Algorithm 10xx: SuiteSparse:GraphBLAS: parallel graph algorithms in the language of sparse linear algebra

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

SuiteSparse:GraphBLAS is a full parallel 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. A description of the parallel implementation of SuiteSparse:GraphBLAS is given, including its novel parallel algorithms for sparse matrix multiply, addition, element-wise multiply, submatrix extraction and assignment, and the GraphBLAS mask/accumulator operation. Its performance is illustrated by solving the graph problems in the GAP Benchmark and by comparing it with other sparse matrix libraries.

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. 2022. Algorithm 10xx: SuiteSparse:GraphBLAS: parallel graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2022), 30 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The GraphBLAS standard defines sparse matrix and vector operations on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms. A full and precise definition of the GraphBLAS specification is provided in *The GraphBLAS C API Specification* [5, 8], based on the mathematical foundations discussed in [22, 23]. Here, we describe a particular OpenMP-based parallel implementation of the GraphBLAS standard, SuiteSparse:GraphBLAS. The implementation assumes a shared-memory model of computing with a number of computational cores consistent with a multicore CPU. A GPU-based implementation is in progress.

In addition to its availability as a Collected Algorithm of the ACM, it appears as the core computational engine of RedisGraph, by Redis Inc, and it is the recommended package for sparse matrix computations in Julia. It is also the built-in sparse matrix multiply in MATLAB R2021a.

The remainder of this paper is structured as follows. Section 2 provides an overview of the objects, methods, and operations in the GraphBLAS specification, and gives an example of how they can be used to write a parallel graph algorithm. Aspects of the parallel implementation of GraphBLAS are discussed in Section 3, followed by a presentation of the many parallel algorithms in SuiteSparse:GraphBLAS: matrix multiply (Section 4), element-wise add and multiply (Sections 5 and 6), submatrix extraction and assignment (Section 7 and 8), and the mask/accumulator phase (Section 9). Section 10 highlights the parallel performance of SuiteSparse:GraphBLAS. Section 11 summarizes related work, followed by future work (Section 12) and final comments and code availability in Section 13.

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

0098-3500/2022/1-ART1 $15.00
https://doi.org/0000001.0000001

2 AN OVERVIEW OF GRAPHBLAS AND ITS IMPLEMENTATION IN SUITESPARSE

The idea behind GraphBLAS is to provide the user a collection of objects and their methods and operations so that a graph algorithm can be expressed using linear algebraic operations with sparse adjacency matrices on different semirings. The goal is both ease of expression of these graph algorithms and high performance. *Graph Algorithms in the Language on Linear Algebra* [25] provides a framework for understanding how graph algorithms can be expressed as matrix computations. For additional background on sparse matrix algorithms, see also [10] and a recent survey paper, [13].

2.1 The GraphBLAS API

A GraphBLAS *operation* is one that takes an optional mask matrix $\mathbf{M}$, which is boolean (or typecasted to boolean), and an optional accumulator operator. If the mask is present, c_{ij} can only be modified if m_{ij} is true. An operation accepts a *descriptor* which modifies its behavior, such as optionally complementing the mask, declaring the mask as *structural*. The set of explicit entries in a sparse matrix is referred to as its sparsity *structure*. Entries not in the sparsity structure have no value (they are not assumed to be numerically zero). When a mask $\mathbf{M}$ is used structurally, any entry in present the structure is treated as true. The descriptor can also transpose the input matrices.

The notation $\mathbf{C}\langle\mathbf{M}\rangle$ is used when the mask is *valued*, where c_{ij} can only be modified if the entry m_{ij} is present and nonzero. The notation $\mathbf{C}\langle s(\mathbf{M})\rangle$ is used when the mask is structural, where only the presence or absence of m_{ij} controls whether or not c_{ij} can be modified; the value of m_{ij} is ignored when using a structural mask.

All matrix operations can be modified by using different binary and unary operators, monoids, or semirings. For example, suppose $\mathbf{C} = \max(\mathbf{A}, \mathbf{B})$ is to be computed, where the structure of $\mathbf{C}$ is the set intersection of $\mathbf{A}$ and $\mathbf{B}$. This is the element-wise multiply operation, but using the binary max operator instead of the binary times operator of the Hadamard product.

A *monoid* is an associative binary operator with an identity element (the *zero* value of the set). Monoids can be used alone, to reduce a matrix to a vector or scalar, such as using the plus (+) monoid to sum each row of a matrix to a scalar. Monoids can also be used in a semiring to redefine matrix multiplication. The GraphBLAS monoid is the same this standard mathematical definition. The formal mathematical definition of a *semiring* is a set with two binary operations, addition and multiplication [18]:

(1) The additive operator is a commutative monoid,
(2) The multiplicative operator is a monoid with an identity element (the *one* value of the set).
(3) Multiplication distributes over addition.
(4) The additive identity (the *zero*) is the same as the multiplicative annihilator.

GraphBLAS includes many semirings that fit this definition, but it allows for a looser definition which is useful for many graph algorithms. All formal semirings are valid GraphBLAS semirings. In the GraphBLAS semiring, the first item in the list above is unchanged. The second is revised, where the multiplicative operator is not a monoid, but merely a binary operator. It need not be associative and need not have an identity value. The third and fourth items are not required for the GraphBLAS semiring.

The semiring of conventional linear algebra is $(+, \times)$, where $\mathbf{C} = \mathbf{AB}$ is defined as

$$c_{ij} = \sum_k a_{ik} \times b_{kj},$$

using the plus (+) monoid and the times ($\times$) multiplicative operator. In the sparse case, the $\times$ operator is applied only to entries in the intersection of the structure of the sparse row $\mathbf{A}_{i*}$ and

method	computation	description
GrB_Matrix_build	C=build(I,J,X,dup)	build a matrix from tuples
GrB_Vector_build	u=build(I,X,dup)	build a vector from tuples
GrB_Matrix_dup	$\mathbf{C} = \mathbf{A}$	duplicate a matrix
GrB_Vector_dup	$\mathbf{u} = \mathbf{b}$	duplicate a vector
operation	computation	description
GrB_mxm	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{AB}$	matrix multiply
GrB_mxv	$\mathbf{w}\langle\mathbf{m}\rangle \odot= \mathbf{Au}$	matrix-vector multiply
GrB_vxm	$\mathbf{w}^{\mathsf{T}}\langle\mathbf{m}^{\mathsf{T}}\rangle \odot= \mathbf{u}^{\mathsf{T}}\mathbf{A}$	vector-matrix multiply
GrB_eWiseAdd	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{A} \oplus \mathbf{B}$	element-wise addition (set union)
GrB_eWiseMult	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{A} \otimes \mathbf{B}$	element-wise multiply (set intersection)
GrB_select	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \text{select}(\mathbf{A})$	element-wise select
GrB_reduce	$\mathbf{w}\langle\mathbf{m}\rangle \odot= \text{reduce}(\mathbf{A})$	reduce matrix to vector
GrB_reduce	$\sigma \odot= \text{reduce}(\mathbf{A})$	reduce matrix to scalar
GrB_assign	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) \odot= \mathbf{A}$	assign
GrB_extract	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{A}(\mathbf{I},\mathbf{J})$	extract
GrB_transpose	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{A}^{\mathsf{T}}$	transpose
GrB_kronecker	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \text{kron}(\mathbf{A},\mathbf{B})$	Kronecker product
GrB_apply	$\mathbf{C}\langle\mathbf{M}\rangle \odot= f(\mathbf{A})$	apply a unary operator

Table 1. Key GraphBLAS methods and operations

the sparse column $\mathbf{B}_{*j}$. Other semirings are useful in different graph algorithms. For example, in a shortest-path problem, the $(\min, +)$ semiring computes

$$c_{ij} = \min_k (a_{ik} + b_{kj}).$$

GraphBLAS also provides a collection of *methods* to create, query, and free each of its different types of objects: matrices, vectors, scalars, types, binary and unary operators, selection operators, monoids, semirings, and a descriptor object used for parameter settings.

Table 1 summarizes the key GraphBLAS methods and operations, from the GraphBLAS C API Specification [5, 8]. The notation $\odot=$ is used for the optional accumulator operator (applied in a set union manner), $\oplus$ refers to any binary operator used in a set union manner, while $\otimes$ refers to any binary operator used in a set intersection manner.

2.2 Example GraphBLAS algorithm

This approach is illustrated with an example of a push/pull direction-optimizing breadth-first search (BFS) [4], and shown in GraphBLAS notation in Algorithm 1. In this algorithm, the sparse matrix $\mathbf{A}$ represents a graph where a_{ij} is the edge (i, j), and $\mathbf{q}$ is a sparse vector whose sparsity structure represents the list of nodes in the current frontier of a BFS. The vector $\mathbf{p}$ is the *parent* vector, where $p_i = j$ if node j is the parent of node i in the BFS tree. If node i has not yet been seen, then the entry p_i does not appear in the sparsity structure of $\mathbf{p}$.

In the BFS, the mask is complemented and used in a structural manner, denoted $\mathbf{C}\langle\neg s(\mathbf{M})\rangle = \mathbf{A}$, where the assignment $c_{ij} = a_{ij}$ is performed only where the entry m_{ij} is *not* present. The values of $\mathbf{M}$ are ignored if it used as a structural mask.

Multiplying $\mathbf{v}\langle\neg s(\mathbf{p})\rangle = \mathbf{A}^{\mathsf{T}}\mathbf{q}$ computes a new vector $\mathbf{v}$, where the entry v_j appears in the sparsity structure of $\mathbf{v}$ only if there is an edge (i, j) and q_i is an entry present in the sparsity structure of $\mathbf{q}$, and where j is previously unvisited. If node j does not yet have a parent, p_j is not present in the sparsity structure of the parent vector $\mathbf{p}$, and thus $\mathbf{p}$ is used as a complemented structural mask. The sparsity structure of $\mathbf{v}$ gives the list of nodes in the next level of the BFS. The value of v_j is also useful, and is determined by the specific semiring. Let the matrix-vector multiplication

ALGORITHM 1: Breadth-first search (BFS) using GraphBLAS [39]

input : n-by-n adjacency matrix $\mathbf{A}$, its transpose $\mathbf{T} = \mathbf{A}^{\mathsf{T}}$, and source node s
output : parent vector $\mathbf{p}$ where $p_i = j$ if j is the parent of i, and $p_s = s$ for the source node

```
p = empty sparse n-by-1 vector
q = empty sparse n-by-1 vector
p(s) = s                                  // s is the root of the BFS tree
q(s) = s                                  // s is only node in the current frontier
while q is not empty do
    determine push or pull (details omitted)
    if push then
        q⟨¬s(p), replace⟩ = A^T q         // using the (any,secondi) or (min,secondi) semiring
    else
        q⟨¬s(p), replace⟩ = T q           // also the (any,secondi) or (min,secondi) semiring
    p⟨s(q)⟩ = q                           // assign parents of nodes in current frontier
```

be redefined where the min operator is used instead of the conventional plus (+), and where the *secondi* operator is used instead of the conventional times ($\times$). The secondi operator is *positional*, where a positional operator depends not on the values of its operands, but on their position in the matrix. The value of secondi(a_{ik}, b_{kj}) is k, which is the row index of the second input scalar operand. The *replace* syntax used in Algorithm 1 means that the result of the operation completely overwrites the output vector, so that $\mathbf{q}$ is replaced with $\mathbf{v}$, in a single statement.

After computing the next frontier of newly-discovered nodes and their parents with a single call to `GrB_vxm` to compute $\mathbf{q}\langle\neg s(\mathbf{p}), \text{replace}\rangle = \mathbf{A}^{\mathsf{T}}\mathbf{q}$, the second step in the BFS is to assign these newly-found parents to the parent vector, with the masked assignment $\mathbf{p}\langle s(\mathbf{q})\rangle = \mathbf{q}$, via `GrB_assign`.

If $\mathbf{A}$ is held in a row-oriented manner, the implicit transpose $\mathbf{A}^{\mathsf{T}}\mathbf{q}$ results in the *push*-style algorithm, where outgoing edges from nodes in the current frontier $\mathbf{q}$ are searched. Now suppose that we also have the explicit transpose, $\mathbf{T} = \mathbf{A}^{\mathsf{T}}$, also in row-oriented format. Then it is simple to see that the expression $\mathbf{v}\langle\neg s(\mathbf{p})\rangle = \mathbf{T}\mathbf{q}$ computes the same thing as $\mathbf{v}\langle\neg s(\mathbf{p})\rangle = \mathbf{A}^{\mathsf{T}}\mathbf{q}$. However, the matrix $\mathbf{T}$ is held in a manner in which the natural algorithm is to compute a sequence of dot-products, one per entry not in the mask. This is identical to the *pull*-style algorithm of a direction optimizing BFS.

The parent tree computed by the (min, secondi) semiring is overly restrictive. In a BFS tree, any parent will do, but the (min, secondi) semiring picks the least numbered parent amongst all candidate parents of a node. This requires more work than necessary. A better monoid than min is based on the *any* operator, unique to SuiteSparse:GraphBLAS. The *any* operator is defined via a nondeterministic choice, any$(x, y) = x$ or y, either one, at the discretion of the operator itself. The *any* monoid has special properties, as described in Section 4.2, and is faster to compute than the min monoid. The details for deciding push versus pull are not shown in Algorithm 1, but they are the same heuristics as used by [4] and are simple to compute with a few calls to GraphBLAS.

Using its C API, the C version is not as succinct as the pseudo-code in Algorithm 1, but it is likewise very simple, and its logic is the same. With GraphBLAS interfaces in Python, Julia, Octave, or MATLAB, the algorithm is almost as simple as the pseudo-code in Algorithm 1. This example illustrates the expressive power of the GraphBLAS approach. The remainder of this paper illustrates how SuiteSparse:GraphBLAS obtains high performance when executing graph algorithms such as this one.

2.3 Key contributions of SuiteSparse:GraphBLAS

SuiteSparse:GraphBLAS introduces many unique and novel algorithms and their parallel implementations. These include new methods for masked matrix multiply, including an extension of prior methods (Gustavson's method [19] and Hashing [15, 29, 30]), by exploiting the mask, using novel finite-state machines for better fine-grain parallelism (Section 4), and special-case monoids. Its sparse matrix addition, element-wise multiply, and mask/accumulator phases use novel parallel meta-algorithms to exploit the properties of each pair of vectors in the input matrices (Sections 5, 6, and 9). Many of the methods for submatrix extraction and assignment are also novel, particularly when the mask and/or accumulator are present (Sections 7 and 8). The key algorithmic and engineering issues encountered in creating these methods include:

- how to exploit the mask to reduce the time complexity,
- how to exploit the GraphBLAS non-blocking mode,
- creating multiple parallelization techniques for different variants of each problem,
- how to mix arbitrary data structures in a matrix operation ($\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{AB}$ has four input matrices, and each one can be in one of 16 formats),
- exploiting important special cases required for particular graph algorithms (such as when the mask and output matrix are aliased), and
- handling the irregular parallelism inherent in sparse matrix computations.

3 DATA STRUCTURES FOR SUITESPARSE:GRAPHBLAS

All of the objects in the GraphBLAS API are opaque, which provides a significant opportunity for optimizing the data structures and algorithms when implementing a parallel GraphBLAS library. This section presents the design of core data structures in SuiteSparse:GraphBLAS and its impact on the parallelization of the methods that operate on them.

3.1 Matrix and vector data structure

The internal data structure for the `GrB_Matrix` and `GrB_Vector` object has changed significantly since the release of Algorithm 1000 [11], the sequential version of SuiteSparse:GraphBLAS (v2.3.3), to exploit new parallel algorithms and to reduce memory usage.

The prior sequential version of SuiteSparse:GraphBLAS provided four different formats: compressed-sparse column (standard CSC), compressed-sparse row (standard CSR), and hypersparse versions of these two formats. The current parallel version (v7.1.2) provides 16 different formats: four sparsity formats, listed below, where each can be held by row or by column, and each can exploit the *iso-value* property, if applicable (see Section 3.2). SuiteSparse:GraphBLAS selects between these 16 formats automatically, but the user application can also provide hints.

- **Compressed-sparse**, or simply **sparse**: If held by column, an m-by-n matrix $\mathbf{A}$ with e entries consists of an integer array `A.p` of size $n+1$, and two arrays of size e: `A.i` and `A.x`. The row indices and values of the `j`th column are held in `A.i[A.p[j]...A.p[j+1]-1]` and `A.x[A.p[j]...A.p[j+1]-1]`, respectively. The total space is $O(n+e)$, or $O(m+e)$ if held by row. A sparse matrix in this format can be viewed as a dense vector of sparse vectors. MATLAB uses this format exclusively for its sparse matrices, held only by column [17].
- **Hypersparse** [6]: This format is like the conventional sparse format, except that the `A.p` array itself is sparse. If $\mathbf{A}$ has k non-empty columns and is held in hypersparse format by column, then `A.p` has length $k+1$, and where typically $k \ll n$. To keep track of which vectors are present, another array `A.h` of length k is required, which is always kept in sorted order. The memory required is $O(k+e)$ and since $k < e$, this is simply $O(e)$. A hypersparse matrix can be viewed as a sparse vector of sparse vectors.

- **Bitmap**: In this format, the numerical values `A.x` are held in a dense array of size $m \times n$. The sparsity structure is held in a boolean array `A.b` of size $m \times n$. If `A.b[i,j]` is true, then its value is `A.x[i,j]`; otherwise `A.x[i,j]` is undefined. The total space is $O(mn)$ which is costly unless e is also $O(mn)$. In this case, the bitmap format is very efficient.
- **Full**: All entries must be present in the matrix in this format, so the sparsity structure is not held. Only the numerical array `A.x` is required, of size $m \times n$.

The matrix type can be nearly anything: boolean, any built-in integer or unsigned integer, single and double precision floating-point, and single and double complex types. In addition, the user application can define its own data types. The only restriction is that the type has a fixed number of bytes, and no constructor/destructor method is required to manage a single scalar of the type.

The GraphBLAS API allows for a non-blocking mode (the default mode) where work can be left pending to be done later. ACM Algorithm 1000 (v2.3.3) used two kinds of pending work; ACM Algorithm 10xx (v7.1.2) introduces another. Algorithm 1000 introduced the idea of *zombies* and *pending tuples*, which still appear. A zombie is an entry in a sparse or hypersparse matrix that is tagged for deletion, but not yet deleted, by setting its index i to $\text{flip}(i)$, a where $\text{flip}(i) = -i - 2$, is a function that is its own inverse. Since their index can still be retrieved, zombies still allow binary searches to be performed in a given sparse vector. A *pending tuple* is an entry that is yet to be inserted into the sparse or hypersparse matrix. They are kept as a list of tuples (with their row index, column index, and value), in order of their insertion, in addition to an operator to be used to assemble duplicate entries. Duplicate pending tuples may appear. New to v7.1.2 is a lazy sort of the vectors of a sparse or hypersparse matrix. If the matrix or vector is *jumbled*, then the indices inside its vectors can appear out of order (but with no duplicates). Some algorithms produce jumbled matrices naturally, and some do not care if their inputs are sorted or jumbled. If the sort is left pending, sometimes a matrix can be left in jumbled form for its entire lifetime, saving time and memory by avoiding the sort entirely. Bitmap and full matrices do not acquire zombies or pending tuples, nor are they ever jumbled. Bitmap and full formats are typically not used for the adjacency matrix of a graph, but they are very useful for vectors or tall-and-thin / short-and-fat matrices that contain information about the properties of the nodes of a graph.

All matrices are held by row, by default, except for the `@GrB` Octave/MATLAB interface, where they are held by column since that is the default for sparse matrices in those packages.

Most sparse matrix algorithms in the literature typically assume column-oriented storage, and so the algorithms in this paper are described as if all matrices are held by column. Internally, SuiteSparse:GraphBLAS manipulates its inputs so that its internal kernels are *agnostic* to the storage format. For example, computing $\mathbf{C} = \mathbf{AB}$ with $\mathbf{A}$ and $\mathbf{B}$ in column-oriented format is identical to computing $\mathbf{C} = \mathbf{BA}$ if both are in row-oriented format. The internal kernels do not distinguish between these two cases. The same code is used for both.

All operations can operate on all sixteen matrix formats in any combination. Most of the time, no format conversion is required, and the algorithms can either handle multiple input formats, or multiple kernels are used. One major exception is the row/column orientation, where mixing formats often requires a transpose of one input matrix.

3.2 Iso-valued matrices and vectors

Many graphs are unweighted, with no particular value associated with each edge. The GraphBLAS C API does not allow for structure-only matrices, however. These two perspectives seem irreconcilable at first glance, but there is a simple solution that saves time and memory when dealing with unweighted graphs in GraphBLAS, with no change to the C API. Algorithm 10xx introduces *iso-valued* matrices and vectors, where all entries in the sparsity structure have the same value.

For an *iso-valued* matrix or vector (or just *iso* for short, from here on), the `A.x` array in all formats shrinks to hold just a single value, the *iso-value* of the matrix or vector. For sparse, hypersparse, and bitmap formats, the sparsity structure must still be held, so these still require $O(n+e)$, $O(e)$, and $O(mn)$ memory, respectively. Full matrices are special. Since its sparsity structure is not represented, an *iso full* matrix takes $O(1)$ memory, regardless of its dimension.

An iso full matrix is useful in conjunction with an operation that typically need not access all its content. For example, if $\mathbf{A}$ is a n-by-n hypersparse matrix with $n = 2^{60}$ with $e \ll n$ entries, the sum of all rows of $\mathbf{A}$ can be computed as $\mathbf{y} = \mathbf{A}\mathbf{x}$ where $\mathbf{x}$ is an iso full vector of length n, or `x=ones(n,1)` in MATLAB notation. Such a vector is impossible to create with a built-in MATLAB expression, but `x=GrB.ones(n,1)` can easily be done in the Octave/MATLAB interface to SuiteSparse:GraphBLAS, taking only a few hundred bytes to represent. MATLAB cannot represent the matrix $\mathbf{A}$ since it would require $O(n+e)$ and n is enormous, but GraphBLAS uses only $O(e)$ space. MATLAB also cannot represent the iso full vector $\mathbf{x}$.

The iso property of the output matrix is exploited when the operation and its input matrices imply the output must be iso. The `GrB_*_build` method also uses a post-processing step where the matrix or vector is checked after it is constructed. A new `GxB_*_build` method is included where the input values are given as a single scalar, and the output is always iso (duplicates are ignored). Even when the output matrix $\mathbf{C}$ is not iso, the inputs of a GraphBLAS operation may still be iso. Whenever the value of an entry a_{ij} is needed, the single iso value of $\mathbf{A}$ is used in its place, resulting in a substantial reduction in memory transfers. With this technique, all parallel methods described in the next sections can handle and exploit iso input matrices.

3.3 Parallel matrix operations

SuiteSparse:GraphBLAS assumes an OpenMP shared-memory model of computing with a modest number of computational cores (say less than O(1000)). The user application itself can be multi-threaded, however, and GraphBLAS operations can be called in parallel, following the rules governed by the GraphBLAS v2.0 C API. Any input matrices can be shared between multiple user threads, so long as `GrB_wait` is used first, to ensure the work to compute them is finished.

The amount of parallelism to exploit and the size of the tasks are determined automatically. The maximum number of threads SuiteSparse:GraphBLAS uses ($p_{\max}$) is given by `omp_get_max_threads`, but this can be reduced via the descriptor or by a global setting. This is essential for composing the parallelism of GraphBLAS with the parallelism (if any) in the user application.

Suppose a parallel phase will do an amount of work equal to w. SuiteSparse:GraphBLAS has a chunk-size parameter κ, equal to 2^{16} by default. The number of threads actually used for this parallel region is $p = \max(1, \min(\lfloor w/\kappa \rfloor, p_{\max}))$. Once p is found, between p and $256p$ tasks are created (typically $8p$), and each task is constructed to perform the same amount of work. More tasks are generated than threads, and a dynamic schedule is typically used. This allows the tasks to vary in their computational time while maintaining a reasonable load balance. It is often difficult to predict the time taken by each task even if they are given equal work, since most tasks perform work with an irregular memory access, which is typical of sparse matrix computations, and the time can be somewhat unpredictable. Given $p =$ `nthreads` threads and `ntasks` tasks, the following is a very common paradigm in SuiteSparse:GraphBLAS:

```
// (1) construct ntasks tasks for nthreads threads
// (2) do the parallel computation using those threads, as:
#pragma omp parallel for num_threads(nthreads) schedule(dynamic,1)
for (int tid = 0 ; tid < ntasks ; tid++)
{
    // do the work for task tid
}
```

When the work for each task is very regular and the time taken by the tasks is predictable, a static schedule is instead. Since these tasks are typically balanced, the parallel algorithms described below have a typical time complexity of $O(w/p + s)$, where $O(w)$ is the time taken by a sequential algorithm and where the setup and wrapup time for the parallel method is $O(s)$. When atomics are used, this time complexity assumes each atomic operation takes $O(1)$ time, which is reasonable since conflicts are uncommon. It also assumes that the hash operations described in Section 4 also take $O(1)$ time, which is true in practice since hash collisions are rare.

The setup includes constructing the set of tasks, and the wrapup may involve a scalar reduction from all of the tasks. This time s is done sequentially in $O(p)$, $O(p \log p)$, or $O(p \log k)$ time, depending upon the algorithm, where the latter case is described in the next section. Most algorithms are essentially fully parallel and reasonably work-efficient assuming that w is the actual work required. A few exceptions are noted, such as the dot-product based matrix multiply methods in Section 4.

3.4 Parallel traversal of a single sparse or hypersparse matrix

Many algorithms described below must traverse multiple input matrices in parallel, and the division of work into parallel tasks must be considered on a case-by-case basis. However, many other algorithms traverse a just a single sparse or hypersparse matrix in parallel. In that case, the following method is used. In the rest of this paper, this method is referred to as *slicing* a matrix.

Suppose the entries of a sparse or hypersparse **A** matrix must be traversed in parallel, with no dependencies between the tasks (or with dependencies resolved via atomics). The matrix can be very tall and thin, or even a single vector, so it is not sufficient to divide up the vectors of the sparse matrix **A**. Instead, the entries in **A** are first divided uniformly into a set of independent tasks. These entries may span partial vectors of **A**, so a binary search of `A.p` is used to determine the starting and ending vectors of each partition, one per task.

Currently, this preprocessing step is done sequentially in $O(p \log k)$ time, where `A.p` has length `k+1` (where $k = n$ if **A** is in compressed-sparse format, or $k \ll n$ if **A** is hypersparse). If p were large this step could easily be done in parallel. The current implementation of SuiteSparse:GraphBLAS assumes p is no more than about $O(1000)$, and thus doing this work in parallel is not worth the overhead. This will be revised in the future for systems with many cores, particularly when computing on the GPU, where the time would be $O(\log k)$ if each task does its own search.

This preprocessing step is used in many algorithms in SuiteSparse:GraphBLAS, and it allows a sparse or hypersparse matrix of dimension m-by-n with $O(e)$ entries to be traversed in parallel in $O(e/p + p \log k)$ time by p threads, where `A.p` has length `k+1`, regardless of m or n.

This parallel traversal allows both the row and column index of every entry to be known by the task. Some methods only need the row index of each entry (if **A** is held by column), or conversely, the column index if **A** is held by row. In that case, the entries of a sparse or hypersparse matrix **A** can be easily divided into tasks without the need for a binary search, where the entire traversal of **A** takes $O(e/p)$ time when using p threads.

4 PARALLEL MATRIX MULTIPLY

The parallel matrix multiply (`GrB_mxm`), matrix-vector multiply (`GrB_mxv`), and vector-matrix multiply (`GrB_vxm`) methods are key operations in GraphBLAS and are typically the most computationally intensive within a given graph algorithm. An overview of the strategy and methods is given, followed by a deep dive into a few selected kernels.

4.1 Overview

`GrB_mxm` has a large set of cases to consider, even for a single semiring. It can operate on either **A** or its transpose, and on either **B** or its transpose. In addition, all four matrices can be held in any mix of sparsity formats (sparse, hypersparse, bitmap, or full, and either held by row or by column). The mask may be present, and if so can be used in a valued-sense or structural sense, and it may be complemented. Some methods can exploit the presence of the accumulator, if it matches the semiring monoid. A `GrB_Vector` of length n is held as an n-by-1 `GrB_Matrix`, so vector computations are not described; they are identical to computations with n-by-1 matrices.

The first step taken by `GrB_mxm` is to reduce this large set of cases via a meta-algorithm that transforms the problem into a canonical form that can be handled by one of four strategies. All of these strategies are *agnostic* to the row/column orientation of their input/output matrices. For simplicity, the remainder of this presentation assumes that all matrices are held by column.

The meta algorithm may need to transpose one or more of its 2 or 3 input matrices, or it may replace $\mathbf{C} = \mathbf{AB}$ with $\mathbf{C} = (\mathbf{B}^\mathsf{T}\mathbf{A}^\mathsf{T})^\mathsf{T}$ via a heuristic rule that attempts to select the fastest method. For example, when computing $\mathbf{C} = \mathbf{A}^\mathsf{T}\mathbf{B}$ it can be thousands of times faster to compute $\mathbf{C} = (\mathbf{B}^\mathsf{T}\mathbf{A})^\mathsf{T}$ instead, if **A** has many entries but **C** and **B** are small or have few entries.

The rules can be modified by user-provided hints. The default meta-rules are as follows, where $\langle\sharp\mathbf{M}\rangle$ denotes either the mask $\langle\mathbf{M}\rangle$, the complemented mask $\langle\neg\mathbf{M}\rangle$, or no mask:

- A *saxpy*-based strategy for $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{AB}$, where the term *saxpy* is an acronym from the dense BLAS function that computes $\alpha\mathbf{x} + \mathbf{y}$ (Section 4.3)
- A *dot product*-based strategy for $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{A}^\mathsf{T}\mathbf{B}$ (Section 4.4).
- Row-scaling ($\mathbf{C} = \mathbf{DB}$) or column-scaling ($\mathbf{C} = \mathbf{AD}$) where **D** is diagonal.
- $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{AB}^\mathsf{T}$ requires an outer-product-based method, which SuiteSparse:GraphBLAS does not have, and thus **A** or **B** are explicitly transposed first, and the saxpy or dot strategy is used.
- For $\mathbf{C}\langle\sharp\mathbf{M}\rangle = (\ldots)^\mathsf{T}$, one of the above strategies is used to compute a temporary result, which is then explicitly transposed to obtain **C**.

Each of these strategies then subdivides into many methods. These methods rely on host of parallel algorithms, depending upon the sparsity formats of their input and output matrices, the presence of the mask matrix, and how many threads are being used.

In a saxpy-based method, the basic computation is the addition of two sparse vectors, `C(:,j)` and `A(:,k)`, where the latter is multiplied by a scale factor, `C(:,j)` $\oplus\!\!=$ `A(:,k)` $\otimes$ `B(k,j)`, where the semiring is $(\oplus, \otimes)$. This is typically identical to a "push" phase of a graph algorithm such as the push/pull direction-optimizing breadth-first search [4]. In a dot product-based method, the basic computation is the dot product of two vectors, `C(i,j) = A(:,i)`$^\mathsf{T}$ `B(:,j)`, for example. The dot product typically corresponds to the "pull" phase of a direction-optimizing graph algorithm.

The next step is to determine the sparsity format of **C** and its iso property, and to select the method. For the saxpy-based strategy, if **M** is sparse or hypersparse, then **C** is hypersparse if **B** is hypersparse, or sparse otherwise. For the dot-product-based strategy, **C** has the same format as the mask **M**, if it is sparse or hypersparse and not complemented. Otherwise, **C** is computed as bitmap or full.

The dot-product methods can require more work than is strictly required, and this is accounted for in the meta-algorithm that selects saxpy versus dot. Let a and b denote the number of entries in `A(:,i)` and `B(:,j)`, respectively. If all of these entries are traversed when computing the sparse dot product, the time is $O(a + b)$. However, the actual work required is to traverse the set intersection, which is smaller than $a + b$. In spite of this, the dot-product methods are very important to many graph algorithms, such as the pull phase of a direction-optimizing breadth-first search.

4.2 Special case monoids

The algorithms described below are used for nearly all monoids and semirings, but some monoids and semirings need special consideration, which greatly speed up many key graph algorithms.

SuiteSparse:GraphBLAS includes an unusual binary operator called the *any* operator. The computation $\text{any}(x, y)$ returns either x, or y, solely at the discretion of the operator itself. The selection is not randomized. Instead, SuiteSparse:GraphBLAS uses the freedom allowed by this operator to return whichever result is fastest to compute. The *any* operator can be used to construct a monoid, since it is both associative and commutative. That is, $\text{any}(x, \text{any}(y, z)) = \text{any}(\text{any}(x, y), z)$, since in both cases, SuiteSparse:GraphBLAS can return x, y, or z, at its sole discretion.

The *any* monoid is useful in two aspects: (1) given a set of identical values to reduce, the result is always the same, and thus the *any* monoid produces a predictable iso-valued result (the same value that the min and max monoids would compute), and (2) there are cases where x and y differ, but the application will be satisfied with either result of $\text{any}(x, y)$. Computing the breadth-first search tree is one such case: if a node found at one level has many possible parent nodes at the prior level, any one of them can become a valid parent in a BFS tree. There are many valid BFS trees. The *any* monoid selects any one of these nodes as the parent. SuiteSparse:GraphBLAS uses this freedom to increase performance by relaxing the atomics in the fine-grain algorithms below, and allowing for a benign race condition. Running the method twice can give a different, yet valid, BFS tree. If a deterministic result is desired, the min or max monoids can be used instead. The GAP Benchmark BFS by Scott Beamer [4] can also return the same non-deterministic result, by selecting any valid parent. Indeed, the *any* monoid was inspired by Beamer's BFS method, as way to express this non-deterministic parent selection in the language of linear algebra over semirings, and is also related to the non-deterministic pivot search in the D2 sparse LU factorization method [14].

GraphBLAS has a simple operator $\text{one}(x, y) = 1$, which is very useful in many graph problems. Consider a $(+, \text{one})$ semiring. When combined with a dense vector $\mathbf{x}$, the computation $\mathbf{y} = \mathbf{Ax}$ with this semiring gives the row degree of the matrix $\mathbf{A}$, where y_i is the number of entries in the ith row of $\mathbf{A}$. Computing y_i can be done in constant time, without the need of traversing all the entries in the ith row, if $\mathbf{A}$ is stored by row. The column degree can be computed as $\mathbf{y} = \mathbf{A}^{\mathsf{T}}\mathbf{x}$. As a result, computing the row degree of a matrix held by row (in sparse or hypersparse format) can be computed in time proportional to the non-empty rows of the matrix. This is asymptotically far faster than the time for a conventional matrix-vector multiply, which takes at least $\Omega(e)$ time for a matrix $\mathbf{A}$ with e entries.

Finally, with many monoids, the reduction can be terminated early, when the value they are computing reaches a known *terminal* value, which is typically (but need not be) the annihilator of the operator. With the logical OR monoid, for example, as soon as a true value is seen, the computation can halt. These monoids are referred to in SuiteSparse:GraphBLAS as *terminal*, and they include min, max, logical OR, logical AND, bitwise OR, bitwise AND, times ($\times$) for integers, and the *any* monoid. The *any* monoid can terminate as soon as any value is found, which is yet another powerful aspect of this unusual operator. The times monoid for floating-point is not considered terminal, because its terminal value would not be zero, but floating-point `NaN`. That condition would be unusual to find in either a graph algorithm or numerical computation, and testing for it would slow down the typical case, so it is not exploited.

4.3 Sparse saxpy

The sparse *saxpy*-based methods are the most complex set of matrix multiplication methods in SuiteSparse:GraphBLAS. They compute $\mathbf{C} = \mathbf{AB}$, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$ for the cases when the vectors of $\mathbf{A}$ and $\mathbf{B}$ lie in the same direction (both held by column or both by row). They are

described here in column-oriented form, but the same algorithms are used for the row-oriented case, with **A** and **B** swapped. The methods divide into six phases, all of which are parallel.

The most novel aspects of the methods present here are (1) the incorporation of the mask, which is essential to exploit during the computation of $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$ to reduce the amount of work and memory required, and (2) the use of finite-state machines for fine-grain parallelism.

These methods do not handle the accumulator operator (as in $\mathbf{C} \odot= \mathbf{AB}$). Instead, these methods compute a new matrix **T** that is initially empty. If the accumulator appears, this temporary matrix is accumulated into the final output **C** matrix in a mask/accumulator step (Section 9).

Some of the methods are coarse-grained if sufficient independent tasks can be constructed for the threads. Coarse grain tasks do not need to use atomics. Others are fine-grain, where tasks share workspace and operate on the same set of results with concurrent operations resolved via atomic operations and the use of novel finite-state machines.

When **C** is computed as sparse or hypersparse, a mix of up to four different kinds of tasks are used for different parts of the output matrix. Any given matrix multiplication can use any combination of the four kinds of tasks for a single matrix **C**, depending on the sparsity structure of **A** and **B**.

(1) *coarse Gustavson*: an extension of Gustavson's algorithm [19], adapted to exploit the mask if present.
(2) *fine Gustavson*: This method is similar to the coarse Gustavson task, except that a group of threads cooperates via atomics to compute a single column of **C**, using a shared workspace, and finite-state machines.
(3) *coarse Hash*: an extension of the hash-based method [15, 29, 30], extended to exploit the mask. In a hash-based method, the Gustavson workspace of size n is replaced with a hash table for each output vector `C(:,j)`, of size equal to the number of operations required to compute it, rounded up to twice the nearest power of two.
(4) *fine Hash*: In this method, a group of threads cooperates, sharing a single hash table via atomics and finite-state machines, to compute a single column of **C**. The method of [29, 30] does not exploit the mask and does not use atomics; it is extended in this work.

When the mask **M** is bitmap or full, either all Hash tasks (fine/coarse) or all Gustavson tasks are used. For the Hash case, a bitmap mask is not scattered into the hash tables but is used in place. Otherwise, a heuristic is used to determine if a sparse or hypersparse mask should be discarded (and applied later), in case the mask is too costly to exploit during the matrix multiply. The heuristic discards the mask if the work to compute **AB** is less than 1% of the number of entries in **M**.

Coarse tasks are allocated to a single thread, and compute a contiguous range of columns, `C(:,j1:j2) = A*B(:,j1:j2)` for a unique set of vectors `j1:j2`. Those vectors are not shared with any other tasks. A fine task works with a team of other fine tasks to compute `C(:,j)` for a single vector `j`. Each fine task in a team computes `A*B(k1:k2,j)` for a unique range `k1:k2`, and sums its results into `C(:,j)` via atomic operations (using the monoid of the semiring) and finite-state machines.

Each of these four kinds of tasks are subdivided into three variants, for $\mathbf{C} = \mathbf{AB}$ when no mask is present, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ with the mask **M**, and $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$ with a complemented mask **M**. This results in a total of 12 different algorithms, each of which are subdivided into six parallel phases. Most of these methods are novel, but what is presented here is a deep dive into two of them, both with a non-complemented mask: $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ with the coarse-grain Gustavson method and the fine-grain Hash-based method. Fine-grain methods for a complemented mask are similar, but with different transitions in the finite-state machine.

Fine tasks are used when there would otherwise be too much work for a single task to compute the single vector `C(:,j)`. Fine tasks share all of their workspace with the team of fine tasks computing

C(:,j). Coarse tasks are preferred since they require less synchronization, but fine tasks allow for better parallelization when **B** has only a few vectors. If **B** consists of a single vector (for GrB_mxv if **A** is in CSC format and not transposed, or for GrB_vxm if **A** is in CSR format and not transposed), then the only way to get parallelism is via fine tasks. If a single thread is used for this case, a single-vector coarse task is used.

4.3.1 **Sparse saxpy, phase 0: work estimates and task creation.** To select between the Hash method or Gustavson's method for each task, the hash table size is first found. The hash table size (s) for a hash task depends on the maximum work required for any vector in that task (which is just one vector for the fine tasks). It is set to twice the smallest power of 2 that is greater than the work to compute that vector (plus the number of entries in M(:,j) for tasks that compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$). This size ensures the results will fit in the hash table, and with ideally only a modest number of collisions. If the hash table size exceeds a threshold (m/16 if C is m-by-n), then Gustavson's method is used instead, with a gather/scatter workspace of size m.

The workspace allocated depends on the type of task. Let s be the hash table size for the task, and **C** is m-by-n and held by column, and where ctype refers to the datatype of **C** and also the type of the monoid. Then the workspace is as follows:

- fine Hash task (shared): int64_t Hf[s] ; ctype Hx[s] ;
- coarse Gustavson task: int64_t Hf[m] ; ctype Hx[m] ;

The Hx array holds the numerical values of C(:,j), using Hx[i] for C(i,j) for the Gustavson method, and Hx[hash(i)] for the hash-based methods, where hash(i) is the hash function used. For fine Hash tasks, the finite state machine for entry hash(i) is held in the lowest 2 bits of Hf[hash(i)], with additional space of 62 bits for an integer index. The coarse Gustavson tasks use Hf[i] as a mark value for set union computations, as a single 64-bit integer.

Once the sparsity format and iso property of **C** is determined, the work analysis is performed, which computes the work required to compute each vector C(:,j). The total work for the entire matrix multiply is then subdivided into tasks, which can be an arbitrary mix of all four types (fine/coarse × Gustavson/Hash). This step may also decide that the mask is too costly to apply during the matrix multiplication. If so, the computation $\mathbf{C} = \mathbf{AB}$ is computed without the mask, and the mask is applied later. The workspaces are allocated for each task, and then each of the five phases below are done completely in parallel, with a barrier between each phase.

4.3.2 **Sparse saxpy, phase 1: symbolic analysis.** In this phase, coarse tasks compute the number of entries in each of their vectors, C(:,j1:j2). Fine tasks simply scatter the mask **M** into their hash tables. This phase does not depend on the semiring.

The hash table for a team of fine tasks is the int64_t array Hf, of size s where s is a power of two. A very simple hash function is used to index into this hash table (hash = mod(257*i,s)), with linear probing if the hash location Hf[h] is occupied. Since s is a power of 2, the mod operation is a bit-mask operation, and multiplying by 257 is computed as (i << 8) + i.

The 64-bit hash entry is packed with two terms: an index h (up to 62 bits) and the state for a finite-state machine (f, which takes 2 bits), to support atomic access to each hash entry. An empty hash position contains h=0 and f=0. Initially, if c_{ij} and the mask entry m_{ij} are to be hashed into Hf[hash] = (h,f), then it contains h=i+1 and f=1. Later on, the states f=2 and f=3 will be used in different ways, depending on the algorithm.

In phase 1, coarse tasks compute the number of entries in each vector of C(:,j1:j2). Two different methods are used in the innermost loop of a coarse task:

(1) If A(:,k) is sparse enough relative to M(:,j), then Algorithm 2 is used, taking $O(a)$ time if a is the number of entries in A(:,k).

ALGORITHM 2: Coarse Gustavson task, phase 1, for $\mathbf{C\langle M\rangle = AB}$

```
mark = 0
for each column j assigned to this task do
    mark = mark + 2                          // clear Hf, now all Hf(0:m-1) < mark
    Cp(j) = 0                                // # of entries in C(:,j)
    for each entry present in M(:,j) do
        Hf(i) = mark
    for each entry b_kj in B(:,j) do
        for each entry a_ik in A(:,k) do
            if Hf(i) == mark then
                // m_ij is 1 and c_ij is not yet seen
                Hf(i) = mark+1               // c_ij has now been seen
                Cp(j) ++
```

(2) For all four kinds of tasks (fine/coarse × Gustavson/Hash), a push/pull optimization is employed, which is a revised variant of Algorithm 2. If `M(:,j)` is much much sparser than `A(:,k)`, then traversing all of `A(:,j)` just to add few entries to `C(:,j)` is costly. So instead, the mask vector `M(:,j)` is traversed, and for each entry, a binary search is performed to find the entry `A(i,k)`. If it appears, the same innermost `if` block is performed. This pull-style optimization is not shown in Algorithm 2, but the change is simple. The inner loop that traverses `A(:,k)` is replaced by a loop that traverses all entries m_{ij} in `M(:,j)`, and then a_{ik} is found with a binary search. If μ and a are the number of entries in `M(:,j)` and `A(:,k)`, this takes $O(\mu \log a)$ time, which is faster than the linear-time method if $\mu \ll a$.

4.3.3 **Sparse saxpy, phase 2: symbolic/numeric work for fine tasks.** Phase 2 is where the fine-grain tasks do the bulk of their work, by computing the structure and values of `C(:,j)` in their hash tables. Coarse tasks do nothing in this phase. There are seven different kinds of fine tasks to consider in this phase: two methods (Gustavson/Hash), and the four kinds of mask (no mask, mask bitmap or full for the Hash method, non-complemented sparse mask, and complemented sparse mask). Only the fine Hash method with a non-complemented mask sparse or hypersparse mask $\mathbf{M}$ is presented here. The fine Hash tasks use atomics and finite-state machines. All tasks in a team computing `C(:,j)` share a single size-`s` hash table. These tasks can be computed over any semiring, but to simplify the presentation, the $(+, \times)$ semiring is assumed.

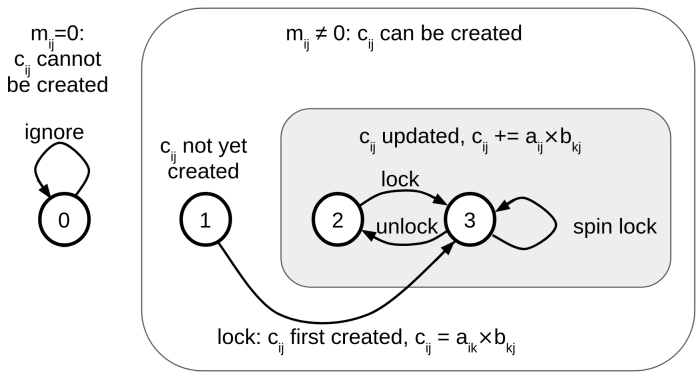
Algorithm 3 is the fine-Hash task for phase 2 when the mask is present, sparse, and not complemented. The state is `f`, which is the lower two bits from `(h,f)= Hf[hash]`. If `(h,f) = (0,0)`, then the hash entry is unlocked and unoccupied; c_{ij} cannot appear because the mask will not allow it, so this state will remain unchanged. A hash collision occurs if `h` is not equal to `i+1`. If `(h,f) = (i+1,1)` then it is unlocked and occupied by $m_{ij} = 1$, c_{ij} has not been seen, and `Hx[hash]` is not initialized. If `(h,f) = (i+1,2)` then it is unlocked and occupied by $m_{ij} = 1$ and `Hx[hash]` holds c_{ij}. The finite-state machine is illustrated in Figure 1, with the following states:

- state 0: $m_{ij} = 0$, and thus c_{ij} cannot be created. This state is permanent.
- state 1: $m_{ij} = 1$, and c_{ij} can be created but it does not yet exist.
- state 2: $m_{ij} = 1$, and c_{ij} has been created and initialized. The state is unlocked. If the monoid can be implemented in an atomic update, this state does not change and all future updates are performed atomically.

ALGORITHM 3: Fine Hash task, phase 2, for $\mathbf{C\langle M\rangle = AB}$

```
for each entry b_kj in B(k1:k2,j) do
    for each entry a_ik in A(:,k) do
        hash = (257i) mod s                          // first hash function for entry c_ij
        while true do
            hf = Hf(hash)                                   // atomic read
            if hf == (i+1,2) and monoid has an OpenMP atomic equivalent then
                Hx(i) += a_ik × b_kj                  // (state 2): atomic update of c_ij
                break
            if hf == (0,0) then
                break                                       // (state 0): m_ij =0, ignore c_ij
            h = hf / 4
            if h == i+1 then
                // hash entry occupied by c_ij
                repeat
                    hf = Hf(hash), Hf(hash) |= 3            // atomic swap and bitwise OR
                until (hf & 3) ≠ 3
                if hf == (i+1,1) then
                    Hx(hash) = a_ik × b_kj                  // (1 → 3) : c_ij is a new entry; atomic write
                else if hf == (i+1,2) then
                    Hx(hash) += a_ik × b_kj                 // (2 → 3) : atomic update of c_ij
                Hf(hash) = (i+1,2)                  // (3 → 2): unlock the entry; atomic write
                break
            hash = (hash+1) mod s  // linear probing if hash entry already occupied by another entry
```

Fig. 1. Finite state machine for $\mathbf{C\langle M\rangle = AB}$ for fine-grain tasks



- state 3: $m_{ij} = 1$, and the state is locked. The entry c_{ij} has already been created, and can be modified with a non-atomic monoid (if the monoid is user-defined, for example), but only by the thread that owns the lock.

If the mask is complemented, then a similar finite-state machine is used, where states 0 and 1 trade places.

4.3.4 **Sparse saxpy, phases 3 and 4: compute final column counts.** In phase 3, fine tasks examine their workspace (the hash table or Gustavson workspace) and compute the number of entries they contain. Next, phase 4 constructs the column pointers `C.p` from the column counts, via a single parallel cumulative sum, for all tasks.

4.3.5 **Sparse saxpy, phase 5: symbolic/numeric phase for coarse tasks, gather for fine tasks.** Teams of fine tasks have already computed their assigned `C(:,j)` vector in their respective workspaces; they now gather their results into the final location of `C(:,j)` in the output matrix **C**. Coarse tasks perform their symbolic and numerical work in phase 5, shown in Algorithm 4.

ALGORITHM 4: Coarse Gustavson task, phase 5, for $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$

```
mark = 0
for each column j assigned to this task do
    mark = mark + 2                          // clear Hf, now all Hf(0:m-1) < mark
    pC = Cp(j)                               // start position of C(:,j)
    for each entry present in M(:,j) do
        Hf(i) = mark
    for each entry b_kj in B(:,j) do
        for each entry a_ik in A(:,k) do
            if Hf(i) == mark then
                // m_ij is 1 and c_ij is not yet seen
                Hf(i) = mark+1               // c_ij has now been seen
                Hx(i) = a_ik × b_kj          // initialize c_ij
                Ci(pC++) = i
            else if Hf(i) == mark + 1 then
                Hx(i) += a_ik × b_kj                                  // update c_ij
        gather all of C(:,j) from the workspace
```

4.4 Sparse masked dot-product

The sparse masked dot-product method is very specialized, computing $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}^{\mathsf{T}}\mathbf{B}$ when the mask **M** is sparse or hypersparse, and not complemented. It is only used when the mask matrix is present. For this case, the matrix **C** has the same sparsity structure as the mask **M**. In case the dot product `C(i,j)=A(:,i)'*B(:,j)` of the two vectors `A(:,i)` and `B(:,j)` is empty, the resulting entry c_{ij} becomes a zombie, and is deleted later. The work divides cleanly into a set of coarse-grain tasks, with no atomics required. Each task iterates over its assigned part of **C** and **M**, computing a set of sparse dot products. Unlike the saxpy method, the dot products require the vectors of **A** and **B** to be sorted on input. How the dot products are computed depends on the sparsity format of **A** and **B**. Let a and b be the number of entries in `A(:,i)` and `B(:,j)`, respectively. If both vectors are sparse, a merge is performed, taking $O(a+b)$ time. If $a \ll b$, the vector `A(:,i)` is scanned and a binary search is performed on `B(:,j)` instead, taking $O(a \log b)$ time. Additional methods exploit the case when **A** and/or **B** are bitmap or full, which require no binary search.

5 PARALLEL ELEMENT-WISE ADD (SET UNION)

The element-wise sparse matrix addition $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ computes a matrix $\mathbf{C}$ whose sparsity structure is the set union of $\mathbf{A}$ and $\mathbf{B}$ (or a subset of that structure if revised by the mask). A binary operator is applied to entries in the intersection of $\mathbf{A}$ and $\mathbf{B}$, while entries in one but not both input matrices are copied to the output matrix unchanged except for possible typecasting.

The parallel sparse matrix addition in SuiteSparse:GraphBLAS is split into three parallel phases when $\mathbf{C}$ is sparse or hypersparse (Methods 1 to 20 in the list below):

- Phase 0: The first phase computes the set of vectors in $\mathbf{C}$, if it is hypersparse. This is the set union of the vectors in $\mathbf{A}$ and $\mathbf{B}$, or if $\mathbf{M}$ is present, not complemented, and hypersparse, then its vectors define the vectors in the hypersparse matrix $\mathbf{C}$.
- Phase 1: The second phase splits $\mathbf{C}$ into parallel tasks, using a mixture of coarse and fine-grain tasks. Coarse grain tasks compute a region `C(:,j1:j2)`; while fine-grain tasks compute a subvector, `C(i1:i2,j)`, which is computed without the need for atomics. Each task then counts the entries in its region, using one of 20 internal kernels, depending on the sparsity of the individual input vectors and the presence of the mask, for each of the cases listed below. This is followed by a cumulative sum.
- Phase 2: The third phase follows the same workflow as the second pass, but computes the structure and values of the entries of $\mathbf{C}$.

The first 10 methods listed below are used for $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$ when no mask is present. Methods 11 and 13 are used for the special case when the mask is aliased with one or both of the two input matrices. Let a, b, and μ denote the number of entries in `A(:,j)`, `B(:,j)`, and `M(:,j)`, respectively. If the mask is complemented, $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$ is computed without the mask, and it is applied later.

(1) `A(:,j)` and `B(:,j)` dense: thus `C(:,j)` is dense. Phase 1 takes $O(1)$ time, while phase 2 is a dense loop.
(2) `A(:,j)` dense, `B(:,j)` sparse: thus `C(:,j)` is dense. Phase 1 takes $O(1)$ time, while phase 2 iterates first over `A(:,j)`, scattering the sparse vector into `C(:,j)`.
(3) `A(:,j)` sparse, `B(:,j)` dense: thus `C(:,j)` is dense, the reverse of Method 2.
(4) `A(:,j)` is empty, and thus `C(:,j)=B(:,j)`.
(5) `B(:,j)` is empty, and thus `C(:,j)=A(:,j)`.
(6) the last entry in `A(:,j)` comes before the first entry in `B(:,j)`, and thus `C(:,j)` is a simple concatenation.
(7) the last entry in `B(:,j)` comes before the first `A(:,j)`, the reverse of Method 6.
(8) `A(:,j)` is much denser than `B(:,j)`, and thus `A(:,j)` is traversed to compute the size of the set union (phase 1 only), with a binary search of `B(:,j)`. This method (and Method 9) are critical when the sparsity of $\mathbf{A}$ and $\mathbf{B}$ differ greatly.
(9) `B(:,j)` is much denser than `A(:,j)`, the reverse of Method 8.
(10) `A(:,j)` and `B(:,j)` about the same sparsity, so a merge of the two sorted lists is used, taking $O(a+b)$ time. This is a very common case.
(11) `A(:,j)` is dense and $\mathbf{B}$ is aliased with $\mathbf{M}$. Only `M(:,j)` is traversed in $O(\mu)$ time.
(12) `B(:,j)` is dense and $\mathbf{A}$ is aliased with $\mathbf{M}$, the reverse of Method 11.
(13) $\mathbf{A}$, $\mathbf{M}$, $\mathbf{B}$ are all aliased to one another.
(14) $\mathbf{C}$ and $\mathbf{M}$ are sparse or hypersparse: the vector `M(:,j)` is traversed, and entries in `A(:,j)` and `B(:,j)` are found with a binary search (if sparse) or $O(1)$ lookup (if dense). The time taken is $O(\mu(\log a + \log b))$. This method works well if $\mathbf{M}$ is very sparse, and is far faster than applying the mask later, since in this case, $\mu \ll a$ and $\mu \ll b$, and applying the mask later could require Method 10 to be used instead, whose time is $O(a+b)$.
(15) `A(:,j)` and `B(:,j)` dense, $\mathbf{M}$ bitmap/full.

(16) `A(:,j)` is empty, **M** bitmap/full: the vector `B(:,j)` is traversed, and entries are copied to `C(:,j)` if permitted the the mask using an $O(1)$ lookup of the mask.
(17) `B(:,j)` is empty, **M** bitmap/full: the reverse of Method 16.
(18) the last entry in `A(:,j)` comes before the first entry in `B(:,j)`, and **M** is bitmap/full: a simple concatenation, where each entry is checked with an $O(1)$ lookup of the mask.
(19) the last entry of `B(:,j)` comes before the first entry in `A(:,j)`, and **M** is bitmap/full: the reverse of Method 19.
(20) `A(:,j)` and `B(:,j)` arbitrary, but with **M** bitmap/full. Like Method 10, except with an $O(1)$ lookup of each entry in the mask. This is a very common case.

The matrix **C** is computed in bitmap form when computing $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ when either **A** or **B** are bitmap or full, except in the case for $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ when **M** is sparse or hypersparse and not complemented. In that case, **C** is computed as either sparse or hypersparse, using the 20 methods above. The bitmap method for $\mathbf{A} \oplus \mathbf{B}$ is not presented here.

6 PARALLEL ELEMENT-WISE MULTIPLY (SET INTERSECTION)

When using the times ($\times$) operator, the element-wise multiply $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ (`C=A.*B` in MATLAB notation) computes the Hadamard product, but in GraphBLAS any binary operator can be used instead. The structure of **C** is the set intersection of **A** and **B**. The parallel element-wise multiply first considers the 7 special cases in the list below. These are typically one-pass or two-pass methods. If none of the first 7 cases hold, a general purpose method (Method 8) is used, which takes three passes, just like many of the element-wise add techniques described in the prior Section 5.

(1) **A** and **B** are both full, with any type of mask: the element-wise multiply (set intersection) is identical to the element-wise add, and so the latter is used (Section 5).
(2) **A** is sparse or hypersparse, and **B** is bitmap or full, and the mask is either not present, sparse/hypersparse but applied later, or bitmap/full. The matrix **C** is sparse or hypersparse. Three different methods are used, all of which slice the matrix **A** (Section 3.4).
   (a) $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ with no mask, and **B** is bitmap. A first pass traverses **A** and does an $O(1)$ lookup into the bitmap of **B** to count the entries in each vector of **C**. A second pass of **A** constructs the structure and values of **C**.
   (b) $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ with no mask, and **B** is full. The structure of **C** is identical to **A**, so no symbolic phase is necessary. A single pass of **A** is performed.
   (c) $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \otimes \mathbf{B}$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \otimes \mathbf{B}$ where **B** and **M** are bitmap or full. This method is identical to Method 2(a) above, except that the mask **M** is also checked, using an $O(1)$-time lookup.
(3) This method is the same as Method 2 above, with the roles of **A** and **B** reversed. That is, **A** is bitmap or full and **B** is sparse or hypersparse. It also subdivides into the three sub-methods listed above.
(4) $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \otimes \mathbf{B}$ where **A** and **B** are both bitmap or full (with at least one of them bitmap), and the mask is sparse or hypersparse and not complemented. The matrix **C** is sparse or hypersparse, and its sparsity structure is a subset of **M**. The sparse matrix **M** is sliced and traversed twice (Section 3.4). The first phase counts the entries in each vector of **C**, and the second phase computes the structure and values of **C**.
(5) $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ where **A** and **B** are bitmap or full (with at least one of them bitmap) and no mask is present. **C** is bitmap. A simple one-pass algorithm computes the structure of values of **C**.
(6) $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \otimes \mathbf{B}$ where **A** and **B** are bitmap or full (with at least one of them bitmap) and the mask is sparse and complemented. **C** is bitmap. The mask is sliced (Section 3.4), and scattered

into the bitmap of **C**. The second phase computes the values and structure of **C**, checking the bitmap of **A** and/or **B** in $O(1)$ time per entry.

(7) $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \otimes \mathbf{B}$ where **A** and **B** are bitmap or full (with at least one of them bitmap) and the mask bitmap or full (either complemented or not). **C** is bitmap. The method is nearly the same as Method 5 above, except the mask is checked with an $O(1)$ lookup, per entry.

(8) Otherwise, a general purpose method is used, described below.

The general purpose element-wise multiply constructs the matrix **C** as sparse or hypersparse, and the mask may be present (which may or may not be complemented). Three phases are used, each of which are parallel.

- Phase 0: the vectors of **C** to compute are determined, and mappings are constructed from the vectors of **A**, **B**, and **M** (if present), if any of them are hypersparse. The matrix **C** is hypersparse if any of the three input matrices are hypersparse.
- Phase 1 constructs the parallel tasks for this phase and phase 2, using the same algorithm described in Phase 1 of Section 5, the element-wise add. Phase 1 then counts the number of entries in each vector of **C** and performs a cumulative sum.
- Phase 2 computes the structure and values of **C**. Each vector `C(:,j)` (or sub-vector, if it is split across different parallel tasks), is computed using one of eight different methods, listed below. The goal of each method is to compute the set intersection of `A(:,j)` and `B(:,j)`, also under control of the mask `M(:,j)` if present.
  - (a) `C(:,j)` is empty if `A(:,j)` or `B(:,j)` are empty, or if their structures do not overlap (an $O(1)$-time check).
  - (b) If no mask is present and `A(:,j)` has many more entries than `B(:,j)`, then `B(:,j)` is traversed, and a binary search is used to find the corresponding entry in `A(:,j)`, taking $O(b \log a)$ time.
  - (c) the same as (b) with **A** and **B** reversed.
  - (d) If `A(:,j)` and `B(:,j)` have roughly the same number of entries, they are scanned together, like the merge of a mergesort, taking $O(a + b)$ time.
  - (e) **M** is sparse or hypersparse (and not complemented). The entries in `M(:,j)` are traversed, and entries in `A(:,j)` and `B(:,j)` are found via binary searches. The time taken is $O(\mu(\log a + \log b))$. If **M** is complemented, it is not used during the element-wise multiply, and applied later after $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ is computed.
  - (f), (g), (h) **M** is bitmap or full. These are the same as methods (b), (c), and (d), except that the mask is also checked, taking $O(1)$ per entry in the intersection of `A(:,j)` and `B(:,j)`.

7 PARALLEL SUBMATRIX EXTRACTION

The GraphBLAS syntax $\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{A}(\mathbf{I},\mathbf{J})$ extracts a submatrix of **A**, or `A(I,J)` in MATLAB notation, where `I` and `J` are ordered lists of row and column indices. The mask and accumulator are currently not exploited by SuiteSparse:GraphBLAS during the extraction of $\mathbf{A}(\mathbf{I},\mathbf{J})$ so only the parallel algorithm for $\mathbf{C} = \mathbf{A}(\mathbf{I},\mathbf{J})$ is described. As an extension the GraphBLAS C API Specification, SuiteSparse:GraphBLAS allows for a strided range of indices to be provided, as the three integers in the MATLAB notation `lo:stride:hi`. If **A** is sparse or hypersparse, the extraction splits into four phases, each of which are parallel.

- Phase 0 finds the vectors that will appear in **C**, and determines the properties of `I` and `J` (whether or not they are sorted or unsorted, if they have duplicates, their minimum and maximum values, and whether or not they form a contiguous range of indices). If **C** will be hypersparse, the `C.h` component is constructed in this phase (some of these vectors may turn out to be empty), and their mapping to the vectors of **A** is found.

- Phase 1 constructs a set of independent tasks for the subsequent phases, as a mix of both coarse and fine-grain tasks. Coarse tasks operate on a set of one or more whole vectors of **C**, while fine-grain tasks construct a subset of a single vector of **C**. No atomics are needed for the fine-grain tasks. Assuming all matrices are held by column, this phase also constructs an inverse of the list `I`, if needed.
- Phase 2 counts the number of entries that will appear in each vector of **C**. This phase uses the same set of methods described below for phase 3, with some simplifications because only the counts are required, not the actual indices or values of the entries in **C**. This is followed by a cumulative sum so that all tasks will know where to place their entries in the output matrix **C**.
- Phase 3 constructs the structure and values `C=A(I,J)`. Each vector `A(:,j)` (or sub-vector `C(i1:i2,j)=A(I(i1:i2),j)`) is traversed using one of twelve methods. Eleven of these 12 methods appear in [11], and are used here nearly unmodified, except the iso property is handled in v7.1.2 of SuiteSparse:GraphBLAS. The 12th method handles the special case where `I` has the form `hi:(-1):lo`, which did not appear in v2.1.3.

8 PARALLEL SUBMATRIX ASSIGNMENT

Submatrix assignment, $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) \odot= \mathbf{A}$ is a very complex operation because of the many possible parameters. Minor variations in the parameters have a large impact on the optimal algorithm to implement the assignment, and the parallelization of each algorithm is often different. Some of the key parameters that affect this method include:

- The sparsity format of the input/output matrix, **C**. Modifying a sparse or hypersparse matrix is entirely different than modifying a bitmap or full matrix.
- The mask matrix **M** can be present or not, and if present, it can be complemented. It may be used in a structural sense or valued sense. Each of these variations leads to entirely different algorithms required. Other special cases can occur, such as when the mask is **C** itself.
- The assignment may affect the entire matrix, as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}$, or it may affect a submatrix, if **I** and **J** are present.
- The accumulator operator can be present or not. If present, it means that entries in **C** cannot be deleted by the assignment.
- The right-hand side can be a matrix **A**, of size $|\mathbf{I}|$-by-$|\mathbf{J}|$, or it can be a scalar. Scalar assignment is very different, as it implicitly expands into a dense matrix of the desired size.
- The *replace* descriptor changes how entries in **C** are modified if not written to by the assignment. If set, an entry c_{ij} is deleted unless it is modified by the assignment. This is written as $\mathbf{C}\langle\mathbf{M}, \textit{replace}\rangle$. The *replace* option can be true or false, and it has a subtle interplay on how it affects the assignment, particularly if the indices **I** and **I** are present, and if the mask **M** is present. It takes a full page table to describe all the variations; refer to the User Guide for details.
- The parallelization of all these variants can vary widely, but fortunately many share similar algorithmic and data access patterns, and can share parallelization techniques.

In ACM Algorithm 1000 (v2.3.5 SuiteSparse:GraphBLAS), its implementation required nearly 4,000 lines of code [11]. In v7.1.2, this has expanded by a factor of 3, not merely because of parallelism. The scope of these different methods is too broad to fully describe here, so instead an overview is given, along with a categorization of how the methods are parallelized.

For many of the methods, the assignment is preceded by a symbolic extraction, where the matrix $\mathbf{S} = \mathbf{C}(\mathbf{I},\mathbf{J})$ is constructed. The values in **S** are not the values of the entries in **C**, but their locations

in the data structure for $\mathbf{C}$. This process is described in [11], except that $\mathbf{S}$ is constructed in parallel using all the methods described in Section 7.

In the *GraphBLAS C API Specification*, the matrix $\mathbf{C}$ and its mask $\mathbf{M}$ have the same size. SuiteSparse:GraphBLAS adds `GxB_subassign`, where $\mathbf{M}$ has the size of the submatrix `C(I,J)`, written as $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle \odot= \mathbf{A}$. The `GrB_assign`, written as $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) \odot= \mathbf{A}$, is translated into the equivalent `GxB_subassign` and so only the latter is described here.

The methods when $\mathbf{C}$ is sparse or hypersparse divide into 10 different categories, depending on how they are parallelized, listed below. The notation $\mathbf{C}\langle\mathbf{M}\rangle$ denotes either a valued or structural mask, whereas $\mathbf{C}\langle s(\mathbf{M})\rangle$ denotes a purely structural mask.

(1) **Parallel traversal of $\mathbf{M}$**: The matrix $\mathbf{M}$ is sliced (Section 3.4).
- $\mathbf{C}(:,:)\langle s(\mathbf{M})\rangle = \mathbf{A}$ where $\mathbf{C}$ is initially empty and $\mathbf{A}$ is full.

(2) **Parallel traversal of all entries in the Cartesian product** $\mathtt{I} \times \mathtt{J}$. All of these methods construct the matrix $\mathbf{S}$.
- $\mathbf{C}(\mathbf{I},\mathbf{J}) = \sigma$
- $\mathbf{C}(\mathbf{I},\mathbf{J}) \odot= \sigma$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle = \sigma$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle \odot= \sigma$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, \mathit{replace}\rangle = \sigma$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, \mathit{replace}\rangle \odot= \sigma$

(3) **Element-wise add of $\mathbf{A} + \mathbf{S}$**: These methods construct the matrix $\mathbf{S}$, and must traverse all entries that would appear in the element-wise addition of $\mathbf{A}+\mathbf{S}$. This is not computed, but the same algorithms used in Section 5 are used to construct the parallel tasks for these methods.
- $\mathbf{C}(\mathbf{I},\mathbf{J}) = \mathbf{A}$. This is identical to `C(I,J)=A` in MATLAB, except that SuiteSparse:GraphBLAS can be 1000x faster than MATLAB (see Section 10.2).
- $\mathbf{C}(\mathbf{I},\mathbf{J}) \odot= \mathbf{A}$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle = \mathbf{A}$ when $\mathbf{A}$ is sparser than $\mathbf{M}$, or if either are bitmap (see Method (5) when this condition does not hold). This is akin to `C(M)=A(M)` in MATLAB, except that SuiteSparse:GraphBLAS can be 248,000x faster than MATLAB.
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle \odot= \mathbf{A}$ where both $\mathbf{M}$ and $\mathbf{A}$ are bitmap (see Method (8) when this condition does not hold).
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}, \mathit{replace}\rangle = \mathbf{A}$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}, \mathit{replace}\rangle \odot= \mathbf{A}$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle = \mathbf{A}$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle \odot= \mathbf{A}$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, \mathit{replace}\rangle = \mathbf{A}$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, \mathit{replace}\rangle \odot= \mathbf{A}$

(4) **Element-wise add of $\mathbf{M} + \mathbf{S}$**: These methods construct the matrix $\mathbf{S}$, and must traverse all entries that would appear in the element-wise addition of $\mathbf{M}+\mathbf{S}$. The term $\mathbf{M}+\mathbf{S}$ itself is not computed, but the same parallel techniques of Section 5 are used. If $\mathbf{M}$ is bitmap or full, all entries in the Cartesian product $\mathtt{I} \times \mathtt{J}$ are traversed.
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}, \mathit{replace}\rangle = \sigma$
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}, \mathit{replace}\rangle \odot= \sigma$

(5) **Fine/Coarse partitioning of $\mathbf{M}$**: the matrix $\mathbf{M}$ is split into a set of fine and coarse tasks, where a coarse task operates on a set of whole vectors of $\mathbf{M}$, and a fine task operates on a subset of a single vector of $\mathbf{M}$.
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle = \sigma$

- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle = \mathbf{A}$ (see also Method (3) above). This assignment has two methods that can be used to implement it. This method is used when $\mathbf{M}$ is sparser than $\mathbf{A}$. This akin to logical indexing in MATLAB (`C(M)=A(M)`), except that SuiteSparse:GraphBLAS can be 248,000x faster than MATLAB (see Section 10.2).
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle \odot= \sigma$

(6) **Simple copy**: the matrix $\mathbf{M}$ or $\mathbf{A}$ is copied into $\mathbf{C}$
- $\mathbf{C}(:,:)\langle s(\mathbf{M})\rangle = \sigma$ where the structure of $\mathbf{M}$ is copied into $\mathbf{C}$, and $\mathbf{C}$ becomes iso-valued. The equivalent in MATLAB would be `C=s*spones(M)`, except that MATLAB does not have a way to represent iso-valued matrices.
- $\mathbf{C}(:,:) = \mathbf{A}$

(7) **Constant time:** these methods take $O(1)$ time and thus are not parallelized. The matrix $\mathbf{C}$ becomes iso-valued.
- $\mathbf{C}(:,:)\langle s(\mathbf{C})\rangle = \sigma$ where $\mathbf{C}$ and the mask are aliased to each other. This is analogous to but far faster than `C=spones(C)` in MATLAB.
- $\mathbf{C}(:,:) = \sigma$ where $\mathbf{C}$ becomes an iso full matrix. This takes $O(1)$ time and memory, and is like `C=s*ones(m,n)`, except that the latter takes $O(mn)$ time and memory in MATLAB.

(8) **Element-wise multiply of $\mathbf{A} \otimes \mathbf{M}$**: This method is uniquely handled with its own parallelization technique. The matrices $\mathbf{M}$ and $\mathbf{A}$ are not bitmap. The matrix $\mathbf{S}$ is not constructed. The parallelization is the same as the element-wise multiplication of $\mathbf{A} \otimes \mathbf{M}$ and thus the methods described in Section 6 are used.
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle \odot= \mathbf{A}$

When $\mathbf{C}$ is full (and one case when it is either bitmap or full), three parallel methods are used for five different kinds of assignments:

(1) **simple:** all entries in $\mathbf{C}$ are modified with a simple parallel loop.
- $\mathbf{C}(:,:) \odot= \sigma$ where $\mathbf{C}$ is full.

(2) **parallel traversal of $\mathbf{A}$**: The matrix $\mathbf{A}$ is sliced (Section 3.4).
- $\mathbf{C}(:,:)\langle\mathbf{A}\rangle = \mathbf{A}$ where $\mathbf{C}$ is bitmap or full.
- $\mathbf{C}(:,:) \odot= \mathbf{A}$ where $\mathbf{C}$ is full.

(3) **parallel traversal of $\mathbf{M}$**: The matrix $\mathbf{M}$ is sliced (Section 3.4).
- $\mathbf{C}(:,:)\langle\mathbf{M}\rangle = \sigma$ where $\mathbf{C}$ is full.

Finally, 16 different methods are used when $\mathbf{C}$ is bitmap, or becomes so because of the assignment. A parallel traversal of the bitmap of $\mathbf{C}$ is simple and not described here in detail. In some of these methods, the sparse matrices $\mathbf{A}$ or $\mathbf{M}$ are sliced (Section 3.4), and used to assign into the bitmap of $\mathbf{C}$. Other methods traverse the Cartesian product of $\mathtt{I} \times \mathtt{J}$ in the bitmap of $\mathbf{C}$.

9 PARALLEL MASK/ACCUMULATOR PHASE

Finally, all of the GraphBLAS operations described so far compute $\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{Z}$ for some computed result $\mathbf{Z}$, followed by the assignment into $\mathbf{C}$ via an optional mask $\mathbf{M}$ and accumulator operator (which can be any binary operator). Many operations exploit the mask themselves, to reduce the time and memory to compute $\mathbf{Z}$, but only the `GrB_assign` and `GxB_subassign` operations fully implement the entire mask/accumulator step themselves. For all of the others, a final mask/accumulator step is performed, described here.

Let $\mathbf{C}$ denote the matrix on input to this step, and let $\mathbf{R}$ denote the final result. Let $\mathbf{T}$ denote the intermediate result from another operation. If the accumulator operator is present, $\mathbf{Z} = \mathbf{C} \oplus \mathbf{T}$ is computed via the element-wise add (Section 5); otherwise, $\mathbf{Z}$ is the same as $\mathbf{T}$.

The next step can be written as the (implicit) copy $\mathbf{R} = \mathbf{C}$, followed by $\mathbf{R}\langle\mathbf{M}\rangle = \mathbf{Z}$, which is the assignment via the mask. The mask may be complemented, and the *replace* option must also be

problem: $N = 100$	$C = A^2$		$C = A^4$		$C = A^8$	
	work: 0.016 Gflop		work: 0.188 Gflop		work: 2.466 Gflop	
threads	time	speedup	time	speedup	time	speedup
1	0.043	1.0	0.297	1.0	3.087	1.0
8	0.006	7.2	0.049	6.1	0.436	7.1
20	0.007	6.1	0.023	12.9	0.194	15.9
40	0.007	6.1	0.022	13.5	0.171	18.1
problem: $N = 400$	$C = A^2$		$C = A^4$		$C = A^8$	
	work: 0.264 Gflop		work: 3.054 Gflop		work: 41.168 Gflop	
threads	time	speedup	time	speedup	time	speedup
1	0.553	1.0	4.900	1.0	52.590	1.0
8	0.099	5.6	0.669	7.3	6.862	7.7
20	0.071	7.8	0.356	13.8	2.921	18.0
40	0.054	10.2	0.296	16.6	2.728	19.3
problem: $N = 1500$	$C = A^2$		$C = A^4$		$C = A^8$	
	work: 3.716 Gflop		work: 43.112 Gflop		work: 584.862 Gflop	
threads	time	speedup	time	speedup	time	speedup
1	7.861	1.0	68.489	1.0	758.972	1.0
8	1.236	6.4	9.422	7.3	97.139	7.8
20	0.819	9.6	4.344	15.8	41.436	18.3
40	0.650	12.1	4.035	17.0	38.540	19.7

Table 2. Wathen matrix results: time in seconds, work in billions of floating point operations (each multiply-add pair counted as two flops)

handled. The operation is similar to both $\mathbf{R} = \mathbf{C} \odot \mathbf{Z}$ via the element-wise add, and $\mathbf{R} = \mathbf{C} \otimes \mathbf{Z}$ via the element-wise multiply, depending on the value of the mask.

As a result, the first phase is identical to the Phase 0 of the element-wise add, which determines the vectors in $\mathbf{R}$. The parallelization in Phase 1 is also identical. The remainder of the method is much like the element-wise add of $\mathbf{C} \oplus \mathbf{Z}$, except for how the cases are during the merge of the two vectors `C(:,j)` and `Z(:,j)`, which is modified by the mask.

10 PERFORMANCE

The performance results below highlight the parallel scaling of `GrB_mxm` (Section 10.1), and provide comparisons with MATLAB (Section 10.2) and the Intel MKL sparse library (Section 10.3). Unless otherwise stated, all results in this section were obtained on an NVIDIA DGX Station (Intel Xeon CPU E5-2698 v4 @ 2.20GHz, a single socket system with 20 hardware cores, 40 threads in OpenMP). The `gcc` compiler (11.2.0 -O3) was used with the GNU OpenMP library, except for the comparison with the Intel MKL sparse library, where the `icx` compiler and the Intel OpenMP library were used. Hyperthreading was left enabled, which is the default.

10.1 Matrix multiply with a regular mesh

This section reports the performance obtained by `GrB_mxm` to compute $\mathbf{C} = \mathbf{A}^2$, $\mathbf{C} = \mathbf{A}^4$, and $\mathbf{C} = \mathbf{A}^8$, in the $(+, \times)$ semiring, using repeated squaring via three matrix-matrix multiplications. The $\mathbf{A}$ matrix arises from the finite-element discretization of an N-by-N 2D mesh [36], or `A=gallery('wathen',N,N)` in MATLAB. Table 2 lists the time for a range of threads and the parallel speedup on three Wathen matrices. The work for each of the nine problems is shown ($\times 10^9$), where each multiply-add pair is counted as two flops. As the problem size increases, `GrB_mxm` obtains nearly linear speedup up to the number of hardware cores, with a peak performance of 15.2 Gflop/sec on the largest problem.

computation	MATLAB (seconds)	SuiteSparse (seconds)	speedup
y=S*x	22.80	2.40	9.49
y=x*S	16.16	1.16	13.92
C=S*F	30.61	9.71	3.15
C=F*S	26.40	1.52	17.32
C=S'	224.73	22.69	9.91
C=S+B	15.56	1.51	10.31
C=S(p,q)	95.62	15.95	6.00

Table 3. SuiteSparse vs MATLAB R2021a with the GAP-Twitter matrix

10.2 GraphBLAS in MATLAB

SuiteSparse:GraphBLAS has its own interface to Octave and MATLAB, where it provides access to all of the features of GraphBLAS, such as `GrB.mxm` for the masked matrix multiply. It also provides overloads for all MATLAB operators and 167 built-in MATLAB functions, so that `C=A*B`, `C(I,J)=A`, etc., can all be written in MATLAB syntax and yet use GraphBLAS matrices (as `@GrB` objects in MATLAB). Matrix multiply is up to 30x faster in GraphBLAS on a 20-core Intel Xeon, as compared with `C=A*B` in MATLAB R2020b and earlier (this author wrote both methods). The results in this section use 20 threads in the `@GrB` interface in MATLAB, since MATLAB turns off hyperthreading by default. SuiteSparse:GraphBLAS nearly always benefits from hyperthreading, however.

An older parallel version of SuiteSparse:GraphBLAS (v3.3.3) is built into MATLAB R2021a for its sparse matrix multiply, but the current version (v7.1.2) is faster still.

Other methods in MATLAB do not yet use GraphBLAS, and thus SuiteSparse:GraphBLAS can be significantly faster than built-in methods in MATLAB. For example, sparse matrix assignment, `C(I,J)=A` can be over 1,000x faster: for a matrix $\mathbf{C}$ of size 25 million by 25 million, with 36 million entries, and a matrix $\mathbf{A}$ of size 5000 by 5000 with 50,000 entries, `C(I,J)=A` takes 270.5 seconds with MATLAB sparse matrices, but only 0.26 seconds using `@GrB` sparse matrices.

Logical indexing for sparse matrices is much faster with `@GrB` matrices as compared to MATLAB, which is `C(M)=A(M)` in MATLAB syntax, and the same syntax with using `@GrB` sparse matrix objects. This is similar to the GraphBLAS notation $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}$, a masked assignment. In GraphBLAS, the statement `C(M)=A(M)` takes 0.4 seconds on a Dell laptop (a 4-core Intel Core i7), if $\mathbf{C}$ is 4.2 million by 4.2 million with 42 million entries, and $\mathbf{M}$ and $\mathbf{A}(\mathbf{M})$ have 4.2 million entries, using all 4 cores. In MATLAB R2021a, the same statement, with the same syntax, takes 28 hours using MATLAB sparse matrices, using a single core. GraphBLAS provides a speedup of nearly 248,000x for this computation, most of which is due to the algorithm, not to parallelism. Given enough time to run MATLAB, problems where SuiteSparse:GraphBLAS would be 1,000,000x faster than MATLAB can be easily constructed.

Table 3 shows the run time of MATLAB 2021a and SuiteSparse:GraphBLAS for a large matrix: the GAP-Twitter matrix, of dimension $n = 61.6$ million with about 1.5 billion entries [12]. In the table, `S` is the Twitter matrix, `x` is a dense row or column vector, `F` is a full matrix of size n-by-4 or 4-by-n, `B` is a random sparse matrix with 150 million entries, and `p` and `q` are random permutation vectors. For these experiments, SuiteSparse:GraphBLAS uses the identical MATLAB syntax, just with GraphBLAS matrices instead of MATLAB built-in matrices.

For the Wathen results in Section 10.1, v7.1.2 is about twice as fast as the v3.3.3 that appears in MATLAB R2021a. The largest problem ($N = 1500$) takes 107.1 seconds to compute `A^8` with the built-in v3.3.3 in the third matrix-matrix multiply, but only 48.4 seconds when using `@GrB` objects with v7.1.2, using the same syntax (both results with 20 threads).

computation	format	MKL method	MKL time (sec)		SuiteSparse	speedup	
			1st	2nd	time (sec)	1st	2nd
y+=S*x	S by row	mkl_sparse_d_mv	2.54	1.27	1.21	2.10	1.05
y+=S*x	S by col	mkl_sparse_d_mv	7.22	7.22	1.98	3.65	3.65
C+=S*F	S by row, F by row	mkl_sparse_d_mm	2.95	1.90	1.98	1.49	**.96**
C+=S*F	S by row, F by col	mkl_sparse_d_mm	6.12	4.99	1.48	4.13	3.37
C+=S*F	S by col, F by row	mkl_sparse_d_mm	28.82	28.82	13.78	2.09	2.09
C+=S*F	S by col, F by col	mkl_sparse_d_mm	78.82	5.17	9.38	8.40	**.55**
C=S+B	S by row	mkl_sparse_d_add	30.77	30.77	1.44	21.37	21.37
C=S'+B	S by row	mkl_sparse_d_add	102.09	27.30	16.29	6.26	1.67
C=S'	S by row	mkl_sparse_convert_csr	77.27	77.27	14.80	5.22	5.22

Table 4. SuiteSparse vs MKL 2022 with the GAP-Twitter matrix

10.3 SuiteSparse:GraphBLAS versus Intel MKL sparse

SuiteSparse:GraphBLAS obtains very competitive performance as compared to the Intel MKL sparse library. This benchmark uses the same Twitter matrix as above, and compares SuiteSparse:GraphBLAS compiled with `icx` and using the latest MKL library (2022.0.0) and the recommended Intel OpenMP libraries, using its 64-bit interface (to match the 64-bit indices used by GraphBLAS). The MKL also has an interface with 32-bit indices, which is faster but which cannot be used for the largest problems.

Many of the MKL functions have an optimization phase that is used in a first call with a given matrix. In this phase, the input matrices are analyzed, and a transpose is made (and kept) if needed. Thus, the first run of MKL can take more time than the second. SuiteSparse:GraphBLAS saves no such state. In Table 4, the two MKL runs are called 1st and 2nd. SuiteSparse:GraphBLAS and the MKL are both parallel, and both use their default number of threads, and the resulting timing when using all cores is shown in the table. SuiteSparse is typically faster, except for two results in bold.

For the computation `C=S'+B`, the MKL library transposes `S` on the first trial, as `T=S'`, and saves the transpose inside the MKL sparse matrix `S`. It then computes `C=T+B` with the saved transpose on the 2nd trial. SuiteSparse:GraphBLAS performs the transpose each time; in other words, computing both the transpose and add with GraphBLAS is faster than computing just the add with MKL.

The computation `y+=S*x` in SuiteSparse:GraphBLAS when `S` is held by column relies on the finite-state machine method in a fine-grain Gustavson method. SuiteSparse:GraphBLAS is 3.65x faster than the best Intel MKL sparse method in this case. For a closely related problem, `y=S*x` in SuiteSparse:GraphBLAS is 9.49x faster than MATLAB (see Table 3).

MKL is significantly faster in only one case, `C+=S*F` where both matrices are held by column. It transposes `S` on the first trial, to create a copy held by row, and then uses it in the subsequent trial (5.17 seconds). Note this time is about the same as the run time when `S` starts out held by row, when GraphBLAS takes just 1.48 seconds. SuiteSparse:GraphBLAS cannot perform this optimization, because the C API Specification requires input matrices to be read-only so that they can be shared between user threads. LAGraph can hold a copy of the transpose in its graph data structure, so this optimization can be done at the LAGraph level [1, 27, 35].

10.4 LAGraph and the GAP benchmark

Performance results of individual GraphBLAS operations are described in the previous sections, but it is essential to consider how all of these methods can be combined into a graph algorithm, and to illustrate the resulting parallel performance of the graph algorithm. The GAP Benchmark specifies six different graph algorithms, to be benchmarked on five different matrices [4]. The reference

graph	nodes	entries in **A**	kind
Kron	134,217,726	4,223,264,644	undirected
Urand	134,217,728	4,294,966,740	undirected
Twitter	61,578,415	1,468,364,884	directed
Web	50,636,151	1,930,292,948	directed
Road	23,947,347	57,708,624	directed

Table 5. Benchmark matrices

Algorithm : package	graph, with run times in seconds				
	Kron	Urand	Twitter	Web	Road
BC : GAP	31.52	46.36	10.82	3.01	1.50
BC : SS	24.06	33.11	9.18	6.82	33.29
BFS : GAP	.31	.58	.22	.34	.25
BFS : SS	.55	1.27	.40	.75	3.15
PR : GAP	19.81	25.29	15.16	5.13	1.01
PR : SS	21.72	27.53	17.07	9.18	1.72
CC : GAP	.53	1.66	.23	.22	.05
CC : SS	2.93	3.76	1.43	2.09	.77
SSSP : GAP	4.91	7.23	2.02	.81	.21
SSSP :SS	15.97	25.36	7.50	8.18	30.80
TC : GAP	374.08	21.82	79.58	22.18	.03
TC : SS	865.43	33.84	225.06	32.36	.26

Table 6. Performance results

codes are highly optimized stand-alone C++ codes, parallelized using OpenMP. These provide an important reference point with which to compare with SuiteSparse:GraphBLAS.

SuiteSparse:GraphBLAS does not have any of these algorithms, however. Instead, the LAGraph effort has been initiated to create graph algorithms implemented using the GraphBLAS kernels [1, 27, 35]. LAGraph is not yet stable as a v1.0 release, but it includes robust implementations of all of the GAP Benchmarks. The following results were obtained with a draft version of LAGraph (v0.9.29), along with SuiteSparse:GraphBLAS v7.1.2.

The six algorithms in the benchmark are: (1) breadth-first search, to construct the BFS tree [39], (2) betweenness-centrality, (3) pagerank, (4) connected components [42], (5) single-source shortest path [32], and (6) triangle counting [3, 37]. The five graphs are shown in Table 5. Table 6 reports the parallel performance results of the GAP Benchmark and LAGraph+SuiteSparse:GraphBLAS.

GraphBLAS is faster than the GAP benchmark for the largest problems for betweenness-centrality. The reason for this is the simplicity of writing the push/pull optimization in GraphBLAS; all that is required is to multiply by the transpose of the matrix, much like the breadth-first search described in Section 2. Push/pull optimization is not implemented in Beamer's BC benchmark algorithm, where it would be more difficult to write without the support of a library such as GraphBLAS. The BC algorithm in LAGraph is extremely simple as compared to the GAP Benchmark method, yet it is significantly faster for the largest problems.

GraphBLAS is somewhat slower that the GAP Benchmark on the breadth-first search. Both exploit the same parallel strategy and both rely on push/pull optimization. GraphBLAS is about 1.5x to 2x slower, and the difference can be attributed to two factors. Beamer's BFS fuses together the matrix-vector multiply and the assignment, which is not done in LAGraph+SuiteSparse:GraphBLAS. The GraphBLAS C API allows for this, but it is not yet implemented. In addition, all of the integers in the GAP Benchmark code are 32-bit, while GraphBLAS uses 64-bit integers for its matrix indices.

The maximum number entries in the benchmark matrices is almost exactly 2^{32}. Any larger, and the GAP Benchmark codes would need to switch to 64-bit indices instead. In this sense, it is the GAP Benchmark code that would need to be revised for larger problems, which certainly arise in practice. Some of the problems that GraphBLAS has solved in practice are much larger than these problems [26], or have extreme dimensions, and so 64-bit indices are essential. SuiteSparse:GraphBLAS supports matrices with dimensions up to 2^{60}, which can be done even on tiny in-network routers [21]; this application could make use of matrices as large as 2^{128} if they were supported.

Triangle counting is about 3x faster in the GAP Benchmark. GraphBLAS computes the result as an entire matrix, using the masked dot-product method described in Section 4.4, and then reduces this result to a single scalar. The GAP Benchmark can fuse these two operations, and never constructs the matrix, which is faster. However, in real applications, the triangle count itself is not as important as operations on those triangles, such as the K-truss algorithm, Burkhardt's recent *triangle centrality* metric [9], or applying a survey operator to the triangles [33]. In this case, the matrix may be needed anyway.

The GraphBLAS C API allows the implementation to fuse together calls to GraphBLAS, via its non-blocking mode, but currently SuiteSparse:GraphBLAS only uses the non-blocking mode to exploit lazy modifications (zombies, pending tuples, and jumbled matrices). Kernel fusion for SuiteSparse:GraphBLAS will be considered in the future (Section 12).

The performance for PageRank is almost identical between the two methods. For the other two graph algorithms, the GAP Benchmark methods are significantly faster in most cases, but even here, most of the results are not worse than 5x for the largest graphs (except for the Road graph, for which GraphBLAS is very slow). This must be balanced against the ease of writing these graph algorithms in the first place, however.

SuiteSparse:GraphBLAS is slow for the Road graph for many problems, mainly because it is a relatively small graph with a high diameter, and thus there is very little scope for parallelism in each call to GraphBLAS. Kernel fusion becomes very important for this case, but SuiteSparse:GraphBLAS does not yet implement this. The overhead of each individual call to GraphBLAS starts to become a significant factor in the computation.

Overall, these results show that the parallel implementation of SuiteSparse:GraphBLAS provides competitive performance as compared with highly-tuned, specialized graph kernels, which only experts can write. By contrast, the ease of use of the GraphBLAS C API, and its interfaces in Python, Julia, Octave, and MATLAB, allow it to be easily employed by any data scientist to write their own graph algorithms.

11 RELATED WORK

SuiteSparse:GraphBLAS is the only complete implementation of the v2.0 *GraphBLAS C API Specification* [5, 8]. Other packages implement portions of GraphBLAS, or are based on similar ideas. These include: the GPI package by IBM [16] (a full implementation of the v1.3 C API Specification), CombBLAS (https://people.eecs.berkeley.edu/~aydin/CombBLAS/html/ [7]), Graphulo (http://graphulo.mit.edu/ [20]), D4M (http://d4m.mit.edu/ [24]), GraphPad [2, 34] Gunrock (https://github.com/gunrock/gunrock-grb, [38]), the GraphBLAS Template Library (Carnegie Mellon Univ., Indiana Univ., and PNNL) (https://github.com/cmu-sei/gbtl), ALP/GraphBLAS (C++) [41], GraphBLAST [40], and an implementation by Lucata (https://lucata.com).

In work done independently of the work reported here, Buluç et al., [28] have developed four novel parallel algorithms for the masked matrix-matrix multiply, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ and $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$. These methods include a hash-based method similar to the masked coarse Hash method described in Section 4.3, and a masked sparse accumulator (MSA) method, similar to the coarse Gustavson method in Section 4.3. Buluç et al. do not consider the following features of the masked matrix

multiplication that appear in SuiteSparse:GraphBLAS: (1) fine-grain methods using atomics and/or finite-state machines, (2) combinations of sparse/hypersparse/bitmap/full matrix formats, (3) the exploitation of the iso properties of the four matrices **C**, **M**, **A**, and **B**, and (4) the ability to use any mix of four kinds of tasks (coarse/fine Hash/Gustavson). All of these algorithmic features are unique to the parallel masked matrix multiply methods presented here.

12 FUTURE WORK

There are many aspects of the GraphBLAS C API that SuiteSparse:GraphBLAS does not yet exploit. Some of the goals for the future are listed below.

- LAGraph v1.0 and more algorithms for LAGraph.
- GPU acceleration: generating CUDA kernels at run time, processed through a JIT. This work is ongoing, in collaboration with Joe Eaton and Corey Nolet of NVIDIA.
- GPU acceleration: with SYCL, in collaboration with Intel and NVIDIA.
- more AVX-accelerated kernels.
- Julia integration: all core sparse matrix operations will be able to use GraphBLAS (conventional $(+, \times)$ semirings) as a recommended sparse linear algebra package. This will also allow easy Julia-friendly access to any GrB methods (data types, semirings, masks, ...).
- further Python integration.
- RedisGraph: enabling further speedups, more features.
- JIT for the CPU, which will enable user-defined types and operators to become just as fast as built-in types and operators.
- aggressive non-blocking, kernel fusion (requires the JIT).
- move `GxB` methods to `GrB` in the v2.x or v3.0 C API Specification.
- `x=A\b` over a field.
- more built-in types (16-bit floating point, complex integers, ...).
- variable-sized types (`GrB_Matrix` of `GrB_matrices`, GMP integers, ...).
- faster kernels: `GrB_mxm` for sampled dense-dense matrix product [31]. This can currently be written in two lines of GraphBLAS, but SuiteSparse:GraphBLAS could be further accelerated to exploit the unique properties of this problem.
- matrices with shallow components (these exploited internally, but not given to the user).

13 SUMMARY

The C API of GraphBLAS provides a novel and expressive way to create graph algorithms, and its parallel implementation in SuiteSparse:GraphBLAS can give competitive performance as compared with highly-tuned parallel C++ codes, which are much more difficult to write.

In addition to being available as a Collected Algorithm of the ACM, SuiteSparse:GraphBLAS is on github at https://github.com/DrTimothyAldenDavis/GraphBLAS. It appears as the built-in sparse matrix multiply in MATLAB R2021a, and is the core graph computational engine of RedisGraph, developed by Roi Lipman, Redis (https://redislabs.com/modules/redis-graph/).

SuiteSparse:GraphBLAS includes its own built-in interfaces to Octave and MATLAB; the former with assistance from John Eaton. There are two Python interfaces, `pygraphblas` and `grblas`, the first by Michael Pelletier of Graphegon, and the second by Erik Welch and Jim Kitchen of Anaconda; the two teams collaborate on a common core interface to SuiteSparse:GraphBLAS, and each has their own Python wrapper (see https://graphegon.github.io/pygraphblas/pygraphblas/index.html and https://anaconda.org/conda-forge/grblas. A Julia interface is under development, by Will Kimmerer, Abhinav Mehndiratta, and Viral Shah, at https://github.com/JuliaSparse/SuiteSparseGraphBLAS.jl.

A CUDA accelerated version of SuiteSparse:GraphBLAS is under development, in collaboration with Joe Eaton and Corey Nolet at NVIDIA.

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, Benjamin Brock, and Carl Yang), and many other collaborators on LAGraph and the Python/Julia/Octave/MATLAB interfaces (Ariful Azad, Mohsen Aznaveh, David Bader, Aydın Buluç, Jinhao Chen, Corey Nolet, Joe Eaton, John Eaton, Lucas Jarman, Will Kimmerer, Jim Kitchens, Abhinav Mehndiratta, Tze Meng Low, Scott McMillan, Tim Mattson, Michel Pelletier, Pat Quillen, Viral Shah, Gábor Szárnyas, Erik Welch, and Yongzhe Zhang). This work is supported by NVIDIA, Intel, MathWorks, MIT Lincoln Laboratory, Redis, Julia Computing, and the National Science Foundation (NSF CNS-1514406).

REFERENCES

[1] LAGraph. https://github.com/GraphBLAS/LAGraph, Mar 2022.
[2] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322, May 2016.
[3] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.
[4] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.
[5] B. Brock, A. Buluç, T. Mattson, S. McMillan, and J. E. Moreira. The GraphBLAS C API specification, v2.0. Technical report, 2021. http://graphblas.org/.
[6] A. Buluç and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, April 2008.
[7] A. Buluç and J. R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.
[8] A. Buluç, T. Mattson, S. McMillan, J. E. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, pages 643–652. IEEE Computer Society, 2017.
[9] P. Burkhardt. Triangle centrality. arXiv/2105.00110, 2021.
[10] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Phila., PA, 2006.
[11] T. A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019.
[12] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.
[13] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.
[14] Timothy A. Davis and Pen-Chung Yew. A nondeterministic parallel algorithm for general unsymmetric sparse lu factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(3):383–402, 1990.
[15] M. Deveci, C. Trott, and S. Rajamanickam. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Computing*, 78:33–46, 2018.
[16] K. Ekanadham, W. Horn, J. Jann, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu. Graph programming interface: Rationale and specification. Technical Report RC25508 (WAT1411-052), IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, November 2014.
[17] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992.
[18] J. S Golan. *Semirings and affine equations over them: theory and applications*. Springer, 2003.
[19] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[20] D. Hutchison, J. Kepner, V. Gadepally, and B. Howe. 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)*, pages 1–9, Sept 2016.

[21] M. Jones, J. Kepner, D. Andersen, A. Buluc, C. Byun, K Claffy, T. Davis, W. Arcand, J. Bernays, D. Bestor, W. Bergeron, V. Gadepally, M. Houle, M. Hubbell, H. Jananthan, A. Klein, C. Meiners, L. Milechin, J. Mullen, S. Pisharody, A. Prout, A. Reuther, A. Rosa, S. Samsi, J. Sreekanth, D. Stetson, C. Yee, and P. Michaleas. Graphblas on the edge: High performance streaming of network traffic, 2022.

[22] J. Kepner. GraphBLAS mathematics. http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf, 2017.

[23] 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. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, Sept 2016.

[24] 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. Dynamic distributed dimensional data model (D4M) database and computation system. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5349–5352, March 2012.

[25] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra.* SIAM, Phila., PA, 2011.

[26] J. Kepner, M. Jones, D. Andersen, A. Buluç, C. Byun, K. Claffy, T. Davis, W. Arcand, J. Bernays, D. Bestor, W. Bergeron, V. Gadepally, M. Houle, M. Hubbell, A. Klein, C. Meiners, L. Milechin, J. Mullen, S. Pisharody, A. Prout, A. Reuther, A. Rosa, S. Samsi, D. Stetson, A. Tse, C. Yee, and P. Michaleas. Spatial temporal analysis of 40,000,000,000,000 internet darkspace packets. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2021.

[27] T. Mattson, T. A. Davis, M. Kumar, A. Buluç, S. McMillan, J. Moreira, and C. Yang. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 276–284, 2019.

[28] Srđan Milaković, Oguz Selvitopi, Israt Nisa, Zoran Budimlić, and Aydin Buluç. Parallel algorithms for masked sparse matrix-matrix products. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[29] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Computing*, 90:102545, 2019.

[31] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan. Sampled dense matrix multiplication for high-performance machine learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 32–41, 2018.

[32] U. Sridhar, M. Blanco, R. Mayuranath, D. G. Spampinato, T. Low, and S. McMillan. Delta-stepping SSSP: From vertices and edges to GraphBLAS implementations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 241–250, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[33] T. Steil, T. Reza, K. Iwabuchi, B. W. Priest, G. Sanders, and R. Pearce. TriPoll: Computing surveys of triangles in massive-scale temporal graphs with metadata. 2021.

[34] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. GraphMat: high performance graph analytics made productive. In *Proceedings of VLDB 2015*, volume 8, pages 1214 – 1225, 2015.

[35] G. Szárnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch. LAGraph: linear algebra, network analysis libraries, and the study of graph algorithms. IPDPS GrAPL'21, 2021.

[36] A. J. Wathen. Realistic eigenvalue bounds for the Galerkin mass matrix. *IMA J. Numer. Anal.*, 7:449–457, 1987.

[37] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[38] C. Yang, A. Buluç, and J. D. Owens. Design principles for sparse matrix multiplication on the GPU. In M. Aldinucci, L. Padovani, and M. Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 672–687. Springer International Publishing, 2018.

[39] C. Yang, A. Buluç, and J. D. Owens. Implementing push-pull efficiently in GraphBLAS. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[40] C. Yang, A. Buluç, and J. D. Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Trans. Math. Softw.*, 48(1), Mar 2022.

[41] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. Preprint, 2020.

[42] Y. Zhang, A. Azad, and A. Buluç. Parallel algorithms for finding connected components using linear algebra. *Journal of Parallel and Distributed Computing*, 144:14–27, 2020.

Received July 2021; revised Aug 2022