations, and (iii) adherence to the underlying mathematics. The C programming language has been chosen as the first target by the Forum due to its relatively smaller feature set and its better interoperability with other high-performance languages such as Fortran and C++. Since this is the first specification of GraphBLAS in any language, our burden has largely been to define the programming concepts for the first time. We believe our design will largely carry over to future specs of the GraphBLAS in other languages.

This paper summarizes the GraphBLAS C API and the motivation behind our decisions. We begin by summarizing the mathematical ideas behind the GraphBLAS and how those ideas influenced our notation. We then explain data structures, algebraic objects, and objects that control the semantics of the functions in the GraphBLAS C API. We then define the core operations in the GraphBLAS C API and the signatures for a subset of the functions within the API. We then present a betweenness centrality algorithm that uses the GraphBLAS C API. We close with results and concluding remarks.

II. GRAPHBLAS MATH

Consider a graph represented as an $n$-by-$n$ adjacency matrix $A$, where $A_{ij}$ is the weight of the edge from vertex $i$ to vertex $j$, and a second $k$-by-$n$ matrix $B$ representing a subset (of size $k$) of the vertices in the graph, such that $B_{ji}$ is 1 if the $j$th element of the subset is vertex $i$ (and all other elements of $B$ are 0). The traditional matrix product $B \times A$ 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 $B$. 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.

This generalized matrix multiplication is performed on an algebraic semiring. A semiring is an algebraic structure over a domain $D$ with two binary operators $\oplus$ and $\otimes$. The addition operator, $\oplus$, is a commutative monoid with an identity element $0$ (not necessarily the number 0) while the multiplication operator, $\otimes$, is a commutative monoid with an identity element $1$ (not necessarily the number 1). The additive identity is also an annihilator for the multiplication operator ($\otimes$), and multiplication distributes over addition. The most common semirings used in the graph algorithms community are shown in Table I.

### Table I: Common Semirings

<table>
<thead>
<tr>
<th>Semiring</th>
<th>Addition ($\oplus$)</th>
<th>Multiplication ($\otimes$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>$\lor$ (or)</td>
<td>$\land$ (and)</td>
</tr>
<tr>
<td>Max</td>
<td>$\max(a, b)$</td>
<td>$\max(a, b)$</td>
</tr>
<tr>
<td>Min</td>
<td>$\min(a, b)$</td>
<td>$\min(a, b)$</td>
</tr>
<tr>
<td>Sum</td>
<td>$a + b$</td>
<td>$a + b$</td>
</tr>
<tr>
<td>Product</td>
<td>$ab$</td>
<td>$ab$</td>
</tr>
<tr>
<td>LogSum</td>
<td>$\log(a) + \log(b)$</td>
<td>$\log(a) \cdot \log(b)$</td>
</tr>
</tbody>
</table>

...
It is often convenient to change the semiring applied to a matrix. This means we must represent the matrix and the semiring separately, and the two come together only when an operation is performed. Mathematically, the ability to change semirings when moving from one GraphBLAS operation to the next impacts the meaning of the implied zero in a sparse representation of the matrix. This element in real arithmetic is the number zero (0), which is the identity of the addition operator and the annihilator of the multiplication operator. As the semiring changes, this implied zero changes to the identity of the addition operator and the annihilator of the multiplication operator for the new semiring. Nothing changes in the stored matrix, but the implied values within the sparse matrix change with respect to a particular operation.

This feature has significant impact on the definitions of GraphBLAS operations. Consider matrix multiplication over the domain $S$ with semiring operators $\oplus$ and $\otimes$:

$$C = A \otimes \circ \cdot B = AB.$$ 

Using index notation familiar in linear algebra

$$C(i, j) = \bigoplus_{k=1}^{l} A(i, k) \otimes B(k, j)$$

for matrices with dimensions $A : S^{m \times l} \quad B : S^{l \times n} \quad C : S^{m \times n}$

The summation notation only works, however, if we redefine the implied zero of the sparse matrices as we change the semiring (to the corresponding additive identity). Depending on the domains associated with the matrix elements and the operations, this can lead to awkward definitions of the operations involving the implied zeros. A cleaner approach based on set notation avoids this problem. For example, we can define the previous matrix multiplication as

$$C(i, j) = \bigoplus_{k \in \text{ind}(A(i, :)) \cap \text{ind}(B(:, j))} (A(i, k) \otimes B(k, j)),$$

where $\text{ind}(A(i, :))$ is the set of the column indices of the elements that are stored in row $i$ of matrix $A$, and $\text{ind}(B(:, j))$ is the set of the row indices of the elements that are stored in column $j$ of matrix $B$.

In other words, the binary operation $\otimes$ is applied to the elements in the intersection of the two sets $\text{ind}(A(i, :))$ and $\text{ind}(B(:, j))$, and the results of this operation are accumulated using the $\oplus$ operator. These notations are equivalent. By defining pairwise operations over set intersections, however, we avoid needing to define how the semiring’s additive identity interacts with the matrix’s implied zeros.

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

### III. GraphBLAS objects

The GraphBLAS C API is built on objects exposed as opaque data types. These objects include
- **Collections**: vectors and matrices.
- **Algebraic objects**: unary and binary operators, monoids, and semirings.
- **Control objects**: descriptors and masks (both one- and two-dimensional).

Functions that manipulate GraphBLAS objects are referred to as **methods**. These methods define the interface to create or destroy GraphBLAS objects, modify their contents, and copy the contents of opaque objects into non-opaque objects (i.e., under direct observation and control of the programmer).

#### A. Collections

The state of a GraphBLAS application is largely captured by collections of values, namely vectors and matrices. The GraphBLAS collections are opaque objects accessible only through the methods in the GraphBLAS C API. The use of opaque data types gives the implementation the flexibility needed to aggressively optimize for different systems.

A GraphBLAS vector $\mathbf{v} = (D, N, \{(i, v_i)\})$ is defined by
- The domain $D$, the data type for the vector elements.
- The size $N > 0$. 

---

**TABLE I: Common semirings used with graph algorithms.**

<table>
<thead>
<tr>
<th>Semiring</th>
<th>operators</th>
<th>domain</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard arithmetic</td>
<td>$+$</td>
<td>$\otimes$</td>
<td>$\mathbb{R}$</td>
<td>0</td>
</tr>
<tr>
<td>max-plus algebras</td>
<td>$\max$</td>
<td>$+$</td>
<td>${-\infty \cup \mathbb{R}}$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>min-max algebras</td>
<td>$\min$</td>
<td>$\max$</td>
<td>$\mathbb{R}_{\geq 0}$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>Galois fields (e.g., GF2)</td>
<td>$\circ$</td>
<td>$\otimes$</td>
<td>${0, 1}$</td>
<td>0</td>
</tr>
<tr>
<td>Power set algebras</td>
<td>$\cup$</td>
<td>$\cap$</td>
<td>$\mathcal{P}(\mathbb{Z})$</td>
<td>$\otimes$</td>
</tr>
</tbody>
</table>

**TABLE II: A mathematical overview of the fundamental GraphBLAS operations supported in this specification.**

<table>
<thead>
<tr>
<th>Operation name</th>
<th>Mathematical description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{mxm}$</td>
<td>$C = A \otimes \circ \cdot B$</td>
</tr>
<tr>
<td>$\text{mxv}$</td>
<td>$C = A \otimes \circ \cdot v$</td>
</tr>
<tr>
<td>$\text{vmx}$</td>
<td>$C = A \otimes \circ \cdot w$</td>
</tr>
<tr>
<td>$\text{eWiseMult}$</td>
<td>$C = A \otimes \circ \cdot B$</td>
</tr>
<tr>
<td>$\text{eWiseAdd}$</td>
<td>$C = A \otimes \circ \cdot v$</td>
</tr>
<tr>
<td>$\text{reduce (row)}$</td>
<td>$C = \bigoplus_i A(:, j)$</td>
</tr>
<tr>
<td>$\text{apply}$</td>
<td>$C = F_u(A)$</td>
</tr>
<tr>
<td>$\text{transpose}$</td>
<td>$C = A^T$</td>
</tr>
<tr>
<td>$\text{extract}$</td>
<td>$C = A(i,j)$</td>
</tr>
<tr>
<td>$\text{assign}$</td>
<td>$C(i,j) \circ \cdot A$</td>
</tr>
<tr>
<td>$w(i)$</td>
<td>$w(i) = u(i)$</td>
</tr>
</tbody>
</table>
• A set of tuples \((i, v_i)\) where \(0 \leq i < N\) and \(v_i \in D\).

A particular value of \(i\) can appear only once in \(v\). We define \(\text{nelem}(v) = N\) and \(\text{L}(v) = \{(i, v_i)\}\). The set \(\text{L}(v)\) is called the \textit{content} of vector \(v\). For a vector \(v\), \(v(i)\) is a reference to \(v_i\) if \((i, v_i) \in \text{L}(v)\) and is undefined otherwise.

A GraphBLAS matrix \(A = \langle D, M, N, \{(i, j, A_{ij})\}\rangle\) is defined by

- The domain \(D\), the data type for the matrix elements.
- The number of rows \(M > 0\) and columns \(N > 0\).
- A set of tuples \(\text{L}(A) = \{(i, j, A_{ij})\}\) where \(0 \leq i < M\), \(0 \leq j < N\), and \(A_{ij} \in D\).

A particular pair of values \(i, j\) can appear only once in \(A\). The set \(\text{L}(A)\) is called the \textit{content} of matrix \(A\). We define \(\text{nrows}(A) = M\) and \(\text{ncols}(A) = N\). For a matrix \(A\), \(A(i, j)\) is a reference to \(A_{ij}\) if \((i, j, A_{ij}) \in \text{L}(A)\) and is undefined otherwise. This points to an important and fundamental difference between the GraphBLAS and traditional sparse matrix libraries for which elements that are not explicitly stored are assumed to have the numerical value 0. As we will see in Section III-B, by specifying these elements as undefined, we avoid the complexity of interpreting implied values in the sparse array definition differently as the semiring changes.

If \(A\) is a matrix and \(0 \leq j < N\), then

\[
A(:, j) = \langle D, M, \{(i, A_{ij}) : (i, j, A_{ij}) \in \text{L}(A)\}\rangle
\]

is a vector called the \textit{j}-th column of \(A\). Correspondingly, if \(A\) is a matrix and \(0 \leq i < M\), then

\[
A(i,:) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \text{L}(A)\}\rangle
\]

is a vector called the \textit{i}-th row of \(A\).

We define the transpose of a matrix in the traditional manner where row and column indices are swapped. Consider a matrix \(A = \langle D, M, N, \{(i, j, A_{ij})\}\rangle\); the transpose of \(A\) is the matrix

\[
A^T = \langle D, N, M, \{(j, i, A_{ij}) : (i, j, A_{ij}) \in \text{L}(A)\}\rangle
\]

\textbf{B. Algebraic objects}

The GraphBLAS differs from traditional sparse linear algebra APIs in that the algebra associated with the data can be varied to match the needs of an application. This provides great deal of flexibility in the domains for the elements of GraphBLAS collections and the operators defined between these elements.

We start with basic binary and unary operators. A GraphBLAS \textit{binary operator} is defined as

\[
F_b = \langle D_1, D_2, D_3, \odot \rangle.
\]

It has three domains—\(D_1\), \(D_2\), and \(D_3\)—and an operation

\[
\odot : D_1 \times D_2 \rightarrow D_3.
\]

A GraphBLAS \textit{unary operator} is defined as

\[
F_u = \langle D_1, D_2, f \rangle.
\]

It has two domains, \(D_1\) and \(D_2\), and an operation \(f : D_1 \rightarrow D_2\). GraphBLAS makes use of two fundamental algebraic structures: monoids and semirings. A GraphBLAS \textit{monoid}

\[
M = \langle D_1, \odot, \mathbf{0} \rangle
\]

is defined by a single domain \(D_1\), an \textit{associative}\(^1\) operation \(\odot : D_1 \times D_1 \rightarrow D_1\), and an identity element \(\mathbf{0} \in D_1\). Let \(F = \langle D_1, D_1, D_1, \odot \rangle\) be a GraphBLAS binary operator, and let \(\mathbf{0}\) be the identity for \(\odot\). Then

\[
M = \langle F, \mathbf{0} \rangle = \langle D_1, \odot, \mathbf{0} \rangle
\]

is the associated GraphBLAS monoid.

The algebraic structure at the core of the GraphBLAS is the semiring. A GraphBLAS \textit{semiring}

\[
S = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0} \rangle
\]

is defined by three domains—\(D_1\), \(D_2\), and \(D_3\); an \textit{associative} additive operation

\[
\oplus : D_3 \times D_3 \rightarrow D_3;
\]

and a multiplicative operation

\[
\otimes : D_1 \times D_2 \rightarrow D_3;
\]

which is the identity for \(\odot\). Let \(F = \langle D_1, D_2, D_3, \odot \rangle\) be a GraphBLAS binary operator, and let \(M = \langle D_3, \oplus, \mathbf{0} \rangle\) be a GraphBLAS monoid; then

\[
S = \langle M, F \rangle = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0} \rangle
\]

is a GraphBLAS semiring. Conversely, for a GraphBLAS semiring \(S = \langle D_1, D_2, D_3, \oplus, \otimes, \mathbf{0} \rangle\), there is always an associated monoid \(M = \langle D_3, \oplus, \mathbf{0} \rangle\) and an associated binary operator \(F = \langle D_1, D_2, D_3, \odot \rangle\).

A UML diagram of the conceptual hierarchy of algebraic objects in GraphBLAS algebra is shown in Figure 1. Note that GraphBLAS semiring differs from the fundamental algebraic semiring in that the GraphBLAS semiring (i) allows input from different domains and can produce an output in a third domain and (ii) does not require the definition of the multiplicative identity.

\textbf{C. Control objects}

The GraphBLAS C API defines two opaque objects that modify the semantics of GraphBLAS methods: masks and descriptors.

A mask is either a one- or a two-dimensional construct. One- and two-dimensional masks, described more formally below, are similar to vectors and matrices, except that they have structure (indices) but no values. Masks are used to control which values from an operation are written to the output object.

A one-dimensional mask \(m = \langle N, \{i\}\rangle\) is defined by its number of elements \(N > 0\) and a set \(\text{L}(m)\) of indices \(\{i\}\) where \(0 \leq i < N\). A particular value of \(i\) can appear at

\footnote{It is expected that implementations will utilize IEEE-754 floating point arithmetic, which is not strictly associative.}
The structure of the complement of a one-dimensional mask is defined as \( m \). We define \( \text{nelem}(m) = N \). We define the structure of a one-dimensional mask as the set \( L(m) \). A two-dimensional mask \( M = \langle M, N, \{(i, j)\} \rangle \) is defined by its number of rows \( M > 0 \), its number of columns \( N > 0 \), and a set \( L(M) \) of tuples \((i, j)\) where \( 0 \leq i < M, 0 \leq j < N \). A particular pair of values \( i, j \) can appear at most once in \( M \). The structure of a two-dimensional mask \( M \) is the set \( L(M) \). We also define \( \text{nrows}(M) = M \) and \( \text{ncols}(M) = N \).

Operations may be directed to use the structural complement of a mask. For a one-dimensional mask, \( m \), this is denoted as \( \neg m \). For a two-dimensional mask, \( M \), this is denoted as \( \neg M \). The structure of the complement of a one-dimensional mask \( m \) is defined as

\[
L(\neg m) = \{i : 0 \leq i < N, i \notin L(m)\},
\]

which is the set of all possible indices that do not appear in \( m \). The structure of the complement of a two-dimensional mask \( M \) is defined as

\[
L(\neg M) = \{(i, j) : 0 \leq i < M, 0 \leq j < N, (i, j) \notin L(M)\},
\]

the set of all possible indices that do not appear in \( M \).

The second control object is the descriptor. Descriptors modify the semantics of GraphBLAS methods by controlling additional optional behaviors. In particular, descriptors specify how other input arguments—vectors, matrices, and masks—should be processed (modified) before the main operation of a method is performed. It is also used to specify whether the output argument should be cleared before assignment.

The descriptor is a lightweight object. It pairs a set of flags representing the possible modifiers with each mask, vector, or matrix argument of a GraphBLAS method. For example, a descriptor may specify that a particular input matrix needs to be transposed or that the structural complement of a mask should be used.

For the descriptors, the arguments of a method are identified through the field names. The output parameter (typically the first parameter in a GraphBLAS method) is indicated by the field name \( \text{GrB_OUTP} \); the mask by the field name \( \text{GrB_MASK} \); and the input vectors and matrices by \( \text{GrB_INP0} \) and \( \text{GrB_INP1} \) (in the order they appear in the signature of the method). A code example showing the creation of a descriptor can be found on lines 14-18 of Figure 3.

**IV. GraphBLAS Execution Model**

Algorithms are expressed as one or more sequences of GraphBLAS method calls. The methods within a sequence occur in the order they are encountered in the program (the Program Order). A new sequence begins with the first GraphBLAS method call and terminates with a call to the \( \text{GrB_wait()} \) method. New sequences can begin following termination of a prior sequence, hence sequences within a single thread are contiguous and do not overlap. A multithreaded program may have a distinct sequence per thread, but those sequences must not share objects unless the shared objects are read-only.

Each method in a sequence and the inputs to the method uniquely and unambiguously define the output GraphBLAS objects. A GraphBLAS program executes in one of two modes: blocking or nonblocking.

- **blocking**: Each method in a sequence completes the GraphBLAS operation before proceeding to the next statement in program order. Output GraphBLAS objects are completely computed and stored in memory before each method returns.
- **nonblocking**: Methods may return after input arguments have been verified. Methods that manipulate only opaque objects may defer their execution. Methods that input non-scalar, non-opaque objects or output non-opaque objects may not defer their execution. When a method is deferred, values associated with the method’s output GraphBLAS objects are not completely computed until (1) the sequence is terminated or (2) a GraphBLAS method that copies values from a GraphBLAS object into a non-opaque object returns. At that point, the GraphBLAS object is said to be complete (i.e., values associated with that object are completely computed and stored in memory).

In a mathematically well-defined sequence with input objects that are numerically well-conditioned, the results from blocking and nonblocking modes should be identical except for effects due to round-off errors associated with floating point arithmetic.

Blocking mode forces an implementation to carry out precisely the GraphBLAS operations defined by the methods and to store output objects to memory between method calls. This mode is valuable for debugging or when an external tool needs to evaluate the state of memory during a sequence.

Nonblocking mode gives an implementation flexibility to choose an execution strategy that might reduce the time required to execute the methods in a sequence. Methods may be placed into a queue and deferred. They can be chained together and fused (e.g., replacing a chained pair of matrix
products with a matrix triple product). Lazy evaluation, greedy evaluation, and asynchronous execution are all valid as long as the final result agrees with the mathematical definition provided by the sequence of GraphBLAS method calls.

A conforming implementation of the GraphBLAS C API running in nonblocking mode may choose to execute “as if” in blocking mode. Furthermore, a sequence in nonblocking mode where every GraphBLAS operation is followed by a call to `GrB_wait()` is equivalent to the same sequence in blocking mode without the calls to `GrB_wait()`.

The context is defined in the GraphBLAS C API when the context of the library invocation is defined. This occurs once before any GraphBLAS methods are called with a call to the `GrB_init()` function. After all GraphBLAS methods are complete, the context is terminated with a call to `GrB_finalize()`. The context can be set only once in the execution of a program (i.e., after `GrB_finalize()` is called, a subsequent call to `GrB_init()` is not allowed).

V. Error Model

GraphBLAS methods return a value of type `GrB_Info` to provide information about the execution of a method available at the time the method returns. The specification of each method lists the allowed return values. Errors fall into two classes: API errors and execution errors. An API error means a GraphBLAS method was called with parameters that violate the rules for that method. Execution errors indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the executing environment. This does not mean that environment errors are the fault of the GraphBLAS implementation. For example, a memory leak is a program error, but it may manifest itself in different points of program execution (or not at all) depending on the platform, problem size, or what else is running at that time.

When a GraphBLAS method is called, the arguments are evaluated for any API errors. If any API errors are found, the method returns without making any changes to the method's arguments and with a return value corresponding to the appropriate API error. If API errors are not found, the method can proceed to carry out the computation associated with the method.

In blocking mode the computation for each method proceeds after testing for any API errors. When the method is finished, it returns the value `GrB_SUCCESS` if the computation completed without errors. If an execution error was found, the method returns a value to indicate the appropriate execution error.

In nonblocking mode, we distinguish between methods that may defer execution (i.e., methods that only read and write opaque objects) and methods that may not defer execution (i.e., methods that read input arrays or methods that force completion of a GraphBLAS object). If the methods allow deferred execution, API errors are tested when the method is called. If no API errors are found, the method may return at this point with the value of `GrB_SUCCESS`. Since execution in nonblocking mode may be deferred, the only information guaranteed to be available when a method returns pertains to the API test, and the return value may not provide any information about the status of the computation.

When the sequence is terminated by a call to `GrB_wait()` a value of `GrB_SUCCESS` is returned if no execution errors were encountered in the execution of the sequence. Other return values from `GrB_wait()` indicate an error occurred during execution of the sequence. Additional information may be available by a call to `GrB_error()`, which returns a pointer to a null terminated string containing any additional error information that might be available.

Methods in nonblocking mode that do not allow deferred execution carry out the computation associated with the method after tests for API errors. When these methods—such as methods to extract values from an opaque object into a non-opaque object—force completion of a GraphBLAS object, the return value is `GrB_SUCCESS` if no execution errors were generated by any of the methods involved in defining the mathematical value of the completed GraphBLAS object. If execution errors were encountered, a return value indicates that an error condition was encountered and, as with the call to `GrB_wait()`, additional information may be provided by a call to `GrB_error()`.

VI. The GraphBLAS C API

Definition of the full GraphBLAS C API is beyond the scope of this paper. Instead, we focus on those parts of the API that are needed to understand the betweenness centrality example in Section VII.

We begin with data types used for the objects in the example. These are listed in Table III. Except for the types that map directly onto C language basic types (`GrB_Info`, `GrB_Index`, and `GrB_Type`), these data types define handles to opaque objects manipulated by GraphBLAS methods. Objects corresponding to algebraic structures (`GrB_Monoid` and `GrB_Semiring`) are constructed from lower-level operators. The GraphBLAS C API provides a mechanism for creating user-defined operators, but for this paper we consider only the predefined operators used in the example (summarized in Table IV). A number of constant literal values are used in the GraphBLAS methods to choose among options or to define return values. The most commonly used GraphBLAS literals are listed in Table V.

We illustrate the principles behind the GraphBLAS C API with a single example: the `GrB_mxm()` operation, shown in

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GrB_Info</code></td>
<td>Return value from any GraphBLAS method</td>
</tr>
<tr>
<td><code>GrB_Index</code></td>
<td>Vector and matrix indices</td>
</tr>
<tr>
<td><code>GrB_Type</code></td>
<td>Type identifier</td>
</tr>
<tr>
<td><code>GrB_Descriptor</code></td>
<td>Opaque GraphBLAS descriptor object</td>
</tr>
<tr>
<td><code>GrB_Monoid</code></td>
<td>Opaque GraphBLAS monoid object</td>
</tr>
<tr>
<td><code>GrB_Semiring</code></td>
<td>Opaque GraphBLAS semiring object</td>
</tr>
<tr>
<td><code>GrB_Matrix</code></td>
<td>Opaque GraphBLAS matrix object</td>
</tr>
<tr>
<td><code>GrB_Vector</code></td>
<td>Opaque GraphBLAS vector object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GrB_Info</code></td>
<td>Return value from any GraphBLAS method</td>
</tr>
<tr>
<td><code>GrB_Type</code></td>
<td>Type identifier</td>
</tr>
<tr>
<td><code>GrB_Monoid</code></td>
<td>Opaque GraphBLAS monoid object</td>
</tr>
<tr>
<td><code>GrB_Semiring</code></td>
<td>Opaque GraphBLAS semiring object</td>
</tr>
<tr>
<td><code>GrB_Matrix</code></td>
<td>Opaque GraphBLAS matrix object</td>
</tr>
<tr>
<td><code>GrB_Vector</code></td>
<td>Opaque GraphBLAS vector object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GrB_Info</code></td>
<td>Return value from any GraphBLAS method</td>
</tr>
<tr>
<td><code>GrB_Type</code></td>
<td>Type identifier</td>
</tr>
<tr>
<td><code>GrB_Monoid</code></td>
<td>Opaque GraphBLAS monoid object</td>
</tr>
<tr>
<td><code>GrB_Semiring</code></td>
<td>Opaque GraphBLAS semiring object</td>
</tr>
<tr>
<td><code>GrB_Matrix</code></td>
<td>Opaque GraphBLAS matrix object</td>
</tr>
<tr>
<td><code>GrB_Vector</code></td>
<td>Opaque GraphBLAS vector object</td>
</tr>
</tbody>
</table>
TIMES
[0x0]TIMES
[0x0]Monoid
[0x0]SCMP
[0x0]Matrix
[0x0]INP0
[0x0]assign
[0x0]extract
[0x0]FP32
[0x0]NULL
[0x0]mxm()
[0x0]MASK
[0x0]build
[0x0]mxm()
[0x0]nrows
[0x0]new
[0x0]mxm
[0x0]mxm()
[0x0]PLUS
[0x0]FP32
[0x0]mxm
[0x0]mxm()
[0x0]Semiring
[0x0]mxm
[0x0]set
[0x0]INP1
[0x0]SUCCESS
eWiseMult
Vector
IDENTITY
nvals
FP32
FP32
new
eWiseAdd
apply
PLUS
new
Matrix
ALL
INT32
OUTP
new

arguments according to the descriptor. Depending on the B
The descriptor field
GrB
the indicated operation is carried out. This produces an internal
domains and sizes of the objects are mathematically consistent,
structural complement of the mask should be used. If the
exist and are true correspond to the elements of the output
matrix that might be replaced by the corresponding elements of Z. Two options are supported:
- Replace mode: If the descriptor field GrB_OUTP is set to
GrB_REPLACE, the values in the C matrix are deleted before masked elements of Z are stored in C. In essence the computed matrix Z replaces the original matrix C.
- Merge mode: Otherwise, the elements from the computation selected by the write mask are written into the output C matrix without changing elements that do not overlap with the mask.

The basic pattern used for GrB_mxm() is used for most of the GraphBLAS operations. For example, the mathematical operations in Table II from GrB_mxm to GrB_reduce use descriptors to modify input matrices or vectors, and write masks. We can’t list all methods in the GraphBLAS API, but for the methods used in the example from Section VII we list method names and descriptions in Table VI.

**VII. EXAMPLE: BETWEENNESS CENTRALITY**
Betweenness centrality (BC) is a popular metric to assess the centrality of vertices in a graph. It is based on shortest paths where the BC score of a vertex \( u \) is the normalized ratio
of the number of shortest paths between any pair of vertices that go through \( v \) to the total number of shortest paths in the graph. Equation 1 formally defines BC where \( \sigma_{st} \) denotes the number of shortest paths from vertex \( s \) to vertex \( t \), and \( \sigma_{st}(v) \) is the number of such paths passing through vertex \( v \):

\[
BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}
\]

The BC score is efficiently computed using Brandes’ algorithm [9], which runs in \( O(mn) \) time on unweighted graphs and avoids the expensive explicit all-pairs shortest paths computation. For each starting vertex \( s \), Brandes’ algorithm computes the BC contributions from the shortest paths starting at \( s \) that pass through every other vertex.

We implemented a batched version of Brandes’ algorithm [2, 10, 11] using the GraphBLAS C API where BC contributions from multiple source vertices are computed simultaneously. Figure 3 shows the subroutine, BC_update, that computes BC contributions from a subset of source vertices. Compared to previous work, the flexibility offered by the GraphBLAS API (i.e., the masks, accumulators, and descriptors) reduces the number of function calls in the main loops and the number of intermediate objects.

At a high level, the BC_update function performs two sweeps over the graph. The forward sweep performs multiple simultaneous breadth-first search traversals (one for each source vertex) and keeps track of the number of independent shortest paths that reach every vertex from the source. This is performed by the do-while loop starting at line 39. The backward sweep rolls back and tallies the BC contributions to every vertex. This is performed by the for loop starting at line 69.

The remainder of this section describes the GraphBLAS implementation in detail. For the sake of brevity and clarity, the examination of GrB_Info return codes and handling of any errors are omitted.

A. Signature

In line 1, a single header file, GraphBLAS.h defines all GraphBLAS collections, algebraic objects, and method signatures. The BC_update expects the following parameters (line 3):

- \( \text{delta} \): an uninitialized vector that will hold the computed BC contributions to each vertex on return. It is initialized to \( n \) 32-bit floating point elements (line 7).
- \( A \): an \( n \times n \) integer adjacency matrix representing an unweighted, directed graph where the presence of an edge is indicated by a stored 1.
- \( s \): an array of source vertex indices (the batch).
- \( nsver \): the number of source vertices in \( s \)

Although not required, this function returns the same type of status code as found in all GraphBLAS methods.
B. BFS Phase Objects

During the forward sweep, the algorithm performs \texttt{nsver} simultaneous BFS traversals. In this implementation it uses 32-bit integer \texttt{(GrB_INT32 domain)} arithmetic and relies on GraphBLAS’s predefined binary operators, \texttt{GrB_PLUS_INT32} and \texttt{GrB_TIMES_INT32}, to declare an addition monoid (\texttt{Int32Add} on line 9) and the arithmetic semiring (\texttt{Int32AddMul} on line 11). If the application needs to process larger graphs, this code should be modified to use 64-bit integer arithmetic and 64-bit indices. A descriptor is used for this phase (starting on line 14) to transpose the first input matrix, structurally complement the mask (if provided), and “replace” the values in the output vector.

The columns of the \texttt{numsp} matrix keep track of the number of independent shortest paths that reach every other vertex from the corresponding source vertices. The dimensions are \texttt{n \times nsver} where each column corresponds to a different source vertex. This matrix is initialized so a single element in each column, corresponding to its source vertex, is set to one. Mathematically,

\[
\text{numsp}(s, i) = 1, \quad 0 \leq i < \text{nsver}. \tag{2}
\]

This is accomplished in lines 20–29 using the \texttt{GrB_Matrix_build} operation. The row index array for this operation comes from the \texttt{s} parameter while the column indices are created in the \texttt{i\_nsver} array – one for each source vertex. An array of size \texttt{nsver} is filled with ones in \texttt{ones}. The call to \texttt{GrB_Matrix_build} specifies the integer addition operator, \texttt{GrB_PLUS_INT32}, in case there are any duplicate entries.

The columns of the \texttt{frontier} matrix contain the current frontiers for the traversals from each source vertex. Integers are stored that correspond to how many shortest paths reached a given vertex during that step. In lines 31–33, this \texttt{n \times nsver} matrix is initialized. Each column of this matrix is initialized to the out vertices of the corresponding source vertex. This is done using the GraphBLAS \texttt{extract} operation on the graph matrix A. The descriptor transposes A, such that the use of the \texttt{s} array specified for the column indices selects each row of A corresponding to the source vertices. The use of \texttt{GrB_ALL} specified for all \texttt{n} row indices ensures that all out neighbors of the selected source vertices are included. The \texttt{numsp} matrix is specified as the mask, which is complemented by the descriptor, and has the effect of removing the source vertices themselves from each frontier (it will remove these elements only if source vertices have edges that point to themselves). Since the \texttt{frontier} matrix is already empty, the descriptor’s \texttt{GrB_REPLACE} parameter has no effect.

The final data structures needed for the BFS phase are a set of matrices that store the current frontier at each step of the BFS phase. This is stored in an array of matrices called \texttt{sigmas}. A set of \texttt{n} of these matrices is dynamically allocated at line 36. Note that the number of these matrices is bounded by the diameter of the graph whose upper bound is the number of vertices in the graph.

C. BFS Phase (Forward Sweep)

The BFS phase of the computation begins with the do-loop on line 39. The first step initializes the \texttt{nx\_nsver \_sigmas[d]} matrix (line 40) and sets it to the current frontier (line 41). The \texttt{apply} operation is used for the assignment and by using the unary operator, \texttt{GrB_IDENTITY_BOOL}, casts the integers in the frontier to booleans. On line 42, the path counts for the current frontier are accumulated using the \texttt{eWiseAdd} operation to perform an element-wise addition of the \texttt{numsp} and \texttt{frontier} matrices. The result is stored in \texttt{numsp}.

The \texttt{GrB_mxm} call on line 43 forms the next frontier in one step by both expanding the current frontier (i.e., discovering the 1-hop neighbors of the set of vertices in the current frontier) and pruning the vertices that have already been discovered. The former is achieved by setting the descriptor, \texttt{desc_tsr}, to use the transpose of the adjacency matrix. The latter is achieved by setting the descriptor to use the structural complement of the mask and by passing the \texttt{numsp} matrix as the mask parameter. The implicit cast of \texttt{numsp} to \texttt{Boolean} allows \texttt{GrB_mxm} to interpret \texttt{numsp} as the set of previously discovered vertices. Note that the descriptor is also set to \texttt{GrB_REPLACE} to ensure that the frontier is overwritten with new values.

The loop ends by computing the number of values in the new frontier using the matrix method, \texttt{GrB_Matrix_nvals}. If the result (stored in \texttt{nvals}) is zero, there are no vertices in the frontier and the forward sweep is completed.

D. Tally Phase Objects

The BC contributions are calculated during the tallying phase that performs a backwards sweep using the previously stored BFS trees. An arithmetic semiring with a floating-point domain is used for this computation. In this example, we choose 32-bit float-point types and declare the necessary monoids and semiring (starting at line 48) based on GraphBLAS’s predefined binary operators.

Starting on line 55, the element-wise inverse of \texttt{numsp} is computed using the \texttt{apply} operation along with the predefined multiplicative inverse unary function defined for 32-bit floating point, \texttt{GrB_MINV_FP32}. Then, the \texttt{bcu} variable that holds the per-source BC contributions is initialized to all ones in order to avoid issues with the treatment of implied zeros (starting at line 59). This fill is accomplished with a variant of the \texttt{assign} operation that allows the same value to be assigned to a subgraph, and that specifies \texttt{GrB_ALL} for both row and column indices. Last, a descriptor object, \texttt{desc_r}, needed by the Tally Phase is initialized starting on line 63. The only parameter needed in this phase is “replace” semantics when using a mask.

E. Tally Phase (Backward Sweep)

After the initialization of a temporary workspace matrix \texttt{w}, the Tally Phase begins on line 69. On line 70, the contributions of each “end” vertex to its predecessors are divided by the number of shortest paths that reach them. This is accomplished
with an eWiseMult operation where the sigma[i] matrix is used to ensure that only paths identified in the BFS phase are assigned to the result. The GrB_mxm call on line 73 discovers predecessors (as opposed to successors in the forward sweep) by its use of the descriptor desc_r (defined in line 63) that uses the adjacency matrix (as opposed to its transpose). The algorithm assumes that the BC contributions are transferred only to direct parents on the BFS tree by passing the previous level of BFS tree (sigma[i-1]) as a mask to GrB_mxm. Last, the elements of the w matrix are scaled by the number of shortest paths with the eWiseMult operation on line 74, and this result is accumulated into the BC update matrix, bcu. This loop terminates when the original source vertices are reached and the columns of bcu contain the BC updates for each source vertex, respectively.

F. Computing BC Updates

To compute the BC updates for all vertices in this batch, the elements in each row of bcu are accumulated using the reduce operation on line 78. This result is biased because bcu began the loop filled with ones; hence all entries in this matrix are greater than the actual update by one. As a result, all elements of the reduction need to be adjusted by the number of source vertices. This is accomplished by accumulating the reduction result (in line 78) with the output vector, delta, that was filled with -nsver on line 77.

The remainder of the subroutine frees resources allocated in this computation. Note that GrB_free_all is a convenience macro (not part of GraphBLAS C API) that expands to GrB_free for each of its parameters.

VIII. RESULTS

The BC code in Figure 3 was tested using the GraphBLAS Template Library (GBTL)[12]. This C++ library is functionally equivalent to the the GraphBLAS C API with similar signatures. The latest development snapshot, including the working BC implementation, is available [13].

Work is in progress to integrate the GraphBLAS C API with Gunrock [14], a highly optimized backend for execution on GPU backends. Another project [15] implemented a variation of the GraphBLAS C API using standard C99. Efforts are underway to add a GraphBLAS C interface to GPI [4]. Last, a GraphBLAS library in Chapel is under development [16].

IX. CONCLUSION

The GraphBLAS forum started its work on standard building blocks for graphs in the language of linear algebra in 2013 [17]. A few years later, we released the mathematical definition of the GraphBLAS [8]. In this paper, we report on the first effort to define a language binding to the GraphBLAS; the GraphBLAS C API.

With the specification complete, work is now underway to create reference implementations of the API. After reference implementations are complete and we validate the core GraphBLAS C API, work will shift to optimizing GraphBLAS C libraries and benchmarking against more established frameworks for graph algorithms.

ACKNOWLEDGMENTS

Aydn Bulu¸c and Carl Yang were supported in part by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. Scott McMillan’s work was supported by the DOD under Contract No. FA8721-05-C-0003 with CMU for the operation of the Software Engineering Institute, a federally funded research and development center [DM-0004492]. The authors thank the members of the GraphBLAS Forum for their input, suggestions, and guidance during the development of the GraphBLAS C API. In particular, we are indebted to Jeremy Kepner and John Gilbert for developing the concept of graph algorithms in the language of linear algebra.

REFERENCES

Fig. 3: C function using GraphBLAS primitives that computes the BC-metric updates $delta$, given Boolean $n \times n$ adjacency matrix $A$, a set of source vertices $s$, and the number of source vertices (i.e., the length of $s$) $nsver$.

```c
#include "GraphBLAS.h" // in addition to other required C headers
GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver) // Compute BC metric
{
  GrB_Index n;
  GrB_Matrix nrows(&n, A); // n = # of vertices in graph
  GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
  GrB_Monoid Int32Add;
  GrB_Monoid_new(&Int32Add, GrB_INT32, GrB_PLUS_INT32, 0);
  GrB_Semiring Int32AddMul;
  GrB_Semiring_new(&Int32AddMul, Int32Add, GrB_TIMES_INT32);
  GrB_Descriptor desc_tsr;
  GrB_Descriptor_new(&desc_tsr);
  GrB_Descriptor_set(desc_tsr, GrB_INPO, GrB_TRAN);
  GrB_Descriptor_set(desc_tsr, GrB_MASK, GrB_SCMP);
  // structural complement of the mask
  GrB_Descriptor_set(desc_tsr, GrB_OUTP, GrB_REPLACE);
  // clear output before result is stored in it.
  GrB_Index i, nsver = malloc(sizeof(GrB_Index) * nsver); // index and value arrays needed to build numsp
  int32_t ones = malloc(sizeof(int32_t) * nsver);
  for (int i = 0; i < nsver; ++i) // i
  i nsver[i] = 1;
  ones[i] = 1;
}
GrB_Matrix numsp; // Its nonzero structure holds all vertices that have
GrB_Matrix_new(&numsp, GrB_INT32, n, nsver); // been discovered and stores number of shortest paths so far
GrB_Matrix build(&numsp, GrB_NULL, GrB_NULL, s, i nsver, ones, nsver, GrB_PLUS_INT32, GrB_NULL); // numsp[s[i], i]=1, i=0, nsver
free(i nsver); free(ones);
GrB_Matrix frontier; // Holds the current frontier where values are path counts.
GrB_Matrix_new(&frontier, GrB_INT32, n, nsver); // Initialized to out vertices of each source node in s.
GrB_Extract(&frontier, numsp, GrB_NULL, A, GrB_ALL, n, nsver, desc_tsr);
//
// The memory for an entry in sigmas is only allocated within the do–while loop if needed
GrB_Matrix *sigmas = malloc(sizeof(GrB_Matrix) * n); // n is an upper bound on diameter
int32_t d = 0; // BFS level number
int32_t nvals = 0; // nvals == 0 when BFS phase is complete
do {
  // The BFS phase (forward sweep)
  GrB_Matrix new(&sigmas[d], GrB_BOOL, n, nsver); // sigmas[d][s] = d’th level frontier from source vertex s
  GrB_Monoid_FP32Add;
  GrB_Monoid_new(&FP32Add, GrB_FP32, GrB_PLUS_FP32, 0); // FP32
  GrB_Monoid_FP32Mul;
  GrB_Semiring_FP32AddMul;
  GrB_Semiring_new(&FP32AddMul, FP32Add, GrB_TIMES_FP32);
  GrB_Matrix nspinv; // inverse of the number of shortest paths
  GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
  GrB_Extract(&nspinv, GrB_NULL, GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1/numsp
  GrB_Matrix bcu; // BC updates for each starting vertex in s
  GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
  GrB_Assign(&bcu, GrB_NULL, GrB_NULL, 0.0, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 0 to avoid sparsity issues
  GrB_Descriptor desc_r; // Descriptor for 1st evisesnalt in tally
  GrB_Descriptor_set(desc_r, GrB_OUTP, GrB_REPLACE);
  // clear output before result is stored in it.
  GrB_Matrix w; // temporary workspace matrix
  GrB_Matrix_new(&w, GrB_FP32, n, nsver);
  for (int i = 1; i d; i++) // tally phase (backward sweep)
  GrB_eWiseMult(&w, sigmas[i], GrB_NULL, FP32Mul, bcu, nspinv, desc_r); // w[i]=1 / nspinv
  // add contributions by successors and mask with that BFS level’s frontier
  GrB_mxm(&w, sigmas[i], GrB_NULL, FP32AddMul, w, desc_r); // w[i]=A + w
  GrB_eWiseMult(&w, bcu, GrB_NULL, GrB_PLUS_FP32, FP32Mul, w, numsp, GrB_NULL); // bcu += w * numsp
}
// subtract “nsver” from every entry in delta (1 extra value per bcu element crept in)
GrB_Assign(delta, GrB_NULL, GrB_NULL, (float) nsver, GrB_ALL, n, GrB_NULL); // fill with –nsver
GrB_Reduce(delta, GrB_NULL, GrB_PLUS_FP32, FP32Add, bcu, GrB_NULL); // add all updates to –nsver
for (int i = 0; i d; i++) // macro that expands GrB_free() for each parameter
GrB_free_all(frontier, numsp, w, bcu, desc_tsr, desc_r); //}
return GrB_SUCCESS;
```