Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra

TIMOTHY A. DAVIS, Texas A&M University, USA

SuiteSparse:GraphBLAS is a full implementation of the GraphBLAS standard, which defines a set of sparse matrix operations on an extended algebra of semirings using an almost unlimited variety of operators and types. When applied to sparse adjacency matrices, these algebraic operations are equivalent to computations on graphs. GraphBLAS provides a powerful and expressive framework for creating graph algorithms based on the elegant mathematics of sparse matrix operations on a semiring. An overview of the GraphBLAS specification is given, followed by a description of the key features and performance of its implementation in the SuiteSparse:GraphBLAS package.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; **Mathematical software**;

Additional Key Words and Phrases: Graph algorithms, sparse matrices, GraphBLAS

ACM Reference Format:
Timothy A. Davis. 2018. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2018), 24 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The GraphBLAS standard [5] defines sparse matrix and vector operations on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms.

For example, consider the matrix-matrix multiplication, $\mathbf{C} = \mathbf{AB}$. Suppose $\mathbf{A}$ and $\mathbf{B}$ are sparse n-by-n Boolean adjacency matrices of two undirected graphs. If the matrix multiplication is redefined to use logical AND instead of scalar multiply, and if it uses the logical OR instead of add, then the matrix $\mathbf{C}$ is the sparse Boolean adjacency matrix of a graph that has an edge (i, j) if node i in $\mathbf{A}$ and node j in $\mathbf{B}$ share any neighbor in common. The OR-AND pair forms an algebraic semiring, and many graph operations like this one can be succinctly represented by matrix operations with different semirings and different numerical types. GraphBLAS provides a wide range of built-in types and operators, and allows the user application to create new types and operators. Expressing graph algorithms in the language of linear algebra provides:

- a powerful way of expressing graph algorithms with large, bulk operations on adjacency matrices with no need to iterate over individual nodes and edges,
- composable graph operations, e.g. $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$,
- simpler graph algorithms in user-code,
- simple objects for complex problems – a sparse matrix with nearly any data type,
- a well-defined graph object, closed under operations,
- and high performance: the bulk graph/matrix operations allow a serial, parallel, or GPU-based library to optimize the graph operations.

Author's address: Timothy A. Davis, Texas A&M University, 3112 TAMU, College Station, TX, 77845, USA, davis@tamu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0098-3500/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

A full and precise definition of the GraphBLAS specification is provided in *The GraphBLAS C API Specification* [5], based on the mathematical foundations discussed in [15, 16].

SuiteSparse:GraphBLAS is the first reference implementation of the GraphBLAS standard, and has been rigorously tested to ensure full conformance with the specification. It is currently a purely sequential package meant to serve as a guide, framework, and testing platform against which parallel implementations can be compared and benchmarked. It is already a complex package, requiring 25,435 of code (and 20,942 lines of comments) in the main library, and 25,465 lines of test code (excluding comments) that verify its compliance with the specification. Including parallelism in this foundational implementation of GraphBLAS would greatly increase its code complexity, making it harder for developers of parallel implementations to use it as a baseline. With the sequential baseline established, the goal for the future is to incorporate parallel methods into SuiteSparse:GraphBLAS, via OpenMP, MPI, and on the GPU.

The remainder of this paper is structured as follows. Section 2 gives an overview of the GraphBLAS concept of writing graph algorithms in the language of linear algebra with sparse matrices and semirings. Section 3 provides an overview of the objects, methods, and operations in the GraphBLAS specification. Key features of the implementation of GraphBLAS are discussed in Section 4. Section 5 gives examples of two graph algorithms written in GraphBLAS. These algorithms and more appear in the `Demo` folder in the software package. Section 6 highlights the performance of SuiteSparse:GraphBLAS, followed by final comments and code availability in Section 7.

2 BASIC CONCEPTS

2.1 Graphs and sparse matrices

Many applications give rise to large graphs, but the are typically very sparse, with n nodes but only $e = O(n)$ edges. Some graphs are *hypersparse* with $e << n$.

Any graph $G = (V, E)$ can be considered as a sparse adjacency matrix A, either square or rectangular. The square case of an n-by-n sparse matrix is useful for representing a directed or undirected graph with n nodes, where either the matrix entry a_{ij} or a_{ji} represents the edge (i, j). In the rectangular case, an m-by-n sparse matrix can be used to represent a bipartite graph, or a hypergraph, depending on the context in which the matrix is used. Edges that do not appear in G are not represented in the data structure of the sparse matrix $\mathbf{A}$. A sparse matrix data structure allows huge graphs to be represented. A dense adjacency matrix for a graph of n nodes would take $O(n^2)$ memory, which is impossible when n is large, whereas a sparse adjacency matrix can be stored in $O(n + e)$ space, or $O(e)$ if stored in a hypersparse format, where e is the number of edges present in the graph. The notation $e = |\mathbf{A}|$ denotes the number of entries in the matrix $\mathbf{A}$.

Values of entries not stored in the sparse data structure have some implicit value. In conventional linear algebra, this implicit value is zero, but it differs with different semirings. Explicit values are called *entries* and they appear in the data structure. The *pattern* of a matrix defines where its explicit entries appear, and can be represented as either a set of indices (i, j), or as a Boolean matrix $\mathbf{S}$ where $s_{ij} = 1$ if a_{ij} is an explicit entry in the sparse matrix $\mathbf{A}$. The entries in the pattern of $\mathbf{A}$ can take on any value, including the implicit value, whatever it happens to be. It need not be the value zero. For example, in the max-plus tropical algebra, the implicit value is negative infinity, and zero has a different meaning.

2.2 Semirings and their use in graph algorithms

Graph Algorithms in the Language on Linear Algebra [18] provides a framework for understanding how graph algorithms can be expressed as matrix computations. For additional background on sparse matrix algorithms, see also [7] and a recent survey paper, [10].

All graph algorithms can be expressed in terms of sparse adjacency matrices, and nearly all can also be written in terms of linear algebraic operations on those matrices, using different semirings. Two examples are given in Section 5: breadth-first search and Luby's method for finding an independent set. Both are modified from the examples in the *GraphBLAS C API Specification*. Additional examples are provide in the `Demo` and `Extras` folders of SuiteSparse:GraphBLAS, including triangle counting, k-truss, and PageRank [21].

Finding the neighbors of a set $\mathcal{X}$ of nodes in a graph, for example, is identical to multiplying the sparse adjacency matrix $\mathbf{A}$ with a sparse vector $\mathbf{x}$ that has one entry for each node in the set $\mathcal{X}$, or $\mathbf{y} = \mathbf{A}'\mathbf{x}$ if a_{ij} is the edge (i, j). In conventional linear algebra, this matrix-vector multiplication would sum the entries to give a non-binary vector $\mathbf{y}$:

$$y_i = \sum_j a_{ji} x_j .$$

What is needed instead is the logical OR instead of the summation; this forms the additive operator of the monoid of the semiring. Instead of a multiply operator, a logical AND is needed instead. This is the multiplicative operator of the semiring. The result is the expression

$$y_i = \bigvee_j a_{ji} \wedge x_j ,$$

where y_i is true if node i is a neighbor of one or more nodes in $\mathcal{X}$. The structure of a sparse matrix-vector multiply is the same as the logical expression above; just the operators are different.

In many graph operations, the innermost loop also has a if-test. For example, in the breadth-first search problem, a node that has already been visited should be excluded from the set of nodes at the next level. This is done in GraphBLAS as a mask operation, $\mathbf{y}\langle\mathbf{m}\rangle = \mathbf{A}'\mathbf{x}$, where y_i can be modified only if the mask entry m_i is true.

Changing the semiring allows the expressiveness of linear algebra over semirings to be applied to graph algorithms. Matrix expressions can be rearranged with standard linear algebra, such as $(\mathbf{AB})' = \mathbf{B}'\mathbf{A}'$ and $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$, giving the graph algorithm developer a simple yet powerful method for writing a succinct graph algorithm and to reason about it. Methods such as matrix multiply, addition, transpose, submatrix extraction and assignment, and the mask also provide a set of bulk operations that can be given to a library to implement, with many operations to perform, as opposed to scalar operations such as a query to a graph to return an edge or a set of neighbors of a node. These bulk operations provide excellent scope for highly-tuned implementations inside a library such as SuiteSparse:GraphBLAS to obtain high performance, whether single-threaded, multi-threaded, distributed, or on the GPU. This approach has proven to be an effective method for accelerating real applications such as the RedisGraph graph database module of Redis (Section 6).

2.3 Related work

SuiteSparse:GraphBLAS is the first fully compliant implementation of the *GraphBLAS C API Specification*, and acts as the primary reference implementation (http://graphblas.org). At least six other projects are developing implementations of the GraphBLAS specification. One of them was recently completed: the GraphBLAS Template Library developed by Carnegie Mellon University, Indiana University, and the Pacific Northwest National Lab. The list includes:

- C++ CombBLAS: https://people.eecs.berkeley.edu/~aydin/CombBLAS/html/ [4].
- Java Graphulo: http://graphulo.mit.edu/ [14].
- MATLAB/Octave D4M: http://d4m.mit.edu/ [17].
- GPI (IBM) [11].
- GraphPad (Intel) [1, 22].

Table 1. SuiteSparse:GraphBLAS Objects

object	description
`GrB_Type`	a scalar data type
`GrB_UnaryOp`	a unary operator $z = f(x)$, where z and x are scalars
`GrB_BinaryOp`	a binary operator $z = f(x, y)$, where z, x, and y are scalars
`GxB_SelectOp`	a unary operator for constructing subgraphs
`GrB_Monoid`	an associative and commutative binary operator and its identity value
`GrB_Semiring`	a monoid that defines the "plus" and a binary operator that defines the "multiply" for an algebraic semiring
`GrB_Matrix`	a 2D sparse matrix of any type
`GrB_Vector`	a 1D sparse column vector of any type
`GrB_Descriptor`	parameters that modify an operation

- Gunrock: https://github.com/gunrock/gunrock-grb [23].
- GraphBLAS Template Library (Carnegie Mellon Univ., Indiana Univ., and PNNL): https://github.com/cmu-sei/gbtl. The implementation of the sequential version of the GraphBLAS Template Library meets the *GraphBLAS C API Specification*, but is currently focused on correctness over performance. It includes a wide range of graph algorithms that use its C++ API, which is also based on linear algebra over semirings. These include breadth-first search, single-source shortest-path methods, all-pairs shortest paths, three centrality metrics, clustering, triangle counting, k-truss enumeration, PageRank, maximal independent set, minimum spanning tree, max flow, and various other graph analytics.

3 SUITESPARSE:GRAPHBLAS METHODS AND OPERATIONS

3.1 Overview

SuiteSparse:GraphBLAS provides a collection of *methods* to create, query, and free each of its nine different types of objects: sparse matrices, sparse vectors, types, binary and unary operators, selection operators, monoids, semirings, and a descriptor object used for parameter settings. These are listed in Table 1. Once these objects are created they can be used in mathematical *operations* (not to be confused with the how the term *operator* is used in GraphBLAS). The `GxB_` prefix is used for all SuiteSparse extensions to the GraphBLAS API.

The GraphBLAS API makes a distinction between *methods* and *operations*. A method is a function that works on a GraphBLAS object, creating it, destroying it, or querying its contents. An operation (not to be confused with an operator) acts on matrices and/or vectors in a semiring. Each object is described below.

3.1.1 Types. A GraphBLAS type (`GrB_Type`) can be any of 11 built-in types (Boolean, integer and unsigned integers of sizes 8, 16, 32, and 64 bits, and single and double precision floating point). In addition, user-defined scalar types can be created from nearly any C `typedef`, as long as the entire type fits in a fixed-size contiguous block of memory (of arbitrary size). All of these types can be used to create GraphBLAS sparse matrices or vectors. All built-in types can typecasted as needed; user-defined types cannot.

3.1.2 Unary operators. A unary operator (`GrB_UnaryOp`) is a function $z = f(x)$. SuiteSparse:GraphBLAS comes with 67 built-in unary operators, such as $z = 1/x$ and $z = -x$, with variants for each built-in type. The user application can also create its own user-defined unary operators.

3.1.3 Binary operators. Likewise, a binary operator (`GrB_BinaryOp`) is a function $z = f(x, y)$, such as $z = x + y$ or $z = xy$. SuiteSparse:GraphBLAS provides 256 built-in binary operators, with variants for each built-in type. User-defined binary operators can also be created.

3.1.4 Select operators. The `GxB_SelectOp` operator is a SuiteSparse extension to the GraphBLAS API. It is used in the `GxB_select` operation to select a subset of entries from a matrix, like `L=tril(A)` in MATLAB. Its syntax is $z = f(i, j, m, n, a_{ij}, k)$ where a_{ij} is the numerical value of the entry in row i and column j in an m-by-n matrix. The parameter k is optional, and is used for operations analogous to the MATLAB statement `L=tril(A,k)`, where entries on or below the kth diagonal are kept and the rest discarded. The output z is true if the entry is to be kept, and false otherwise.

3.1.5 Monoids. The scalar addition of conventional matrix multiplication is replaced with a *monoid*. A monoid (`GrB_Monoid`) is an associative and commutative binary operator $z = f(x, y)$ where all three domains are the same (the types of x, y, z), and where the operator has an identity value o such that $f(x, o) = f(o, x) = x$. Performing matrix multiplication with a semiring uses a monoid in place of the "add" operator, scalar addition being just one of many possible monoids. The identity value of addition is zero, since $x + 0 = 0 + x = x$. GraphBLAS includes eight built-in operators suitable for use as a monoid: min (with an identity value of positive infinity), max (whose identity is negative infinity), add (identity is zero) multiply (with an identity of one), and four logical operators: AND, OR, exclusive-OR, and Boolean equality. User-created monoids can be defined with any associative and commutative operator with an identity value.

A monoid can also be used in a reduction operation, like `s=sum(A)` in MATLAB. MATLAB provides the plus, times, min, and max reductions of a real or complex sparse matrix as `s=sum(A)`, `s=prod(A)`, `s=min(A)`, and `s=max(A)`, respectively. In GraphBLAS, any monoid can be used (min, max, plus, times, AND, OR, exclusive-OR, equality, or any user-defined monoid, on any user-defined type).

3.1.6 Semirings. A *semiring* (`GrB_Semiring`) consists of an additive monoid and a multiplicative operator. Together, these operations define the matrix multiplication $\mathbf{C} = \mathbf{AB}$, where the monoid is used as the additive operator and the semiring's multiplicative operator is used in place of the conventional scalar multiplication in standard matrix multiplication via the plus-times semiring. A user application can define its own monoids and semirings.

A semiring can use any built-in or user-defined binary operator $z = f(x, y)$ as its multiplicative operator, as long as the type of its output, z matches the type of the semiring's monoid. The user application can create any semiring based on any types, monoids, and multiply operators, as long these few rules are followed.

Just considering built-in types and operators, SuiteSparse:GraphBLAS can perform `C=A*B` in 960 unique semirings. With typecasting, any of these 960 semirings can be applied to matrices `C`, `A`, and `B` of any of the 11 types, in any combination. In addition, each of the four matrices in the masked matrix multiplication $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$ can be in any of four formats (standard CSR and CSC, and hypersparse CSR and CSC). This gives $960 \times 11^3 \times 256 = 327,106,560$ possible kinds of sparse matrix multiplication supported by SuiteSparse:GraphBLAS, and this is counting just built-in types, operators, and data formats. By contrast, MATLAB provides just one data format (standard CSC), and two semirings for its sparse matrix multiplication `C=A*B`: plus-times-double and plus-times-complex, not counting the typecasting that MATLAB does when multiplying a real matrix times a complex matrix. All of the 327 million forms of matrix multiplication methods in SuiteSparse:GraphBLAS are typically just as fast as computing `C=A*B` in MATLAB using its own native sparse matrix multiplication methods. (Caveat: this author wrote the built-in `C=A*B` in the current version of MATLAB when `A` and/or `B` are sparse.)

3.1.7 Descriptor. A *descriptor* object, `GrB_Descriptor`, provides a set of parameter settings that modify the behavior of GraphBLAS operations, such as transposing an input matrix or complementing a `Mask` matrix (see Section 3.1.9 for details).

3.1.8 Non-blocking mode. GraphBLAS includes a *non-blocking* mode where operations can be left pending, and saved for later. This is very useful for submatrix assignment (`C(I,J)=A` in MATLAB notation, where `I` and `J` are integer vectors), or or scalar assignment (`C(i,j)=x` where `i` and `j` are scalar integers). Because of how MATLAB stores its matrices, adding and deleting individual entries is very costly. By contrast, this is very fast in SuiteSparse:GraphBLAS, taking only $O(e \log e)$ time:

```
GrB_Matrix A ;
GrB_Matrix_new (&A, GrB_FP64, m, n) ;
for (int k = 0 ; k < e ; k++)
{
    compute a value x, row index i, and column index j
    // A (i,j) = x
    GrB_Matrix_setElement (A, x, i, j) ;
}
```

but the equivalent method in MATLAB takes $O(e^2)$ time:

```
A = sparse (m,n) ;   % an empty sparse matrix
for k = 1:e
    compute a value x, row index i, and column index j
    A (i,j) = x ;
end
```

Building a matrix all at once via `GrB_Matrix_build` in SuiteSparse:GraphBLAS and via `sparse` in MATLAB is equally fast; both methods below take $O(e \log e)$ time, in SuiteSparse:GraphBLAS:

```
for (int k = 0 ; k < e ; k++)
{
    compute a value x, row index i, and column index j
    I [k] = i ;
    J [k] = j ;
    X [k] = x ;
}
GrB_Matrix A ;
GrB_Matrix_new (&A, GrB_FP64, m, n) ;
GrB_Matrix_build (A, I, J, X, e, GrB_SECOND_FP64) ;
```

And in MATLAB:

```
for k = 1:e
    compute a value x, row index i, and column index j
    I (k) = i ;
    J (k) = j ;
    X (k) = x ;
end
A = sparse (I,J,X,m,n) ;
```

Exploiting non-blocking mode, SuiteSparse:GraphBLAS can do both incremental and all-at-once methods equally fast, but in MATLAB only the all-at-once method is efficient. Allowing for fast

incremental updates allows the user to write simpler code to construct a matrix, and enables much faster incremental updates to a matrix that has already been constructed.

3.1.9 The accumulator and the mask. Most GraphBLAS operations can be modified via transposing input matrices, using an accumulator operator, applying a mask or its complement, and by clearing all entries the matrix `C` after using it in the accumulator operator but before the final results are written back into it. All of these steps are optional, and are controlled by a descriptor object that holds parameter settings that control the following options:

- the input matrices `A` and/or `B` can be transposed first.
- an accumulator operator can be used, like the plus in the MATLAB statement `C=C+A*B`. The accumulator operator can be any binary operator, and an element-wise "add" (set-union) is performed using the operator.
- an optional *mask* can be used to selectively write the results to the output. The mask is a sparse Boolean matrix `Mask` whose size is the same size as the result. If `Mask(i,j)` is true, then the corresponding entry in the output can be modified by the computation. If `Mask(i,j)` is false, then the corresponding entry in the output is protected and cannot be modified by the computation. The `Mask` matrix acts exactly like logical matrix indexing in MATLAB, with one minor difference: in GraphBLAS notation, the mask operation is $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$, where the mask $\mathbf{M}$ appears only on the left-hand side. In MATLAB, it would appear on both sides as `C(Mask)=Z(Mask)`. If no mask is provided, the `Mask` matrix is implicitly all true. This is indicated by passing `GrB_NULL` as the `Mask` argument in GraphBLAS operations.

This process can be described in mathematical notation as:

$\mathbf{A} = \mathbf{A}'$, if requested via descriptor (first input option)
$\mathbf{B} = \mathbf{B}'$, if requested via descriptor (second input option)
$\mathbf{T}$ is computed according to the specific operation
$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$, accumulating and writing the results back via the mask $\mathbf{M}$

The application of the mask and the accumulator operator is written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ where $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ denotes the application of the accumulator operator, and $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ denotes the masked assignment via the Boolean matrix $\mathbf{M}$. The expression $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ is computed as follows:

if no accumulator operator, $\mathbf{Z} = \mathbf{T}$; otherwise $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$
if requested via descriptor (replace option), all entries cleared from $\mathbf{C}$
if `Mask` is `NULL`
    $\mathbf{C} = \mathbf{Z}$ if `Mask` is not complemented; otherwise $\mathbf{C}$ is not modified
else
    $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ if `Mask` is not complemented;
    otherwise $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{Z}$

The accumulator operator ($\odot$) acts like a sparse matrix addition, except that any operator can be used. The pattern of $\mathbf{C} \odot \mathbf{T}$ is the set-union of the patterns of $\mathbf{C}$ and $\mathbf{T}$, and the operator is applied only on the set-intersection of $\mathbf{C}$ and $\mathbf{T}$. Entries in neither the pattern of $\mathbf{C}$ nor $\mathbf{T}$ do not appear in the pattern of $\mathbf{Z}$. That is:

for all entries (i, j) in $\mathbf{C} \cap \mathbf{T}$ (that is, entries in both $\mathbf{C}$ and $\mathbf{T}$)
    $z_{ij} = c_{ij} \odot t_{ij}$
for all entries (i, j) in $\mathbf{C} \setminus \mathbf{T}$ (that is, entries in $\mathbf{C}$ but not $\mathbf{T}$)
    $z_{ij} = c_{ij}$
for all entries (i, j) in $\mathbf{T} \setminus \mathbf{C}$ (that is, entries in $\mathbf{T}$ but not $\mathbf{C}$)
    $z_{ij} = t_{ij}$

Table 2. SuiteSparse:GraphBLAS Operations

function name	description	GraphBLAS notation
`GrB_mxm`	matrix-matrix mult.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix mult.	$\mathbf{w}'\langle\mathbf{m}'\rangle = \mathbf{w}' \odot \mathbf{u}'\mathbf{A}$
`GrB_mxv`	matrix-vector mult.	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise, set-union	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise, set-intersection	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{i}, \mathbf{j})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
`GrB_assign`	assign submatrix	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i}, \mathbf{j}) = \mathbf{C}(\mathbf{i}, \mathbf{j}) \odot \mathbf{A}$ $\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GxB_subassign`	assign submatrix	$\mathbf{C}(\mathbf{i}, \mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i}, \mathbf{j}) \odot \mathbf{A}$ $\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
`GxB_select`	apply select op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, \mathbf{k})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, \mathbf{k})$
`GrB_reduce`	reduce to vector reduce to scalar	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$ $s = s \odot [\oplus_{ij} \mathbf{A}(i, j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$
`GxB_kron`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{kron}(\mathbf{A}, \mathbf{B})$

3.2 GraphBLAS methods and operations

The matrix (`GrB_Matrix`) and vector (`GrB_Vector`) objects include additional methods for setting a single entry, extracting a single entry, making a copy, and constructing an entire matrix or vector from a list of *tuples*. The tuples are held as three arrays `I`, `J`, and `X`, which work the same as `A=sparse(I,J,X)` in MATLAB, except that any type matrix or vector can be constructed, and any operator can be used to combine duplicates. A complete list of methods can be found in the User Guide provided with the software package.

Table 2 lists all SuiteSparse:GraphBLAS operations in the GraphBLAS notation where $\mathbf{AB}$ denotes the multiplication of two matrices over a semiring. Upper case letters denote a matrix, and lower case letters are vectors. Each operation takes an optional `GrB_Descriptor` argument that modifies the operation. The input matrices $\mathbf{A}$ and $\mathbf{B}$ can be optionally transposed, the mask $\mathbf{M}$ can be complemented, and $\mathbf{C}$ can be cleared of its entries after it is used in $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ but before the $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ assignment. Vectors are never transposed via the descriptor.

The notation $\mathbf{A} \oplus \mathbf{B}$ denotes the element-wise operator that produces a set-union pattern (like `A+B` in MATLAB). The notation $\mathbf{A} \otimes \mathbf{B}$ denotes the element-wise operator that produces a set-intersection (like `A.*B` in MATLAB). Reduction of a matrix $\mathbf{A}$ to a vector reduces the ith row of $\mathbf{A}$ to a scalar w_i, like `w=sum(A')` in MATLAB.

4 SUITESPARSE:GRAPHBLAS IMPLEMENTATION

The GraphBLAS API provides a great deal of flexibility in its implementation details. All of its objects are opaque, and their contents can only be modified by calling GraphBLAS functions. This

section describes how the GraphBLAS API is implemented in the SuiteSparse:GraphBLAS software package. As a reference implementation, the library is entirely self-contained, relying no other matrix or graph library except for a threading library (OpenMP or POSIX) for the thread-safety of user threads, and the standard C library functions (such as `malloc`, `memcpy`, `printf`, etc).

4.1 Matrix and vector data structure

The GraphBLAS matrix and vector objects are opaque to the end-user, and can only be accessed via GraphBLAS methods and operations. Their implementation has a significant impact on the performance of GraphBLAS, so an overview of the data structure is given here.

In SuiteSparse:GraphBLAS (version 2.2.2), a GraphBLAS matrix (the `GrB_Matrix` object) is stored in one of four different formats: compressed-sparse column (standard CSC), compressed-sparse row (standard CSR), and hypersparse versions of these two formats (hyper CSR and hyper CSC). A `GrB_Vector` is always stored in standard CSC format, as a single column vector.

The basics of the standard CSC data structure are identical to the MATLAB sparse matrix [12], and also the sparse matrix format used in CSparse [7]. For a matrix `A` of size `m`-by-`n`, an integer array `A.p` of size `n+1` holds the column "pointers", which are references to locations in the `A.i` and `A.x` arrays, each of which are of size equal to, or greater, than the number of entries in the matrix. The row indices of entries in column `j` of `A` are held in `A.i[A.p[j]...A.p[j+1]-1]`, and the values are held in the same positions in the `A.x` array. Row and column indices are zero-based, which matches the internal data structure of a MATLAB sparse matrix. The MATLAB user is provided the illusion of 1-based indexing in a MATLAB M-file, whereas the GraphBLAS user sees a 0-based indexing. To facilitate submatrix extraction (`C=A(I,J)`) and assignment (`C(I,J)=A`), row indices are always kept sorted, just like MATLAB. No duplicate entries are held in this format. The total memory space for the standard CSC format is $O(n + e)$ if the matrix is m-by-n with e entries.

The standard CSR format is identical to the standard CSC format, except it stores the matrix as a set of sparse row vectors. The CSR format (either standard or hypersparse) is the default format for all `GrB_Matrix` objects. This is because most graph algorithms for directed graphs traverse the outgoing edges of a node, and `A(i,j)` is natural to use for the edge (i, j). The total memory space for the standard CSR format is $O(m + e)$ if the matrix is m-by-n with e entries.

Both of these standard formats require memory space that is proportional to one of the matrix dimensions. This is fine for numerical computations, since a matrix with fewer than n or m entries would be singular. However, graphs and sub-graphs can often be hypersparse, with fewer entries than the number of nodes ($e << \min(m, n)$). To accommodate this, the package includes hypersparse forms of the CSC and CSR formats [3]. In a hypersparse format, the pointer array `A.p` itself becomes sparse. It is replaced with two arrays: `A.p` and `A.h` of length `A.plen+1`, where `A.p` is a pointer for each non-empty vector (row vectors for hyper CSR or column vectors for hyper CSC), and `A.h` is a list of indices of row or column vectors that contain at least one entry. Both the hyper CSR and hyper CSC formats require only $O(e)$ memory.

SuiteSparse:GraphBLAS always uses the CSR format, unless the user application requests a CSC format. It selects automatically between hypersparse or standard CSR, based on the number of non-empty rows, and converts between the two as needed. The user application can modify this heuristic, either for all matrices or for selected matrices. It can modify the default format for all matrices, and it can force all matrices or any particular matrix to always remain in standard or hypersparse format.

The type of the entries of `A` can be almost anything. GraphBLAS provides 11 built-in types (boolean, integers and unsigned integers of size 8, 16, 32, and 64 bits, and single and double precision floating-point). In addition, a user application can define any new type, without requiring a recompilation of the GraphBLAS library. The only restriction on the type is that it must have

a constant size, and it must be possible to copy a value with a single `memcpy`. MATLAB provides sparse logical, double, and complex double. GraphBLAS can be easily extended to handle complex double sparse matrices via a user-defined complex type; an example is given in the `Demo` folder of SuiteSparse:GraphBLAS.

MATLAB drops its entries with a numerical value of zero, but this is never done in GraphBLAS. A "zero" is simply an entry that is not stored in the data structure, and the value of this implicit entry depends on the semiring. If the matrix is used in the conventional plus-times semiring, the implicit value is zero. If used in max-plus, the implicit entry is $-\infty$. This value is not stored in `A`; it depends on the semiring, and any matrix can be used in any semiring. Thus, dropping a numerical entry whose value happens to be zero cannot be automatically done in GraphBLAS. It can of course be done as an operation executed in a user application, if desired.

SuiteSparse:GraphBLAS adds an additional set of features to this data structure: *zombies* and *pending tuples*, which enable fast insertion and deletion of entries. These features enable submatrix assignment to be many times faster the equivalent operations in MATLAB, even when blocking mode is used.

A *zombie* is an entry in the data structure that has been marked for deletion. This is done by "negating" the row index of the entry. More precisely, its row index `i` is changed to `(-i-2)`, to accommodate zero-based row indices. Zombies allow for fast deletion, and they also permit binary searches of a sparse vector to performed, even if it contains zombies.

A *pending tuple* is an entry that has not yet been added to the compressed-sparse vector part of the data structure. Pending tuples are held in an unsorted list of row indices, column indices, and values. Duplicates may appear in this list. The matrix also keeps track of a single operator to be used to combine duplicate entries. A matrix can have both zombies and pending tuples.

Temporary matrices, internal in SuiteSparse:GraphBLAS, need not have their own `p`, `h`, `i`, and/or `x` arrays. Any or all of these can be shallow pointers to the components of other matrices. This feature facilitates operations such as typecasting a user input matrix to a required type. In this case, only `x` needs to be constructed for the temporary typecasted matrix, while the pattern (`p`, `h`, and `i`) are shallow copies of the input matrix.

A final component of the `GrB_Matrix` and `GrB_Vector` is an optional workspace that is allocated if `C` is created via `C=A*B` when using Gustavson's method for sparse matrix-matrix multiply [13]. The space is $O(n)$ for the CSR format, or $O(m)$ for the CSC format. This space is essential for obtaining the right asymptotic time complexity, particular when multiplying a sparse matrix times a sparse vector. The first time `C=A*B` is computed, the time taken is $\Omega(n)$ (for CSR) or $\Omega(m)$ (for CSC) to allocate and initialize this space. Subsequent computations of `C=A*B` do not have this n or m component in their time complexity.

A GraphBLAS vector (`GrB_Vector`) is simply held as an `n`-by-1 matrix, although the two types (`GrB_Matrix` and `GrB_Vector`) are different and unique types in the user-level application. Internally, any `GrB_Vector` can be typecasted into a `GrB_Matrix`, and while this is technically available to the GraphBLAS user application, it is not documented, not guaranteed to work in all future versions of the library, and not part of the *GraphBLAS C API Specification.*

4.2 Matrix operations

Details of the matrix multiply, element-wise, submatrix extraction, and submatrix assignment operations are given below. Unless specified otherwise, the matrix $\mathbf{C}$ is m-by-n. For this discussion, the matrices are assumed to be in CSC format, either standard or hypersparse. The CSR versions of the algorithms are the transpose of the CSC algorithms. The `GrB_apply`, `GrB_reduce`, `GrB_transpose`, `GxB_select`, and `GxB_kron` operations are not described here since their implementation is fairly straight-forward.

All methods can operate on all four matrix formats in any combination. For example, the four matrices for $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ give rise to 256 variants of the sparse matrix-matrix multiply. The user interface for all variants is the same, and the user application need not know or care what formats are being used internally, other than to experience their impact on performance of the user application.

4.2.1 Matrix multiply. Multiplying two sparse matrices (`GrB_mxm`), matrix-vector multiply (`GrB_mxv`), or vector-matrix multiply (`GrB_vxm`) rely on three methods:

(1) a variant of Gustavson's algorithm [13],
(2) a heap-based method [3], and
(3) a dot-product formulation.

Each of the methods has a masked variant that computes $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, and variants that operate on hypersparse matrices. The methods operate on both CSR and CSC format, but the discussion below assumes the CSC format.

In the first method, when no mask is present, the work is split into a symbolic analysis phase that finds the pattern of $\mathbf{C}$ and a numerical phase that computes its values. To multiply $\mathbf{C} = \mathbf{AB}$ where $\mathbf{C}$ is m-by-n, $\mathbf{A}$ is m-by-k and $\mathbf{B}$ is k-by-n, both phases take only $O(n + f)$ time, assuming all matrices are in CSC format, and assuming the $O(m)$ workspace is already allocated and initialized, where f is the number of "multiply-adds" computed (in the semiring). If the $O(m)$ workspace has already been allocated by a prior matrix multiplication, the terms m and k do not appear in the time complexity, which is very important when n and f are both much less than m or k.

A breadth-first search, for example, must compute $\mathbf{c} = \mathbf{Ab}$ where $\mathbf{b}$ is a sparse vector. The result $\mathbf{c}$ is the set of neighbors of all nodes i where b_i is true. This set of neighbors is much smaller than the number of nodes, m, in the graph. Assuming the workspace of size $O(m)$ has already been allocated and initialized, the time to compute this set is simply $O(f)$, or the sum total of the adjacency sets for all nodes in the current level, represented by $\mathbf{b}$ (a column vector, so $n = 1$).

To obtain this time $O(f)$ that does not depend on m, SuiteSparse:GraphBLAS maintains two of pre-allocated workspaces of size m, used internally and reused for subsequent operations. One is uninitialized, and used for numerical gather/scatter operations in the numerical phase. The other is an initialized integer array, `mark`, that is used for the set-union computation in the symbolic phase, and for the mask `M` in the numerical phase. The entire space is cleared in constant time. When initialized, `mark[i]<flag` holds for all `i`; to set `mark[i]` as true, `mark[i]=flag` is done, and clearing the entire `mark` array simply requires `flag` to be incremented. Both workspaces are allocated if they are not present, this is be done just once for the entire breadth-first search, for example. Thus, in an amortized sense, these size m arrays do not contribute an $O(m)$ component to the time complexity of computing $\mathbf{C} = \mathbf{AB}$ in each iteration. The $O(m)$ work appears just once in the entire breadth-first search algorithm. If m is large compared with $|\mathbf{A}| + |\mathbf{B}|$, Gustavson's method is not used, and the heap-based method is used instead.

Both $\mathbf{C}$ and $\mathbf{B}$ have n columns, and even if many of those columns are entirely empty, the time complexity of $O(n + f)$ still depends on n, if the matrices are in standard format. A hypersparse format need only operate on the non-empty columns of $\mathbf{B}$ and $\mathbf{C}$, however, so the time complexity drops to $O(\bar{n}_B + f)$ where $\bar{n}_B < n$ is the number of non-empty columns of $\mathbf{B}$.

Constructing $\mathbf{C}$ takes $\Omega(n)$ time and space if it is stored in standard compressed sparse-column form with a pointer array `C.p` of size $n + 1$. The term n could dominate in some cases, if n is much larger than f. Thus SuiteSparse:GraphBLAS stores its matrices in hypersparse format if $\bar{n} < n/16$.

The result of computing the matrix product $\mathbf{AB}$ may be written into $\mathbf{C}$ via a mask, $\mathbf{M}$, via $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$. If the mask is present (and not complemented), only the subset of entries appearing in the mask are computed. This greatly reduces the time and memory usage. In this method, the symbolic analysis is skipped. A matrix $\mathbf{T} = \mathbf{AB}$ is computed whose pattern is assumed to be a subset

of the mask matrix **M**. Entries in **AB** outside the mask need not be computed, and are discarded if they are computed.

For example, if **L** is the strictly lower triangular part of an unweighted graph **A**, then $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{L}^2$ finds the number of triangles in the graph, where c_{ij} is the number of triangles the containing the edge (i, j) [2]. Not all of $\mathbf{L}^2$ is computed or stored, but only the entries corresponding to entries in the mask, **L**. This greatly reduces the time and memory complexity of the masked matrix multiply, as compared with computing all of $\mathbf{L}^2$ first and then applying the mask, as would be done in the MATLAB expression `C=(L^2).*L`.

The $O(m)$ workspace required for Gustavson's method is kept in the output matrix `C`. It could be kept outside the matrix, in global workspace or in thread-local workspace, and reused for different matrices, but using a global workspace would mean that GraphBLAS would not be thread-safe if the user application is multithreaded. Thread-local workspace adds complexity to the implementation, and would be more space if more than one user thread were to operate on the matrix. It is thus kept in matrix `C`. It can be freed by the user application if desired.

When this initialized workspace is available, Gustavson's method takes $O(\bar{n} + f)$ time when the matrices are hypersparse, or $O(n + f)$ when they are in standard CSC format. The $O(m)$ size of the workspace is prohibitive if $e << m$, however. Thus, a heap-based method is also implemented [3]. It constructs each non-empty column j of the output **C** as a k-way merge, if column $\mathbf{B}_{*j}$ has k entries. The merge is managed with a heap of size k, where $k << m$ typically holds. As a result, the workspace of the heap-based method is very small. The time complexity to compute a single column $\mathbf{C}_{*j}$ is no more than $O(f_j \log f_j)$, if computing this column takes $O(f_j)$ numeric operations. Thus, the total work is $O(f \log f)$ which can be much smaller than $O(n + f)$ if $f << n$, and the memory space is much smaller. None of the matrix dimensions m or n appear in the time or memory complexity of the heap-based method. As a result, a matrices of size 2^{60}-by-2^{60} can easily be created and operated on in as little as $O(1)$ time and space, if the matrices have $O(1)$ entries.

SuiteSparse:GraphBLAS uses a complex heuristic to select between these three methods. Gustavson's method works best when the matrices are not hypersparse and if its $O(m)$ workspace does not dominate the memory space required. The heap-based method is best for hypersparse matrices, or when **A** or **B** are diagonal or permutation matrices. The dot product formulation is useful in very specific cases. It can be very fast if **C** is small compared to **A** and **B**, or if $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ is computed with a very sparse mask. Since it is difficult to always make the best choice automatically, the user application can also select the method to be used via the `GrB_Descriptor`.

Currently, matrix multiply and submatrix assignment (`GrB_mxm`, `GrB_mxv`, `GrB_vxm`, `GrB_assign`, and `GxB_subassign`) are the only operations that exploit the mask during computations. All other operations compute their result and only then do they write their results into the output matrix via the mask. Future versions of SuiteSparse:GraphBLAS may exploit the mask more fully to reduce the time and space required.

4.2.2 Element-wise operations. Element-wise operations in GraphBLAS can use any binary operator. They are applied in either a set-union (`GrB_eWiseAdd`) or set-intersection (`GrB_eWiseMult`) manner. The set-union is like a sparse matrix addition in MATLAB, `C=A+B`. If an entry appears in just `A` or `B`, the value is copied into `C` without applying the binary operator. The set-intersection is like `C=A.*B` in MATLAB, which is the Hadamard matrix product if the binary multiply operator is used. For both methods, however, any binary operator can be used, which is only applied to entries in the set-intersection.

The implementation of `GrB_eWiseAdd` is straightforward, taking $O(n + |\mathbf{A}| + |\mathbf{B}|)$ time where n is the number of columns in the three matrices, if the matrices are in standard CSC format. Each column of **C** is computed with a merge of the corresponding columns of **A** and **B**, since columns

are always kept with sorted row indices. Unless the matrices have fewer than $O(n)$ entries, this time is optimal. The time is optimal for the hypersparse case: $O(|\mathbf{A}| + |\mathbf{B}|)$.

The `GrB_eWiseMult` operation is more subtle, since the lower bound on the ideal time is the size of the set-intersection, $\Omega(|\mathbf{C}|)$, which is smaller than either $|\mathbf{A}|$ or $|\mathbf{B}|$. For each column j, if the number of nonzeros in $\mathbf{A}_{*j}$ and $\mathbf{B}_{*j}$ are similar, then a conventional merge is used, taking $O(|\mathbf{A}_{*j}| + |\mathbf{B}_{*j}|)$ time. This is just like set-union "add," except that entries outside the set-intersection are not copied into $\mathbf{C}$. Suppose instead that the jth column of $\mathbf{A}$ has far fewer entries than the jth column of $\mathbf{B}$. In this case, during the merge, each entry a_{ij} is examined and a trimmed binary search is used to find the corresponding entry in $\mathbf{B}_{*j}$. The search is trimmed since $\mathbf{B}_{*j}$ is sorted, and once an entry i is found this result is used to reduce the search for subsequent row indices $i' > i$.

Additional tests for special cases reduce the time even further. For example, if the last entry in column $\mathbf{A}_{*j}$ comes before the first entry in $\mathbf{B}_{*j}$, then the intersection is empty and no traversal is done of either $\mathbf{A}_{*j}$ or $\mathbf{B}_{*j}$.

These tests are performed column-by-column. In the extreme case, if $\mathbf{A}$ is very sparse and $\mathbf{B}$ is completely dense, the time complexity of `GrB_eWiseMult` is $O(n + |\mathbf{A}| \log m)$. This is much smaller than the $O(n^2)$ time that would result if a simple merge is used, like the set-union "add" method used in `GrB_eWiseAdd`. For the hypersparse case, the $O(n)$ term drops from the time complexity, and is replaced with $O(\bar{n}_A + \bar{n}_B)$ if $\bar{n}_A$ and $\bar{n}_B$ are the number of non-empty columns of $\mathbf{A}$ and $\mathbf{B}$.

4.2.3 Submatrix extraction. `GrB_extract` extracts a submatrix, $\mathbf{C} = \mathbf{A}(\mathbf{i}, \mathbf{j})$ where $\mathbf{i}$ and $\mathbf{j}$ are integer vectors, like `C=A(I,J)` in MATLAB notation. It is a meta-algorithm that selects different methods for traversing the matrix $\mathbf{A}$, depending on the length $|\mathbf{i}|$ of the row index list $\mathbf{i}$ and the number of entries in a given column of the matrix $\mathbf{A}$. The eleven cases are shown in Table 3.

A CSC matrix `C` is built one column `C(:,k)` at a time, by examining `A(:,j)` where `j=J[k]`. Let $a = |\mathbf{A}_{*j}|$ be the number of entries in the jth column of $\mathbf{A}$, and let $c = |\mathbf{C}_{*k}|$ be the number of entries in the kth column of $\mathbf{C}$. If `I` has duplicate entries, then $c > a$ is possible; otherwise $c \le a$ must hold.

The GraphBLAS standard allows for two different ways of specifying the index sets `I` and `J`. An index set can either be an explicit list of integers (one per index), or it can be a special value `GrB_ALL`, which acts just like the colon in `A(:,j)` in MATLAB.

This can be problematic if `A` is hypersparse. Suppose `A` has dimension n-by-n with $n = 2^{60}$, but contains only a million entries. Extracting the top left submatrix, `C=A(1:n/2,1:n/2)`, would require the creation of an index list `I` of size $n/2$, which is impossible. Thus, SuiteSparse:GraphBLAS provides a mechanism similar to the MATLAB colon notation, `lo:stride:hi`, which can be specified with only three integers and can thus be used for arbitrarily large matrices.

If `I` is a large explicit list of size $m/256$ or larger, then a multiset inverse of `I` is created in $O(m+|\mathbf{i}|)$ workspace, where m is the number of rows of `A`, taking $O(m + |\mathbf{i}|)$ time. The inverse is a multiset if `I` contains duplicate entries. This workspace is needed only for cases 9 to 11, below. This is the only workspace needed for `GrB_extract`, and it is not allocated until the first time cases 9 to 11 are triggered.

Except for cases 1, 2, and 4, the method starts with a binary search for the first row index, `I[0]`, taking at most $O(\log a)$ time. The method then considers eleven cases for each column of `C` and `A`; the first case that holds for a particular column `j` is utilized. The time shown in Table 3 excludes the time to invert `I` for cases 9 to 11. There is no simple expression for the total time to construct all of $\mathbf{C} = \mathbf{A}(\mathbf{i}, \mathbf{j})$, since each column is treated differently in one of the above eleven cases. The lower bound of $\Omega(|\mathbf{C}|)$ can be obtained for some simple cases under modest assumptions. Other cases such as `C=A(imin:imax,J)` are nearly optimal, taking no more than $O(|\mathbf{C}| + |\mathbf{j}| \log \max a)$ time, where $\max a$ is the number of entries in the densest column of `A(:,J)`, and where the list `J` can have any format (`GrB_ALL`, a colon-like expression, or an explicit list).

Table 3. Meta-algorithm for C(:,k)=A(I,j)

case	description	$O(\ldots)$
1	`A(:,j)` is dense and `I` is `GrB_ALL` (the colon `A(:,j)` in MATLAB notation): the entire column is copied.	c
2	`A(:,j)` is dense and `I` has any form: `C(:,k)=A(I,j)`, by scanning each index in `I`	c
3	`I` is a single row index: the entry is extracted in $O(1)$ time.	$\log a$
4	`A(:,j)` is sparse and `I` is `GrB_ALL`: the entire column is copied.	c
5	`I` is a contiguous list of stride 1 (`C(:,k)=A(imin:imax,j)`): the entire column is copied starting at `imin` until reaching `imax`.	$\log a + c$
6	The inverse of `I` is too costly to compute, or the size of `I` is small compared with the number of entries in `A(:,j)`: a binary search in `A(:,j)` is performed for each entry in `I`. If found, the entry is appended to `C(:,k)`. No sort is needed. Case 6 is faster than cases 9 to 11 when $\|\mathbf{i}\|$ is smaller than a.	$\|\mathbf{i}\| \log a$
7	`I` is given using a colon-like notation, `I=imin:stride:imax`, with `stride` >= 0. Each entry in `A(:,j)` is scanned. Testing if this row index appears in `I` and finding the new row index in `C(:,k)`, takes $O(1)$ time. No sort is needed.	a
8	The same as case 7, except `stride` < 0. The entries in `A(:,j)` are scanned in reverse order. No sort is needed.	a
9	The list `I` is not in order (and may include duplicates): all of `A(:,j)` is examined. If `A(i,j)` is nonzero, then all row indices `t` for which `t=I[k]` (if any) are appended to the column `C(:,k)`, along with their positions in `A(:,j)`. This traversal relies on the multi-inverse of `I`. Once the traversal of the pattern of `A(:,j)` is completed, the row indices and positions in `C(:,k)` are sorted. Next, the corresponding numerical values are copied from `A(I,j)` to `C(:,k)`. The time to invert `I` is $O(m + \|\mathbf{i}\|)$ time, but this is done only once.	$c \log c + a$
10	The list `I` is sorted but has duplicates: all of `A(:,j)` is examined. This method is just like case (9), except that no sort is needed, and it is done in a single pass.	$c + a$
11	the list `I` is sorted, with no duplicates (a simplified version of case (10)).	a

4.2.4 Submatrix assignments. The `GrB_assign` and `GxB_subassign` operations modify a submatrix of `C`. The GraphBLAS operation `GrB_assign` computes $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i},\mathbf{j}) = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$, where the mask matrix $\mathbf{M}$ has the same size as $\mathbf{C}$. SuiteSparse:GraphBLAS adds an extension to the GraphBLAS specification, via the operation `GxB_subassign`, which computes $\mathbf{C}(\mathbf{i},\mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$. In this case, the mask matrix is the same size as the submatrix $\mathbf{C}(\mathbf{i},\mathbf{j})$. Details of the two operations are shown in Table 4.

These two operations are the most intricate operations in SuiteSparse:GraphBLAS, taking 3,908 lines of C to implement and 3,984 lines of comments to describe (not including support functions also used for other operations, such as typecasting). They fully exploit non-blocking mode to obtain high performance.

If a sequence of assignments uses the same accumulator operator, portions of the assignment are postponed, via zombies and pending tuples. In most cases, the method starts with a specialized form of submatrix extraction, $\mathbf{S} = \mathbf{C}(\mathbf{i},\mathbf{j})$, where the scalar entries in $\mathbf{S}$ are pointers to where the

Table 4. GrB_assign and GxB_subassign

Step	GrB_assign	GxB_subassign
1	$\mathbf{S} = \mathbf{C}(\mathbf{I},\mathbf{J})$	$\mathbf{S} = \mathbf{C}(\mathbf{I},\mathbf{J})$
2	$\mathbf{S} = \mathbf{S} \odot \mathbf{A}$	$\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{A}$
3	$\mathbf{Z} = \mathbf{C}$	$\mathbf{C}(\mathbf{I},\mathbf{J}) = \mathbf{S}$
4	$\mathbf{Z}(\mathbf{I},\mathbf{J}) = \mathbf{S}$	
5	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	

entries reside in **C**. Next $\mathbf{S} = \mathbf{S} \odot \mathbf{A}$ is computed, where entries that appear in both **S** and **A** are modified directly in **C** (if such an entry appears as a zombie in **C**, it comes back to life). Entries that appear in **S** but not **A** become zombies in **C**. Entries that appear in **A** but not **S** become pending tuples in **C**.

No workspace is required beyond that needed by the submatrix extraction to compute **S**. As a result, if **C** is m-by-n, in most cases no size m or size n workspace is needed (unless **S** is built, using cases 9 to 10 of Table 3). This makes the method much more efficient for very large, very sparse matrices, as compared to `C(I,J)=A` in MATLAB, even when the non-blocking mode is not exploited. In particular, for one square matrix `C` of dimension 3 million, containing 14.3 million nonzeros, `C(I,J)=A` takes 87 seconds in MATLAB on a MacBook Pro but only 0.74 seconds in SuiteSparse:GraphBLAS (where `A` is 5500-by-7000 with 38,500 nonzeros, and `I` and `J` are randomly generated). This 0.74 seconds includes the time to return the result back to MATLAB as a valid MATLAB sparse matrix with all zombies and pending tuples removed.

The fast incremental update of a matrix is one of the reasons RedisGraph obtains such high performance (see Section 6). A common use case for graph analytics is to add a batch of updates to a matrix/graph, by modifying, adding, and deleting edges. With the algorithm used in SuiteSparse:GraphBLAS, each scalar update to a sparse vector of length k takes just $O(\log k)$ time, with the rest of the update of the entire matrix left pending. Eventually when the matrix needs to be completed with all zombies deleted and pending tuples assembled, the extra work is no more than $O(n + e + p \log p)$, where e is the number of entries in the original matrix (prior to any updates) and p is the number of pending tuples waiting to be added, assuming the matrix is in standard CSC or CSR form. The $O(n)$ component does not appear if the matrix is hypersparse.

The `GrB_Matrix_setElement` method modifies a single entry, `C(i,j)=x` where `i` and `j` are scalars. It first performs a binary search of column `C(:,j)` to see if the entry is present. If so, the value is modified; a zombie may come back to life in this case. If it is not present, the entry is added to the list of pending tuples.

4.3 Compile-time user-defined objects

The nine different objects in Table 1 are all created at run-time. This provides a great deal of flexibility for constructing new types and semirings, but it comes at the cost of performance. For example, computing `C=A*B` with sparse double precision matrices is about 10% faster than the same computation in MATLAB, using the `GrB_FP64` built-in type. GraphBLAS does not have a built-in complex type. It can be constructed at run-time, but the resulting `C=A*B` computation in `GxB_mxm` is 2 to 3 times slower than the same computation in MATLAB.

To address this problem, SuiteSparse:GraphBLAS includes a mechanism for creating user-defined types, operators, monoids, and semirings when SuiteSparse:GraphBLAS is compiled (the first six objects in Table 1). Matrix, vectors, and descriptors cannot be defined at compile time. The user

```
#ifdef GxB_USER_INCLUDE
    #include <complex.h>
    static inline void my_complex_plus (double complex *z,
        const double complex *x, const double complex *y)
    {
        (*z) = (*x) + (*y) ;
    }
    static inline void my_complex_times (double complex *z,
        const double complex *x, const double complex *y)
    {
        (*z) = (*x) * (*y) ;
    }
#endif

// the complex type: My_Complex, based on the ANSI C11 double complex type
GxB_Type_define(My_Complex, double complex) ;

// My_Complex_plus: add two complex values
GxB_BinaryOp_define(My_Complex_plus, my_complex_plus, My_Complex, My_Complex, My_Complex) ;

// My_Complex_plus: multiply two complex values
GxB_BinaryOp_define(My_Complex_times, my_complex_times, My_Complex, My_Complex, My_Complex) ;

// My_Complex_plus_monoid: complex addition, with an identity of 0+0*i
GxB_Monoid_define(My_Complex_plus_monoid, My_Complex_plus, CMPLX(0,0)) ;

// My_Complex_plus_times: the complex-plus-times semiring
GxB_Semiring_define(My_Complex_plus_times, My_Complex_plus_monoid, My_Complex_times) ; }
```

Fig. 1. Compile-time construction of the double complex plus-times semiring

simply writes a short `m4` script using six `m4` macros of the form `GxB_*_define`, which mimic the `GrB_*_new` functions that construct these objects at run-time.

An example is shown in Figure 1. With this mechanism, sparse complex double precision matrices can be computed in GraphBLAS just as quickly as they can in MATLAB, or slightly faster. The operation `C=A*B` with the pre-defined complex plus-times semiring in GraphBLAS is about 10% faster than `C=A*B` in MATLAB, as compared to 2x to 3x slower when the same complex plus-times semiring is created a run-time.

An alternative strategy would be to use C++ templates, but the GraphBLAS API is in C, not C++.

4.4 Testing

Testing is a vital component of SuiteSparse:GraphBLAS. SuiteSparse:GraphBLAS includes a `Test` folder that contains a full MATLAB implementation of the *GraphBLAS API Specification*, so that each method and operation in the C-callable SuiteSparse:GraphBLAS can be tested and their results compared with the GraphBLAS specification. The tests must ensure correct results and the correct asymptotic performance of the algorithms, for three reasons:

- SuiteSparse:GraphBLAS is meant to be a robust and high-performance sequential implementation that is suitable for inclusion in a final application. The package already appears in Debian and Ubuntu Linux distributions and forms the basis of the RedisGraph module of Redis, a commercial graph database package. It must be well-tested so that its results can be relied upon in production use.

- It provides a high-performance sequential baseline, with the right asymptotic performance, against which the performance of future parallel implementations can be compared. There is little point in a creating a parallel GraphBLAS that is slower than a good sequential implementation.
- It provides an exact implementation of the *GraphBLAS C API Specification*, so that future parallel implementations can check their results for correctness.

To accomplish these goals, each GraphBLAS method and operation is written as a very simple and short MATLAB function that can be visually inspected to conform to the *GraphBLAS C API Specification*. MATLAB has high-performance sparse matrix computations, but the MATLAB implementation is not meant to be fast. It is meant as a perfectly compliant implementation of the specification.

MATLAB supports all of the built-in types of GraphBLAS in its dense matrix data structure, but MATLAB sparse matrices can only be double or complex. As a result, the MATLAB implementation in the `Test` folder uses two dense matrices for each GraphBLAS matrix: one to hold the numerical values (in varying sizes of integers and floating-point values) and a dense binary matrix (`logical` in MATLAB) to hold the pattern. This is of course very slow for large problems, so the performance of this dense-matrix-based MATLAB implementation should not be compared with SuiteSparse:GraphBLAS.

The MATLAB functions that implement the *GraphBLAS C API Specification* are the `GB_spec_*.m` files, and are a total of 1,330 lines of MATLAB and another 1,384 lines of comments. They are carefully commented so they can be compared with the GraphBLAS API to ensure their compliance.

For example, `GrB_assign` and the related function `GrB_extract` take a total of 3,908 lines of C to implement in SuiteSparse:GraphBLAS and 3,984 lines of comments to describe. The same functions in the concise `GB_spec_*.m` functions take only 161 lines of code and 208 lines of comments. Of the 161 lines of m-file code, 33 lines are error checks that are not necessary to illustrate and implement the method.

The testing framework includes a MATLAB `mexFunction` interface to each GraphBLAS function (written in C), so that any computation can be done twice, in C and in MATLAB. The results are then compared with the fast C implementation of SuiteSparse:GraphBLAS. Most of the tests check for exactly identical results, even the same roundoff error for floating-point computations. Since the MATLAB implementation relies on dense matrices, these tests can only be done for small test problems.

This approach adds a challenge to the test coverage question. A test framework must ensure accurate results while at the same time ensuring 100% test coverage. However, test coverage tools for C such as `gcov` cannot be used in a MATLAB `mexFunction`. MATLAB has a test coverage reporting mechanism, but it can only evaluate the test coverage of MATLAB m-files, not `mexFunctions` in C.

To meet this challenge, SuiteSparse:GraphBLAS includes its own implementation of a test coverage mechanism. To test the code, a code preprocessor analyzes the C version of the package and emits a test version with code coverage statements inserted. The `mexFunction` interface then calls this annotated version of the C-callable library, and collects statement coverage counts as the tests proceed.

At 25,465 lines of C and MATLAB (excluding 11,990 lines of comments), the testing framework is longer than the C code of the SuiteSparse:GraphBLAS library itself. Only 1,330 lines of the testing framework are the `GB_spec_*.m` files, which provide a simple, elegant, and easy-to-read implementation that is almost a line-by-line transliteration of the GraphBLAS specification. This provides a powerful means to ensure its compliance with the spec.

The entire testing framework is thus extremely rigorous. Fortunately, it only needs to be done once. Now that SuiteSparse:GraphBLAS is available, and has been rigorously shown to exactly implement the Specification, future parallel implementations do not have to write such rigorous tests. They can simply compare their results with SuiteSparse:GraphBLAS.

5 EXAMPLE GRAPH ALGORITHMS IN GRAPHBLAS

This section describes two of the graph algorithms provided as demos in the SuiteSparse:GraphBLAS package: breadth-first search, and Luby's method for finding a maximal independent set. Both are modified versions of examples that appear in the GraphBLAS specification [5].

5.1 Breadth-first search

Figure 2 gives an algorithm in GraphBLAS for computing the breadth-first search of a graph. Each step consists of a matrix-vector multiplication, which finds the nodes in the next level of the traversal. Space does not permit all the details of the C API syntax to be described, so the algorithm is repeated in pseudo-MATLAB notation in Figure 3. The single call to `GrB_vxm` computes a masked matrix-vector product using the logical OR-AND semiring. Internally, it first computes `t'=q'*A` using the Boolean OR-AND semiring. It then clears all entries in `q`, via the `GrB_REPLACE` parameter in the descriptor. Finally, it assigns `t` to `q` using a complemented mask, which is `q(~v)=t` in MATLAB notation.

5.2 Luby's maximal independent set algorithm

A second algorithm provided with SuiteSparse:GraphBLAS is an implementation of Luby's parallel method for finding a maximal independent set [20]. SuiteSparse:GraphBLAS is currently a sequential package, but it expresses the algorithm using whole-graph operations that could be implemented in parallel in future versions.

The *maximal independent set* problem is to find a set of nodes S such that no two nodes in S are adjacent to each other (an independent set), and all nodes not in S are adjacent to at least one node in S (and thus S is maximal since it cannot be augmented by any node while remaining an independent set).

In each phase, all candidate nodes are given a random score. If a node has a score higher than all its neighbors, then it is added to the independent set. All new nodes added to the set cause their neighbors to be removed from the set of candidates. The process must be repeated for multiple phases until no new nodes can be added. This is because in one phase, a node `i` might not be added because one of its neighbors `j` has a higher score, yet that neighbor `j` might not be added because one of its neighbors `k` is added to the independent set instead. The node `j` is no longer a candidate and can never be added to the independent set, but node `i` could be added to S in a subsequent phase.

Each phase of Luby's algorithm consists of nine calls to GraphBLAS operations. The inner loop of Luby's method is shown in Figure 4. The descriptor `r_desc` causes the result to be cleared first, and `sr_desc` selects that option in addition to complementing the mask.

The two matrix-vector multiplications are the important parts and also take the most time. They also make interesting use of semirings and masks. The first one discussed here computes the largest score of all the neighbors of each node in the candidate set:

```
// compute the max probability of all neighbors
GrB_vxm (neighbor_max, candidates, NULL, GxB_MAX_FIRST_FP32, prob, A, r_desc) ;
```

`A` is a symmetric Boolean matrix and `prob` is a sparse real vector of type `FP32`, which is a `float` in C. `prob(j)` is nonzero only if node `j` is a candidate. The pre-defined `GxB_MAX_FIRST_FP32` semiring uses `z=FIRST(x,y)=x` as the "multiply" operator. Column `A(:,j)` is the adjacency of

```
GrB_Info bfs                // BFS of a graph (using vector assign & reduce)
(
    GrB_Vector *v_output,   // v [i] is the BFS level of node i in the graph
    const GrB_Matrix A,     // input graph, treated as if boolean in semiring
    GrB_Index s             // starting node of the BFS
)
{
    GrB_Index n ;                           // # of nodes in the graph
    GrB_Vector q = NULL ;                   // nodes visited at each level
    GrB_Vector v = NULL ;                   // result vector
    GrB_Descriptor desc = NULL ;            // Descriptor for vxm
    GrB_Matrix_nrows (&n, A) ;              // n = # of rows of A
    GrB_Vector_new (&v, GrB_INT32, n) ;     // Vector<int32_t> v(n) = 0
    GrB_Vector_new (&q, GrB_BOOL, n) ;      // Vector<bool> q(n) = false
    GrB_Vector_setElement (q, true, s) ;    // q[s] = true, false elsewhere
    GrB_Descriptor_new (&desc) ;
    GrB_Descriptor_set (desc, GrB_MASK, GrB_SCMP) ;     // invert the mask
    GrB_Descriptor_set (desc, GrB_OUTP, GrB_REPLACE) ;  // clear q first

    bool successor = true ; // true when some successor found
    for (int32_t level = 1 ; successor && level <= n ; level++)
    {
        // v<q> = level, using vector assign with q as the mask
        GrB_assign (v, q, NULL, level, GrB_ALL, n, NULL) ;
        // q<!v> = A ||.&& q ; finds all the unvisited
        // successors from current q, using !v as the mask
        GrB_vxm (q, v, NULL, GrB_LOR_LAND_BOOL, q, A, desc) ;
        // successor = ||(q)
        GrB_reduce (&successor, NULL, GrB_LOR_BOOL_MONOID, q, NULL) ;
    }
    *v_output = v ;         // return result
    GrB_free (&q) ;
    GrB_free (&desc) ;
    return (GrB_SUCCESS) ;
}
```

Fig. 2. GraphBLAS breadth-first search: given an adjacency matrix A and a source node s, perform a BFS traversal of the graph, setting v[i] to the level in which node i is visited.

```
v = zeros (1,n) ;       % v(k) is the BFS level (1 for source node s)
q = false (1,n) ;       % boolean vector of size n
q (s) = true ;          % q(k) is true if node k is in current level
for level = 1:n
    v (q) = level ;     % set v(i)=level where q(i) is true
    % new q = all unvisited neighbors of current q:
    t = A'*q ;          % where '*' is the OR-AND semiring
    q = false (1,n) ;   % clear q of all entries
    q (~v) = t ;        % q (i) = t (i) but only where v(i) is zero
    if (~any (q)) break ;
end
```

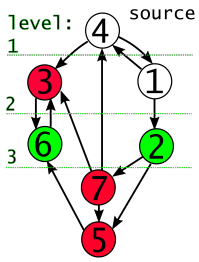

Fig. 3. Breadth-first search using pseudo-MATLAB notation. The only aspect of this code fragment that is not pure MATLAB is the computation of A'*q, since MATLAB does provide matrix multiplication with the OR-AND semiring. The graphic shows the computation of t=A'*q using the OR-AND semiring, where q is at level three (nodes 2 and 6). Level 4 will be nodes 5 and 7; node 3 is excluded since v is used as a complemented mask and v[3] is nonzero.

```
// Iterate while there are candidates to check.
GrB_Index nvals ;
GrB_Vector_nvals (&nvals, candidates) ;
int64_t last_nvals = nvals ;

while (nvals > 0)
{
    // compute a random probability scaled by inverse of degree
    GrB_apply (prob, candidates, NULL, set_random, degrees, r_desc) ;

    // compute the max probability of all neighbors
    GrB_vxm (neighbor_max, candidates, NULL, GxB_MAX_FIRST_FP32, prob, A, r_desc) ;

    // select node if its probability is > than all its active neighbors
    GrB_eWiseAdd (new_members, NULL, NULL, GrB_GT_FP64, prob, neighbor_max, NULL) ;

    // add new members to independent set.
    GrB_eWiseAdd (iset, NULL, NULL, GrB_LOR, iset, new_members, NULL) ;

    // remove new members from set of candidates c = c & !new
    GrB_apply (candidates, new_members, NULL, GrB_IDENTITY_BOOL, candidates, sr_desc) ;

    GrB_Vector_nvals (&nvals, candidates) ;
    if (nvals == 0) { break ; }                 // early exit condition

    // Neighbors of new members can also be removed from candidates
    GrB_vxm (new_neighbors, candidates, NULL, GxB_LOR_LAND_BOOL, new_members, A, NULL) ;
    GrB_apply (candidates, new_neighbors, NULL, GrB_IDENTITY_BOOL, candidates, sr_desc) ;

    GrB_Vector_nvals (&nvals, candidates) ;

    // this will not occur, unless the input is corrupted somehow
    if (last_nvals == nvals) { printf ("stall!\n") ; exit (1) ; }
    last_nvals = nvals ;
}
```

Fig. 4. Luby's maximal independent set method in GraphBLAS

node `j`, and the dot product `prob'*A(:,j)` applies the `FIRST` operator on all entries that appear in the intersection of `prob` and `A(:,j)`, `z=FIRST(prob(i),A(i,j))`, which is just `prob(i)` if `A(i,j)` is present. If `A(i,j)` is not an explicit entry in the matrix, then this term is not computed and does not take part in the reduction by the `MAX` monoid.

Thus, each term `z=FIRST(prob(i),A(i,j))` is the score, `prob(i)`, of all neighbors `i` of node `j` that have a score. Node `i` does not have a score if it is not also a candidate and so this is skipped. These terms are then "summed" up by taking the maximum score, using `MAX` as the additive monoid.

Finally, the results of this matrix-vector multiply are written to the result, `neighbor_max`. The `r_desc` descriptor has the `REPLACE` option enabled. Since `neighbor_max` does not also take part in the computation `prob'*A`, it is simply cleared first. Next, is it modified only in those positions `i` where `candidates(i)` is true, using `candidates` as a mask. This sets the `neighbor_max` only for candidate nodes, and leaves the other components of `neighbor_max` as zero (implicit values not in the pattern of the vector).

All of the above work is done in a single matrix-vector multiply, with an elegant use of the max-first semiring coupled with a mask. The matrix-vector multiplication is described above as if

Table 5. Performance of triangle counting on an IBM Minsky system, 20 Power8 cores, 8 hardware threads per core. Performance is in terms of 10^6 edges/sec (higher is better).

matrix			performance (10^6 edges/sec)			
	nodes	edges	Graph	without GraphBLAS		
name	10^3	10^6	BLAS	simple	fast	parallel
loc-brightkite_edges	.06	0.2	13.2	6.2	**33.8**	138.1
soc-Epinions1	.08	0.4	6.5	3.3	**20.0**	124.2
email-EuAll	.27	0.4	11.4	1.4	**26.0**	132.8
cit-HepPh	.04	0.4	10.3	5.1	**30.9**	177.8
soc-Slashdot0902	.08	0.5	8.4	3.6	**21.1**	101.4
amazon0302	.26	0.9	16.2	11.0	**36.0**	273.5
loc-gowalla_edges	.20	1.0	5.0	2.4	**14.8**	99.6
roadNet-TX	1.38	1.9	12.7	12.9	**25.4**	275.7
flickrEdges	.11	2.3	2.3	1.7	**4.0**	40.6
cit-Patents	3.77	16.5	**6.0**	4.9	4.5	104.3
friendster	119.43	1,800.0	**5.7**	2.1	2.6	56.3
g-16764930	4.19	16.8	**24.5**	18.9	9.0	282.1
g-268386306	67.11	268.4	**24.3**	20.2	15.7	271.3

it uses dot products of the row vector `prob'` with columns of `A`, but SuiteSparse:GraphBLAS does not compute it that way. Sparse dot products are typically much slower than the optimal method for multiplying a sparse matrix times a sparse vector, except in special cases. The result is the same, however.

6 PERFORMANCE

GraphBLAS allows a user application to express its graph algorithms in a powerful, expressive, and simple manner. Details of the graph data structure and the implementation of the basic methods can be left to the author of the GraphBLAS library. The key question, however, is whether or not performance of such an application can match that of a graph algorithm written without GraphBLAS.

In [8], two algorithms were written in GraphBLAS, and without GraphBLAS (both sequentially and in parallel). The first algorithm is to count the number of triangles in a graph, which is the number of unique cliques of size 3 in the graph. The second is construct the k-truss of a graph. In a 3-truss, all edges in the graph are removed except for those in one or more triangles. In a k-truss, each edge that remains must appear in at least $k-2$ triangles. Both codes are very simple to write using GraphBLAS.

The methods were tested on a large set of graphs from the MIT GraphChallenge (https://graphchallenge.mit.edu). Table 5 lists the performance of triangle counting algorithm in GraphBLAS, and three implementations without GraphBLAS: a simple sequential implementation, a highly-tuned sequential code (the column "fast"), and a highly-tuned code in OpenMP. The results show that triangle counting in SuiteSparse:GraphBLAS is competitive with a highly-tuned sequential method ("fast"), and even faster for some larger graphs. An asymptotically optimal yet simple C implementation of triangle counting ("simple") is much slower than an implementation using SuiteSparse:GraphBLAS.

Performance results of the k-truss methods with $k = 3$ are shown in Table 6. K-truss in SuiteSparse:GraphBLAS is also simple and the performance is competitive with a highly optimized

Table 6. 3-truss performance (10^6 edges/sec, higher is better)

matrix			performance (10^6 edges/sec)		
	nodes	edges	Graph	without GraphBLAS	
name	10^3	10^6	BLAS	sequential	parallel
loc-brightkite_edges	0.06	0.2	1.4	2.3	10.5
soc-Epinions1	0.08	0.4	0.5	1.1	6.6
email-EuAll	0.27	0.4	0.4	0.7	7.4
cit-HepPh	0.04	0.4	1.0	1.9	14.4
soc-Slashdot0902	0.08	0.5	0.8	1.6	8.6
amazon0302	0.26	0.9	2.2	3.9	29.7
loc-gowalla_edges	0.20	1.0	0.4	0.8	9.2
roadNet-TX	1.38	1.9	10.8	15.1	56.6
flickrEdges	0.11	2.3	0.2	0.4	3.5
cit-Patents	3.77	16.5	0.9	1.4	11.5
friendster	119.43	1800.0	0.1	0.2	1.0
GenBank/A2a	170.73	180.3	6.2	8.5	35.1
GenBank/V1r	214.01	232.7	11.5	18.5	60.2
g-268386306	67.11	268.4	10.2	38.9	190.7
g-1073643522	268.44	1073.6	10.1	38.6	199.9

Table 7. RedisGraph compared with other graph databases: 1-hop path query time. Normalized is relative to the time for RedisGraph (lower is better)

Matrix	Measure	RedisGraph	TigerGraph	Neo4j	Neptune	JanusGraph	ArangoDB
graph500	time (msec)	0.39	6.3	21.0	13.5	27.2	91.9
	normalized	1	16.2	53.8	34.6	69.7	235.6
twitter	time (msec)	0.80	24.1	205.0	289.2	394.8	1,674.7
	normalized	1	30.1	256.3	361.5	493.5	2,093.4

and complex algorithm in pure C, rarely taking more than twice the time as the highly-optimized, sequential versions in pure C. These results demonstrate that GraphBLAS can be an efficient library that allows end users to write simple yet fast code. Full details of this experiment are available in [8]. The code for these experiments is in `Extras` folder of the software bundle, and at http://suitesparse.com.

SuiteSparse:GraphBLAS has been released as part of the RedisGraph database module of the Redis database systems, by RedisLabs, Inc [6]; Roi Lipman is the primary developer. Based on SuiteSparse:GraphBLAS Version 1.2.0, RedisGraph 1.0 is much faster than competing graph databases. Two graphs were used for the following experiment reported in [6], and the results are summarized here: `graph500` with 2.4 million nodes and 64 million edges, from http://graph500.org, and `twitter` with 41.6 million nodes and 1.47 billion edges, from http://an.kaist.ac.kr/traces/WWW2010.html. The twitter graph appears as the `SNAP/twitter7` matrix from the SNAP collection at Stanford [19], and also appears in the SuiteSparse Matrix Collection (http://sparse.tamu.edu/SNAP/twitter7) [9]. The results were obtained on a 32-core system with 244 GB of RAM. Table 7 reports the results of a breadth-first search in RedisGraph and five other graph databases. The experiment tests a one-hop breadth-first search for 300 source nodes (that is, find the neighbors of 300 nodes in a graph). Table 8 reports the tests for 300 two-hop breadth-first searches for 300 source nodes. Finally, Table 9

Table 8. RedisGraph compared with other graph databases: 2-hop path query time. Normalized is relative to the time for RedisGraph (lower is better)

Matrix	Measure	RedisGraph	TigerGraph	Neo4j	Neptune	JanusGraph	ArangoDB
graph500	time (msec)	30.7	71.0	4,180.0	8,250.0	24,050.0	16,650.0
	normalized	1	2.3	136.2	268.7	783.4	542.3
twitter	time (msec)	503.0	460.0	18,340.0	27,400.0	27,780.0	28,980.0
	normalized	1	0.9	36.5	54.5	55.2	57.6

Table 9. RedisGraph compared with other graph databases: 6-hop path query time. Normalized is relative to the time for RedisGraph (lower is better). A dash if the database took more than 2.5 hours.

Matrix	Measure	RedisGraph	TigerGraph	Neo4j	Neptune	JanusGraph	ArangoDB
graph500	time (msec)	1614	1780	1.3×10^6	-	-	-
	normalized	1	1.1	813.3	-	-	-
twitter	time (msec)	78,730	63,000	-	-	-	-
	normalized	1	0.8	-	-	-	-

reports the tests for 300 6-hop breadth-first searches for 300 source nodes, where a dash appears if the database timed out after 2.5 hours. These results demonstrate that SuiteSparse:GraphBLAS provides competitive performance against these other commercial graph database systems, even though SuiteSparse:GraphBLAS is still a single-threaded package. Version 2.2.2 of the package is faster than the Version 1.2.0 used in these experiments, as well.

7 SUMMARY

The SuiteSparse:GraphBLAS library provides an efficient and highly optimized single-threaded implementation of the GraphBLAS API standard [5]. It allows users to write powerful and expressive graph algorithms with simple user-level code, whose performance is competitive with a highly-tuned code written by an expert. The package is available as Collected Algorithm 9xx of the ACM, at http://suitesparse.com, as part of the RedisGraph distribution, and is also available as a pre-packaged Debian and Ubuntu Linux distro in the `suitesparse` package.

ACKNOWLEDGMENTS

GraphBLAS has been a community effort (http://graphblas.org), and this package would not be possible the efforts of the GraphBLAS Steering Committee (David Bader, Aydın Buluç, John Gilbert, Jeremy Kepner, Tim Mattson, and Henning Meyerhenke), the GraphBLAS API Committee (Aydın Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang), and Roi Lipman of RedisLabs. This work is supported by the National Science Foundation, under grant NSF CNS-1514406, and by MIT Lincoln Laboratory.

REFERENCES

[1] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. 2016. GraphPad: Optimized Graph Primitives for Parallel and Distributed Platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 313–322. https://doi.org/10.1109/IPDPS.2016.86

[2] A. Azad, A. Buluç, and J. Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 804–811. https://doi.org/10.1109/IPDPSW.2015.75

[3] A. Buluç and J. R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536313

[4] A. Buluç and J. R. Gilbert. 2011. The Combinatorial BLAS: design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509. https://doi.org/10.1177/1094342011403516 arXiv:https://doi.org/10.1177/1094342011403516

[5] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. 2017. *The GraphBLAS C API Specification.* Technical Report. http://graphblas.org/

[6] P. Cailliau. 2018. Benchmarking RedisGraph 1.0 . https://redislabs.com/blog/new-redisgraph-1-0-achieves-600x-faster-performance-graph-databases

[7] T. A. Davis. 2006. *Direct Methods for Sparse Linear Systems.* SIAM, Phila., PA. https://doi.org/10.1137/1.9780898718881

[8] T. A. Davis. 2018. Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and K-truss. In *2018 IEEE High Performance Extreme Computing Conference (HPEC).* (doi to appear; in the meantime, see http://faculty.cse.tamu.edu/davis/publications_files/Davis_HPEC18.pdf).

[9] T. A. Davis and Y. Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (Dec. 2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663

[10] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566. https://doi.org/10.1017/S0962492916000076

[11] K. Ekanadham, W. Horn, J. Jann, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and Yu H. 2014. *Graph Programming Interface: Rationale and Specification.* Technical Report RC25508 (WAT1411-052). IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY.

[12] J. R. Gilbert, C. Moler, and R. Schreiber. 1992. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356. https://doi.org/10.1137/0613024

[13] F. G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978), 250–269. https://doi.org/10.1145/355791.355796

[14] D. Hutchison, J. Kepner, V. Gadepally, and B. Howe. 2016. From NoSQL Accumulo to NewSQL Graphulo: Design and utility of graph algorithms inside a BigTable database. In *2016 IEEE High Performance Extreme Computing Conference (HPEC).* 1–9. https://doi.org/10.1109/HPEC.2016.7761577

[15] J. Kepner. 2017. *GraphBLAS Mathematics.* Technical Report. MIT Lincoln Lab . http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf

[16] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC).* 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[17] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee. 2012. Dynamic distributed dimensional data model (D4M) database and computation system. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 5349–5352. https://doi.org/10.1109/ICASSP.2012.6289129

[18] J. Kepner and J. Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* SIAM, Phila., PA. https://doi.org/10.1137/1.9780898719918

[19] J. Leskovec and A. Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[20] M. Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.* 15, 4 (1986). https://doi.org/10.1137/0215074

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

[22] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. 2015. GraphMat: High performance graph analytics made productive. In *Proceedings of VLDB 2015*, Vol. 8. 1214 – 1225. https://doi.org/10.14778/2809974.2809983

[23] C. Yang, A. Buluç, and J. D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati (Eds.). Springer International Publishing, 672–687. https://doi.org/10.1007/978-3-319-96983-1_48

Received June 2018; revised November 2018; accepted (month) 2018