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 $\mathbf{A}$, 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.

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 $\mathbf{0}$ (not necessarily the number 0) while the *multiplication* operator, $\otimes$, is a commutative monoid with an identity element $\mathbf{1}$ (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 used with graph algorithms.

Semiring	operators $\oplus$	operators $\otimes$	domain	$\mathbf{0}$	$\mathbf{1}$
Standard arithmetic	$+$	$\times$	$\mathbb{R}$	0	1
max-plus algebras	max	$+$	$\{-\infty \cup \mathbb{R}\}$	$-\infty$	0
min-max algebras	min	max	$\infty \cup \mathbb{R}_{\geq 0}$	∞	0
Galois fields (*e.g.*, GF2)	xor	and	$\{0, 1\}$	0	1
Power set algebras	$\cup$	$\cap$	$\mathcal{P}(\mathbb{Z})$	$\varnothing$	U

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 $\mathbb{S}$ with semiring operators $\oplus$ and $\otimes$:

$$\mathbf{C} = \mathbf{A} \oplus . \otimes \mathbf{B} = \mathbf{AB}.$$

Using index notation familiar in linear algebra

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

for matrices with dimensions

$$\mathbf{A} : \mathbb{S}^{m \times l} \qquad \mathbf{B} : \mathbb{S}^{l \times n} \qquad \mathbf{C} : \mathbb{S}^{m \times n}$$

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

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

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

In other words, the binary operation $\otimes$ is applied to the elements in the intersection of the two sets $\mathbf{ind}(\mathbf{A}(i,:))$ and $\mathbf{ind}(\mathbf{B}(:,j))$, 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.

TABLE II: A mathematical overview of the fundamental GraphBLAS operations supported in this 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	$\mathbf{C}(\mathbf{i},\mathbf{j}) \odot\!= \mathbf{A}$
	$\mathbf{w}(\mathbf{i}) \odot\!= \mathbf{u}$

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 $\mathbf{v} = \langle D, N, \{(i, v_i)\}\rangle$ is defined by

- The domain D, the data type for the vector elements.
- The size $N > 0$.

- 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 $\mathbf{v}$. We define $\mathbf{nelem}(\mathbf{v}) = N$ and $\mathbf{L}(\mathbf{v}) = \{(i, v_i)\}$. The set $\mathbf{L}(\mathbf{v})$ is called the *content* of vector $\mathbf{v}$. For a vector $\mathbf{v}$, $\mathbf{v}(i)$ is a reference to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$ and is undefined otherwise.

A GraphBLAS matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\}\rangle$ 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 $\mathbf{L}(\mathbf{A}) = (i, j, A_{ij})$ 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 $\mathbf{A}$. The set $\mathbf{L}(\mathbf{A})$ is called the *content* of matrix $\mathbf{A}$. We define $\mathbf{nrows}(\mathbf{A}) = M$ and $\mathbf{ncols}(\mathbf{A}) = N$. For a matrix $\mathbf{A}$, $\mathbf{A}(i, j)$ is a reference to A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$ 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 $\mathbf{A}$ is a matrix and $0 \leq j < N$, then

$$\mathbf{A}(:, j) = \langle D, M, \{(i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$$

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

$$\mathbf{A}(i, :) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$$

is a vector called the i-th *row* of $\mathbf{A}$.

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

$$\mathbf{A}^T = \langle D, N, M, \{(j, i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle.$$

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 a 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 *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

$$\odot : D_1 \times D_2 \to D_3.$$

A GraphBLAS *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 $f : D_1 \to D_2$. GraphBLAS makes use of two fundamental algebraic structures: monoids and semirings. A GraphBLAS *monoid*

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

is defined by a single domain D_1, an *associative*[1] operation $\odot : D_1 \times D_1 \to D_1$, 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 *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 *associative* additive operation

$$\oplus : D_3 \times D_3 \to D_3;$$

a multiplicative operation

$$\otimes : D_1 \times D_2 \to D_3;$$

and an element $\mathbf{0} \in D_3$, which is the identity for $\oplus$. Let $F = \langle D_1, D_2, D_3, \otimes\rangle$ 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 $F = \langle D_1, D_2, D_3, \otimes\rangle$.

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.

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

[1]It is expected that implementations will utilize IEEE-754 floating point arithmetic, which is not strictly associative.

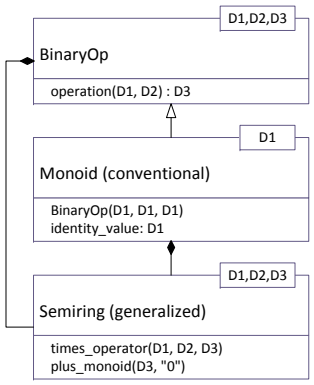

Fig. 1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

most once in $\mathbf{m}$. We define $\mathbf{nelem}(\mathbf{m}) = N$. We define the *structure* of a one-dimensional mask as the set $\mathbf{L}(\mathbf{m})$. A two-dimensional mask $\mathbf{M} = \langle M, N, \{(i,j)\}\rangle$ is defined by its number of rows $M > 0$, its number of columns $N > 0$, and a set $\mathbf{L}(\mathbf{M})$ of tuples (i,j) where $0 \le i < M$, $0 \le j < N$. A particular pair of values i, j can appear at most once in $\mathbf{M}$. The *structure* of a two-dimensional mask $\mathbf{M}$ is the set $\mathbf{L}(\mathbf{M})$. We also define $\mathbf{nrows}(\mathbf{M}) = M$ and $\mathbf{ncols}(\mathbf{M}) = N$.

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

$$\mathbf{L}(\neg\mathbf{m}) = \{i : 0 \le i < N, i \notin \mathbf{L}(\mathbf{m})\},$$

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

$$\mathbf{L}(\neg\mathbf{M}) = \{(i,j) : 0 \le i < M, 0 \le j < N, (i,j) \notin \mathbf{L}(\mathbf{M})\},$$

the set of all possible indices that do not appear in $\mathbf{M}$.

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 GrB_OUTP; the mask by the field name GrB_MASK; and the input vectors and matrices by GrB_INP0 and 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 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 mode 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.

TABLE III: GraphBLAS data types.

Data type	Description
GrB_Info	Return value from any GraphBLAS method
GrB_Index	Vector and matrix indices
GrB_Type	Type identifier
GrB_Descriptor	Opaque GraphBLAS descriptor object
GrB_Monoid	Opaque GraphBLAS monoid object
GrB_Semiring	Opaque GraphBLAS semiring object
GrB_Matrix	Opaque GraphBLAS matrix object
GrB_Vector	Opaque GraphBLAS vector object

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

TABLE IV: Some predefined GraphBLAS operators.

Operator	Description
GrB_TIMES_INT32	Binary operation, returns product of two 32-bit integer values
GrB_PLUS_INT32	Binary operation, returns sum of two 32-bit integer values
GrB_PLUS_FP32	Binary operation, returns sum of two 32-bit floating-point values
GrB_TIMES_FP32	Binary operation, returns product of two 32-bit floating-point values
GrB_MINV_FP32	Unary operation, returns multiplicative inverse of the input 32-bit floating-point value
GrB_IDENTITY_BOOL	Unary operation, returns input boolean value

TABLE V: GraphBLAS literals used in Section VII.

Literal	Description
GrB_OUTP	Descriptor field for the output argument.
GrB_MASK	Descriptor field for the mask.
GrB_INP0	Descriptor field for the first input argument.
GrB_INP1	Descriptor field for the second input argument.
GrB_SCMP	Descriptor value to indicate use of the structural complement of the mask.
GrB_TRAN	Descriptor value to indicate use of the transpose of the corresponding input matrix.
GrB_REPLACE	Descriptor value to indicate that the output object should be replaced by the result of the method.
GrB_ALL	All of an object's indices in order.
GrB_NULL	*Null* value used to indicate when a parameter is not provided and a default behavior should be used.
GrB_SUCCESS	Return value indicating that a method has returned without encountering an error condition.
GrB_BOOL	Identifier for boolean type.
GrB_INT32	Identifier for 32-bit integer type.
GrB_FP32	Identifier for 32-bit floating point type.

Figure 2. GrB_mxm() takes three input matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$; computes a matrix product of $\mathbf{A}$ and $\mathbf{B}$; and either copies the result into the matrix $\mathbf{C}$ or accumulates the product into the matrix $\mathbf{C}$. Based on the arguments and the descriptor, the semantics of GrB_mxm() can vary considerably. The function of the descriptor is consistent across all methods, hence its use for GrB_mxm() is applicable to all methods.

The semantics of GraphBLAS methods associated with the operations from Table II follow a similar pattern:

1) The internal matrices and mask used in the computation are formed from the input parameters. Their domains and dimensions are tested for consistency.
2) The indicated computations are carried out using the internal matrices and producing an internal result.
3) The internal result is written into the output matrix, possibly under control of a mask.

In the case of the GrB_mxm operation, internal matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and mask $\mathbf{Mask}$ are formed from the corresponding arguments according to the descriptor. Depending on the values in the descriptor fields GrB_INP0 and GrB_INP1, $\mathbf{A}$ and/or $\mathbf{B}$ may be the transpose of the corresponding argument. The descriptor field GrB_MASK may also indicate that the structural complement of the mask should be used. If the domains and sizes of the objects are mathematically consistent, the indicated operation is carried out. This produces an internal matrix $\mathbf{T}$ equal to the product of matrices $\mathbf{A}$ and $\mathbf{B}$. (We emphasize that an implementation of the GraphBLAS is not required to materialize the matrix $\mathbf{T}$).

If an optional binary accumulator function accum is provided, it is used to combine the elements of matrix $\mathbf{C}$ and the internal matrix $\mathbf{T}$. This forms a new internal matrix $\mathbf{Z}$.

At this point the elements of $\mathbf{Mask}$ are used as a *write mask* to select which elements of $\mathbf{Z}$ are used to form the final output result. Basically, the elements of the boolean write mask that exist and are true correspond to the elements of the output matrix that might be replaced by the corresponding elements of $\mathbf{Z}$. Two options are supported:

- *Replace mode*: If the descriptor field GrB_OUTP is set to GrB_REPLACE, the values in the $\mathbf{C}$ matrix are deleted before masked elements of $\mathbf{Z}$ are stored in $\mathbf{C}$. In essence the computed matrix $\mathbf{Z}$ *replaces* the original matrix $\mathbf{C}$.
- *Merge mode*: Otherwise, the elements from the computation selected by the write mask are written into the output $\mathbf{C}$ 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.

TABLE VI: Methods used in the example in section VII.

Method Name	Description
GrB_Monoid_new	Creates a new monoid with specified domain, operator, and identity element.
GrB_Semiring_new	Creates a new semiring with specified domain, monoid, and operators.
GrB_Vector_new	Creates a new vector with specified domain and size.
GrB_Matrix_new	Creates a new matrix with specified domain and dimensions.
GrB_Matrix_nrows	Retrieves the number of rows in a matrix.
GrB_Matrix_nvals	Retrieves the number of stored elements (tuples) in a matrix.
GrB_Descriptor_new	Creates a new (empty) descriptor.
GrB_Descriptor_set	Sets the content (details of an operation) for a field of an existing descriptor.
GrB_Matrix_build	Copies elements from tuples into a matrix.
GrB_mxm	Performs matrix multiplication over a semiring.
GrB_eWiseMult	Performs an element-wise multiplication on the elements of two matrices.
GrB_eWiseAdd	Performs an element-wise addition on the elements of two matrices.
GrB_extract	Extracts a subgraph from an input matrix and copies them into an output matrix.
GrB_assign	Assigns an input scalar value to each element of a specified subgraph.
GrB_apply	Applies a unary operator to matrix elements.
GrB_reduce	Reduces across matrix rows into a vector.

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 v is the normalized ratio

Fig. 2: The GrB_mxm() function signature, parameters, and return values.

a) Signature:

```
GrB_Info GrB_mxm(GrB_Matrix              *C,
                 const GrB_Matrix         Mask,
                 const GrB_BinaryOp       accum,
                 const GrB_Semiring       op,
                 const GrB_Matrix         A,
                 const GrB_Matrix         B,
                 const GrB_Descriptor     desc);
```

b) Parameters:

C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the matrix product. On output, the matrix holds the results of this operation.

Mask (IN) A "write" mask that controls which results from this operation are stored into the output matrix C (optional). If no mask is desired, GrB_NULL is specified. The mask dimensions must match those of the matrix C, and the domain of the Mask matrix must be of type bool or any "built-in" GraphBLAS type.

accum (IN) A binary operator for accumulating entries with an existing C entries. Use GrB_NULL for assignment rather than accumulation.

op (IN) Semiring used in the matrix-matrix multiply: op $= \langle D_1, D_2, D_3, \oplus, \otimes, 0 \rangle$.

A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the multiplication.

B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the multiplication.

desc (IN) Operation descriptor (optional). If a *default* descriptor is desired, GrB_NULL should be used. Valid fields are as follows:

Argument	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared before result is stored.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for operation.

c) Return Values:

GrB_SUCCESS	Blocking mode: operation completed successfully. Nonblocking mode: input argument consistency tests passed.
GrB_PANIC	Unknown internal error.
GrB_INVALID_OBJECT	At least one of the argument objects is in an invalid state – caused by a previous execution error.
GrB_OUT_OF_MEMORY	Not enough memory available. for operation.
GrB_UNINITIALIZED_OBJECT	One or more of the GraphBLAS objects has not been initialized.
GrB_NULL_POINTER	The output matrix pointer is NULL.
GrB_DIMENSION_MISMATCH	Matrix dimensions are incompatible.
GrB_DOMAIN_MISMATCH	The domains of the matrices are incompatible with the accumulator, semiring, or mask domains.

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 σ_{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:

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

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):

`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 nsver simultaneous BFS traversals. In this implementation it uses 32-bit integer (GrB_INT32 domain) arithmetic and relies on GraphBLAS's predefined binary operators, GrB_PLUS_INT32 and GrB_TIMES_INT32, to declare an addition monoid (Int32Add on line 9) and the arithmetic semiring (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 numsp matrix keep track of the number of independent shortest paths that reach every other vertex from the corresponding source vertices. The dimensions are n×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,

$$\texttt{numsp}(\texttt{s}_i, i) = 1, \text{ for } 0 \leq i < \texttt{nsver}. \tag{2}$$

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

The columns of the 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 n×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 extract operation on the graph matrix A. The descriptor transposes A, such that the use of the s array specified for the column indices selects each row of A corresponding to the source vertices. The use of GrB_ALL specified for all n row indices ensures that all out neighbors of the selected source vertices are included. The 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 frontier matrix is already empty, the descriptor's 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 sigmas. A set of 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 n×nsver sigmas[d] matrix (line 40) and sets it to the current frontier (line 41). The apply operation is used for the assignment and by using the unary operator, 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 eWiseAdd operation to perform an element-wise addition of the numsp and frontier matrices. The result is stored in numsp.

The 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, 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 numsp matrix as the mask parameter. The implicit cast of numsp to Boolean allows GrB_mxm to interpret numsp as the set of previously discovered vertices. Note that the descriptor is also set to 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, GrB_Matrix_nvals. If the result (stored in 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 numsp is computed using the apply operation along with the predefined multiplicative inverse unary function defined for 32-bit floating point, GrB_MINV_FP32. Then, the 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 assign operation that allows the same value to be assigned to a subgraph, and that specifies GrB_ALL for both row and column indices. Last, a descriptor object, 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 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 assures 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

Aydın Buluç 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

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

[8] 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.

[9] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

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

[11] E. Robinson, "Complex graph algorithms," *Graph Algorithms in the Language of Linear Algebra*, vol. 22, p. 59, 2011.

[12] 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.

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

[14] 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," *ArXiv e-prints*, Jan. 2017.

[15] B. Cook, "GraphBLAS c99 library," https://github.com/bobcgausa/GraphBLAS.

[16] A. Azad and A. Buluç, "Towards a GraphBLAS library in Chapel," in *Intl. Parallel & Distributed Processing Symposium Workshop (IPDPSW*, 2017.

[17] 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.

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) $nsver$.

```
#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;                                      // Monoid <int32_t,+,0>
  GrB_Monoid_new(&Int32Add,GrB_INT32,GrB_PLUS_INT32,0);
  GrB_Semiring Int32AddMul;                                 // Semiring <int32_t,int32_t,int32_t,+,*,0>
  GrB_Semiring_new(&Int32AddMul,Int32Add,GrB_TIMES_INT32);

  GrB_Descriptor desc_tsr;                                  // Descriptor for BFS phase mxm
  GrB_Descriptor_new(&desc_tsr);
  GrB_Descriptor_set(desc_tsr,GrB_INP0,GrB_TRAN);           // transpose of the adjacency matrix
  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_nsver[i] = i;
    ones[i] = 1;
  }
  GrB_Matrix numsp;                                         // Its nonzero structure holds all vertices that have
  GrB_Matrix_new( , GrB_INT32, n, nsver);              // been discovered and stores number of shortest paths so far.
  GrB_Matrix_build( ,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,s,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_apply(&(sigmas[d]),GrB_NULL,GrB_NULL,GrB_IDENTITY_BOOL,frontier,GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
    GrB_eWiseAdd( ,GrB_NULL,GrB_NULL,Int32Add,numsp,frontier,GrB_NULL);       // numsp += frontier (accum path counts)
    GrB_mxm(&frontier,numsp,GrB_NULL,Int32AddMul,A,frontier,desc_tsr);             // f<!numsp> = A' +.* f (update frontier)
    GrB_Matrix_nvals(&nvals,frontier);                      // number of nodes in frontier at this level
    d++;
  } while (nvals);

  GrB_Monoid FP32Add;                                       // Monoid <float,+,0.0>
  GrB_Monoid_new(&FP32Add,GrB_FP32,GrB_PLUS_FP32,0.0f);
  GrB_Monoid FP32Mul;                                       // Monoid <float,*,1.0>
  GrB_Monoid_new(&FP32Mul,GrB_FP32,GrB_TIMES_FP32,1.0f);
  GrB_Semiring FP32AddMul;                                  // Semiring <float,float,float,+,*,0.0>
  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_apply(&nspinv,GrB_NULL,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,1.0f,GrB_ALL,n, GrB_ALL,nsver,GrB_NULL); // filled with 1 to avoid sparsity issues

  GrB_Descriptor desc_r;                                    // Descriptor for 1st ewisemult in tally
  GrB_Descriptor_new(&desc_r);
  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=d-1; i>0; i--)  { // ---------------------- Tally phase (backward sweep) ----------------------
      GrB_eWiseMult(&w,sigmas[i],GrB_NULL,FP32Mul,bcu,nspinv,desc_r);         // w<sigmas[i]>=(1 ./ nsp).*bcu

      // add contributions by successors and mask with that BFS level's frontier
      GrB_mxm(&w,sigmas[i-1],GrB_NULL,FP32AddMul,A,w,desc_r);                 // w<sigmas[i-1]> = (A +.* w)
      GrB_eWiseMult(&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,GrB_PLUS_FP32,bcu,GrB_NULL);        // add all updates to -nsver

  for(int i=0; i<d; i++) { GrB_free(sigmas[i]); } free(sigmas);
  GrB_free_all(frontier,numsp,nspinv,w,bcu,desc_tsr,desc_r);    // macro that expands GrB_free() for each parameter
  GrB_free_all(Int32AddMul,Int32Add,FP32AddMul,FP32Add,FP32Mul);
  return GrB_SUCCESS;
}
```