Design of the GraphBLAS API for C

Aydın Buluç, Tim Mattson, Scott McMillan, José Moreira, Carl Yang

***Abstract*—The GraphBLAS effort aims to standardize linear-algebraic building blocks for graph computations. A time consuming part of this standardization effort is to translate the mathematical specification to an actual Application Programming Interface (API) that (i) is faithful to the mathematics as much as possible and (ii) enables efficient implementations on modern hardware. This paper documents the efforts taken by the C language specification subcommittee and presents the main concepts, constructs, and objects within the GraphBLAS API.**

I. Introduction

Graphs and matrices are fundamental abstractions in computer science and applied mathematics, respectively. Graphs have been used to represent matrices, and especially sparse matrices, where they are often useful to conceptualize dependencies between rows or columns. Consequently graphs became a popular abstraction in sparse matrix research [1]. Conversely and more recently, matrices have started to pay back their dues and helped increase the performance of graph algorithms due to matrices being a better match for computer architectures. Many graph algorithms have been mapped to the language of linear algebra over the years [2].

High-performance systems and libraries that allow efficient implementation of graph algorithms have been built in recent years [3], [4], [5], [6]. This had led to a concern among the community that a fragmentation of concepts and abstractions might occur. The mapping of graph algorithms into the language of matrices and vectors were sufficiently well understood that the community decided to establish an effort to standardize the fundamental operations [7]. This has led to the formation of the GraphBLAS Forum [8], a loosely coupled group of researchers and practitioners from academia, industry and federally-funded research organizations. The mathematical foundations of the GraphBLAS released first, which is summarized in a recent paper [9].

A subcommittee from the general GraphBLAS forum took the task to map the mathematics to an actual programming language. The authors of this paper form that subcommittee. We had to define the concepts, the objects, and function signatures. We had to balance multiple and often conflicting objectives: (i) simplicity and ease of use, (ii) enabling high-performance implementations, and (iii) adherence to the underlying mathematics.

This paper provides a high-level summary of the GraphBLAS application programming interface (API) specification for the C language. It also explains the rationale behind many design choices and provides an accessible introduction to the specification contents.

II. GraphBLAS Math

Graphs permit a dual representation as an collection of vertices/edges or as a matrix. The matrix representation of a graph can take different forms. A common one uses an *Adjacency matrix*; for which the rows and columns designate the vertices of a graph and nonzero $(i, j)^{th}$ matrix elements signify an edge between vertices i and j. The degree of vertices in a graph are almost always a small compared to the number of vertices; hence, the matrices used to represent graphs are sparse.

Multiplying an Adjacency matrix times a second matrix is equivalent to a breath first search from multiple starting locations. Whole classes of algorithms based on this basic pattern can be constructed from this core operation. We can extend the range of graph algorithms by keeping the basic memory access pattern of a matrix-matrix multiplication, but varying the operations used in the operation. By carefully choosing operations that support the algebraic properties of commutativity and associativity, familiar algebraic relations between matrix objects are retained thereby enabling composable graph algorithms.

We do this using the concept of an algebraic semiring. The most common semirings used in the Graph Algorithms community are shown in table I.

TABLE I: Common Semirings used with Graph Algorithyms.

Semiring	operators		Domain
Standard Arithmetic	$\oplus \equiv +$	$\otimes \equiv \times$	$\mathbb{R}$
max-plus Algebras	$\oplus \equiv max$	$\otimes \equiv +$	$\{-\infty \cup \mathbb{R}\}$
max-min Algebras	$\oplus \equiv max$	$\otimes \equiv \times$	$\infty \cup \mathbb{R}_{\leq 0}\}$
Finite (Galois) Fields (e.g. GF2)	$\oplus \equiv xor$	$\otimes \equiv and$	$\{0, 1\}$
Power set Algebras	$\oplus \equiv \cup$	$\otimes \equiv \cap$	$\mathbb{Z}$

In the C-binding to the GraphBLAS, this means we define a separate object for the semiring that is passed into functions. Since in many cases the full semiring is not required, we also support passing monoids or even operators; which basically means the semiring is implied but not explicitly stated.

In addition to matrix multiplication, a range of operations are included in the GraphBLAS. These are summarized in table II.

III. Basic Concepts

The GraphBLAS C API is built on a collection of objects exposed to the C programmer as opaque data types. These objects include

- *collections*: vectors and matrices.
- *algebraic objects*: operators, monoids, and semirings.
- *control objects*: descriptors and masks (both vector and matrix).

TABLE II: A Mathematical overview of the fundamental GraphBLAS operations supported in this specification. $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are GraphBLAS matrices, $\mathbf{u}$ and $\mathbf{v}$ are GraphBLAS vectors, i and j are indices, and v is a scalar indicating the value of an element of a GraphBLAS object. $f()$ is a function and m and n are integers indicating the size of GraphBLAS object dimensions. In most cases, 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	$\mathbf{C}$	$\oplus=$	$\mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{u}$	$\oplus=$	$\mathbf{A} \oplus . \otimes \mathbf{v}$
vxm	$\mathbf{u}$	$\oplus=$	$\mathbf{v} \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C}$	$\oplus=$	$\mathbf{A} \otimes \mathbf{B}$
eWiseAdd	$\mathbf{C}$	$\oplus=$	$\mathbf{A} \oplus \mathbf{B}$
reduce (row)	$\mathbf{u}$	$\oplus=$	$\oplus_j \mathbf{A}(:,j)$
apply	$\mathbf{C}$	$\oplus=$	$f(\mathbf{A})$
transpose	$\mathbf{C}$	$\oplus=$	$\mathbf{A}$
extract	$\mathbf{C}$	$\oplus=$	$\mathbf{A}(\mathbf{i},\mathbf{j})$
assign	$\mathbf{C}(\mathbf{i},\mathbf{j})$	$\oplus=$	$\mathbf{A}$
buildMatrix	$\mathbf{C}$	$\oplus=$	$\mathbb{S}^{m \times n}(\mathbf{i},\mathbf{j},\mathbf{v},\oplus_{dup})$
buildVector	$\mathbf{u}$	$\oplus=$	$\mathbb{S}^{n}(\mathbf{i},\mathbf{v})$
extractTuples	$(\mathbf{i},\mathbf{j},\mathbf{v})$	$=$	$\mathbf{A}$

Functions that manipulate these objects are referred to as *methods*. These methods fully define the interface to GraphBLAS objects to create or destroy them, modify their contents, and copy the contents of opaque objects into non-opaque or *transparent* objects the contents of which are under direct control of the programmer.

In this section, we introduce the basic objects. Then we consider the GraphBLAS C API execution and error models.

A. Vectors

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

- a domain D
- a 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 only appear at most 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 $\mathbf{v}$. We also define the set:

$$\mathbf{ind}(\mathbf{v}) = \{i : (i, v_i) \in \mathbf{L}(\mathbf{v})\}$$

(called the *structure* of $\mathbf{v}$), and $\mathbf{D}(\mathbf{v}) = D$. 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.

B. Matrices

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

- a domain D,
- a number of rows $M > 0$ and columns $N > 0$
- a set of tuples (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 only appear at most once in $\mathbf{A}$. We define $\mathbf{ncols}(\mathbf{A}) = N$, $\mathbf{nrows}(\mathbf{A}) = M$ and $\mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\}$. The set $\mathbf{L}(\mathbf{A})$ is called the *content* of matrix $\mathbf{A}$. We also define the sets:

$$\mathbf{indrow}(\mathbf{A}) = \{i : \exists(i, j, A_{ij}) \in \mathbf{A}\}$$

$$\mathbf{indcol}(\mathbf{A}) = \{j : \exists(i, j, A_{ij}) \in \mathbf{A}\}$$

These are the sets of nonempty rows and columns of $\mathbf{A}$, respectively.) The *structure* of matrix $\mathbf{A}$ is the set

$$\mathbf{ind}(\mathbf{A}) = \{(i, j) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}$$

and $\mathbf{D}(\mathbf{A}) = D$. 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.

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 $\mathbf{A}$.

Given a matrix

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

its *transpose* is another matrix

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

C. Operators

A GraphBLAS *binary operators*

$$F_b = \langle D_1, D_2, D_3, \odot\rangle$$

is defined by three domains, D_1, D_2, D_3, and an operation

$$\odot : D_1 \times D_2 \rightarrow D_3$$

For a given GraphBLAS operators

$$F_b = \langle D_1, D_2, D_3, \odot\rangle$$

we define $\mathbf{D}_1(F_b) = D_1$, $\mathbf{D}_2(F_b) = D_2$, $\mathbf{D}_3(F_b) = D_3$, and $\bigodot(F_b) = \odot$. Note that $\odot$ could be used in place of either $\oplus$ or $\otimes$.

A GraphBLAS *unary operators*

$$F_u = \langle D_1, D_2, f\rangle$$

is defined by two domains, D_1, D_2, and an operation $f : D_1 \rightarrow D_2$. For a given GraphBLAS operators

$$F_u = \langle D_1, D_2, f\rangle$$

we define $\mathbf{D}_1(F_u) = D_1$, $\mathbf{D}_2(F_u) = D_2$, and $\mathbf{f}(F) = f$.

D. Monoids

A GraphBLAS *generalized monoid* (or *monoid* for short)

$$M = \langle D_1, \odot, 0\rangle$$

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 $0 \in D_1$. For a given GraphBLAS monoid

$$M = \langle D_1, \odot, 0\rangle$$

we define $\mathbf{D}_1(M) = D_1$, $\bigodot(M) = \odot$ and $\mathbf{0}(M) = 0$. A GraphBLAS monoid is equivalent to the conventional *monoid* algebraic structure.

Let

$$F = \langle D_1, D_1, D_1, \odot\rangle$$

be a GraphBLAS binary operator with element $0 \in D_1$. Then

$$M = \langle F, 0\rangle = \langle D_1, \odot, 0\rangle$$

is a GraphBLAS monoid.

E. Semirings

A GraphBLAS *semiring* (or *semiring* for short)

$$S = \langle D_1, D_2, D_3, \oplus, \otimes, 0\rangle$$

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 $0 \in D_3$ For a given GraphBLAS semiring

$$S = \langle D_1, D_2, D_3, \oplus, \otimes, 0\rangle$$

we define $\mathbf{D}_1(S) = D_1$, $\mathbf{D}_2(S) = D_2$, $\mathbf{D}_3(S) = D_3$, $\bigoplus(S) = \oplus$, $\bigotimes(S) = \otimes$, and $\mathbf{0}(S) = 0$.

Let $F = \langle D_1, D_2, D_3, \otimes\rangle$ be a operator and let $A = \langle D_3, \oplus, 0\rangle$ be a monoid, then

$$S = \langle A, F\rangle = \langle D_1, D_2, D_3, \oplus, \otimes, 0\rangle$$

is a semiring.

Note: There must be one GraphBLAS monoid in every semiring which serves as the semiring's additive operator and specifies the same domain for its inputs and output parameters.

A UML diagram of the conceptual hierarchy of object classes in GraphBLAS algebra (binary operators, monoids and semirings) is shown in Figure 1.

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

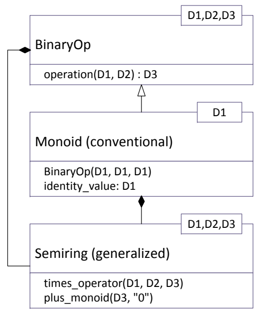

Fig. 1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the 'add' function, and a binary operator with three domains for the 'multiply' function.

F. Masks

A mask can be either a one- or a two-dimensional construct. One- and two-dimensional masks, described more formally below, are similar to vectors and matrices, respectively, except that they have structure (indices) but no values. Masks are used to control which values from a 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 $0 \le i < N$. A particular value of i can only appear at most once in $\mathbf{m}$. We define $\mathbf{n}(\mathbf{m}) = N$. We also define the set

$$\mathbf{ind}(\mathbf{m}) = \{i : i \in \mathbf{L}(\mathbf{m})\}$$

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 only appear at most once in $\mathbf{M}$. We define $\mathbf{ncols}(\mathbf{M}) = N$, and $\mathbf{nrows}(\mathbf{M}) = M$. The *structure* of a two-dimensional mask $\mathbf{M}$ is the set:

$$\mathbf{ind}(\mathbf{M}) = \{(i,j) : (i,j) \in \mathbf{L}(\mathbf{M})\}$$

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 masks this is denoted as $\neg\mathbf{M}$. The structure of the complement of an one-dimensional mask $\mathbf{m}$ is defined as:

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

It 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:

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

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

1) Descriptors: Descriptors, the last argument in all GraphBLAS methods, are used to control optional behaviors of operations. In particular, descriptors specify how the other input arguments that correspond to collections – vectors, matrices and masks – should be processed (modified) before the main operation of a method is performed.

The descriptor is a lightweight object. It pairs a set of flags representing the possible modifiers with each collection argument of the GraphBLAS method. For example, a descriptor may specify that a particular input matrix needs to be transposed or that a mask needs to be structurally complemented (defined in Section III-F) before using it in the operation.

For the purpose of constructing descriptors, the arguments of a method that can be modified are identified by specific field names. The output parameter (typically the first parameter in a GraphBLAS method) is indicated by the field name, GrB_OUTP. The mask is indicated GrB_MASK field name. The input parameters corresponding to the input vectors and matrices are indicated by GrB_INP0 and GrB_INP1, in the order they appear in the signature of the GraphBLAS method.

G. Execution Model

A program using the GraphBLAS C API constructs GraphBLAS objects, manipulates them to implement a graph algorithm, and then extracts values from the GraphBLAS objects as the result of the algorithm. Functions defined within the GraphBLAS C API that manipulate GraphBLAS objects are called *methods*.

Graph algorithms are expressed as an ordered collection of GraphBLAS method calls defined by the order they are encountered in a program. This is called the *Program Order*. Each method in the collection uniquely and unambiguously defines the output GraphBLAS objects based on the GraphBLAS operation and the input GraphBLAS objects.

We define a sequence of GraphBLAS method calls, or when the meaning is clear a *sequence*, as a well defined ordered collection of GraphBLAS method calls. The initiation of a sequence is the first method call that modifies a GraphBLAS object, The end of the sequence is either (1) the first GraphBLAS method that reads values from a GraphBLAS object into a non-opaque data structure or (2) a GraphBLAS wait method. We collectively refer to these methods as *terminating methods*. The set of operations between the initiation of a sequence and its termination mathematically define the result of that sequence.

There are two modes A GraphBLAS program can use for execution: *blocking* and *nonblocking*.

- *blocking*: In blocking mode, each method in a sequence completes the GraphBLAS operation defined by the method before proceeding to the next statement in program order. Output GraphBLAS objects defined by a method are stored in memory and are available to other C functions after each method returns.
- *nonblocking*: In nonblocking mode, each method may return once the input arguments have been inspected and verified to define a well formed GraphBLAS operation. The GraphBLAS operation and the state of any GraphBLAS objects are undefined when a method returns until the terminating method in the sequence returns.

An application executing in nonblocking mode is not required to return immediately after input arguments have been verified; in essence a conforming implementation of the GraphBLAS C API running in nonblocking mode may choose to execute "as if" in blocking mode. Further, a sequence in nonblocking mode where every GraphBLAS operation is followed by an GrB_wait() call is equivalent to the same sequence in blocking mode with GrB_wait() calls removed.

Nonblocking mode allows for any execution strategy that satisfies the mathematical definition of the sequence. The methods can 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 or asynchronous execution are all valid as long as the final result agrees with the mathematical definition provided by the sequence of GraphBLAS method calls appearing in program order.

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. It is valuable for debugging or in cases where an external tool such as a debugger needs to evaluate the state of memory during a sequence.

In a mathematically well-defined sequence with input objects that are well-conditioned, the results from blocking and nonblocking modes should be identical outside of effects due to round-off errors associated with floating point arithmetic. Due to the great flexibility afforded to an implementation when using nonblocking mode, we expect execution of a sequence in nonblocking mode to potentially complete execution in less time.

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(). In the current version of the GraphBLAS C API, the context can only be set once in the execution of a program; i.e. After GrB_finalize() is called a following call to GrB_init() is not allowed.

H. Error Model

All GraphBLAS methods return a value of type GrB_info to provide information available to the system at the time the method returns. In blocking mode, that information pertains to the full computation and the return values defined for each method in the specification provide information concerning the condition of the computation.

In nonblocking mode, the information pertains to a consistency check of the arguments to the method. Any method that terminates a sequence must return information about the status of that sequence of method calls. A return value of GrB_SUCCESS indicates that the method returned correctly and that the sequence produced the result defined by the sequence of GraphBLAS operations. Other return values from the method indicate that an error was found during execution of the sequence. When possible, that return value will provide information concerning the cause of the error. Additional information is returned in the null terminated character string, err which is always the last argument to any method that may terminate a sequence.

Errors fall into two groups: API errors and execution errors. An API error means a GraphBLAS method was called with parameters that violate the rules for that method. API errors are deterministic and consistent across platforms and implementations. 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.

IV. GraphBLAS C API

Data types defined in the GraphBLAS C API

Generic function of graphBlas operations.

- Form objects from arguments, potentially using the descriptor
- carry out the indicated operation.
- Store results into the output object, potentially using a Mask

Describe matrix multiply in detail.

Then briefly define the operators used in our example.

- GrB_Vector_new(delta,GrB_FP32,n);
- GrB_Matrix_new(&sigma, GrB_INT32, n, n);
- GrB_Monoid_new(&Int32Add,GrB_INT32,GrB_PLUS_I32,0);
- GrB_Semiring_new(&Int32AddMul,Int32Add,GrB_TIMES_I32);
- GrB_Descriptor_new(&desc);
- GrB_Matrix_nrows(&n, A);
- GrB_assign(&q, 1, s);
- GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP);
- GrB_eWiseAdd(&p,GrB_NULL,GrB_NULL,Int32AddMul,p,q,GrB_NULL);
- GrB_eWiseMult(&t4,GrB_NULL,GrB_NULL,FP32Mul,t4,t3,GrB_NULL);
- GrB_reduce(&sum,GrB_NULL,Int32Add,q,GrB_NULL);
- GrB_mxv(&t3,GrB_NULL,GrB_NULL,FP32AddMul,A,t2,GrB_NULL);
- GrB_vxm(&q,p,GrB_NULL,Int32AddMul,q,A,desc)
- GrB_free(FP32AddMul);

V. 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 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 s to t, and $\sigma_{st}(v)$ is the number of such paths passing through vertex v.

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

BC is efficiently computed using Brandes' algorithm [10], which runs in $O(mn)$ time or unweighted graphs and avoids the expensive explicit all-pairs shortest paths computation. For each starting vertex s, Brandes' algorithm calls a subroutine that computes the BC contributions of to every other vertex. This subroutine, BC_update, is shown in Figure 2 using the C GraphBLAS API.

BC_update performs two sweeps over the graph. The forward sweep performs a breadth-first search where it also keeps track of the number of independent shortest paths that reach each vertex. This is performed by the do-while loop in lines 38-44. The backward sweep rolls back and tallies the BC contributions to each vertex. This is performed by the for loop in lines 61-71.

In the forward sweep, variable `p` keeps track of the number of independent shortest paths that reach each vertex, variable `q` contains the current frontier, and `sigma` stores the final breadth-first search tree. GrB_vxm call in line 40 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 before. This is achieved by setting the descriptor object `desc` to use the structural complement of the mask and by passing the variable `p` as the mask parameter. The implicit cast of `p` to Boolean allows GrB_vxm to interpret `p` as the set of previously discovered vertices. The GrB_reduce call in line 42 calculates the number of newly discovered vertices in that iteration and stores it in variable `sum`. The forward sweep is over when `sum` is zero.

VI. Results

We ran our BC code and it worked. Let's just show that it worked and that the performance doesn't totally suck.

VII. Conclusion

The C binding to the GraphBLAS standard is excellent and will have a huge impact on the development of Graph Algorithms.

Acknowledgments

This work is supported by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231.

References

[1] A. George, J. R. Gilbert, and J. W. Liu, *Graph theory and sparse matrix computation*. Springer Science & Business Media, 2012, vol. 56.

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

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

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

[5] 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 *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '16. New York, NY, USA: ACM, 2016, pp. 72–81.

[6] 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.

[7] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conference (HPEC '13)*. IEEE, 2013, (position paper).

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

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

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

Fig. 2: C function to compute the BC-metric vector, delta, given a boolean $n \times n$ adjacency matrix A and a source vertex s(which should be empty on input).

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "GraphBLAS.h"

GrB_info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_index s)
{
  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_Matrix sigma;                              // Matrix<int32_t> sigma(n,n)
  GrB_Matrix_new(&sigma, GrB_INT32, n, n);       // sigma[d,k] = shortest path count to node k at level d

  GrB_Vector q;
  GrB_Vector_new(&q, GrB_INT32, n);              // Vector<int32_t> q(n) of path counts
  GrB_assign(&q, 1, s);                          // q[s] = 1

  GrB_Vector p;
  GrB_Vector_new(&p, GrB_INT32, n);              // Vector<int32_t> p(n) shortest path counts so far
  GrB_assign(&p, q);                             // p = q

  GrB_Monoid Int32Add;                           // Monoid <int32_t,+,0>
  GrB_Monoid_new(&Int32Add,GrB_INT32,GrB_PLUS_I32,0);
  GrB_Semiring Int32AddMul;                      // Semiring <int32_t,int32_t,int32_t,+,*,0,1>
  GrB_Semiring_new(&Int32AddMul,Int32Add,GrB_TIMES_I32);

  GrB_Descriptor desc;                           // Descriptor for vxm
  GrB_Descriptor_new(&desc);
  GrB_Descriptor_set(desc,GrB_MASK,GrB_SCMP);    // structural complement of the mask

   // BFS phase

  int32_t d = 0;                                 // BFS level number
  int32_t sum = 0;                               // sum == 0 when BFS phase is complete
  do {
    GrB_assign(&sigma,GrB_NULL,GrB_NULL,q,d,GrB_ALL,n,GrB_NULL); // sigma[d,:] = q
    GrB_vxm(&q,p,GrB_NULL,Int32AddMul,q,A,desc);                 // q = # paths to nodes reachable from current level
    GrB_eWiseAdd(&p,GrB_NULL,GrB_NULL,Int32AddMul,p,q,GrB_NULL); // accumulate path counts on this level
    GrB_reduce(&sum,GrB_NULL,Int32Add,q,GrB_NULL);               // sum path counts at this level
    d++;
  } while (sum);

   // BC computation phase    .... (t1,t2,t3,t4) are temporary vectors

  GrB_Semiring FP32AddMul;        // Semiring <float,float,float,+,*,0.0,1.0>
  GrB_Semiring_new(&FP32AddMul,GrB_FP32,GrB_FP32,GrB_FP32,GrB_PLUS_F32,GrB_TIMES_F32,0.0,1.0);

  GrB_Monoid FP32Add;             // Monoid <float,float,float,+,0.0>
  GrB_Monoid_new(&FP32Add,GrB_FP32,GrB_FP32,GrB_FP32,GrB_PLUS_F32,0.0);

  GrB_Monoid FP32Mul;                            // Monoid <float,float,float,*,1.0>
  GrB_Monoid_new(&FP32Mul,GrB_FP32,GrB_FP32,GrB_FP32,GrB_TIMES_F32,1.0);

  GrB_Vector t1; GrB_Vector_new(&t1,GrB_FP32,n);
  GrB_Vector t2; GrB_Vector_new(&t2,GrB_FP32,n);
  GrB_Vector t3; GrB_Vector_new(&t3,GrB_FP32,n);
  GrB_Vector t4; GrB_Vector_new(&t4,GrB_FP32,n);
  for(int i=d-1; i>0; i--)
  {
    GrB_assign(&t1,GrB_NULL,GrB_NULL,1,GrB_ALL,GrB_NULL);                  // t1 = 1+delta
    GrB_eWiseAdd(&t1,GrB_NULL,GrB_NULL,FP32Add,t1,*delta,GrB_NULL);
    GrB_assign(&t2,GrB_NULL,GrB_NULL,sigma,i,GrB_ALL,n,GrB_NULL);          // t2 = sigma[i,:]
    GrB_eWiseMult(&t2,GrB_NULL,GrB_NULL,GrB_DIV_F32,t1,t2,GrB_NULL);       // t2 = (1+delta)/sigma[i,:]
    GrB_mxv(&t3,GrB_NULL,GrB_NULL,FP32AddMul,A,t2,GrB_NULL);               // add contributions made by successors of a node
    GrB_assign(&t4,GrB_NULL,GrB_NULL,sigma,i-1,GrB_ALL,n,GrB_NULL);        // t4 = sigma[i-1,:]
    GrB_eWiseMult(&t4,GrB_NULL,GrB_NULL,FP32Mul,t4,t3,GrB_NULL);           // t4 = sigma[i-1,:]*t3
    GrB_eWiseAdd(delta,GrB_NULL,GrB_NULL,FP32Add,*delta,t4,GrB_NULL);      // accumulate into delta
  }
  GrB_free(&sigma);
  GrB_free(q); GrB_free(p);
  GrB_free(Int32AddMul); GrB_free(Int32Add); GrB_free(FP32AddMul);
  GrB_free(FP32Add); GrB_free(FP32Mul);
  GrB_free(desc);
  GrB_free(t1); GrB_free(t2); GrB_free(t3); GrB_free(t4);

  return GrB_SUCCESS;
}
```