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, along with the performance obtained when it is used to solve the graph problems in the GAP Benchmark.

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. 2021. Algorithm 10xx: SuiteSparse:GraphBLAS: parallel graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2021), 35 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The GraphBLAS standard [7, 8] 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* [7, 8], based on the mathematical foundations discussed in [18, 19].

Here, we describe a particular OpenMP-based parallel implementation of the GraphBLAS standard, SuiteSparse:GraphBLAS (or SS:GrB for short). In additional to its availability as a Collected Algorithm of the ACM, as the core computational engine of RedisGraph, by Redis Labs, and it is used for the built-in sparse matrix multiply (`C=A*B`) 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 SS:GrB: matrix multiply (Section 4), element-wise add and multiply (Sections 5 and 6), submatrix extraction and assignment (Section 7 and 8), transpose (Section 9), and the mask/accumulator phase (Section 10). Section 11 highlights the parallel performance of SS:GrB for a suite of benchmark graph algorithms. Section 12 summarizes related work, followed by 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.

0098-3500/2021/1-ART1 $15.00
https://doi.org/0000001.0000001

2 AN OVERVIEW OF GRAPHBLAS AND ITS IMPLEMENTATION IN SUITESPARSE

2.1 Semirings and their use in graph algorithms

Graph Algorithms in the Language on Linear Algebra [21] 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, [11].

In GraphBLAS, a directed or undirected graph is represented via its sparse adjacency matrix, and operating on this matrix with different *semirings* provides a mechanism for creating graph algorithms, particularly when combined with the *mask*. This approach is illustrated with an example of a push/pull direction-optimizing breadth-first search [4]. Let the sparse matrix $\mathbf{A}$ represent a graph where a_{ij} is the edge (i, j), and let $\mathbf{q}$ be a sparse vector whose sparsity pattern represents the list of nodes in the current frontier of a BFS. Let $\mathbf{p}$ denote 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 pattern of $\mathbf{p}$.

A masked assignment, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}$, acts like a bulk if-statement, where the assignment $c_{ij} = a_{ij}$ is performed only where the entry m_{ij} is present in the boolean mask matrix $\mathbf{M}$ and equal to 1. In the BFS, this mask is complemented and used in a structural manner, denoted $\mathbf{C}\{\neg\mathbf{M}\} = \mathbf{A}$, 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 $\mathbf{v}\{\neg\mathbf{p}\} = \mathbf{A}^{\mathsf{T}}\mathbf{q}$ computes a new vector $\mathbf{v}$, where the entry v_j appears in the sparsity pattern of $\mathbf{v}$ only if there is an edge (i, j) and q_i is an entry present in the sparsity pattern 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 pattern of the parent vector $\mathbf{p}$, and thus $\mathbf{p}$ is used as a complemented structural mask. The sparsity pattern 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 be redefined where the `MIN` operator is used instead of the conventional `PLUS`, and where the function `SECONDI` is used instead of the conventional `TIMES`. Rather than the `PLUS_TIMES` semiring of conventional linear algebra, which would compute

$$c_j = \sum_k a_{ik} \times b_k j$$

for $\mathbf{C} = \mathbf{AB}$, the unusual `MIN_SECONDI` semiring can be used instead, which computes

$$c_j = \min_k \text{secondi}(a_{ik}, b_{kj}).$$

A *semiring* is simply a *monoid* (an associative and commutative operator with an identity value) and a binary multiplicative operator. In the conventional semiring used in numerical linear algebra, the monoid is `PLUS` and the multiplicative operator is `TIMES`. For the breadth-first search, the monoid is `MIN` and the binary operator is `SECONDI`, instead.

The operator `SECONDI` is unique to SuiteSparse:GraphBLAS. It is a *positional* operator, whose result depends not on the values of its operands, but on their position in the matrix, so that $\text{secondi}(a_{ik}, b_{kj})$, as computed in the expression above, is equal to k, which is the row index of the second input scalar operand.

Using this semiring to compute $\mathbf{v}\{\neg\mathbf{p}\} = \mathbf{A}^{\mathsf{T}}\mathbf{q}$, the value of v_j is the smallest row index of any node i so that the edge (i, j) exists in the graph and where q_i appears in the input vector, and where p_j does not yet appear in the sparsity pattern of $\mathbf{p}$. In other words, it is the node id of the new parent of the newly discovered node j.

The second step in the BFS is to assign this newly-found parent to the parent vector, with the masked assignment $\mathbf{p}\{\mathbf{v}\} = \mathbf{v}$, and then by assigning $\mathbf{v} = \mathbf{q}$ the algorithm traverses to the next frontier.

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 a copy of $\mathbf{A}$ held in a column-oriented form; call it $\mathbf{T}$ (which can also be considered as the explicit transpose of $\mathbf{A}$ but held in row-oriented format). Then it is simple to see that the expression $\mathbf{v}\{\neg\mathbf{p}\} = \mathbf{Tq}$ computes the same thing as $\mathbf{v}\{\neg\mathbf{p}\} = \mathbf{A}^{\mathsf{T}}\mathbf{q}$. However, the matrix 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.

This parent tree is overly restrictive, however. In practice, any parent will do, but the `MIN_SECONDI` picks the least numbered parent. A better monoid than `MIN` is the based on the `ANY`, unique to SuiteSparse:GraphBLAS. With this operator, $f(x, y) = x$ or y, either one, at the discretion of the operator itself. The `ANY` monoid has special properties, as described in Section 4.1, and is faster to compute than the `MIN` monoid.

The GraphBLAS-based push/pull BFS [31] is shown below, combining pseudo-MATLAB with GraphBLAS notation, and revised by using the `ANY_SECONDI` semiring. The input consists of the source node s, and the adjacency matrix $\mathbf{A}$ and its explicit transpose $\mathbf{T}$, both held in row-oriented form. The details for deciding push versus pull are not shown, but they are the same heuristics as used by [4] and are simple to compute in GraphBLAS.

```
function p = bfs (A, T, s)
    n = size (A,1)          % the graph has n nodes
    p = sparse (n,1)        % empty sparse n-by-1 vector, the parent vector
    q = sparse (n,1)        % empty sparse n-by-1 vector, the current frontier
    p (s) = s               % s is the root, its own parent
    q (s) = s               % s is the only node in the current frontier
    while (q is not empty)
        % decide push vs pull (details omitted) ...
        % The matrix-vector multiply below uses the ANY_SECONDI semiring, with
        % the "replace" semantic so that q is overwritten with the next frontier.
        if (push)
            q{!p,replace} = A'*q
        else % pull
            q{!p,replace} = T*q
        p{q} = q
```

Using its C API, the above algorithm in C code is not as succinct, but its logic is the same. With GraphBLAS interfaces in Python, Julia, Octave, or MATLAB, the algorithm is almost as simple as the pseudo-code shown above (only the push versus pull heuristic would need to be added, for a few extra lines of code). 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.

2.2 GraphBLAS methods and operations

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. The goal is both ease of expression of these graph algorithms, plus high performance.

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

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* (in which case, only the pattern of $\mathbf{M}$ is used and any entry in the pattern is treated as true), and optionally transposing the input matrices. The BFS written with GraphBLAS relies on a complemented structural mask, for example.

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 $\mathbf{C}\{\mathbf{M}\}$ 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 pattern 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.

A *monoid* is an associate and commutative binary operator $f(x, y)$ that has an identity value, *id*, so that $f(x, \mathrm{id}) = f(\mathrm{id}, x) = x$. 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. A semiring consist of a monoid and a multiplicative binary operator. The semiring of conventional linear algebra is `PLUS_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` multiplicative operator. Other semirings are useful in different graph algorithms. For example, in a shortest-path problem, the `MAX_PLUS` semiring computes $c_{ij} = \max_k(a_{ik} + b_{kj})$.

The table below summarizes the key GraphBLAS methods and operations, including `GrB*` methods in the GraphBLAS C API Specification [7, 8], and `GxB*` extensions unique to SS:GrB. 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.

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)
`GxB_select`	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathrm{select}(\mathbf{A})$	element-wise select
`GrB_reduce`	$\mathbf{w}\langle\mathbf{m}\rangle \odot= \mathrm{reduce}(\mathbf{A})$	reduce matrix to vector
`GrB_reduce`	$\sigma \odot= \mathrm{reduce}(\mathbf{A})$	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`	$\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathrm{kron}(\mathbf{A}, \mathbf{B})$	Kronecker product
`GrB_apply`	$\mathbf{C}\langle\mathbf{M}\rangle \odot= f(\mathbf{A})$	apply a unary operator

3 SUITESPARSE:GRAPHBLAS PARALLEL IMPLEMENTATION

All of the objects in the GraphBLAS API are opaque to the application that uses GraphBLAS, which provides a significant opportunity for optimizing the data structures and algorithms when implementing a parallel GraphBLAS library.

3.1 Matrix and vector data structure

In particular, the internal data structure for the `GrB_Matrix` and `GrB_Vector` has changed significantly since the release of Algorithm 1000 [12], the sequential version of SuiteSparse:GraphBLAS (v2.3.3), to exploit new parallel algorithms and to reduce memory usage.

The prior sequential version of SS:GrB 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 (v5.1.5) provides 16 different formats. There are four sparsity formats, listed below. Each can be held by row or by column, and each can exploit the *iso-value* property, if applicable (see Section 3.2). SS:GrB selects between these 16 formats automatically, but a method is also provided for the user application to provide hints as to the data structure(s) to use.

- **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 $O(n + e)$. 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 [15].
- **Hypersparse** [5]: 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 $k << n$. 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. A `GrB_Vector` is never held in hypersparse format.
- **Bitmap**: In this format, the numerical values `A.x` are held in a dense array of size $m \times n$. The sparsity pattern is held in a boolean array `A.b` of size $m \times n$. 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 pattern 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 their own data types.

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 used two kinds of pending work; ACM Algorithm 10xx (v5.1.5) introduces another.

Algorithm 1000 introduced the idea of *zombies* and *pending tuples*, which still appear in v5.1.5. A zombie is an entry in a sparse or hypersparse matrix that is tagged for deletion, but not yet deleted, by "negating" its row or column index in the `A.i` array (offset by 2 since indices start at zero). Since their index can still be retrieved, zombies allow binary searches to still 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.

New to v5.1.5 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. Some algorithms produce jumbled matrices naturally, and some are oblivious to whether or not 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 a sort.

Bitmap and full matrices do not acquire zombies or pending tuples, nor are they ever jumbled. These 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.

By default, SS:GrB selects automatically between the four basic formats (sparse, hypersparse, bitmap, or full). The user application can limit this choice, by allowing a matrix to be in only a subset of these four formats. All matrices are held by row, by default, except for the `@GrB` Octave/MATLAB interface, where they are held be column since that is the default for sparse matrices in those packages.

3.2 Iso-valued matrices and vectors

In many applications, graphs are unweighted, with no particular value associated with each edge. The GraphBLAS C API does not allow for structure-only matrices to be specified, however. All entries in the sparsity pattern of a matrix or vector must have a value associated with them. 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 (v5.1.5) introduces the idea of *iso-valued* matrices and vectors, where all entries in the sparsity pattern have the same value (typically 1, but any value can be assigned).

For an *iso-valued* matrix or vector (or just "*iso*" for short, from here on in this paper), the `A.x` array in all formats shrinks down in size to hold just a single value, the *iso-value* of the matrix or vector. For sparse, hypersparse, and bitmap formats, the sparsity pattern must still be held, so these require $O(n + e)$, $O(e)$, and $O(mn)$ memory, respectively. Full matrices are special. Since its sparsity pattern 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 $e << n$ 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 SS:GrB, 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 can easily create such matrices using only $O(e)$ space.

SS:GrB exploits *iso-valued* matrices and vectors (or just *iso* for short) by detecting many cases where they naturally occur in the GraphBLAS C API. Graph algorithms that rely on GraphBLAS can exploit these cases to reduce time and memory, sometimes asymptotically, so they are described in detail below.

- Positional operators: when used as multiplicative operator in a semiring, such as the example given in Section 2.1, these operators do not result in an iso matrix $\mathbf{C}$, however it is computed. A positional operator is one that does not depend on the value of its inputs, but on their position in the matrix that contains them. For example, the `SECONDI` operator used in a

semiring, where $t = a_{ik}b_{kj}$ is computed, returns the row index of the second operand, or the value k. All of the cases below exclude positional operators.

- `GrB_eWiseAdd`: when computing $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$, the pattern of $\mathbf{C}$ is the set union of $\mathbf{A}$ and $\mathbf{B}$, using a binary operator (it need not be scalar addition, but can be any binary operator). Entries in the sparsity pattern of $\mathbf{A}$ but not $\mathbf{B}$, and in $\mathbf{B}$ but not $\mathbf{A}$, are copied directly into $\mathbf{C}$. Therefore, unless $\mathbf{A}$ and $\mathbf{B}$ are full, $\mathbf{C}$ can be iso only if $\mathbf{A}$ and $\mathbf{B}$ are iso. If their iso values are a and b, respectively, then $a = b = f(a, b)$ must hold for $\mathbf{C}$ to be iso, and its iso value is equal to that of $\mathbf{A}$ and $\mathbf{B}$.
- `GrB_eWiseMult`: $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ is the Hadamard product, and is like the MATLAB notation `C=A.*B`, where the pattern of $\mathbf{C}$ is the intersection of $\mathbf{A}$ and $\mathbf{B}$ (the binary operator need not be scalar multiplication). Four operators provide special cases: `PAIR` ($f(x, y) = 1$), `FIRST` ($f(x, y) = x$), `SECOND` ($f(x, y) = y$), and `ANY` ($f(x, y) = x$ or y, chosen arbitrarily). The `PAIR` operator always produces an iso result. If $\mathbf{B}$ is iso and the operator is `SECOND` or `ANY`, then $\mathbf{C}$ is iso with the same iso value as $\mathbf{B}$. Likewise, if $\mathbf{A}$ is iso and the operator is `FIRST` or `ANY`, then $\mathbf{C}$ is iso with the same iso value as $\mathbf{A}$. If both $\mathbf{A}$ and $\mathbf{B}$ are iso, then $\mathbf{C}$ is iso regardless of then operator, with an iso value of $c = f(a, b)$.
- `GrB_mxm`: matrix multiplication $\mathbf{C} = \mathbf{AB}$ has a complex set of rules that govern the iso property of $\mathbf{C}$, depending on the iso properties of $\mathbf{A}$ and $\mathbf{B}$, and the semiring used. If given a set of identical values $\{t, t, t, \ldots t\}$ to reduce to a single value, some monoids will always produce the result as t; namely, the `MIN`, `MAX`, `LAND` (logical and), `LOR` (logical or), `BAND` (bitwise and), `BOR` (bitwise or), and `ANY` monoids. These entries are produced by the multiplicative operator $t = a_{ik}b_{kj}$, and if $\mathbf{A}$ and $\mathbf{B}$ are both iso, then t is the same regardless of where it is computed. If $\mathbf{B}$ is iso and the multiplicative operator is `SECOND` or `ANY`, then $\mathbf{C}$ is also iso with an iso value $c = b$, if one of these monoids is used. If $\mathbf{A}$ is iso and the multiplicative operator is `FIRST` or `ANY`, then $\mathbf{C}$ is also iso with an iso value $c = a$, if one of these monoids is used. If both $\mathbf{A}$ and $\mathbf{B}$ are iso, and one of these monoids is used, then $\mathbf{C}$ is iso with its value coming from the multiplicative operator, $c = t = f(a, b)$. For the `PAIR` operator, $t = 1$ and all of these monoids produce an iso matrix $\mathbf{C}$, regardless of the iso properties of $\mathbf{A}$ and $\mathbf{B}$. In addition, so do the `EQ`, `LXNOR`, and `TIMES` monoids when used with `PAIR`. Full matrices add an additional special case. If $\mathbf{A}$ and $\mathbf{B}$ are iso full matrices, then $\mathbf{C}$ is iso full for all semirings whose multiplicative operator is not a positional operator. This allows two iso full n-by-n matrices to be multiplied in at most $O(\log n)$ time and $O(1)$ space for any monoid, even user-defined ones.
- `GrB_Matrix_build_FP64 (C,I,J,X,nvals,dup)`: this function constructs a matrix $\mathbf{C}$ using three arrays (`I`, `J`, and `X`, each of length `nvals`) that hold the row indices, column indices and values of the tuples to be included in the matrix. The `dup` operator is used to combine duplicate entries. After constructing the matrix, SS:GrB examines it to see if all entries are identical. If so, it reduces the `A.x` array from its current size (`nvals` if there are no duplicates) down to size one. SS:GrB also adds two functions `GxB_Matrix_build_Scalar` `GxB_Vector_build_Scalar` that replace the `X` array with a single scalar, and which do not have the `dup` operator. These methods construct iso-valued matrices by design, thus saving time and memory in their construction.
- `GrB_reduce`: reducing an iso matrix with e entries to a scalar takes only $O(\log e)$ time, for any monoid. Reducing a matrix to a vector using a monoid (`REDUCE`) is identical to a matrix-vector multiply, $\mathbf{v} = \mathbf{Ax}$ where $\mathbf{x}$ is an iso full vector and where the semiring is `REDUCE_FIRST`. Thus, the rules for `GrB_mxm` listed above are used to exploit the iso property when reducing a matrix to a vector.
- `GrB_apply`: applying a unary operator, or binary operator with one input being a scalar as $\mathbf{C} = f(\mathbf{A})$, $\mathbf{C} = f(\mathbf{A}, \sigma)$, or $\mathbf{C} = f(\sigma, \mathbf{A})$ where σ is a scalar, results in an iso matrix $\mathbf{C}$ if

the operator is PAIR, ONE, ANY, FIRST (in the case of $\mathbf{C} = f(\sigma, \mathbf{A})$), or SECOND (in the case of $\mathbf{C} = f(\mathbf{A}, \sigma)$), regardless of the iso property of $\mathbf{A}$. In addition, if $\mathbf{A}$ is iso, then so is $\mathbf{C}$ unless the operator is positional.

- GxB_select: applying a select operator produces an iso result $\mathbf{C} = \text{select}(\mathbf{A})$ if $\mathbf{A}$ is iso (for example, the syntax C=tril(A) in MATLAB notation, which returns the lower triangular part of $\mathbf{A}$). If the select operator is EQ_ZERO or EQ_THUNK for any matrix type, or NONZERO for boolean matrices, then $\mathbf{C}$ is iso regardless of whether or not $\mathbf{A}$ is iso.
- GrB_assign: the rules for this method are complicated, but many cases produce an iso result.
  - For scalar assignment with no accumulator operator, $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) = \sigma$, if $\mathbf{C}$ starts out as empty, or has an iso value equal to σ, then it remains iso. In the special case of $\mathbf{C}\{\mathbf{C}\} = \sigma$, using a structural mask, $\mathbf{C}$ always becomes iso with an iso value of σ, requiring only $O(1)$ time. This assignment can be used to replace a non-iso matrix $\mathbf{C}$ with an iso-valued matrix with the given value σ, much like C=s*spones(C) in MATLAB notation, except that the latter takes $O(e)$ time while $\mathbf{C}\{\mathbf{C}\} = \sigma$ takes $O(1)$ time. The scalar assignment $\mathbf{C}(:,:) = \sigma$, with no mask, always constructs an iso full matrix $\mathbf{C}$ in $O(1)$ time and memory. To construct an iso matrix $\mathbf{C}$ from a non-iso matrix $\mathbf{A}$, as in C=spones(A) in MATLAB notation, the assignment $\mathbf{C}\{\mathbf{A}\} = \mathbf{1}$ can be used, with $\mathbf{A}$ used as a structural mask and where $\mathbf{C}$ is initially empty.
  - For scalar assignment with an accumulator, $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) \odot= \sigma$, if $\mathbf{C}$ starts out as empty, or has an iso value c such that $c = c + s$, then it remains iso.
  - For matrix assignment with no accumulator operator, $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) = \mathbf{A}$ results in an iso matrix $\mathbf{C}$ if $\mathbf{A}$ is iso and $\mathbf{C}$ starts out empty or iso with the same iso value as $\mathbf{A}$.
  - For matrix assignment with an accumulator operator, $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{I},\mathbf{J}) \odot= \mathbf{A}$ results in an iso matrix $\mathbf{C}$ if $\mathbf{A}$ is iso and $\mathbf{C}$ starts out empty or is iso with an iso value such that $c = c + a$.
- GrB_extract: extracting a submatrix, $\mathbf{C} = \mathbf{A}(\mathbf{I},\mathbf{J})$ creates an iso result $\mathbf{C}$ if $\mathbf{A}$ is iso.
- GrB_transpose: transposing an iso matrix, $\mathbf{C} = \mathbf{A}^{\mathsf{T}}$ creates an iso result $\mathbf{C}$ if $\mathbf{A}$ is iso.

When the output matrix $\mathbf{C}$ is not iso, it can often be the case that the input matrices are iso. In this case, 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-valued input matrices.

3.3 Parallel matrix operations

Details of the parallel matrix multiply (Section 4), element-wise add (set union, Section 5), element-wise multiply (set intersection, Section 6), submatrix extraction (Section 7), submatrix assignment operations (Section 8), transpose (Section 9), and the mask/accumulator phase (Section 10) are given in the following sections.

By default, SS:GrB holds its matrices in row-oriented for all formats: sparse, hypersparse, bitmap, and full. However, most sparse matrix algorithms in the literature typically assume column-oriented storage, so that is assumed for the methods described below for consistency. Internally, SS:GrB 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.

The parallel operations GrB_apply, GrB_reduce (to scalar), GrB_kronecker, GxB_select, and the parallel methods such as GrB_*_build are not described here since their implementation is

fairly straight-forward. Many of them use the parallel traversal of a single matrix as described in the next section.

The amount of parallelism to exploit and the size of the tasks are determined automatically. The maximum number of threads SS:GrB can use is given from `omp_get_max_threads` (by default), but this can be reduced via the descriptor or by a global setting. Suppose a parallel phase will do an amount of work equal to w, and let $p_{\max}$ be the maximum number of threads it can use. SS:GrB has a chunk-size parameter κ, equal to 2^{16} by default (this can also be modified). Then the number of threads actually used for this parallel region is $t = \max(1, \min(\lfloor w/\kappa \rfloor), p_{\max})$. Once t is found, between t and $256t$ tasks are created (typically $8t$), 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 $t =$ `nthreads` threads and `ntasks` tasks, the following is a very common paradigm in SS:GrB:

```
// (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.

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 Sections 4, and the bucket-style transpose in Section 9.

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 $\mathbf{A}$ 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 $\mathbf{A}$. Instead, the entries in $\mathbf{A}$ are first divided uniformly into a set of independent tasks. These entries may span partial vectors of $\mathbf{A}$, 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` ($k = n$ if $\mathbf{A}$ is in compressed-sparse format, or $k << n$ if $\mathbf{A}$ is hypersparse). If p were large this step could easily be done in parallel. The current implementation of SuiteSparse:GraphBLAS assumes p is at most a few hundred, 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 $\mathbf{A}$ is held by column), or conversely, the column index if $\mathbf{A}$ is held by row. In that case, the entries of a sparse or hypersparse matrix $\mathbf{A}$ can be easily divided into tasks without the need for a binary search, where the entire traversal of $\mathbf{A}$ takes $O(e/p)$ time when using p threads.

4 PARALLEL MATRIX MULTIPLY

The parallel matrix multiply (`GrB_mxm`), matrix-vector multiply (`GrB_mxv`), or vector-matrix multiply (`GrB_vxm`) methods are key kernels in GraphBLAS and are typically the most computationally intensive within a given graph algorithm. They 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. Internally, 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.

Reducing a matrix to a vector using `GrB_reduce`, by applying a monoid to its rows or columns, is implemented as a matrix-vector multiply (or vector-matrix), where the vector is full and iso-valued, and thus many the algorithms in this section are used for that case.

Depending on how the methods are counted, SS:GrB has 48 unique methods for parallel matrix-matrix, vector-matrix, and matrix-vector multiplication, depending upon the sparsity format of the input matrices and the presence of the mask matrix, for each of its many different semirings. Most of these 40 methods are described below, where only the most novel ones are described in full in the interest of space.

Two fundamentally different methods are used: *saxpy*-based and *dot product*-based methods. 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.

Via its `descriptor` input, `GrB_mxm` can be requested to operate on either $\mathbf{A}$ or its transpose, and on either $\mathbf{B}$ 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 first step taken by `GrB_mxm` is to reduce this large set cases. In particular, a matrix held by row but not transposed is identical to a matrix held by column and transposed via the descriptor. The mask matrix cannot be transposed via the descriptor, but it may be held in a different format than required ($\mathbf{C}$ is stored by row but $\mathbf{M}$ is by column). If necessary, the mask matrix is transposed first.

This phase reduces any problem down to just four cases, where all matrices are now assumed to be held by column, and the matrix multiply algorithm is then chosen. All of these decisions can be controlled by the user application, but the default 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.

- $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{AB}$: a saxpy-based (Sections 4.2 and 4.3), unless one or both matrices are diagonal, in which case a row or column scaling method is used (Section 4.7).
- $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{AB}^{\mathsf{T}}$: this would require an outer-product-based method, which SS:GrB does not have, and thus $\mathbf{B}$ is explicitly transposed first, and the saxpy-based method is used.
- $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{A}^{\mathsf{T}}\mathbf{B}$: a dot-product-based (Section 4.4, 4.5, and 4.6).
- $\mathbf{C}\langle\sharp\mathbf{M}\rangle = (\mathbf{AB})^{\mathsf{T}}$: a saxpy-based (Sections 4.2 and 4.3), and after $\mathbf{C}$ is computed, it is explicitly transposed.

The next step is to determine the sparsity format of $\mathbf{C}$ and its iso property (see Section 3.2 for the latter), and to select the method. For the saxpy-based methods, if $\mathbf{M}$ is sparse or hypersparse, then $\mathbf{C}$ is hypersparse if $\mathbf{B}$ is hypersparse, or sparse otherwise. Otherwise, the sparsity format of $\mathbf{C}$ is determined as follows:

- If $\mathbf{B}$ is hypersparse, then so is $\mathbf{C}$.
- If $\mathbf{B}$ is sparse, and $\mathbf{A}$ is sparse or hypersparse, then $\mathbf{C}$ is sparse.
- If $\mathbf{B}$ is sparse, and $\mathbf{A}$ is bitmap or full, then $\mathbf{C}$ is sparse if $\mathbf{B}$ has many empty columns (50%) or bitmap otherwise.
- If $\mathbf{B}$ is bitmap or full, and $\mathbf{A}$ is sparse or hypersparse, then $\mathbf{C}$ is sparse if $\mathbf{A}$ is very sparse (fewer than $m/20$ entries if $\mathbf{A}$ is m-by-n) or bitmap otherwise.
- If $\mathbf{B}$ and $\mathbf{A}$ are bitmap or full, $\mathbf{C}$ is bitmap.

If the above rules determine that $\mathbf{C}$ should be computed in sparse or hypersparse form, the sparse saxpy method is used (see Section 4.2). If $\mathbf{C}$ is to be in bitmap form, the bitmap saxpy method is used (see Section 4.3).

For the dot-product-based methods, the following rules are used:

- If $\mathbf{C}$ is full and the accumulator operator is present and matches the monoid of the semiring, and no mask is present, then the method in Section 4.6 is used.
- If $\mathbf{M}$ is present, sparse, and not complemented, the method in Section 4.4 is used, and $\mathbf{C}$ is sparse.
- Otherwise, the general dot-product method in Section 4.5 is used, and $\mathbf{C}$ is computed in bitmap form.

The dot-product methods can require more work than is strictly required. 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.

4.1 Special case monoids

The algorithms described below are used for nearly all monoids and semirings, but some monoids and semirings need special consideration.

SS:GrB includes an unusual binary operator called `ANY`. The computation $f(x, y)$ returns either x, or y, solely at the discretion of the operator itself. The selection is not randomized. Instead, SS:GrB 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, $f(x, f(y, z)) = f(f(x, y), z)$, since in both cases, SS:GrB 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 ($\{t, t, \ldots t, t\}$), the result is always t, and thus the `ANY` monoid produces a predictable iso-valued result, and (2) there are cases where x and y differ, but the application will be satisfied with either result of $f(x, y)$. Computing the breadth-first search tree is one such case: if a node found at one level has k possible

parent nodes at the prior level, then 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. SS:GrB 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. An example is given in Section 4.2.3. The GAP Benchmark BFS by Scott Beamer 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 idea in the language of linear algebra over semirings.

Consider a PLUS_PAIR semiring (of any type). The PAIR operator is $f(x, y) = 1$ and is the multiplicative operator, and the monoid is PLUS. When combined with a dense vector x, the computation $y = Ax$ 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 similarly. 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. With the LOR monoid (logical OR), for example, as soon as a true value is seen, the computation can halt. These monoids are referred to in SS:GrB as *terminal*, and they include MIN, MAX, LOR, LAND, BOR (bitwise OR), BAND (bitwise AND), TIMES (for integers only), and ANY. 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, so it is not exploited.

4.2 Sparse saxpy

The sparse *saxpy*-based methods are the most complex set of matrix multiplication methods in SS:GrB. 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 terms of column-oriented form, but the same algorithms work the same if the matrices are held by row. The methods divide into six phases, all of which are parallel.

These methods do not handle the accumulator operator (as in $\mathbf{C} \odot= \mathbf{AB}$); that is handled in a subsequent step if it appears. 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, in which tasks share workspace and operate on the same set of results, with concurrent operations resolved via atomic operations.

If the rules determine that $\mathbf{C}$ is sparse or hypersparse, then 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 $\mathbf{C}$, depending on the sparsity structure of the matrices $\mathbf{A}$ and $\mathbf{B}$.

(1) *coarse Gustavson*: an extension of Gustavson's algorithm [16], but 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 $\mathbf{C}$, using a shared workspace.

(3) *coarse Hash*: an extension of the hashed-based method [13, 24, 25], extended to exploit the mask.
(4) *fine Hash*: In this method, a group of threads cooperates, sharing a single hash table via atomics, to compute a single column of **C**. The method of [24, 25] does exploit the mask and does not use atomics; it is extended in this work. The method of [16] did not consider parallelism, but its coarse-grain parallelism is straightforward. The novel contribution here is the fine task parallelism and atomics, particularly when exploiting the mask.

In the case when the mask **M** is bitmap or full, either all Hash tasks (fine/coarse) or all Gustavson tasks are used. For the Hash case, the 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 the case that it 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).

Each of these four kinds of tasks are subdivided into at least 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**.

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.

4.2.1 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, and the hash table size is set to `m`, to serve as the gather/scatter workspace for Gustavson's method.

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.

- fine Gustavson task (shared): `int8_t Hf[m] ; ctype Hx[m] ;`
- fine Hash task (shared): `int64_t Hf[s] ; ctype Hx[s] ;`
- coarse Gustavson task: `int64_t Hf[m] ; ctype Hx[m] ;`
- coarse Hash task: `int64_t Hf[s] ; ctype Hx[s] ; int64_t Hi[s] ;`

Note that the `Hi` array is needed only for the coarse Hash task. Additional workspace is allocated to construct the list of tasks, but this is freed before **C** is constructed.

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.

4.2.2 Sparse saxpy, phase 1: symbolic analysis. The symbolic analysis phase is performed in phase 1. 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, if the mask is present, except that a fine Hash task never scatters a bitmap or full mask into its hash table (any mask entry can be found in $O(1)$ via a direct lookup into **M** itself). If a fine Gustavson task operating on `C(:,j)` is using the mask, then `Hf[i] = 1` is performed for every entry m_{ij} in the mask, whether it is complemented or not. 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 60 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)`. If no mask is present, the task uses the conventional Gustavson method (essentially the same as the method below, except with no mask, and the `mark` is incremented one each iteration). If the mask is complemented, c_{ij} can be computed if m_{ij} is zero, and the following algorithm is used:

```
Coarse Gustavson task, phase 1, for C<!M>=A*B
mark = 0 ;
for j = j1 to j2
    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(i,j): set Hf[i] = mark
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            if (Hf[i] < mark)  // if true, M(i,j) is zero and C(i,j) is not yet seen
                Hf[i] = mark+1 // C(i,j) has been seen
                Cp[j]++
```

If the mask is present and not complemented, a similar algorithm is used, except the innermost `if` test becomes `if (Hf[i] == mark)`. In addition, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ when the mask is not complemented relies on two different methods for traversing the innermost loop:

(1) If `A(:,k)` is sparse enough relative to `M(:,j)`, then the method above is used, taking $O(a)$ time if a is the number of entries in `A(:,k)`.
(2) For all four kinds of tasks (fine/coarse Gustavson/Hash), when the mask is present and not complemented, a push/pull optimization is employed. 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 the algorithm above, but the change is simple. The inner loop that traverse `A(:,k)` is replaced by a loop that traverses all entries m_{ij} in `M(:,j)`, and then

`A(i,k)` is found with a binary search. If μ and a are the number of entries in `M(:,j)` and `A(:,k)`, this takes $O(m \log a)$ time, which is faster than the linear-time method if $m << a$.

The three types of coarse Hash tasks for phase 1 are analogous, handling the cases for $\mathbf{C} = \mathbf{AB}$, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, and $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$. Each single thread is responsible for computing the number of nonzeros in each vector of `C(:,j1:j2)`, except that they do not use a Gustavson workspace of size `m`. Instead, they use a hash table of size `s`, which is much less than `m`. Each hash entry is represented by two 64-bit integers: `Hf[hash]`, which contains the same `mark` as the coarse Gustavson task above, and `Hi[hash]`, which contains the row index of the entry c_{ij} present in that position.

```
Coarse Hash task, phase 1,f or C=A*B with no mask
mark = 0 ;
for j = j1 to j2
    mark = mark + 1 // clear Hf, now all Hf[0:m-1] < mark
    Cp[j] = 0 ;     // # of entries in C(:,j)
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            marked = false
            for (hash = (257*i) % m ; ; hash = (hash+1) % m)
                marked = (Hf[hash] == mark)
                found = marked && (Hi[hash] == i)
                if (found || !marked) break
            if (!marked)
                Hf[hash] = mark    // add C(i,j) to this hash entry
                Hi[hash] = i
                Cp[j]++
```

The coarse Hash methods for phase 1 that exploit the mask are similar. The same above algorithm is used if $\mathbf{M}$ is present but bitmap or full, except that a constant-time lookup for the mask entry is performed in the innermost loop. In that case, the mask need not be hashed, as done in the methods below. Otherwise, when the mask is present, sparse, and complemented, the following states are used. Let `h = Hi[hash]` and `f = Hf[hash]` for a given hash entry.

- `f < mark`: unoccupied, m_{ij} is zero, and c_{ij} is not yet seen.
- `h == i, f == mark` : m_{ij} is one, c_{ij} is ignored.
- `h == i, f == mark+1` : m_{ij} is zero, and c_{ij} has been seen.

To compute the number of entries in `C(:,j)`, the mask `M(:,j)` is first scattered into the hash table using the following method. This method is used for both phase 1 and phase 5 of the coarse Hash method, for both cases of the mask (complemented or not complemented). This method uses the same `mark` technique to flag entries currently in the hash table, which can be then cleared for the next iteration in $O(1)$ time, simply by incrementing the `mark`.

```
Hash M(:,j) for a coarse hash task, for phase 1 or 5:
for each entry M(i,j) in M(:,j)
    if (M(i,j) == 0) continue   // skip if M(i,j) is zero
    for (hash = (257*i) % m ; ; hash = (hash+1) % m)
        if (Hf[hash] < mark)
            Hf[hash] = mark
            Hi[hash] = i
            break
```

The phase-1 coarse Hash methods when the mask is present are given below. The two methods are similar, except that the coarse Hash task when $\mathbf{M}$ is present but not complemented uses the same two methods as for the coarse Gustavson task, traversing `M(:,j)` and using a binary search in `A(:,j)`, or traversing `A(:,j)` directly. The second method below when the mask is not

complemented assumes that `A(:,j)` is traversed, to simplify the presentation. The nonzero pattern of `C(:,j)` must be a subset of the pattern of `M(:,j)`, so if a row index `i` is not found in the hash table, it cannot be included in `C(:,j)`.

```
Coarse Hash task, phase 1, for C<!M>=A*B
mark = 0 ;
for j = j1 to j2
    mark = mark + 2 // clear Hf, now all Hf[0:s-1] < mark
    Cp[j] = 0 ;     // # of entries in C(:,j)
    hash all entries in M(:,j)
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            for (hash = (257*i) % m ; ; hash = (hash+1) % m)
                if (Hf[hash] < mark)    // M(i,j) zero, so C(i,j) appears
                    Hf[hash] = mark+1   // C(i,j) is a newly found entry
                    Hi[hash] = i
                    Cp[j]++
                    break
                if (Hi[hash] == i)
                    break               // C(i,j) already seen
Coarse Hash task, phase 1, for C<M>=A*B
mark = 0 ;
for j = j1 to j2
    mark = mark + 2 // clear Hf, now all Hf[0:s-1] < mark
    Cp[j] = 0 ;     // # of entries in C(:,j)
    hash all entries in M(:,j)
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            for (hash = (257*i) % m ; ; hash = (hash+1) % m)
                if (Hf[hash] < mark) break  // M(i,j) zero, ignore C(i,j)
                if (Hi[hash] == i)          // if true, M(i,j) is 1
                    if (Hf[hash] == mark)   // if true, C(i,j) is new
                        Hf[hash] = mark+1   // C(i,j) is a newly found entry
                        Cp[j]++
                    break
```

4.2.3 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 pattern 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, uncomplemented sparse mask, and complemented sparse mask). All of the tasks use atomics, and all tasks in a team computing `C(:,j)` share a single workspace, either a size-`m` Gustavson workspace for the fine Gustavson tasks, or a size-`s` hash table for the fine Hash tasks. In the interest of space, not all tasks can be fully described here. These task can be computed over any semiring, but to simplify the presentation, the `PLUS_TIMES` semiring is assumed.

The fine Gustavson task, with no mask, is the simplest. A team of threads computes the symbolic pattern and numerical values for a single vector, `C(:,j)=A*B(:,j)`. The `int8` work array `Hf` of size `m` is initially all zero. Each entry `Hf[i]` acts as the state of a single finite-state machine that controls the computation of c_{ij}. The following state transitions are used, modified by atomic operations.

- `Hf[i]` is initially 0.
- $0 \rightarrow 3$: to lock, if c_{ij} is seen for the first time
- $2 \rightarrow 3$: to lock, if c_{ij} has been seen already
- $3 \rightarrow 3$: spin lock
- $3 \rightarrow 2$: to unlock; now c_{ij} has been seen

A single task computes `C(:,j)+=A*B(k1:k2,j)` for a unique range `k1:k2`, shown below. If `f==2` is found in the first atomic read, the entry is unlocked but initialized, so `C(i,j)+=t` is computed with a single atomic update. The first time c_{ij} is seen, `Hf[i]` is zero, so the spin-lock is entered. Contention is expected to be low, since the spin-lock is entered only if c_{ij} has not yet been seen.

```
Fine Gustavson task, phase 2, for C=A*B with no mask
for each entry B(k,j) in B(k1:k2,j)
    for each entry A(i,k) in A(:,k)
        t = A(i,k) * B(k,j)
        f = Hf[i]               // atomic read
        if (f == 2)             // if true, update C(i,j)
            Hx[i] += t          // atomic update of C(i,j)
            continue
        do { f = Hf[i] ; Hf[i] = 3 } while (f == 3) // atomic swap
        if (f == 0)
            Hx[i] = t           // C(i,j) is a new entry; atomic write
        else
            Hx[i] += t          // atomic update of C(i,j)
        Hf[i] = 2               // flag/unlock the entry
```

For the `ANY` monoid, the task relies a benign race condition for higher performance:

```
Fine Gustavson task, phase 2, no mask, with the ANY monoid
for each entry B(k,j) in B(k1:k2,j)
    for each entry A(i,k) in A(:,k)
        t = A(i,k) * B(k,j)
        f = Hf[i]               // atomic read
        if (f == 2) continue    // if true, C(i,j) stays the same
        Hx[i] = t               // atomic write, benign race condition
        Hf[i] = 2               // flag/unlock the entry
```

The fine Gustavson task, phase 2, when the mask is present, sparse, and not complemented is given below. The following state transitions are used:

- `Hf[i]` is 0 if m_{ij} not present or $m_{ij} = 0$.
- $0 \rightarrow 1$: has already been done in phase 0 if $m_{ij} = 1$.
- $1 \rightarrow 3$: to lock, if c_{ij} is seen for the first time
- $2 \rightarrow 3$: to lock, if c_{ij} has been seen already
- $3 \rightarrow 3$: spin lock
- $3 \rightarrow 2$: to unlock; now c_{ij} has been seen

The method below is simplified; it assumes that `A(:,k)` is traversed, but just as in the coarse Gustavson task, `M(:,j)` can be traversed instead, with a binary search of `A(:,j)`.

```
Fine Gustavson task, for C<M>=A*B where M is sparse
for each entry B(k,j) in B(k1:k2,j)
    for each entry A(i,k) in A(:,k)
        t = A(i,k)*B(k,j)
        f = Hf[i]               // atomic read
        if (f == 2)             // if true, update C(i,j)
            Hx[i] += t          // atomic update of C(i,j)
            continue
        if (f == 0) continue    // M(i,j)=0; ignore C(i,j)
        do { f = Hf[i] ; Hf[i] = 3 } while (f == 3) // atomic swap
        if (f == 1)
            Hx[i] = t           // C(i,j) is a new entry; atomic write
        else
            Hx[i] += t          // atomic update of C(i,j)
        Hf[i] = 2               // flag/unlock the entry
```

The case where the mask is sparse, present, and complemented is nearly identical. The only thing that changes is the role of states 0 and 1 are reversed for each finite-state machine. In addition, since c_{ij} need not be a subset of `M(:,j)`, the entire vector `A(:,j)` must be traversed. Traversing `M(:,j)` and using a binary search of `A(:,j)` is not an option.

The fine Hash tasks are analogous to the fine Gustavson tasks, but the atomic access to the hash table is much more complex. Each team of tasks shares an `int64` array, `Hf`, and a numerical array, `Hx`, both of size `s` where `s` is a power of two.

When no mask is present (or when the mask is bitmap or full), the following algorithm is used. It starts with the size-`s` array `Hf` set to zero. Each entry `Hf[hash]` splits into the pair `(h,f)` where `h` is 60 bits in size, and `f` is 2 bits. If `(h,f) = (0,0)` then the hash entry is unlocked and unoccupied. If `(h,f) = (i+1,2)` then the hash entry is unlocked and occupied by c_{ij}. If `(h,f) = (...,3)` then the hash entry is locked. Three state transitions are used, managed via atomic operations:

- $0 \rightarrow 3$: to lock, if c_{ij} is seen for the first time
- $2 \rightarrow 3$: to lock, if c_{ij} has been seen already
- $3 \rightarrow 3$: spin lock
- $3 \rightarrow 2$: to unlock; now c_{ij} has been seen

Just as in the fine Gustavson task, if the entry c_{ij} is already initialized and appears in the hash table, the entry is found right away, and the spin-lock is not entered.

```
Fine Hash task, for C=A*B with no mask
for each entry B(k,j) in B(k1:k2,j)
    for each entry A(i,k) in A(:,k)
        t = A(i,k)*B(k,j)
        for (hash = (257*i) % m ; ; hash = (hash+1) % m)
            hf = Hf[hash]           // atomic read
            if (hf == (i+1,2))      // if true, C(i,j) already present
                Hx[i] += t          // atomic update of C(i,j)
                break
            h = hf >> 2
            if (h == 0 || h == i+1)
                // h == 0: unoccupied; h == i+1: occupied by C(i,j)
                do { hf = Hf[hash] ; Hf[hash] |= 3 } while ((hf & 3) == 3) // atomic swap
                if (hf == (0,0))
                    Hx[hash] = t            // C(i,j) new entry; atomic write
                    Hf[hash] = (i+1,2)      // unlock the entry; atomic write
                    break
                else if (hf == (i+2,2))
                    Hx[hash] += t           // atomic update of C(i,j)
                    Hf[hash] = (i+1,2)      // unlock the entry; atomic write
                    break
                Hf[hash] = hf               // unlock with prior value
```

The fine Hash task for phase 2 when the mask is present, sparse, and not complemented will see the following states for each hash entry, `(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. 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+2,2)` then it is unlocked and occupied by $m_{ij} = 1$ and `Hx[hash]` holds c_{ij}. Four state transitions are used, via atomics:

- $0 \rightarrow 0$: to ignore, if $m_{ij} = 0$
- $1 \rightarrow 3$: to lock, if c_{ij} is seen for the first time
- $2 \rightarrow 3$: to lock, if c_{ij} seen already
- $3 \rightarrow 2$: to unlock; now c_{ij} has been seen

```
Fine Hash task, for C<M>=A*B
for each entry B(k,j) in B(k1:k2,j)
    for each entry A(i,k) in A(:,k)
        t = A(i,k)*B(k,j)
        for (hash = (257*i) % m ; ; hash = (hash+1) % m)
            hf = Hf[hash]           // atomic read
            if (hf == (i+1,2))      // if true, C(i,j) already present
                Hx[i] += t          // atomic update of C(i,j)
                break
            if (hf == (0,0)) break          // M(i,j)=0, ignore C(i,j)
            h = hf >> 2
            if (h == i+1)                   // if true, occuppied by C(i,j)
                do { hf = Hf[hash] ; Hf[hash] |= 3 } while ((hf & 3) == 3)  // atomic swap
                if (hf == (i+1,1))
                    Hx[hash] = t            // C(i,j) new entry; atomic write
                else if (hf == (i+2,2))
                    Hx[hash] += t           // atomic update of C(i,j)
                Hf[hash] = (i+1,2)          // unlock the entry; atomic write
                break
```

The fine Hash task for phase 2 when the mask is present and complemented is similar. If `(h,f) = (0,0)`, then the hash entry is unlocked and unoccupied. If `(h,f) = (i+1,1)` then it is unlocked and occupied by $m_{ij} = 1$. In this case, c_{ij} cannot appear in the result and is ignored. If `(h,f) = (i+2,2)` then it is unlocked and occupied by c_{ij}. Four state transitions are used, managed via atomic operations:

- $1 \rightarrow 1$: to ignore, if $m_{ij} = 1$
- $0 \rightarrow 3$: to lock, if c_{ij} seen for the first time
- $2 \rightarrow 3$: to lock, if c_{ij} seen already
- $3 \rightarrow 2$: to unlock; now c_{ij} has been seen

```
Fine Hash task, for C<!M>=A*B
for each entry B(k,j) in B(k1:k2,j)
    for each entry A(i,k) in A(:,k)
        t = A(i,k)*B(k,j)
        for (hash = (257*i) % m ; ; hash = (hash+1) % m)
            hf = Hf[hash]           // atomic read
            if (hf == (i+1,2))      // if true, C(i,j) already present
                Hx[i] += t          // atomic update of C(i,j)
                break
            if (hf == (i+1,1)) break        // M(i,j)=1, ignore C(i,j)
            h = hf >> 2
            if (h == 0 || h == i+1)
                // h == 0: unoccupied; h == i+1: occupied by C(i,j)
                do { hf = Hf[hash] ; Hf[hash] |= 3 } while ((hf & 3) == 3) // atomic swap
                if (hf == (0,0))
                    Hx[hash] = t            // C(i,j) new entry; atomic write
                    Hf[hash] = (i+1,2)      // unlock the entry; atomic write
                    break
                elseif (hf == (i+1,2))
                    Hx[hash] += t           // atomic update of C(i,j)
                    Hf[hash] = (i+1,2)      // unlock the entry; atomic write
                    break
                Hf[hash] = hf               // unlock with prior value; atomic write
```

4.2.4 Sparse saxpy, phase 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.

4.2.5 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 this phase, they gather their results from their workspaces into the final location of `C(:,j)` in the output matrix **C**. Coarse tasks perform their symbolic and numerical work in phase 5.

When no mask is present, the coarse Gustavson method is the same as [10, 16], except the work has been split into parallel tasks. The gather of the final result for all three phase-5 coarse Gustavson tasks is done in one of two ways: if a vector `C(:,j)` has many entries (more than $m/16$) then the entire size-m workspace is scanned and `C(:,j)` is constructed with sorted row indices. Otherwise the pattern of `C(:,j)` is constructed in the order it is computed, and either left jumbled (as pending work) or sorted explicitly. Then the values are gathered from the workspace.

The coarse Gustavson task when the mask is present and complemented is shown below, except that only one method for gathering the final `C(:,j)` is shown. The case when the mask is present but not complemented is very similar. The primary difference is the test `(Hf[i] < mark)`, which becomes `(Hf[i] == mark)`, which denotes that $m_{ij} = 1$ and c_{ij} is not yet seen.

```
Coarse Gustavson task, phase 5, for C<!M>=A*B
mark = 0 ;
for j = j1 to j2
    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(i,j): set Hf[i] = mark
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            if (Hf[i] < mark)  // if true, M(i,j) is zero and C(i,j) is not yet seen
                Hf[i] = mark+1
                Hx[i] = A(i,k)*B(kj)        // initialize C(i,j)
                Ci[pC++] = i
            else if (Hf[i] == mark + 1)
                Hx[i] += A(i,k)*B(kj)       // update C(i,j)
    gather all of C(:,j) from the workspace
```

For the three kinds of coarse Hash tasks for phase 5, the hash tables are managed in the same way as in phase 1. With no mask, phase 5 of the coarse Hash task is listed below.

```
Coarse Hash task, phase 5, for C=A*B with no mask
mark = 0 ;
for j = j1 to j2
    mark = mark + 1 // clear Hf, now all Hf[0:m-1] < mark
    pC = Cp[j]      // start position of C(:,j)
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            for (hash = (257*i) % m ; ; hash = (hash+1) % m)
                if (Hf[hash] == mark)
                    if (Hi[hash] == i)
                        Hx[hash] += A(i,k)*B(k,j)   // update C(i,j)
                        break
                else
                    Hf[hash] = mark
                    Hi[hash] = i
                    Hx[hash] = A(i,k)*B(k,j)        // initialize C(i,j)
                    Ci[pC++] = i
                    break
    gather all of C(:,j) from the workspace
```

The method above is essentially identical to the Hash method of [24, 25], except for the use of the mark, which allows the hash table to be cleared in constant time for each iteration, just by incrementing the mark.

When the mask is complemented, the following coarse Hash algorithm is used:

```
Coarse Hash task, phase 5, for C<!M>=A*B
mark = 0 ;
for j = j1 to j2
    mark = mark + 2 // clear Hf, now all Hf[0:s-1] < mark
    pC = Cp[j]     // start position of C(:,j)
    hash all entries in M(:,j)
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            for (hash = (257*i) % m ; ; hash = (hash+1) % m)
                if (Hf[hash] < mark)
                    Hf[hash] = mark + 1
                    Hi[hash] = i
                    Hx[hash] = A(i,k)*B(k,j)        // initialize C(i,j)
                    Ci[pC++] = i
                if (Hf[hash] == i)
                    if (Hf[hash] == mark+1)
                        Hx[hash] += A(i,j)*B(k,j)   // update C(i,j)
                    break
    gather all of C(:,j) from the workspace
```

Finally, the coarse Hash task for phase 5 when the mask is not complemented is below:

```
Coarse Hash task, phase 5, for C<M>=A*B
mark = 0 ;
for j = j1 to j2
    mark = mark + 2 // clear Hf, now all Hf[0:s-1] < mark
    pC = Cp[j]     // start position of C(:,j)
    hash all entries in M(:,j)
    for each entry B(k,j) in B(:,j)
        for each entry A(i,k) in A(:,k)
            for (hash = (257*i) % m ; ; hash = (hash+1) % m)
                if (Hf[hash] < mark) break  // M(i,j) zero, ignore C(i,j)
                if (Hi[hash] == i)          // if true, i is found
                    if (Hf[hash] == mark)   // if true, C(i,j) is new
                        Hf[hash] = mark+1
                        Hx[hash] = A(i,k)*B(k,j)    // initialize C(i,j)
                        Ci[pC++] = i
                    else
                        Hx[hash] += A(i,k)*B(k,j)   // update C(i,j)
                    break
    gather all of C(:,j) from the workspace
```

4.3 Bitmap saxpy

Three different algorithms are used for $\mathbf{C} = \mathbf{AB}$, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$: (1) when $\mathbf{A}$ is sparse/hypersparse and $\mathbf{B}$ is bitmap/full, (2) when $\mathbf{A}$ is bitmap/full and $\mathbf{B}$ is sparse/hypersparse, and (3) when both $\mathbf{A}$ and $\mathbf{B}$ are bitmap/full. All three methods can exploit the mask, either complemented or otherwise, in any sparsity format (sparse, hypersparse, bitmap, or full). In all three methods, the resulting matrix $\mathbf{C}$ is computed in bitmap format. All three methods could exploit the accumulator operator, if it is present and if it matches the semiring monoid, but this feature is not yet implemented.

All methods can exploit any kind of mask matrix **M**. If **M** is sparse or hypersparse, it is first scattered into the bitmap of **C**. Each entry `Cb(i,j)` in the **C** bitmap is an entire byte, and can be in one of four states. When the methods finish, the mask is cleared from the bitmap of **C**, so that it contains just 1s and 0s.

- `Cb(i,j) = 0`: the entry c_{ij} is not present and m_{ij} is zero.
- `Cb(i,j) = 1`: the entry c_{ij} is present and m_{ij} is zero.
- `Cb(i,j) = 2`: the entry c_{ij} is not present and m_{ij} is one.
- `Cb(i,j) = 3`: the entry c_{ij} is present and m_{ij} is one.

4.3.1 Bitmap saxpy: A sparse/hypersparse, B bitmap/full. This method subdivides into three entirely different algorithms.

(1) The first method is used if the number of threads is smaller than the number of columns of **B**. This method relies on coarse-grain tasks. Each task operates on up to 4 columns of **B** at a time, and loads and transposes `B(:,j1:j2)` for this panel into a thread-local workspace `G`. Next, `H=A*G` is computed, where both `H` and `G` are held by row (assuming **A** and **B** are in column-oriented format). After the panel `H` is computed, `C(:,j1:j2)+=H` is performed. This approach allows the innermost loop to access memory in stride one, as each sparse entry `A(i,j)` can be used for all entries in a single row of `H` and `G`.

(2) The second method relies on atomic operations, using a set of fine-grain tasks. Each fine task computes `C<#M>+=A(:,k1:k2)*B(k1:k2,j)` for a single column `j`, and a contiguous range `k1:k2`. Atomic updates are used to directly write into the result **C**. This method is used if the numerical intensity (the work required per entry in **C**) is low and if the workspace for the third method would otherwise be prohibitive.

(3) The third method also constructs fine-grain tasks, where task `t` computes its local term `W(:,t)=A(:,k1:k2)*B(k1:k2,j)`. Once all the tasks are finished, the second phase (after a barrier) reduces these local terms `C<#M>+=W(:,t)`, where each task is given its own unique range of entries in **C** to operate on. No atomics are used, but a significant amount of workspace is required.

4.3.2 Bitmap saxpy: A bitmap/full, B sparse/hypersparse. To compute $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{AB}$ when **A** is bitmap/full and **B** is sparse/hypersparse, the rows of **A** and **C** are subdivided into panels, which are computed one at a time. The outer iterations compute `C<#M>(p1:p2,:)+=A(p1:p2,:)*B` for a range of rows `p1:p2` in the panel. To compute the panel, `G=A(p1:p2,:)` is loaded into a shared bitmap matrix `G`, just large enough for a single panel. Next, `H=G*B` is computed where `H` is also held in bitmap format, and where each coarse-grain task computes one section of `H` using a 2D assignment of tasks, as `H(i1:i2,k1:k2) = G(i1:i2,:)*B(:,k1:k2)`. Finally, after `H` is computed for this outer panel, it is accumulated in parallel into **C**, as `C<#M>(p1:p2,:)+=H`, also using a 2D task assignment to the matrix `H`.

4.3.3 Bitmap saxpy: both A and B bitmap/full. Finally, a third algorithm computes $\mathbf{C}\langle\sharp\mathbf{M}\rangle = \mathbf{AB}$ when both **A** and **B** are bitmap or full. A 2D task decomposition is used where each task computes its own region of **C**. All tasks exploit the mask, if present.

4.4 Sparse masked dot-product

The sparse matrix 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 then the mask matrix is present. For this case, the matrix **C** has the same sparsity pattern 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**, using one of 12 different algorithms. The sparse/hypersparse methods are identical so only the sparse case is listed:

- methods 1 to 4: **A** and **B** are both bitmap or full: a simple dense vector computation, where the bitmap of **A** and/or **B** must be tested, resulting in 4 different variants.
- methods 5 and 6: **A** is bitmap/full and **B** is sparse: The method iterates over the sparse vector `B(:,j)` and does an $O(1)$ lookup of the matching entry in `A(:,i)`, either full (method 5) or bitmap (method 6).
- methods 7 and 8: **A** is sparse and **B** is bitmap/full: the reverse of methods (5) and (6).
- methods 9 to 12 handle case when both **A** and **B** are sparse. For method 9, a quick check is made if the last entry of **A** comes before the first entry of **B**, or visa versa, and if this holds then there is no work to do.
- method 10: `A(:,i)` has many more entries than `B(:,j)`. In this case, the search of entries in `A(:,i)` is accelerated with a binary search, so the work is at most $O(b \log a)$ if a and b denote the number of entries in the two vectors.
- method 11: `B(:,j)` has many more entries than `A(:,i)`; the reverse of method 10, taking at most $O(a \log b)$ time.
- method 12: otherwise, a set intersection is computed in $O(a + b)$ time.

4.5 Bitmap unmasked dot-product

Computing $\mathbf{C} = \mathbf{A}^{\mathsf{T}}\mathbf{B}$ using the dot product method, or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A}^{\mathsf{T}}\mathbf{B}$ can be very costly, since it requires $O(mn)$ sparse dot products if **C** is m-by-n. However, if **C** is small compared with **A** and **B**, this can be a very useful method. It is required for the "pull" phase of the push/pull breadth-first search, for example, and is also used in the direction-optimizing Betweenness-Centrality method described in Section 11.

Since all entries in **C** must be considered, the simplest approach for computing **C** is to construct it in bitmap form. The same 12 dot product methods described in Section 4.4 can then be used, but on all the entries, not just the entries in the sparsity pattern of the mask.

4.6 Dot-product with full accumulator

The PageRank computation described in Section 11 requires the computation of $\mathbf{C} \odot\!= \mathbf{A}^{\mathsf{T}}\mathbf{B}$ where **C** is a dense vector, and where the accumulator operator is present and matches the monoid of the semiring. This is a very specialized method, but critical to the performance of iterative methods such as PageRank or computing the Fielder vector of the Laplacian of a graph. Each dot product can be computed independently, and there is no mask, so the parallelism is simple. A similar set of 12 different kinds of sparse dot products (described in Section 4.4) is used for this method.

4.7 Sparse row and column scaling

Finally, SuiteSparse:GraphBLAS includes two methods for scaling the rows or columns of a matrix, as $\mathbf{C} = \mathbf{DB}$ or $\mathbf{C} = \mathbf{AD}$, where **D** is a diagonal matrix. The parallelism of these to methods is simple. The column scaling divides up the matrix **A** using the slicing technique described in Section 3.4, since each operation requires knowledge of the column index. The row scaling does not require a complex division of work, since each entry in **B** can be examined for its row index (assuming it is stored in column-oriented form) and then $c_{ij} = d_{ii} \otimes b_{ij}$ can be computed.

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 pattern is the set union of $\mathbf{A}$ and $\mathbf{B}$ (or a subset of that pattern 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 SS:GrB 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 pattern 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, $\mu << a$ and $\mu << b$, 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.

Nine different methods are used when **C** is bitmap, listed below. The first three methods are used when the mask is not present. Methods 24 to 26 use a sparse or hypersparse mask, and in this case it is very efficient to first scatter the entries of the mask into the bitmap of **C** (this technique is described in Section 4.3).

(21) $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$ where all three matrices are in bitmap form.
(22) **C** and **A** are bitmap; **B** is sparse or hypersparse. The matrix **A** is copied into **C** in $O(mn)$ time, and then the sparse/hypersparse matrix **B** is traversed and its entries added to **C**.
(23) **C** and **B** are bitmap; **A** is sparse or hypersparse; the reverse of Method 21.
(24) $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ where **M** is sparse and all other matrices are bitmap or full. All entries must be considered, in $O(mn)$ time if the matrices are m-by-n.
(25) $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ where **M** and **B** are sparse or hypersparse, and **C** and **A** are bitmap or full. The matrix **A** is copied into **C** where permitted by the mask. Next, the sparse **B** is traversed and added to **C**, where permitted by the mask.
(26) $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ where **M** and **A** are sparse or hypersparse, and **C** and **B** are bitmap or full. The reverse of Method 25.
(27) $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ where all four matrices are bitmap or full. The mask **M** is in bitmap form so it is not scattered into the bitmap of **C**. Time is $O(mn)$.
(28) $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ where **B** is sparse or hypersparse and all other matrices are bitmap or full. This method is like Method 25, except that the mask is not scattered into the bitmap of **C**.
(29) $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$ where **A** is sparse or hypersparse and all other matrices are bitmap or full. The reverse of Method 28.

The parallelism of all the methods listed above is straight-forward. Most of the algorithms need to traverse a single sparse or hypersparse matrix in parallel. This is required for matrix **B** in Methods 22, 25, and 28, above, for **A** in Methods 23, 26, and 29, and to scatter the mask **M** into the bitmap of **C** (Methods 24 to 26). In this case, the single sparse matrix is sliced and traversed in parallel (Section 3.4). Traversing all entries in a bitmap matrix is is very trivial to do in parallel.

6 PARALLEL ELEMENT-WISE MULTIPLY (SET INTERSECTION)

When using the `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 pattern 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 pattern and values of **C**.
   (b) $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ with no mask, and **B** is full. The pattern 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 pattern 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 pattern 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 pattern 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 pattern 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 pattern 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. 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 patterns 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 SS:GrB 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, SS:GrB 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 pattern 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 [12], and are used here nearly unmodified, except the iso property is handled in v5.1.5 of SS:GrB. 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. In ACM Algorithm 1000 (v2.3.5 SS:GrB), its implementation required nearly 4,000 lines of code [12]. In v5.1.5, this has expanded by a factor of 3, not merely because of parallelism. The method has been split into 27 unique methods

when **C** is sparse or hypersparse, 5 methods when it is full, and 16 methods when it is bitmap. The scope of these different 48 methods is too broad to fully describe here, so instead an overview is given, along with a categorization of how the 48 methods are parallelized.

In the *GraphBLAS C API Specification*, the matrix **C** and its mask **M** have the same size. SS:GrB adds `GxB_subassign`, where **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 *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 $\mathbf{C}\langle\mathbf{M}, replace\rangle$.

Finally, two kinds of assignment are permitted: matrix assignment or scalar assignment. In the latter, a single scalar σ is expanded, as if it were a dense matrix of size $|\mathbf{I}|$-by-$|\mathbf{J}|$, for the assignment $\mathbf{C}(\mathbf{I},\mathbf{J}) = \sigma$, just like the MATLAB notation `C(I,J)=s` for a scalar `s`.

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 **C**. This process is described in [12], except that **S** is constructed in parallel using all the methods described in Section 7.

The 27 methods when **C** is sparse or hypersparse divide into 10 different categories, depending on how they are parallelized. In the list of methods below, $\mathbf{C}\langle\mathbf{M}\rangle$ denotes either a valued or structural mask, whereas $\mathbf{C}\{\mathbf{M}\}$ denotes a purely structural mask.

(1) **Parallel traversal of M**: The matrix **M** is sliced (Section 3.4).
   - $\mathbf{C}(:,:)\{\mathbf{M}\} = \mathbf{A}$ where **C** is initially empty and **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 **S**.
   - $\mathbf{C}(\mathbf{I},\mathbf{J}) = \sigma$
   - $\mathbf{C}(\mathbf{I},\mathbf{J}) \odot= \sigma$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle = \sigma$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle \odot= \sigma$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, replace\rangle = \sigma$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, replace\rangle \odot= \sigma$

(3) **Element-wise add of A + S**: These methods construct the matrix **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 SS:GrB can be 1000x faster than MATLAB (see Section 11).
   - $\mathbf{C}(\mathbf{I},\mathbf{J}) \odot= \mathbf{A}$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle = \mathbf{A}$ when **A** is sparser than **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 SS:GrB can be 248,000x faster than MATLAB.
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle \odot= \mathbf{A}$ where both **M** and **A** are bitmap (see Method (8) when this condition does not hold).
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}, replace\rangle = \mathbf{A}$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}, replace\rangle \odot= \mathbf{A}$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle = \mathbf{A}$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}\rangle \odot= \mathbf{A}$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, replace\rangle = \mathbf{A}$
   - $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\neg\mathbf{M}, replace\rangle \odot= \mathbf{A}$

(4) **Element-wise add of M + S**: These methods construct the matrix **S**, and must traverse all entries that would appear in the element-wise addition of **M** + **S**. The term **M** + **S** itself is not computed, but the same parallel techniques of Section 5 are used. If **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 M**: the matrix **M** is split into a set of fine and coarse tasks, where a coarse task operates on a set of whole vectors of **M**, and a fine task operates on a subset of a single vector of **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 **M** is sparser than **A**. This akin to logical indexing in MATLAB (`C(M)=A(M)`), except that SS:GrB can be 248,000x faster than MATLAB (see Section 11).
- $\mathbf{C}(\mathbf{I},\mathbf{J})\langle\mathbf{M}\rangle \odot= \sigma$

(6) **Simple copy**: the matrix **M** or **A** is copied into **C**
- $\mathbf{C}(:,:)\{\mathbf{M}\} = \sigma$ where the pattern of **M** is copied into **C**, and **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 **C** becomes iso-valued.
- $\mathbf{C}(:,:)\{\mathbf{C}\} = \sigma$ where **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 **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 A ⊗ M**: This method is uniquely handled with its own parallelization technique. The matrices **M** and **A** are not bitmap. The matrix **S** is not constructed. The parallelization is the same as the element-wise multiplication of **A** ⊗ **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 **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 **C** are modified with a simple parallel loop.
- $\mathbf{C}(:,:) \odot= \sigma$ where **C** is full.

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

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

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

9 PARALLEL TRANSPOSE

SS:GrB includes 8 different parallel methods for computing the transpose of a matrix, $\mathbf{C} = \mathbf{A}^{\mathsf{T}}$. These methods can also typecast the matrix and/or apply a unary or binary operator (for the latter, a binary operator is used with one input bound to a scalar).

The two parallel methods when $\mathbf{A}$ is bitmap or full are simple and not described here.

Two methods handle row and column vectors. A `GrB_Vector` cannot be transposed by a user application, but internally they can be. Vectors and matrices with a single row or column are transposed with their own specialized methods. A sparse n-by-1 column vector $\mathbf{u}$ can be easily converted into a hypersparse 1-by-n matrix $\mathbf{w}$ held in column form, where the row indices `u.i` become the hyperlist `w.h`. This enables the transpose of vectors of arbitrary dimension, which cannot be done in MATLAB since it does not have hypersparse matrices and so would take $O(n)$ time and memory. These two methods parallelize efficiently.

Four methods are used in the general case, and a heuristic selects the one to use. The most flexible method converts the input matrix to tuple form, as three arrays `I` (row indices) `J` (column indices), `X` (values), and then uses the parallel `build` method to convert these tuples into the transpose, after reversing the roles of `I` and `J`.

The final three methods are all variants of a bucket sort: (1) a sequential method, (2) a method relying on atomics, and (3) a non-atomic method. Transposing column-oriented matrices $\mathbf{A}$ into $\mathbf{C}$, the entry a_{ij} is placed in the ith bucket of $\mathbf{C}$. A workspace of size at least m is required, if $\mathbf{A}$ is m-by-n, so this method cannot be used for large hypersparse matrices. The method is simple with one thread, so that is implemented as a special case. The non-atomic method is fast in practice with p threads when p is small but it requires $O(pm)$ workspace. Thus it does not scale at all when p is large. The atomic method uses $O(m)$ space, independent of p, and scales well.

The `build` method easily parallelizes via a parallel mergesort of the tuples, and allows for the transpose of huge hypersparse matrices, but it can take more work (by a logarithmic factor) than using a bucket sort. The work for the build is $O(e \log e)$ if $\mathbf{A}$ has e entries, while the bucket methods require $O(e + m + n)$ work. The parallel time is $O(e \log e)/p)$ and $O((e + m + n)/p)$, respectively, for these methods.

10 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 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.

11 PERFORMANCE

11.1 GraphBLAS in MATLAB

SuiteSparse:GraphBLAS has its own interface to Octave and MATLAB, where it provides access to all of the features of GraphBLAS via 64 unique functions, 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). SuiteSparse:GraphBLAS v3.3.3 is built into MATLAB R2021a for its sparse matrix multiply, so `C=A*B` is now equally fast with both MATLAB sparse matrices and GraphBLAS matrices.

Other methods in MATLAB do not yet use GraphBLAS, but SS:GrB 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, on a 20-core Intel Xeon with MATLAB R2020b.

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 SS:GrB would be 1,000,000x faster than MATLAB can be easily constructed.

11.2 LAGraph and the GAP benchmark

Rather than provide performance results of the individual GraphBLAS operations described in the previous sections, it is far more useful 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 [4] specifies six different graph algorithms, to be benchmarked on five different matrices. The reference 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, 22, 28]. 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 the June 24, 2021 version of LAGraph (in the `reorg` branch), along with SuiteSparse:GraphBLAS v5.1.5, on an NVIDIA DGX Station (20-core Intel Xeon CPU E5-2698 v4 @ 2.20GHz, a single socket system with 40 threads in OpenMP, with gcc 5.4.0 -O3). Hyperthreading was left enabled, which is the default on this system.

The six algorithms in the benchmark are: (1) breadth-first search, to construct the BFS tree, [31], (2) betweenness-centrality, (3) pagerank, (4) connected components [32], (5) single-source shortest path [26], and (6) triangle counting [3, 29]. The five graphs are shown in Table 1. Table 2 reports the parallel performance results of the GAP Benchmark and LAGraph+SS:GrB.

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 1. Benchmark matrices

Algorithm :	graph, with run times in seconds				
package	Kron	Urand	Twitter	Web	Road
BC : GAP	31.52	46.36	10.82	3.01	1.50
BC : SS	23.05	32.72	8.79	6.49	34.76
BFS : GAP	.31	.58	.22	.34	.25
BFS : SS	.53	1.23	.35	.70	3.05
PR : GAP	19.81	25.29	15.16	5.13	1.01
PR : SS	21.94	27.87	17.61	9.37	1.67
CC : GAP	.53	1.66	.23	.22	.05
CC : SS	3.44	4.60	1.59	2.03	1.01
SSSP : GAP	4.91	7.23	2.02	.81	.21
SSSP :SS	17.13	25.20	8.63	9.10	49.91
TC : GAP	374.08	21.82	79.58	22.18	.03
TC : SS	927.98	33.66	237.10	34.64	.19

Table 2. Performance results

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 2.1. 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+SS:GrB. 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. Note that 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, and so 64-bit indices are essential.

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. However, in real applications, the triangle count itself is of little interest. Of far greater interest is an subsequent method that makes use of the triangles themselves, such as the K-truss algorithm, or Burkhardt's recent *triangle centrality* metric [9].

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.

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.

SS:GrB 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 SS:GrB 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.

12 RELATED WORK

SuiteSparse:GraphBLAS is just one of two full implementations of the *GraphBLAS C API Specification* [7, 8]. The other full implementation is the GPI package by IBM [14]. Other packages that implementation portions of GraphBLAS, or are based on similar ideas include CombBLAS (https://people.eecs.berkeley.edu/~aydin/CombBLAS/html/ [6]), Graphulo (http://graphulo.mit.edu/ [17]), D4M (http://d4m.mit.edu/ [20]), GraphPad ([2, 27]) Gunrock (https://github.com/gunrock/gunrock-grb, [30]), the GraphBLAS Template Library (Carnegie Mellon Univ., Indiana Univ., and PNNL) (https://github.com/cmu-sei/gbtl), and an implementation by Emu Technology (https://www.emutechnolgy.com).

Buluç et al., [23] in work done independently of the work reported in this paper, have developed four novel parallel algorithms for the masked matrix-matrix multiply, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$ and $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$ where all input matrices are sparse (not hypersparse, bitmap, or full). These include a (1) hash-based method similar to the masked coarse Hash method described in Section 4.2, and (2) a masked sparse accumulator (MSA) method, similar to the coarse Gustavson method in Section 4.2. They describe two algorithms mostly unrelated to the methods described in this paper: (3) a heap-based method, and (4) a mask compressed accumulator method (MCA). The first three methods support both a complemented and non-complemented mask, while the MCA method only supports a non-complemented mask. The traversal of the mask in the MCA method is analogous to traversing `M(:,j)` and using a binary search of `A(:,k)` discussed in Section 4.2.

The fine-grain methods presented in this paper are unique to this paper, and not considered by Buluç et al., as are the combinations of sparse/hypersparse/bitmap/full considered here, and the exploitation of the iso properties of the four matrices **C**, **M**, **A**, and **B**. In SuiteSparse:GraphBLAS, a single matrix multiply can use any mix of four kinds of tasks (coarse/fine Hash/Gustavson), which is also unique to this paper.

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

REFERENCES

[1] LAGraph. https://github.com/GraphBLAS/LAGraph, 2021.

[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] Ariful Azad, Aydın Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.

[4] Scott Beamer, Krste Asanovic, and David 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] 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.

[6] 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.

[7] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. The GraphBLAS C API specification. Technical report, 2017.

[8] Aydin Buluç, Tim Mattson, Scott McMillan, José E. Moreira, and Carl 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] Paul 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, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016.

[12] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019.
[13] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Computing*, 78:33–46, 2018.
[14] K. Ekanadham, W. Horn, J. Jann, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and Yu H. Graph programming interface: Rationale and specification. Technical Report RC25508 (WAT1411-052), IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, November 2014.
[15] 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.
[16] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.
[17] 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.
[18] J. Kepner. GraphBLAS mathematics. http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf, 2017.
[19] 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.
[20] 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.
[21] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra.* SIAM, Phila., PA, 2011.
[22] Tim Mattson, Timothy A. Davis, Manoj Kumar, Aydın Buluç, Scott McMillan, Jose Moreira, and Carl 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.
[23] S. Milaković, I. Nisa, O. Selvitopi, Z. Budimlić, and A. Buluç. Parallel algorithms for masked sparse matrix-matrix products. draft manuscript.
[24] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın 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.
[25] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Computing*, 90:102545, 2019.
[26] 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.
[27] 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.
[28] Gábor Szárnyas, David A. Bader, Timothy A. Davis, James Kitchen, Timothy G. Mattson, Scott McMillan, and Erik Welch. LAGraph: linear algebra, network analysis libraries, and the study of graph algorithms. IPDPS GrAPL'21, 2021.
[29] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.
[30] 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.
[31] Carl Yang, Aydın Buluç, and John 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.
[32] Yongzhe Zhang, Ariful Azad, and Aydın Buluç. Parallel algorithms for finding connected components using linear algebra. *Journal of Parallel and Distributed Computing*, 144:14–27, 2020.

Received July 2021