2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-3577-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPSW52791.2021.00047

Introduction to GraphBLAS 2.0

Benjamin Brock*, Aydın Buluç†*, Timothy G. Mattson‡, Scott McMillan§, José E. Moreira¶

* EECS Department, University of California, Berkeley, CA
† Computational Research Department, Lawrence Berkeley National Laboratory, Berkeley, CA
‡ Parallel Computing Labs, Intel, Ocean Park, WA
§ Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA
¶ IBM Thomas J. Watson Research Center, Yorktown Heights, NY

***Abstract*—The GraphBLAS is a set of basic building blocks for constructing graph algorithms in terms of linear algebra. They are first and foremost defined mathematically with the goal that language bindings will be produced for a wide range of programming languages. We started with the C programming language and over the last four years have produced multiple versions of the GraphBLAS C API specification. In this paper, we describe our next version of the C GraphBLAS specification. It introduces a number of major changes including support for multithreading, import/export functionality, and functions that use the indices of matrix/vector elements. Since some of these changes introduce small backwards compatibility issues, this is a major release we call *GraphBLAS 2.0*.**

I. Introduction

There are multiple ways to represent graphs. One exploits the connection between graphs and sparse matrices resulting in graph algorithms expressed as linear algebra [9]. The linear algebra community (in the 1970's and 1980's) developed a core set of building blocks for writing linear algebra algorithms called the Basic Linear Algebra Subprograms or the *BLAS*. We have built on this concept to define the BLAS of Graph algorithms or the *GraphBLAS*. The mathematical definition of the GraphBLAS was released in 2016 [8]. A year later we completed the binding of the GraphBLAS to the C programming language [5].

Shortly after the release of the GraphBLAS C specification, a number of implementations of the GraphBLAS were developed [7], [1], [6], [11], [15], [14], [12]. With feedback from the community of GraphBLAS implementors (especially the SuiteSparse [7] and GBTL [1] developers) and a group working on a library of high level algorithms that use the GraphBLAS (LAGraph [10]), we enhanced the specification through four releases culminating in the GraphBLAS 1.3 specification [4] in 2019. These were minor releases by which we mean the APIs did not violate backwards compatibility.

The next step in the growth of GraphBLAS requires more profound changes. These changes introduce a small number of backwards compatibility issues and change foundational concepts within the API. Hence, the next release of the GraphBLAS is a major release, *GraphBLAS 2.0*. We refer to prior releases (1.0, 1.0.2, 1.1.0, 1.2.0, and 1.3.0) collectively as *GraphBLAS 1.X*.

In this paper, we introduce the GraphBLAS 2.0 specification. We start with a section that summarizers the motivation behind some of the larger changes made in moving from GraphBLAS 1.X to GraphBLAS 2.0. We then describe the new features defined in GraphBLAS 2.0.

- Multithreading and how the GraphBLAS interact with the host language memory model.
- Expanded concept of an execution context to support hierarchical multithreading and to prepare for a future version of the GraphBLAS that supports distributed computing.
- An error model that is easier to support in a multithreaded implementation of the GraphBLAS.
- A new *GrB_scalar* GraphBLAS object.
- Methods to import and export data between GraphBLAS matrices and common array formats.
- Methods to serialize GraphBLAS objects.
- Operations that use the indices of elements of GraphBLAS objects.

The paper closes with a brief description of minor enhancements in the 2.0 specification and concluding comments.

II. Motivation

As the GraphBLAS API and computer architectures evolved over the last few years, several limitations of the existing GraphBLAS API became clear. While the version 1.X of the GraphBLAS API touched on the subject of multithreading, it was underspecified. As computers and processors get more heterogeneous [13], either having different cores on a processor or having both CPUs and various types of accelerators on the same compute node, a flat notion of parallelism is no longer realistic even on a single compute node. This is true for parallelism available on distributed-memory computers even though we do not cover those aspects in this paper.

This paper makes two contributions along this area. The first is a deeper dive into details of exploiting multithreading for GraphBLAS programs, including a coverage of pitfalls and recommended usage. The second is the introduction of the GraphBLAS Context object (`GrB_Context`) that has been alluded to in the past [3]. The GraphBLAS Context object specifies where in the address space each GraphBLAS object (matrix or vector) lies and how the resources (e.g., threads) are allocated to those objects. Due to the large variation in specifying those resources by different platforms and concurrency environments, a majority of the details of

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00047

how a `GrB_Context` object is initialized is implementation-defined.

Until now, GraphBLAS API did not allow its operators and semirings to access the vector and matrix indices. This had the benefit of keeping these functions and semiring operations to be relatively stateless and functional. This approach of disabling access to index values predates GraphBLAS and similar linear-algebraic libraries, such as Combinatorial BLAS, do the same. Whenever a graph algorithm needs indices, those index values were stored in the values array. During the computation, these index values were unpacked from the values array. Clearly this is inefficient in terms of storage and bandwidth as the same information is stored and streamed twice: once as part of index array and once as part of value arrays. More importantly, for common use cases, it requires user-defined operators and semirings just to be able to unpack the index values from the values arrays. This leads to additional performance penalties in implementations of the GraphBLAS C API, such as SuiteSparse::GraphBLAS, because of a function pointer call required for each scalar operation. For these reasons, we have expanded the API to allow access to matrix and vector index values in a few key GraphBLAS operations.

III. MULTITHREADING

The GraphBLAS C specification versions 1.X deliberately deferred consideration of multithreaded execution. A conformant implementation could utilize multiple threads inside a GraphBLAS method, but the behavior of an application calling GraphBLAS methods from multiple threads was not addressed. This was not an oversight. Execution of GraphBLAS method calls from multiple threads is complicated. We chose to ignore those issues so we could focus on translating the GraphBLAS mathematical specification into C.

With GraphBLAS 2.0, we are addressing multithreading directly. We require in the 2.0 specification that a conformant implementation of the GraphBLAS specification be *thread safe*. A GraphBLAS library is thread safe when independent method calls (*i.e.*, GraphBLAS objects are not shared between method calls) from multiple threads in a race-free program return the same results as would follow from their sequential execution in some interleaved order. This is a common requirement in software libraries. Thread safety requires an implementation of a library to carefully manage (or if possible, eliminate) shared data structures inside a library.

Thread safety applies to the behavior of multiple independent threads. In the more general case for multithreading, threads are not independent. That is, they share variables and mix read and write operations to those variables across threads. This requires the program to address the memory consistency model of the host programming environment.

A memory consistency model is defined by the programming language used to implement a library, the programming language used to write the application making calls to the library, and the architecture of the hardware on which the application runs. The memory consistency model defines the rules that specify the result that can be returned from reading the value of a variable. This sounds simple, but consider the complexity of multiple concurrent threads issuing loads and stores to variables across a complex memory hierarchy. At any point, a variable may exist in a register, in multiple levels of cache, in DRAM, NVRAM, memory pages managed by a Translation Look-aside Buffer (TLB), etc.

The GraphBLAS 2.0 specification does not define its own memory model. Instead we define what must be done by a programmer calling GraphBLAS methods and by the implementor of a GraphBLAS library so an implementation of the GraphBLAS can work correctly using the memory model of the host environment (*i.e.*, the compiler and the processors on which the program is running). A detailed description of the C memory consistency model requires many more pages then would fit in this paper. The essential elements of this model can be briefly summarized as follows.

- Concurrent execution of threads, that is, they are unordered with respect to each other and scheduled such that each thread gets an even chance to execute (*fair scheduling*).
- A *happens-before* relationship defines an ordering constraint between threads. This is done through an event that defines a *synchronized-with* relationship between threads.
- A synchronized-with relation is usually expressed in terms of loads and stores of atomic variables.
- The memory model defines constraints on the order of memory operations with respect to the atomic loads and stores used in a synchronized-with event. Three memory orders are of interest to multithreading in the GraphBLAS:
  - *Acquire*: Associated with an atomic *load* operation, no reads or writes that follow the atomic load in program order can be reordered to *precede* the atomic-load.
  - *Release*: Associated with an atomic *store* of a variable, no writes prior to the atomic-store in program order can be reordered to *follow* the atomic-store. These writes are visible to other threads that execute an atomic-load of the same variable with the acquire memory order.
  - *sequential consistency*: A sequentially consistent load is an acquire operation. A sequentially consistent store is a release operation. Sequential consistency is a stronger order than acquire-release since it requires a single total order for memory operations *and* that all threads observe that same order.

A program that violates the rules defined by a memory consistency model contains data races; that is, reads and writes that are unordered across threads making the final value of a variable undefined. Modern programming languages such as Java, C, C++ as well as multithreaded APIs such as OpenMP specify that a program that contains any data races is invalid and the results of that program are undefined. Hence, a programmer writing multithreaded code must go to great lengths to assure that their code is free of data races.

The GraphBLAS C specification adds a significant complication to reasoning about memory consistency models. A GraphBLAS object at any point in an application is defined by a collection of GraphBLAS methods in program order. We call this collection of GraphBLAS method calls the *sequence* that defines the GraphBLAS object at a given point in the program. An implementation of the GraphBLAS specification can reorder operations in a sequence or even fuse operations to create more efficient but mathematically equivalent executions of a sequence. This means that at any point in a program, the state of a computation over GraphBLAS objects is potentially ambiguous and a clear happens-before relationship cannot be defined.

To address this problem, we have added the concept of *completion* to the GraphBLAS specification. A GraphBLAS object is said to be *complete* when it can be used in a happens-before relationship with a method call that reads the variable on another thread. Take the example of a pair of GraphBLAS method calls where a method on one thread writes an object and a method on a second thread reads that object. To establish a happens before relationship and support race free execution, the programmer must do the following.

- Express the collection of ordered GraphBLAS method calls that define the sequence for the GraphBLAS object.
- Put the object into a *complete* state which causes computations on the object to finish and data structures internal to the GraphBLAS implementation to be resolved in memory so they are available to another thread.
- Define the synchronized-with relation between threads using the approach provided by the host programming environment (*i.e.*, the GraphBLAS specification does not define a multithreading programming model).
- Stipulate use of acquire and release operations so variables shared between threads are updated and available.
- Use the GraphBLAS object in a method that follows the synchronized-with relation on the second thread.

An object, obj, is forced into a state of completion using GrB_wait():

```
GrB_wait(obj, GrB_COMPLETE);
```

This method will not return until the sequence that defines *obj* completes the computations defined by the sequence and data structures internal to the opaque object are in a state that can be safely shared between threads, where we use the word “safe” with respect to the memory consistency model of the host programming environment.

Figure 1 contains an example of a properly synchronized GraphBLAS program. The program is written with a C-style pseudo-code. For brevity, we omit details about the argument lists to the GraphBLAS method calls. We use OpenMP [2] to express multithreading.[1] The parallel construct should fork two threads which are distinguished by their id. Thread 0 computes a shared object named Esh and calls GrB_wait() to force Esh into the GrB_COMPLETE state. Following completion of Esh, thread 0 writes to flag to indicate that Esh is ready to use. Thread 1 carries out a local computation and then waits at a spin lock for an update to the variable flag. This establishes a synchronized-with relation between threads

```
#include <omp.h>
#include "GraphBLAS.h"

int main()
{
   int flag = 0;  // Synchronization flag
   GrB_Matrix Esh, Hres, Dres;

   GrB_init(GrB_NONBLOCKING);

   omp_set_num_threads(2);
   #pragma omp parallel shared(Esh, Hres, Dres)
   {
       if(omp_get_num_threads() != 2) exit();
       int id = omp_get_thread_num();

       if(id == 0){
           GrB_Matrix A, B, C, D;

           // A user written function (not shown)
           Load_and_initialize(A,B,C,D,Esh,Dres);

           // simplified ... most args  omitted
           GrB_mxm(C,   A, B);
           GrB_mxm(Esh, D, C);

           GrB_wait(Esh, GrB_COMPLETE);

           #pragma omp atomic write release
              flag = 1;

           GrB_mxm(Dres, A, Esh);
           GrB_wait(Dres, GrB_COMPLETE);
       }
       else if(id==1){
           int tmp = 0;
           GrB_Matrix E, F, G;

           // A user written function (not shown)
           Load_and_initialize(E, F, G, Hres);

           // simplified ... most args omitted
           GrB_mxm(G, E, F);

           while(tmp == 0){
               #pragma omp atomic read acquire
                  tmp = flag;
           }
           GrB_mxm(Hres, G, Esh);
           GrB_wait(Hres, GrB_COMPLETE);
       }
   }  // end parallel region. A barrier is implied

   // Dres and Hres are available at this point.

   GrB_finalize();
}
```

Fig. 1. GraphBLAS C-style pseudo code with two threads sharing a matrix Esh with a spin-lock on the atomic update to flag with acquire and release memory orders. Calls to GrB_wait() are needed when GraphBLAS objects (Esh, Dres, and Hres) are written in one thread and read in another.

[1] While we use OpenMP in this paper, the GraphBLAS C specification is not tied to OpenMP. A conforming implementation of the GraphBLAS must work with any multithreading API that follows the C or C++ memory consistency model.

0 and 1. Note that writes and reads to flag are done with an OpenMP atomic operation and the memory order on the write to flag is release while the read of flag uses the acquire memory order. The use of acquire-release semantics means that memory operations on each thread are ordered with respect to their atomic operations and the second thread can expect that the value of Esh, including any internal data structures used to implement the opaque object, are available and visible. Note that calls to GrB_wait() are needed for Dres and Hres (the final results from the two threads) to assure that they are complete and available to other threads after the parallel region has finished.

An implementation of the GraphBLAS 2.0 specification is responsible for any additional synchronized-with relations required by internal data structures used to implement the opaque GraphBLAS objects. We attempted to make the implementors job easier by requiring the programmer writing an application that uses the GraphBLAS to use the acquire and release memory orders (rather than one of the weaker memory orders). In cases where an opaque GraphBLAS object is supported by static data structures, the implementor should not need any additional synchronization due to the rules on memory operations (specifically those not involving the atomic variables) when using the acquire-release memory orders.

IV. Execution Context

Calls to GraphBLAS functions occur within a *context*. In GraphBLAS 1.X, there is a single context for the entire program. It establishes the mode for the execution of GraphBLAS functions (GrB_BLOCKING or GrB_NONBLOCKING). As the GraphBLAS evolves to include multithreaded and (soon) distributed execution, the scope of GraphBLAS's context must expand to include features of a parallel execution such as the number of threads, thread affinities, MPI Communicators, and potentially much more. Furthermore, the execution of an application may embed parallel executions in complex ways. For example, a common pattern in high performance computing is to have a top level distributed execution using MPI with multithreaded execution on each node using OpenMP. Hence, our new expanded context must also enable hierarchical parallel execution strategies.

In Figure 2 we present the functionality added to the GraphBLAS specification to add the concept of hierarchical execution contexts to the GraphBLAS. A program must start with the top level context with a call to GrB_init(). This is unchanged from GraphBLAS 1.X. A context nested within an "outer" context is added with a call to the method GrB_Context_new(). This function includes an output parameter for a handle to a GraphBLAS context with the type GrB_Context. As with other GraphBLAS objects, this is an opaque type. This new context can use the existing GraphBLAS modes. Contexts are hierarchical and defined with respect to a parent context that is specified by the parent argument. Passing GrB_NULL implies that the "top level" context is to be used. In addition, GrB_Context_new() takes a void* pointer to a structure containing information about the new context. The contents of this structure is *implementation defined* which means the GraphBLAS specification does not define this structure but a conforming implementation of the GraphBLAS must include documentation that defines it. For example, with OpenMP, this might be a number of threads to use in the context. It might also include information about the places to run threads or the affinity of threads. Such details are too varied across implementations to standardize so the specification gives an implementor flexibility to do what is needed for their specific situation.

```
GrB_Info GrB_init(GrB_Mode  mode);

GrB_Info GrB_Context_new(GrB_Context  *ctx,
                         GrB_Mode      mode,
                         GrB_Context   parent,
                         void         *exec);

GrB_Info GrB_Vector_new(GrB_Vector    *v,
                        GrB_Type       d,
                        GrB_Index      nsize,
                        GrB_Context    ctx);

GrB_Info GrB_Matrix_new(GrB_Matrix    *A,
                        GrB_Type       d,
                        GrB_Index      nrows,
                        GrB_Index      ncols,
                        GrB_Context    ctx);

GrB_Info GrB_Context_switch(<GrB Object> *obj,
                            GrB_Context   newCtx);
```

Fig. 2. GraphBLAS methods for context functionality.

When writing parallel code, one of the greatest challenges is to manage locality of data to minimize the costs of data movement. We expose locality management to the GraphBLAS programmer by associating a GraphBLAS Matrix or Vector with a context. Hence, in GraphBLAS 2.0 we have added a new, optional argument to the constructors of GraphBLAS vectors and matrices for the context the vector or matrix belongs to. We require that all the GraphBLAS matrices and Vectors in a GraphBLAS method share a context. In doing so, a GraphBLAS implementation can use this shared context to manage data movement without exposing low-level details to the application programmer. This means we need a method to switch an object's context, GrB_Context_switch() where in this case by <GrB object> we refer to GrB_Vector or GrB_Matrix.

Finally, the handle to a context is an opaque GraphBLAS object. Any resources needed to support that object can be deleted by a call to GrB_free() after which that object behaves as an uninitialized object. A call to GrB_finalize() frees all GrB_Context objects.

V. Error Model

The error model as defined in GraphBLAS 1.X created challenges for the implementor we did not intend. These challenges impacted the way the GraphBLAS interacts with multithreaded execution. Hence, in GraphBLAS 2.0 we made major changes to how the error model works. The GraphBLAS C specification defines two kinds of errors: API errors and execution errors.

API errors indicate that a GraphBLAS method call was malformed, with arguments that violate the rules for that method. API errors are deterministic and consistent across platforms and implementations. They are never deferred, even in nonblocking mode. If a GraphBLAS method returns with an API error, it is guaranteed that none of the arguments of the method, or any other program data, have been modified.

Execution errors indicate that something went wrong during the execution of a well formed GraphBLAS method invocation. (If the invocation were not well formed, it would have returned with an API error.) The occurrence of execution errors can depend on various circumstances specific to a run. Sometimes they are clearly caused by program errors (*e.g.*, GrB_INDEX_OUT_OF_BOUNDS). Sometimes they are clearly caused by internal errors in the implementation (*e.g.*, GrB_PANIC). Sometimes they are caused by either or even environment limitations (*e.g.*. GrB_OUT_OF_MEMORY).

In GraphBLAS nonblocking mode, the execution of the actual operations performed by the method can be deferred. Consequently, execution errors and their reporting can also be deferred. To constraint the scope of execution errors, GraphBLAS 2.0 introduces a new *materializing* variant of GrB_wait:

```
GrB_wait(obj, GrB_MATERIALIZE);
```

A successful return from this method indicates that all previous method invocations with `obj` as an OUT or INOUT argument have completed (*i.e.*, the same effect as `GrB_wait(obj, GrB_COMPLETE)`) and no more errors can be generated from those methods. (It also guarantees that no more execution time will be charged to those methods.)

The `GrB_wait(obj,GrB_MATERIALIZE)` variant always includes the `GrB_wait(obj,GrB_COMPLETE)` variant. In particular, a thread can call a sequence of methods with `obj` as an OUT/INOUT argument and then call `GrB_wait(obj,GrB_COMPLETE)`. After that, with proper synchronization, a second thread can continue the sequence and end it with a call to `GrB_wait(obj,GrB_MATERIALIZE)`. Until the materializing call, any method invocation in the second thread can report an error from any of the previous methods in the sequence, including those in the first thread. This is implementation dependent, as an implementation can always make the completing wait method equivalent to a materializing wait.

When the execution of a method, deferred or not, causes an execution error, the state of the OUT or INOUT argument of that method is undefined. An implementation-specific string about the error state of an object can be obtained through a call to

```
GrB_error(&str,obj);
```

which returns in `str` a pointer to the string. The call is thread safe. Two threads can call the method concurrently, even on the same object, without any synchronization as long as the conditions from Section III hold. (Although each has to pass a different string pointer.) The contents of the string are implementation-defined. It is always legal to return an empty string.

VI. GRB_SCALAR

A GraphBLAS *scalar* (`GrB_Scalar`) is an opaque container for a single element of a GraphBLAS domain, either predefined or user-defined. Just like GraphBLAS vectors and matrices, GraphBLAS scalars can be empty. Table I lists the methods used to define an manipulate `GrB_Scalar` objects. `GrB_Scalar` objects serve two main purposes.

First, they significantly reduce the number of nonpolymorphic variants of a method, because the scalar argument is always of type `GrB_Scalar`, as opposed to one of the predefined types of `void*` for user-defined types. This reduction in variants also helps with the clarity of polymorphic code. For example, it is easy to forget that the standard C constant "`true`" is actually of type "`int`" and not "`bool`". Similarly, scalars of user-defined types are always passed as `void*`, which makes it easy to misuse one user-defined type for another. In contrast, a `GrB_Scalar` always has the type assigned to it when it was created through `GrB_Scalar_new` or `GrB_Scalar_dup`.

The second major application of GraphBLAS scalars is that, in being opaque objects, they make the behavior of some methods more uniform. As an example, consider the current variants of `GrB_Vector_extractElement` and `GrB_Matrix_extractElement`. Since the operation extracts the value of a (possibly nonexistent) element into a nonopaque data structure, the program has to (i) test for the possibility of the element not existing (indicated by a GrB_NO_VALUE return code) and (ii) immediately retrieve the value (if present), without the possibility of deferring execution. A variant with `GrB_Scalar` as the output bypasses both of these problems.

Other examples are the matrix and vector variants of the `GrB_reduce` operation when reducing that produces a scalar value. With the current variants, the methods return the monoid identity when there is nothing to reduce. A variant with `GrB_Scalar` output can instead return an empty container, similar to what happens when a matrix is reduced to a vector. Moreover, we can now define reduction to scalar that takes `GrB_BinaryOp` as the reducing function.

Table II lists the GraphBLAS methods that are currently targeted for extension with `GrB_Scalar` variants in GraphBLAS 2.0 and beyond. The first to be released will be methods that have scalar outputs. We are not yet deprecating the variants with explicitly typed elements, but we may choose to do so in the future.

TABLE I
GrB_Scalar MANIPULATION METHODS.

Method	Description
GrB_Scalar_new(GrB_Scalar*, GrB_Type)	Create a GraphBLAS scalar of certain domain
GrB_Scalar_dup(GrB_Scalar*, const GrB_Scalar)	Duplicate an existing GraphBLAS scalar into a new one
GrB_Scalar_clear(GrB_Scalar)	Empty the contents of a GraphBLAS scalar
GrB_Scalar_nvals(GrB_Index*, const GrB_Scalar)	Return number of elements in a GraphBLAS scalar (0 or 1)
GrB_Scalar_setElement(GrB_Scalar, <type>)	Set the value of the element of a GraphBLAS scalar
GrB_Scalar_extractElement(<type>*, const GrB_Scalar)	Extract the value of the element of GraphBLAS scalar, if present

TABLE II
GRAPHBLAS METHODS TO BE EXTENDED WITH GrB_Scalar VARIANTS IN GRAPHBLAS 2.0 AND BEYOND. (const KEYWORDS OMITTED.)

GrB_Monoid_new(GrB_Monoid*, GrB_BinaryOp, GrB_Scalar)
GrB_Vector_setElement(GrB_Vector, GrB_Scalar, GrB_Index)
GrB_Vector_extractElement(GrB_Scalar, GrB_Vector, GrB_Index)
GrB_Matrix_setElement(GrB_Matrix, GrB_Scalar, GrB_Index, GrB_Index)
GrB_Matrix_extractElement(GrB_Scalar, GrB_Matrix, GrB_Index, GrB_Index)
GrB_assign(GrB_Vector, GrB_Vector, GrB_BinaryOp, GrB_Scalar, GrB_Index*, GrB_Index, GrB_Descriptor)
GrB_assign(GrB_Matrix, GrB_Matrix, GrB_BinaryOp, GrB_Scalar, GrB_Index*, GrB_Index, GrB_Index*, GrB_Index, GrB_Descriptor)
GrB_apply(GrB_Vector, GrB_Vector, GrB_BinaryOp, GrB_BinaryOp, GrB_Scalar, GrB_Vector, GrB_Descriptor)
GrB_apply(GrB_Vector, GrB_Vector, GrB_BinaryOp, GrB_BinaryOp, GrB_Vector, GrB_Scalar, GrB_Descriptor)
GrB_apply(GrB_Matrix, GrB_Matrix, GrB_BinaryOp, GrB_BinaryOp, GrB_Scalar, GrB_Matrix, GrB_Descriptor)
GrB_apply(GrB_Matrix, GrB_Matrix, GrB_BinaryOp, GrB_BinaryOp, GrB_Matrix, GrB_Scalar, GrB_Descriptor)
GrB_apply(GrB_Vector, GrB_Vector, GrB_BinaryOp, GrB_IndexUnaryOp, GrB_Vector, GrB_Scalar, GrB_Descriptor)
GrB_apply(GrB_Matrix, GrB_Matrix, GrB_BinaryOp, GrB_IndexUnaryOp, GrB_Matrix, GrB_Scalar, GrB_Descriptor)
GrB_select(GrB_Vector, GrB_Vector, GrB_BinaryOp, GrB_IndexUnaryOp, GrB_Vector, GrB_Scalar, GrB_Descriptor)
GrB_select(GrB_Matrix, GrB_Matrix, GrB_BinaryOp, GrB_IndexUnaryOp, GrB_Matrix, GrB_Scalar, GrB_Descriptor)
GrB_reduce(GrB_Scalar, GrB_BinaryOp, GrB_Monoid, GrB_Vector, GrB_Descriptor)
GrB_reduce(GrB_Scalar, GrB_BinaryOp, GrB_BinaryOp, GrB_Vector, GrB_Descriptor)
GrB_reduce(GrB_Scalar, GrB_BinaryOp, GrB_Monoid, GrB_Matrix, GrB_Descriptor)
GrB_reduce(GrB_Scalar, GrB_BinaryOp, GrB_BinaryOp, GrB_Matrix, GrB_Descriptor)

VII. DATA TRANSFER

The problem of moving data into and out of the GraphBLAS is an important issue that can be approached from two different angles, each with different requirements. First, it is necessary to have a mechanism for moving data into and out of the GraphBLAS using *known, non-opaque formats*. This is necessary to allow GraphBLAS to interact with other libraries and so that data can be stored in a standard format. However, it is sometimes advantageous to allow an implementation-defined, *opaque format* that need not be compatible between implementations. An opaque format can allow for more efficient data movement, since it does not require data be converted to a specific format.

A. Import/Export API

In order to allow for easier interaction with other libraries, as well as more straightforward storage of GraphBLAS objects, we have added an import/export API in GraphBLAS 2.0. The import/export API supports a number of commonly used formats, including CSR, CSC, COO, and dense matrix formats, as well as dense and sparse vector formats. To import an object stored in one of these formats into the GraphBLAS, users can invoke the GrB_Matrix_import or GrB_Vector_import method, passing in the size and type of the object, pointers to arrays defining the object data, and a GrB_Format argument indicating the format of the data being imported. A new GraphBLAS object will be constructed using the provided external matrix or vector data.

To export a GraphBLAS matrix for external use, users first invoke the GrB_Matrix_exportSize method, passing in the GraphBLAS matrix to be exported as well as the desired export format. The method will return sizes for each of the output arrays that the user can then allocate using their desired method. This allows users to use a custom allocator, a memory-mapped file, or some other method for memory allocation. After allocating the external arrays, the user then must invoke the GrB_Matrix_export method, which will export the matrix using the specified format. Corresponding methods have been added for GraphBLAS vectors.

In addition, the GrB_Matrix_exportHint method has been included that allows GraphBLAS implementations to provide a hint about which format might be most efficient for exporting a matrix. For example, if a GraphBLAS matrix is internally stored in CSR, the implementation might choose to return the CSR format as a hint. Note, however, that users are free to arbitrarily pick a format for import and export. Likewise, the implementation may refuse to provide a hint by returning GrB_NO_VALUE. A corresponding method has been added for GraphBLAS vectors.

B. Serialize/Deserialize API

While the import/export API provides a convenient way to move data in and out of GraphBLAS matrices and vectors in a number of predefined formats, this is not necessarily the most efficient method of data transport. It is likely that some GraphBLAS applications, for example those in a distributed setting, will want to extract data

TABLE III
NON-OPAQUE MATRIX FORMATS FOR IMPORT/EXPORT.

Format	Parameters	
GrB_CSR_MATRIX Compressed sparse row.	indptr	a pointer to an array of size nrows+1 and the i'th index will contain the starting index of the i'th row in the values and indices arrays. The elements of each row are not required to be sorted by column index.
	indices	a pointer to an array of size number of stored elements, where each element contains the corresponding element's column index.
	values	a pointer to an array of size number of stored elements, where each element contains the corresponding value.
GrB_CSC_MATRIX Compressed sparse column.	indptr	a pointer to an array of size ncols+1 and the i'th index will contain the starting index of the i'th column in the values and indices arrays. The elements of each column are not required to be sorted by row index.
	indices	a pointer to an array of size number of stored elements, where each element contains the corresponding element's row index.
	values	a pointer to an array of size number of stored elements, where each element contains the corresponding element's value.
GrB_COO_MATRIX Sparse coordinate format. Elements are not required to be sorted in any order.	indptr	a pointer to an array of size number of stored elements, where each element contains the corresponding element's column index.
	indices	a pointer to an array of size number of stored elements, where each element contains the corresponding element's row index.
	values	a pointer to an array of size number of stored elements, where each element contains the corresponding value.
GrB_DENSE_ROW_MATRIX Dense row-major format.	indptr	unused and may be set to NULL
	indices	unused and may be set to NULL
	values	an array of size number of columns times number of rows, where element i,j is located at index i*ncols + j.
GrB_DENSE_COL_MATRIX Dense column-major format.	indptr	unused and may be set to NULL
	indices	unused and may be set to NULL
	values	array of size number of columns times number of rows, where element i,j is located at index i + j*nrows.
GrB_SPARSE_VECTOR Sparse vector format.	indices	an array of size number of elements, where element i contains the index of the corresponding element.
	values	array of size number of elements, where value of element i is located at index i.
GrB_DENSE_VECTOR Dense vector format.	indices	unused and may be set to NULL
	values	array of size number of elements, where element i is located at index i.

in an arbitrary, opaque, serialized stream of bytes which can easily be sent over the wire. However, this serialized object need not be interpretable by the program or indeed even other implementations of the GraphBLAS. This allows implementations to use custom serialization mechanisms, which can save both space and compute time. To this end, we have added an API for serializing and deserializing GraphBLAS matrices. To serialize a matrix, users must first call the `GrB_Matrix_serializeSize` method, which will return the size in bytes of the buffer needed to serialize the matrix. Then, users can call the `GrB_Matrix_serialize` method, which will serialize the matrix into an opaque stream of bytes in the user-provided buffer. The serialized matrix can then be deserialized into a GraphBLAS object using the `GrB_Matrix_deserialize` method, which will construct a new GraphBLAS matrix from the serialized data. Corresponding methods have been added for GraphBLAS vectors.

VIII. OPERATING WITH INDICES

In GraphBLAS 1.X, elementwise operations such as apply specify a unary operator that operates only on the values stored in a matrix or vector. GraphBLAS 2.0 expands the set of operator types to include one that operates on both the stored value *and* the indices describing the value's location in the matrix or vector. Figure 3 shows an example of how these new operators can be used to either select a subset of the stored values using the new select operation, or modify the stored values based on its location using new variants of the apply operation.

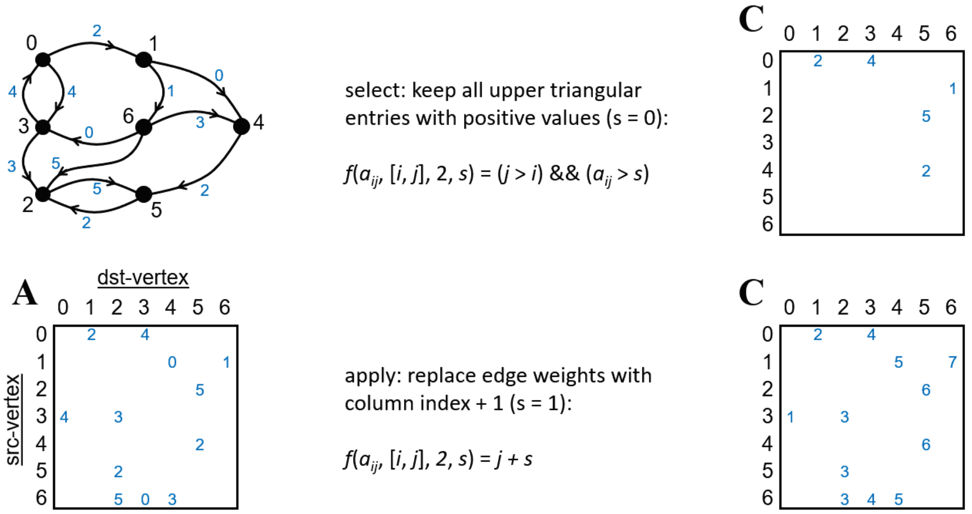
A. Index Unary Operators

GraphBLAS 2.0 supports the creation of arbitrary user-defined index unary operators that can operate on either vectors or matrices. The following `GrB_IndexUnaryOp_new` method is used to construct these operators:

```
GrB_Info GrB_IndexUnaryOp_new(
   GrB_IndexUnaryOp   *index_unary_op,
   void (*index_unary_func)(
         void*,        // out     = C(i,j) or w(i)
         const void*,  // in      = A(i,j) or u(i)
         GrB_Index*,   // indices = [i,j]  or [i]
         GrB_Index,    // n       = 2      or 1
         const void*), // s       (user specified)
   GrB_Type            d_out,
   GrB_Type            d_in,
   GrB_Type            d_s);
```

The first argument specifies the handle to the operator object that is being constructed. The second argument is a pointer to the user-defined function, `index_unary_func`, that performs the computation on the stored values and indices. The final three GrB_Type parameters define the types that correspond to `void*` parameters of the user-defined function, `out`, `in1`, and `s`, respectively.

There are five parameters to the user-defined function, `index_unary_func`, as indicated by the comments in the code above. The `out` parameter is the result of the function.


select: keep all upper triangular
entries with positive values (s = 0):
f(a_ij, [i, j], 2, s) = (j > i) && (a_ij > s)
apply: replace edge weights with
column index + 1 (s = 1):
f(a_ij, [i, j], 2, s) = j + s
A
dst-vertex
src-vertex
C
C
0 1 2 3 4 5 6

(a) Original weighted graph and its adjacency matrix.

(b) Examples of operators for select and apply operations

(c) Matrices resulting from the operations.

Fig. 3. When using index unary operators with apply and select operations, the value and location (indices) of stored values are provided as well as an additional scalar, s, that is provided by the user via the apply and select parameters. The select operation uses an operator that returns boolean to determine which stored elements are kept (unchanged) or annihilated from the input matrix, while the operator for apply computes new values for every stored element in the input.

The next three parameters (in, indices, and n) define the value and location of a stored element in the matrix or vector. When used in an operation on a matrix, indices is a pointer to an array of two indices (row and column), and n is set to 2. When used in an operation on a vector, indices is a pointer to an array of one index (row), and n is set to 1.

The last argument to the user-defined function, s, is an additional scalar value that is specified by the user application (passed through the select or apply operation) and can be used in the calculation of the out value. For example, if you want to “select” elements with a certain value, s can be set to that value and the operator performs a comparison between the stored value (A(i,j) or u(i)) and s and returns the result as a boolean. The s parameter can also be used in comparisons with the indices to select values based on their location in the matrix.

The following is an example of the code used to define an index unary operator that selects values in the upper triangular portion of a matrix that are greater than a user-defined constant (passed as the s parameter) and returns a boolean. This creates the operator for the select example shown on the top right of Figure 3):

```
void my_triu_eq_INT32(void         *out,
                      const void   *in,
                      GrB_Index    *indices,
                      GrB_Index     n,
                      const void   *s)
{
   assert (n == 2);  // perform test in debug mode
   *out =((indices[1] > indices[0]) && // j > i
         (((int)*in) > ((int)*s)));    // a_ij > s
}

GrB_IndexUnaryOp myTriuEqINT32;
GrB_IndexUnaryOp_new(
      &myTriuEqINT32, my_triu_eq_INT32,
      GrB_BOOL, GrB_INT32, GrB_INT32);
```

TABLE IV
PREDEFINED INDEX UNARY OPERATORS.

Operator	Description
GrB_{ROW,COL}INDEX	replace with a row or column index (plus s)
GrB_DIAGINDEX	replace with a diagonal index (plus s)
GrB_{TRIL,TRIU}	keep the elements below/above diagonal s
GrB_{DIAG,OFFDIAG}	keep diagonal s, or remove diagonal s
GrB_{ROW,COL}LE	keep the set of rows or columns $<$ s
GrB_{ROW,COL}GT	keep the set of rows or columns $\geq$ s
GrB_VALUE{EQ,NE,LT, LE,GT,GE}	keep any element based on a comparison of its stored value with s

To support this new functionality, a set of predefined operators is also defined by the specification and summarized in Table IV. While the operators listed in this table can be used by both apply and select operations, the descriptions starting with “keep” correspond to operators that return boolean values intended for use with select, while descriptions starting with “replace” correspond to operators returning other types intended for use with apply. Operators like VALUEEQ only access the stored value that is passed and can be used in operations on either vectors or matrices. Likewise, operators like ROWINDEX only access the first entry of the indices array and can also be used in operations on either vectors or matrices. However, operators such as COLINDEX, DIAGINDEX, TRIL, or TRIU access both row and column indices and are only intended for use with matrices. If these operators are used in operations involving vectors, an index array of one element is passed which will result in undefined behavior.

B. *Index Variants of* apply

The new variants of the apply operation allow the use of an index unary operator to compute new stored values for a vector or matrix as a function of the existing stored values, the indices of their locations, and a scalar. The mathematical notation for these operations are given as follows:

$$\begin{aligned}\mathbf{w}\langle \mathbf{m}, r\rangle &= \mathbf{w} \odot f(\mathbf{u}, \mathrm{ind}(\mathbf{u}), 1, s)\\ \mathbf{C}\langle \mathbf{M}, r\rangle &= \mathbf{C} \odot f(\mathbf{A}^{[\mathsf{T}]}, \mathrm{ind}(\mathbf{A}^{[\mathsf{T}]}), 2, s)\end{aligned}$$

Note that when the input matrix, `A`, is transposed, the index values used in the computation correspond to locations after the transpose is applied. The signature of these operations follow the standard order of other GraphBLAS operations:

```
// Vector variant of apply using IndexUnaryOp
GrB_Info GrB_apply(GrB_Vector               w,
                   const GrB_Vector         m,
                   const GrB_BinaryOp       accum,
                   const GrB_IndexUnaryOp   f,
                   const GrB_Vector         u,
                   <type>                   s,
                   const GrB_Descriptor     desc);

// Matrix variant of apply using IndexUnaryOp
GrB_Info GrB_apply(GrB_Matrix               C,
                   const GrB_Matrix         M,
                   const GrB_BinaryOp       accum,
                   const GrB_IndexUnaryOp   f,
                   const GrB_Matrix         A,
                   <type>                   s,
                   const GrB_Descriptor     desc);
```

Where `w` and `C` will hold the results of the operation and `u` and `A` are the input objects. Each operation supports an optional mask and accumulation operator. Different flags in the descriptor control how the output is written (replace or merge), how the mask is used (structure and/or complement), and whether the input matrix is transposed (matrix variant only). The `f` parameter specifies which index unary operator is to be used. The `s` parameter is the scalar value that is passed to the `s` argument of the `GrB_IndexUnaryOp`.

A common use for these functions is to replace stored values with their row index (vectors and matrices) or column index (matrix) only. With a matrix, the former replaces edge weights with its source vertex index while the latter replaces it with the destination index. The predefined operator `DIAGINDEX` can be used to replace matrix elements with their diagonal index. The following code uses the predefined column index operator from Table IV to perform the apply operation shown in the lower right portion of Figure 3:

```
GrB_Matrix C, A;
...
GrB_apply(C, GrB_NULL, GrB_NULL,
          GrB_COLINDEX_UINT64T, A, 1UL, GrB_NULL);
```

C. *The* select *Operation*

The new select operation provides the equivalent of a functional input mask. It uses an index unary operator that returns a boolean value to determine which elements to keep (when the operator returns true) and which to annihilate (when the operator returns false). The mathematical notation for these operations are given as follows:

$$\begin{aligned}\mathbf{w}\langle \mathbf{m}, r\rangle &= \mathbf{w} \odot \mathbf{u}\langle f(\mathbf{u}, \mathrm{ind}(\mathbf{u}), 1, s)\rangle\\ \mathbf{C}\langle \mathbf{M}, r\rangle &= \mathbf{C} \odot \mathbf{A}^{[\mathsf{T}]}\langle f(\mathbf{A}^{[\mathsf{T}]}, \mathrm{ind}(\mathbf{A}^{[\mathsf{T}]}), 2, s)\rangle\end{aligned}$$

Where angle brackets on the right hand side of the equations are used to indicate the masking behavior of this operation. The signatures of this operation are identical to the index variants of apply:

```
// Vector variant of select
GrB_Info GrB_select(GrB_Vector               w,
                    const GrB_Vector         m,
                    const GrB_BinaryOp       accum,
                    const GrB_IndexUnaryOp   f,
                    const GrB_Vector         u,
                    <type>                   s,
                    const GrB_Descriptor     desc);

// Matrix variant of select
GrB_Info GrB_select(GrB_Matrix               C,
                    const GrB_Matrix         M,
                    const GrB_BinaryOp       accum,
                    const GrB_IndexUnaryOp   f,
                    const GrB_Matrix         A,
                    <type>                   s,
                    const GrB_Descriptor     desc);
```

Common uses for this operation include selecting regions of a matrix (TRIL, TRIU, ROWLE, COLLE, ROWGT, COLGT), select regions of a vector (ROWLE, ROWGT), or select elements of either a vector or matrix based on its value when compared to the scalar `s` (VALUE* where * is EQ, NE, LT, LE GT, or GE). The following code uses the operator created earlier in this section to perform the select operation shown in the upper right portion of Figure 3:

```
GrB_Matrix C, A;
...
GrB_apply(C, GrB_NULL, GrB_NULL,
          myTriuEqINT32, A, 0UL, GrB_NULL);
```

IX. Cleanup and Miscellany

Creating a formal specification is difficult. We strive for perfection but inevitably we make mistakes or leave important corner cases unaddressed. In this section, we cover a few changes made in the GraphBLAS to “cleanup” the specification that are beyond simple typographical errors.

The first change involves the definition of enumerations in the specification. Enumerations provide symbolic names for literal constants. With our goal of providing implementors of the GraphBLAS maximum flexibility to produce optimal libraries conforming to the specification, we aggressively exploited opaqueness in the specification. It turns out, however, that when it comes to the symbolic names that make up an enumeration, it is important to *not* leave them opaque. Defining specific values for each symbol lets programs correctly link to different libraries that implement the GraphBLAS. Therefore, any tables in the specification that list the elements of an enumeration will now also specify the values they must correspond to. Most notably this has been done for the preexisting specification of GrB_Info and has been applied to the new GrB_Format enumeration that lists the supported matrix formats for import and export.

The other change to the specification occurs in the definition of the GrB_Matrix_build() and GrB_Vector_build() methods. We neglected to cover the case of a user not wanting to define a "duplicate" function (the dup binary operator) that is called to handle the case when more than one value is specified for the same location to produce the value that will be stored in the vector or matrix. In previous releases, this parameter was required, but in this release it is optional. If a value of GrB_NULL is provided as the duplicate function, the result is that duplicates are now treated as an execution error.

X. Conclusion

The GraphBLAS 2.0 specification was a collaborative effort. In addition to the GraphBLAS C API Specification working-group (*i.e.*, the authors of this paper), we worked closely with the group behind the SuiteSparse implementation of the GraphBLAS [7] and the group behind the LAGraph [10] library. With this close collaboration between specification-designers and library-implementors, we hope to see implementations of the GraphBLAS conformant with the GraphBLAS 2.0 specification shortly after its official release.

With the GraphBLAS 2.0 specification complete, our work will shift to enhancements to the GraphBLAS to support execution on distributed systems. We also anticipate enhancements to support heterogeneous systems where devices such as GPUs and CPUs are addressed from a single program. These goals should be clear from the definition of the execution context described in this paper. Additional enhancements will undoubtedly be included in future versions of the GraphBLAS as we continue our ongoing collaboration with teams implementing software based on the GraphBLAS. These collaborations have been quite productive and we foresee even richer interactions in the future.

Acknowledgments and Disclaimers

We thank the members of the GraphBLAS forum. Benjamin Brock and Aydın Buluç were supported in part by the DOE Office of Advanced Scientific Computing Research under contract number DEAC02-05CH11231 and in part by NSF under Award No. 1823034. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM21-0299].

References

[1] Graphblas template library (GBTL). https://github.com/cmu-sei/gbtl.

[2] OpenMP API for multithreaded programming. www.openmp.org.

[3] Benjamin Brock, Aydin Buluç, Timothy G Mattson, Scott McMillan, José E Moreira, Roger Pearce, Oguz Selvitopi, and Trevor Steil. Considerations for a distributed GraphBLAS API. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 215–218. IEEE, 2020.

[4] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and José Moreira. The GraphBLAS C API Specification. *GraphBLAS. org, Tech. Rep.*, version 1.3.0, 2019.

[5] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *IEEE Intr. Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652, 2017.

[6] Bob Cook. GraphBLAS c99 library. https://github.com/bobcgausa/GraphBLAS.

[7] Timothy A Davis. Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[8] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José Moreira, John Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[9] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[10] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *Proc. GrAPL'19, Workshop on Graphs, Architectures, Programming, and Learning*, 2019.

[11] J. E. Moreira, M. Kumar, and W. P. Horn. Implementing the GraphBLAS C API. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 298–309, 2018.

[12] Michel Pelletier. pygraphblas: A Python API for GraphBLAS and LAGraph. https://github.com/Graphegon/pygraphblas.

[13] Jeffrey S Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity. Technical report, US DOE Office of Science (SC), Washington, DC (United States), 2018.

[14] Carl Yang, Aydın Buluç, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *arXiv preprint arXiv:1908.01407*, 2019.

[15] Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. GBTL-CUDA: Graph algorithms and primitives for GPUs. In *IEEE Intr. Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 912–920. IEEE, 2016.