Parallel GraphBLAS with OpenMP *

Mohsen Aznaveh† Jinhao Chen† Timothy A. Davis† Bálint Hegyi‡
Scott P. Kolodziej† Timothy G. Mattson§ Gábor Szárnyas‡¶

Abstract

SuiteSparse:GraphBLAS is a complete implementation of the GraphBLAS standard. It provides a powerful and expressive framework for creating graph algorithms based on the elegant mathematics of sparse matrix operations on a semiring. Algorithms written with the GraphBLAS achieve high performance with minimal development time. Multithreaded parallelism through OpenMP provides additional speedup, which we illustrate on a 20-core Intel® Xeon® E5-2698 CPU system when solving various problems (triangle counting, k-truss, breadth-first search, Bellman-Ford, local clustering coefficient, and a sparse deep neural network problem). This wide variety of algorithms illustrates the expressiveness of the GraphBLAS API to create new graph algorithms. We present performance results with these algorithms on a set of large real-world graphs, using the newly developed SuiteSparse:GraphBLAS v3.0.1.

1 Introduction

The GraphBLAS standard [3] defines sparse matrix and vector operations on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms. Kepner and Gilbert [11] provide a framework for understanding how graph algorithms can be expressed as matrix computations. This approach leads to high performance since the library treats operations over graphs as bulk operations on adjacency matrices. User code need not deal with individual nodes and edges. Writing graph algorithms with GraphBLAS reduces development time as well, as illustrated by the results in this paper.

*With support from NSF CNS-1514406, NVIDIA, Intel, MIT Lincoln Lab, Redis Labs, and IBM. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

†Department of Computer Science and Engineering, Texas A&M University, College Station, TX

‡Department of Measurement and Information Systems, Budapest University of Technology and Economics

§Intel Corporation, Oregon

¶MTA-BME Lendület Cyber-Physical Systems Res. Group

To show the performance benefits of this parallel GraphBLAS implementation, results for several large-scale, computationally intensive graph-based problems are reported. These problems include the following:

1. **Triangle counting.** Count the number of triangles (cliques of size three) in a graph.
2. **4-Truss.** Compute the 4-truss subgraph of a graph, in which each edge of the subgraph is incident on at least two triangles.
3. **A breadth-first search of a graph.** The search starts at node zero and finds all nodes reachable from the starting node.
4. **The Bellman-Ford shortest path algorithm.** For vertex zero, find the shortest path distance to all other vertices in a directed, weighted graph with no self-edges.
5. **Computing the local clustering coefficient (LCC) for all vertices in a graph.** The LCC is the ratio between the number of edges between neighbors of a given node and the maximum possible number of edges between these neighbors.
6. **Sparse deep neural network forward propagation [10].** Given a deep neural network with known weights and sparse connectivity between layers, compute the outputs of the network given several input sets.

2 Overview of GraphBLAS objects, methods, and operations

SuiteSparse:GraphBLAS provides a collection of *methods* to create, query, and free each of its ten different types of objects. Once these objects are created they can be used in mathematical *operations* (not to be confused with how the term *operator* is used in GraphBLAS). The ten types are described below. User applications can also define their own data types, operators, monoids, and semirings.

(1) Types: A GraphBLAS type (`GrB_Type`) can be any of 11 built-in types (Boolean, integer and unsigned

Table 1: SuiteSparse:GraphBLAS Operations

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

integers of sizes 8, 16, 32, and 64 bits, and single and double precision floating point). User-defined scalar types can be created from nearly any C `typedef`.

(2) Unary operators: A unary operator (`GrB_UnaryOp`) is a function $z = f(x)$.

(3) Binary operators: Likewise, a binary operator (`GrB_BinaryOp`) is a function $z = f(x, y)$, such as $z = x + y$ or $z = xy$.

(4) Select operators: The `GxB_SelectOp` operator is a SuiteSparse extension to the GraphBLAS API. It is used in the `GxB_select` operation to select a subset of entries from a matrix, like `L=tril(A)` in MATLAB.

(5) Monoids: The scalar addition of conventional matrix multiplication is replaced with a *monoid*. A monoid (`GrB_Monoid`) is an associative and commutative binary operator $z = f(x, y)$ where all three domains are the same (the types of x, y, and z) and where the operator has an identity value o such that $f(x, o) = f(o, x) = x$. Performing matrix multiplication with a semiring uses a monoid in place of the "add" operator, scalar addition being just one of many possible monoids.

(6) Semirings: A *semiring* (`GrB_Semiring`) consists of a monoid and a "multiply" operator. Together, these operations define the matrix "multiplication" $\mathbf{C} = \mathbf{AB}$, where the monoid is used as the additive operator and the semiring's "multiply" operator is used in place of the conventional scalar multiplication in standard matrix multiplication via the plus-times semiring.

(7) Descriptors: A *descriptor* `GrB_Descriptor` with parameter settings for GraphBLAS operations.

(8) Scalars: A sparse scalar `GxB_Scalar`, currently only used as the input `k` to the `GxB_select` operation.

(9) Vectors: A sparse vector, `GrB_Vector`.

(10) Matrices: A sparse matrix, `GrB_Matrix`.

2.1 Non-blocking mode GraphBLAS includes a *non-blocking* mode where operations can be left pending, and saved for later. This is very useful for submatrix assignment (like `C(I,J)=A` in MATLAB), particularly when `A` is small compared to `C`. This exploitation of the non-blocking mode is essential for some problems, but has little effect on graph problems presented here, other than to simplify the reading of input graphs.

2.2 The accumulator and the mask An optional accumulator operator (*odot*) and mask matrix ($\mathbf{M}$) can be specified, written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ where $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ denotes the application of the accumulator operator, and $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ denotes the mask operator via the Boolean matrix $\mathbf{M}$. The mask matrix is used to selectively write results into the output matrix. For the

computation $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$, the assignment of the entry $c_{ij} = z_{ij}$ is performed only if the mask entry m_{ij} is true. If false, c_{ij} is not modified. The mask can be negated, as in $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{Z}$, in which case the assignment is performed only where the mask $\mathbf{M}$ is false. The mask/accumlator step also accepts a REPLACE option, via the descriptor. If the REPLACE option is enabled, the matrix $\mathbf{C}$ is cleared of all entries after it is used in the right-hand side (say in the accumulator step $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$), but before the result $\mathbf{Z}$ is written back into $\mathbf{C}$ via the mask. By default, the REPLACE option is not set. The accumulator operator can be any binary operator, either one of the 256 built-in operators, or a user-defined operator.

GraphBLAS methods and operations The matrix (`GrB_Matrix`) and vector (`GrB_Vector`) objects include additional methods for setting a single entry, extracting a single entry, making a copy, and constructing an entire matrix or vector from a list of *tuples*. The tuples are held as three arrays `I`, `J`, and `X`, which work the same as `A=sparse(I,J,X)` in MATLAB, except that any type matrix or vector can be constructed.

Table 1 lists all GraphBLAS operations in the GraphBLAS notation where $\mathbf{AB}$ denotes the multiplication of two matrices over a semiring. Upper case letters denote a matrix, and lower case letters are vectors. The notation $\mathbf{A} \oplus \mathbf{B}$ denotes the element-wise operator that produces a set-union pattern (like `A+B` in MATLAB). The notation $\mathbf{A} \otimes \mathbf{B}$ denotes the element-wise operator that produces a set-intersection (like `A.*B` in MATLAB). An optional accumulator operator ($\odot$) and mask matrix ($\mathbf{M}$) can be specified, written as $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$ where $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ denotes the application of the accumulator operator, and $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ denotes the mask operator via the Boolean matrix $\mathbf{M}$. The mask matrix is used to selectively write results into the output matrix.

3 Parallelism in SuiteSparse:GraphBLAS

SuiteSparse:GraphBLAS Version 2.3.5 is to appear as a Collected Algorithm of the ACM [6]. While its sequential performance is good, as illustrated in that paper, it does not exploit any parallelism at all. Version 3.0.1 has been released (July 31, 2019), with exploitation of multi-threaded parallelism expressed through OpenMP, presented in this paper. A GPU-accelerated version is also in progress, and an MPI version is planned in the more distant future.

3.1 Parallel algorithmic choices

There are many ways of creating a parallel algorithm. In nearly all methods described below, we have implemented algorithms that scale well as the problems get larger. Given the number of OpenMP threads to use, `nthreads`, we first determine the number of independent tasks to create, typically some modest multiple of `nthreads` (say `ntasks=64*nthreads`). We then employ a parallel pragma of the following form

```
#pragma omp parallel for num_threads(nthreads)   \
   schedule(dynamic,1)
for (int tid = 0 ; tid < ntasks ; tid++)
{
    // compute task tid with no synchronization
    // required between tasks
}
```

In some cases, we add a reduction clause, such as `reduction(+,nzombies)` to add up the number of *zombies* found (see below for a discussion of zombies). In most algorithms, the work is perfectly distributed between the tasks. There are exceptions to this, such as the bucket-transpose method described in Section 3.9, or the bucket-reduction in Section 3.7, which do not scale to a large number of threads. However, in those cases, we are typically able to select a scalable algorithm when needed, automatically as depending on the problem at hand.

OpenMP tasking is used for the parallel mergsort.

3.2 GrB_mxm: Parallel matrix multiplication

The sequential version of SuiteSparse:GraphBLAS includes three different forms of matrix-matrix multiply: Gustavson's method [8], a heap-based method [2], and a dot-product based method. Each of these has a masked variant to compute $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, and the dot product variant can also compute $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}^{\mathrm{T}}$ during the computation of $\mathbf{AB}^{\mathrm{T}}$. To compute $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{AB}$ with a negated mask using the Gustavson or heap-based method, the full matrix multiplication $\mathbf{Z} = \mathbf{AB}$ is first computed, followed by a separate mask/accumulator step $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{Z}$.

By default, all matrices in SuiteSparse:GraphBLAS are held in compressed-sparse row (CSR) format, but the matrices can also be held in compressed-sparse column format (CSC). This discussion assumes the default CSR format.

Gustavson's method and the heap-based method are both *saxpy*-based, where the ith row is computed as a sum of scaled sparse vectors. In MATLAB notation, assuming the conventional plus-times semiring, the ith row is computed as:

```
for k = find (A(i,:))
    C(i,:) = C(i,:) + A(i,k) * B(k,:)
end
```

Gustavson's method uses a size-n gather/scatter workspace, where `C` is m-by-n. In the parallel

case, each thread requires its own workspace. SuiteSparse:GraphBLAS keeps a set of workspaces that can be used in subsequent operations, to reduce the time to initialize this space. Some uses of $\mathbf{C} = \mathbf{AB}$ take less than $O(m)$ time, so it is essential to avoid the $O(m)$ time need to initialize this space. However, for large numbers of threads, the Gustavson method does not scale well. The heap-based method avoids this problem, but (at least in our current implementation) it is not as fast as Gustavson's method. It merges the vectors of `B` using a heap of size `nnz(A(i,:))`. In both methods, all rows of `C` can be computed in parallel.

Our current implementation divides the work into a single task for each thread, where the tasks are chosen to balance the operation count in each task. Each submatrix of `C` is computed in parallel, and then the resulting submatrices are concatenated together.

For both methods, the matrix $\mathbf{A}$ is divided into t partitions, for t threads. Then the ith thread computes $\mathbf{C}_i = \mathbf{A}_i\mathbf{B}$, and when all threads finish, the submatrices are concatenated. The partitions are chosen so as to balance the operation count for each thread. If $\mathbf{A}$ has too few rows for this to be effective, the partitions are chosen to split individual rows of $\mathbf{A}$, and the resulting matrix $\mathbf{C}$ must be summed for these rows, not just concatenated.

The dot-product method takes a different approach. It computes `C=A*B'`, if the matrices are in CSR format. Each entry `C(i,j)` is computed independently. The method is not well-suited for general matrix-matrix multiplication, since all mn dot products must be computed. The time is thus $\Omega(mn)$, so this method is only used if m or n are 1. However, if the mask is present, only entries in the mask need be computed. In this case, the dot product method can be much faster than Gustavson's method or the heap-based method, for computing the masked-dot-product, $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}^{\mathrm{T}}$.

SuiteSparse:GraphBLAS automatically selects the method to use, although the user application can make this selection instead. By default, GraphBLAS selects the masked-dot-product method for triangle counting, LCC, the pull phase of the push/pull BFS and Bellman-Ford, and the simple BFS. The saxpy-based Gustavson or heap-based methods are used in the K-truss, the push phase of the push/pull BFS and Bellman-Ford, and some forms of triangle counting. PageRank uses an unmasked dot product method.

The SuiteSparse implementation of `GrB_mxm` also includes two specialized matrix multiplication methods, in which one of the matrices `A` or `B` are diagonal. These two methods are easy to parallelize, and are fast sequentially as well. PageRank uses the scaling method to initialize its Markov transition matrix, and the sparse deep neural network problem uses it to apply the bias to each column.

By default, the matrix-matrix multiply algorithm is selected automatically via a heuristic, and typically this heuristic chooses the fastest method for the problem at hand.

3.3 GrB_eWiseAdd and GrB_eWiseMult: Parallel element-wise operations Two element-wise operations compute the equivalent of `C=A+B` (sparse matrix add) and `C=A.*B` (the sparse Hadamard product). These operations are easy to parallelize, where each row can be computed independently by its own task. However, the resulting algorithm can be poorly load-balanced if the matrices have a few dense rows. In particular, if `A` and `B` are large `GrB_Vector` objects, then this coarse-grain parallelism allows for only a single thread to do most the work.

Thus, we use purely coarse-grain parallelism only when it provides enough work for each thread. Otherwise, a mix of fine/coarse-grain tasks are used. Each row vector that needs multiple threads is broken into multiple fine-grain tasks. It is not sufficient to simply divide up the index space. For example, consider adding two vectors `c=a+b` of length `n`, and dividing up the work as follows:

```
s = n/2 ;
c(1:s)   = a(1:s)   + b(1:s) ;
c(s+1:n) = a(s+1:n) + b(s+1:n) ;
```

If all the entries in `a` and `b` are in the first half, then this gives no speedup. The value `s` must instead be chosen to split the work in half. A binary search is used compute `s`. For each candidate value of `s`, two inner binary searches partition `a` and `b` into entries in rows 1 to `s` and `s+1` to `n`. All vectors are held with sorted indices. The sizes of these partitions determine if `s` must be increased or decreased. These nested binary searches take $O(\log^2 n)$ time in the worst case, or take $O(\log n(\log|a| + \log|b|))$ work in the general case, where $|a|$ denotes the number of entries in the sparse vector `a`. However, each of the f tasks can be constructed in parallel.

In general, the work for a dense row of `C` is split into as many fine tasks as needed. If f tasks as used, the preprocessing to construct these tasks takes $O(f \log^2 n)$ time, but f is normally small, $f < 32p$ if p OpenMP threads are used. The number of fine tasks is chosen based on the number of entries in `a` and `b`, so that each task operates on at least 4096 entries.

The work is split into four phases, each of which are fully parallel: (1) find the vectors present in `C`, (2) split the work into fine/coarse tasks, (3) count the

number of entries in each vector of the result (followed by a cumulative sum to determine where each vector resides in the result `C`), and (4) compute each vector of the result. Each phase is fully parallel, and handles any matrix or vector with no sequential bottleneck. In important cases, phase (3) takes less time in practice than phase (4), and is asymptotically faster as well. For example, if the vector `a` is much sparser than `b`, the size of `c` (the set union) can be found by computing the size of the set intersection instead, in time $O(|a| \log |b|)$, where $|a| << |b|$. Phase (4) for this case takes $O(|a|+|b|)$ time.

3.4 GrB_extract: Parallel submatrix extraction The GraphBLAS operation `GrB_extract` computes the equivalent of `C=A(I,J)` in MATLAB. The parallel algorithm in SuiteSparse:GraphBLAS divides into four distinct phases, just like the element-wise operations: (1) find the vectors of `C` and the properties of `I` and `J`, (2) split the work into fine and coarse tasks, (3) count the number of entries in each vector of the result `C` (followed by a parallel cumulative sum), and (4) extract the entries from `A` into `C`. The first phase is trivial if `A` and `B` are standard CSR or CSC matrices, but it must construct a set union if the both are hypersparse. The second phase constructs fine and coarse grain tasks, using a similar method as the element-wise operations, `GrB_eWiseAdd` and `GrB_eWiseMult`. Phase (3) is a 'dry-run', doing all the work of the extraction but not saving the results. All it does is to count the number of entries in each vector of `C`. This may seem inefficient, but it leads to better parallelism. Phase (3) is typically much faster than phase (4), since it is a read-only process, and it is asymptotically faster in many cases.

3.5 GrB_assign: Parallel submatrix assignment Submatrix assignment is the most complex method in GraphBLAS, in terms of code complexity and algorithm variations. From the user perspective, it looks deceptively simple, computing $\mathbf{C}(\mathbf{i},\mathbf{j})\langle\mathbf{M}\rangle\odot = \mathbf{A}$ if an accumulator operator is present (where $\odot$ denotes any binary operator). MATLAB refers to the mask operation as *logical indexing*, where `C(M)=A(M)` in MATLAB is equivalent to the GraphBLAS notation $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}$. However, in MATLAB, the mask $\mathbf{M}$ cannot be combined with the $(\mathbf{i},\mathbf{j})$ indexing operation. With no mask, $\mathbf{C}(\mathbf{i},\mathbf{j}) = \mathbf{A}$ is the same in MATLAB (`C(I,J)=A`). The GraphBLAS notation $\mathbf{C}(\mathbf{i},\mathbf{j})\odot = \mathbf{A}$ is the expression `C(I,J)=C(I,J)+A` in MATLAB. There is no REPLACE option in MATLAB.

The complete parallel implementation of this method takes 6K lines of code. It subdivides into 32 different cases: $\mathbf{A}$ may be a matrix or a scalar, the accumulator may or may not be present, the mask may or may not be present, the mask may or may not be complemented, and the REPLACE option may or may not be used. Of these 32 cases, some are redundant, and so SuiteSparse:GraphBLAS has 22 different functions for these 32 cases.

In all cases, the pattern of `C` may be changed, but this is always postponed. It is left unfinished if non-blocking mode is enabled, or finished at the end of the assignment otherwise. This design decision greatly facilitates a parallel algorithm for each of these 32 cases.

If the submatrix assignment needs to delete an entry, it is not deleted right away. Instead, it is marked for future deletion, but "negating" its index (the actual negation of a column index `j` is `flip(j)=-j-2`, to allow for zero-based indices. The flip function is its own inverse, just like negation for one-based indices. These entries marked for deletion are called *zombies*. Removing a single zombie is costly, but all zombies can be removed from matrix, in parallel, in the same time as removing a single zombie.

If the submatrix assignment needs to add an entry, it is not added right away. Instead, it is placed in an unsorted list of *pending tuples*, each with a row index, column index, and value. Pending tuples are assembled all at once, when the work is finished, and added into the matrix with a parallel sparse matrix addition.

Since the pattern of `C` does not change during the assignment, the assignment can start with a symbolic extraction. The matrix `S=C(I,J)` can be computed, in parallel. The entries in `S` are not the values of the corresponding entry in `C`, however. Instead, the entries are pointers back into `C` itself. Thus, to modify an entry in `C`, it can be found quickly by traversing the matrix `S`.

Most but not all submatrix assignment methods start with the construction of the matrix `S`. Three cases are best done without `S`. For one of the 32 cases, it is sometimes useful to use `S`, and sometimes faster not to create it at all, and the decision is based on the sparsity of `A` and the mask `M`.

The 32 cases are listed below, along with the parallel strategy used for the method. The 32 cases use 22 functions but only 4 different parallel strategies, listed in the last column of the table:

- `IxJ`: the method must examine all positions in `C` in the Cartesian product `IxJ`, even where no entries exist. The space `IxJ` is subdivided into independent tasks.
- `M` or `S`: the method must examine all entries in the mask `M` or the matrix `S`. This one matrix is subdivided into coarse and fine tasks, of roughly the same amount of work. A coarse task takes one

or more entire rows of the matrix, while a fine task takes a subset of a single row (this is useful for handling dense rows in `M` or `S`).

- `S+A` or `S+M`: This is identical to how the sparse matrix addition of two matrices performed. The same scheduling method is used from `GrB_eWiseAdd_Matrix`.

- `M.*A`: this is the same method as used by `GrB_eWiseMult_Matrix`, for the two matrices `M` and `A`. Only entries in the intersection of `M` and `A` need be examined.

M	¬	R	+	A	S	Method	Para.
-	-	-	-	x	S	`C(I,J) = x`	`IxJ`
-	-	-	-	A	S	`C(I,J) = A`	`S+A`
-	-	-	+	x	S	`C(I,J) += x`	`IxJ`
-	-	-	+	A	S	`C(I,J) += A`	`S+A`
-	-	r				(same as 1st 4 methods)	
-	c	-				(no work)	
-	c	r			S	`C(I,J)<!,repl>=[ ]`	`S`
M	-	-	-	x	-	`C(I,J)<M> = x`	`M`
M	-	-	-	A	-	`C(I,J)<M> = A`	`M`
M	-	-	-	A	S	`C(I,J)<M> = A`	`S+A`
M	-	-	+	x	-	`C(I,J)<M> += x`	`M`
M	-	-	+	A	-	`C(I,J)<M> += A`	`M.*A`
M	-	r	-	x	S	`C(I,J)<M,repl> = x`	`S+M`
M	-	r	-	A	S	`C(I,J)<M,repl> = A`	`S+A`
M	-	r	+	x	S	`C(I,J)<M,repl> += x`	`S+M`
M	-	r	+	A	S	`C(I,J)<M,repl> += A`	`S+A`
M	c	-	-	x	S	`C(I,J)<!M> = x`	`IxJ`
M	c	-	-	A	S	`C(I,J)<!M> = A`	`S+A`
M	c	-	+	x	S	`C(I,J)<!M> += x`	`IxJ`
M	c	-	+	A	S	`C(I,J)<!M> += A`	`S+A`
M	c	r	-	x	S	`C(I,J)<!M,repl> = x`	`IxJ`
M	c	r	-	A	S	`C(I,J)<!M,repl> = A`	`S+A`
M	c	r	+	-	S	`C(I,J)<!M,repl> += x`	`IxJ`
M	c	r	+	A	S	`C(I,J)<!M,repl> += A`	`S+A`

In table above, the first 5 columns define the method: (1) if `M` is present, (2) if `M` is complemented, (3) if the REPLACE option is used in the descriptor, (4) if an accumulator operator is present, and (5) if the assignment is a matrix assignment (`A`), or a scalar assignment (`x`). The 6th column denotes whether or not the `S` matrix is constructed. The corresponding assignment expression is shown under the "Method" column and "Para." denotes the parallelism strategy that is used.

All of these methods are either asymptotically optimal, or to within a log factor of being optimal. For the parallel strategies, all tasks generated are fully independent, and roughly the same size. The methods all scale to a large number of tasks since the tasks can be either "coarse" (operating on one or more rows) or "fine" (where individual rows can be subdivided, all the way down to single entries).

The graph algorithms discussed in this paper use only a small subset of these methods. The simple BFS requires `C(:,:)<M>=x` and `C(:,:)<M,repl>=A`, where the colon denotes `GrB_ALL` in GraphBLAS notation, for all rows or columns. The push/pull BFS also uses `C(:,:)+=x` and `C(:,:)<M>+=x`, to construct the BFS tree. For both BFS methods, `C` and `A` are vectors. PageRank relies on a single `C(:,:)=x` vector assignment.

The performance of the submatrix assignment is typically either just as fast as the same operation in MATLAB, when one or two threads are used, or sometimes vastly asymptotically faster. For example, consider a very large assignment `C(I,J)=A`, where `C` has dimension 12 million with 230 million nonzeros. The matrix `A` is 5500-by-7800 with about 38,500 entries. The index vectors are randomly chosen. For this simple MATLAB expression, MATLAB R2018a takes about 30 minutes. Computing the same thing with a single thread in GraphBLAS takes about 5 seconds, and 40 threads reduces the time to 0.75 seconds. Non-blocking mode was not exploited; GraphBLAS took this time to return a completed matrix back to MATLAB, as a valid MATLAB sparse matrix, through a MATLAB mexFunction interface, with no unfinished work. This represents a speedup of 2,660x over MATLAB, for a computation that is a simple one-line expression in MATLAB, namely `C(I,J)=A`.

3.6 GrB_apply: Parallel unary operators The `GrB_apply` operation applies an unary operator to each entry in `A`, as `C=f(A)`. It is easy to parallelize, but only if dense vectors of `A` are allowed to be split into multiple tasks. This is done in a simple manner that does not require a nested binary search (as used in `GrB_eWiseAdd`). The entries of `A` are split equally. If a vector has many entries, it is split amongst several tasks ("fine" tasks). All the tasks are the same size.

3.7 GxB_select: Parallel selection operators SuiteSparse:GraphBLAS adds the `GxB_select` operation as an extension to the API Specification. It selects a subset of entries from a matrix, keeping only those for which the select operator is true. In this way, the select operator acts much like a functional mask. The selector can depend on the row and column index, the dimension of `A`, and the value of the entry. There are two kinds of built-in operators: those that depend solely on the position of the entry (like `tril` and `triu`), and those that depend on the value (such as keeping only nonzero values). The parallelism for the two kinds of operators is slightly different. Both are split into an analysis phase that counts the number of entries in each vector of the

result, and a execution phase that constructs the result. Methods that depend only on the position require only a binary search for each vector in the analysis phase. Computation is divided into tasks in much the same as `GrB_apply`.

`GxB_select` was added to SuiteSparse:GraphBLAS for the 2018 Graph Challenge, since it was needed to compute the lower triangular part of a matrix for the triangle counting problem [5]. It was also used in the sparse deep neural network problem as the ReLU function, which must drop entries at each layer, keeping only those greater than zero [7].

3.8 GrB_reduce: Parallel reduction GraphBLAS has two kinds of reduction operations: reduction to a vector, and reduction to a scalar; both are called via `GrB_reduce`. At first glance, scalar reduction is simple to compute in parallel, but to improve vectorization for built-in reduction operators, the entries are split into panels of size 8 to 64 entries, depending on the operator and data type. The innermost loop is thus no longer a reduction, and all iterations are independent. As a result, it vectorizes better, giving a 5x speedup for the `max` and `min` operators, over the non-vectorized method. The panel method is used inside each parallel task.

The reduction to a vector can be computed in two ways. For an m-by-n matrix `A`, either all the entries in each row can be reduced to a scalar, leading to a vector of size m, or all entries in each column can be reduced to a scalar, leading to a vector of size n. If the matrix is stored by row, the first method is straight-forward, although dense vectors must be handled with fine-grain tasks to obtain sufficient parallelism. In that case, the results from multiple threads working in a single row must be combined to get the final result.

Reducing the columns of a matrix in CSR format is more difficult. If the matrix is extremely sparse ($|A| < n/16$), this is accomplished by discarding the row indices, and using `GrB_build` to build a vector of dimension n, with the reduction operator as the operator for removing duplicate entries (see Section 3.10), taking $O(|A| \log |A|)$ time, if a single thread is used. This relies on a parallel mergesort for parallelism. No part of the time complexity depends on the matrix dimension, so a hypersparse matrix of dimension 2^{60}-by-2^{60} can be reduced quickly on a low-powered laptop to a vector of dimension 2^{60}, in well under a second, depending on the number of entries.

Otherwise, if the matrix is not extremely sparse, a bucket method is used. Each task reduces a set of rows of $\mathbf{A}$ to its own vector of dimension n, and a final parallel step reduces these vectors to a single vector of dimension n. This takes pn workspace for p threads, so to ensure that the workspace does not dominate the method, at most $p = |A|/n$ threads are used. This implies that the bucket method does not scale well to very many threads. In that case, the mergesort method with `GrB_build` is used instead. The selection is based on which method takes the least total workspace.

3.9 GrB_transpose: Parallel transpose SuiteSparse:GraphBLAS implements two kinds of parallel transpose, and selects between them automatically. The first one constructs a list of tuples from `A`, swaps them, and sorts the result, using `GrB_build` to construct the output matrix. This method is well-suited for hypersparse matrices, since no part of the time or memory complexity depends on the matrix dimensions. The second method uses a method like bucket-sort. In the first phase, each task counts entries in each vector of the output, for its input rows (assuming `A` is in CSR format). The next step sums up those counts, and the final step builds the result. Like the bucket method for `GrB_reduce`, each of the p tasks requires size n workspace, and so the number of threads is limited to $p = |A|/n$. The method thus does not scale well to very large numbers of threads, but it does get good parallelism in spite of this, for large matrices.

3.10 GrB_*_build: Parallel matrix and vector build The `GrB_Matrix_build` operation creates a CSR or CSC matrix from a list of unsorted tuples (each with a row index, column index, and value). The parallel method divides into five phases. Phase 1 makes a copy of the user input. Phase 2 sorts the tuples, using a parallel mergesort. The parallel mergesort algorithm uses OpenMP explicit task constructs to implement a parallel divide-and-conquer algorithm. Task constructs recursively partition the arrays until the result matches the base-case size. At that point we sort the arrays using a sequential quick sort algorithm. We then merge the results in parallel inside the OpenMP explicit task constructs. Phase 3 finds the non-empty vectors and the duplicate entries in $O(e/p)$ time. Phase 4 constructs the vector pointers, and list of non-empty vectors. The final phase assembles the tuples. All phases except the parallel mergesort in phase 2 take $O(e/p)$ time, with p threads and an input of e tuples. `GrB_Vector_build` creates an analogous sparse vector.

This method is well-suited for constructing hypersparse matrices, since no part of the time or memory complexity depends on the matrix dimensions. The output is always constructed as hypersparse, and then converted to standard CSR or CSC format, if appropriate.

Matrix	40-Thread Speedup Relative to 1 Thread					40-Thread Edge Computation Rate (10^6 edges/s)				
	Triangle Counting	4-Truss	BFS	Bellman-Ford	LCC	Triangle Counting	4-Truss	BFS	Bellman-Ford	LCC
datagen-8_9-fb	26.6	27.7	3.5	18.5	25.8	15.3	0.6	285.0	161.6	6.7
datagen-9_2-zf	16.9	19.6	*	*	7.9	63.0	3.2	*	*	12.5
cit-Patents	16.1	19.7	2.6	13.6	11.7	87.8	11.7	25.4	78.8	23.0
g-1073643522-268435456	11.2	16.6	3.6	9.0	8.4	268.3	252.4	10.5	1.1	86.0
graph500-scale25-ef16	30.5	*	3.9	18.9	30.2	1.8	0.1	186.6	220.2	0.6
MAWI/201512020330	5.8	13.4	9.7	2.4	5.7	104.5	86.2	48.7	21.2	23.4

Table 2: Parallel GraphBLAS Computational Results.

4 Problem Descriptions

Using built-in operators, SuiteSparse:GraphBLAS has 1040 unique semirings that can be used in a wide variety of graph algorithms (independent set, breadth-first search, centrality metrics, and so on). MATLAB has just two semirings that it can apply to its sparse matrices: `PLUS_TIMES_FP64` and `PLUS_TIMES_COMPLEX`. Along with the masking operation, this gives GraphBLAS a distinct edge (pun intended) in writing complex graph algorithms. We describe a variety of these graph problems below, to both illustrate the expressive power of GraphBLAS, and to test and demonstrate the performance gains of our parallel OpenMP-based implementation of GraphBLAS relative to the serial implementation. Our sequential GraphBLAS is identical to our OpenMP parallel code with a single thread.

4.1 Triangle Counting A *triangle* in a graph is a clique of size 3. The best matrix formulation we are aware of is by Wolf et al. [16], a variant of Cohen's method [4]. If `A` is a symmetric adjacency matrix of a graph, and `L=tril(A)` denotes the lower triangular part of `A`, then the MATLAB expression `sum(sum(L*L).*L)` computes the number of triangles in the graph. This is expressed as $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{L}^2$ in GraphBLAS notation, followed by a summation of all entries in $\mathbf{C}$ to a scalar. If `L` is stored by row (the default in SuiteSparse:GraphBLAS and also in the work of Wolf et al.) then this can also be computed as $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{L}\mathbf{U}^{\mathrm{T}}$ or `C=(L*U').*L` in MATLAB notation, where `U=triu(A)` is the upper triangular part of `A` (since $\mathbf{A}$ is symmetric, $\mathbf{L} = \mathbf{U}^{\mathrm{T}}$). This latter method is used for these experiments as it is overall the best of the different variants tried in both [5] and [16]. First, `L` and `U` are computed with `GxB_select`. Next `GrB_mxm` computes `C` using a masked dot-product matrix multiply, and `GrB_reduce` sums up `C` for the result.

4.2 4-Truss The k-truss $\mathbf{C}$ of a graph $\mathbf{A}$ is a subgraph with the same number of nodes, but where each edge in the k-truss appears in at least $k-2$ triangles in $\mathbf{A}$. The term c_{ij} is the number of triangles containing the edge (i, j), which is defined as the *support* of the edge (i, j). In the 4-truss, any edges with support less than $k - 2 = 2$ are removed, and thus all edges in the 4-truss are in at least two triangles. The process is iterative; to compute the first support, which is to let $\mathbf{C} = \mathbf{A}$ and then compute $\mathbf{C}\langle\mathbf{C}\rangle = \mathbf{C}^2$. Since $\mathbf{C}$ is symmetric, this can also be computed with $\mathbf{C}\langle\mathbf{C}\rangle = \mathbf{C}\mathbf{C}^{\mathrm{T}}$. This is done with `GrB_mxm`. Next, `GxB_select` is used to drop any edges with edge weight less than the support, $k-2$. The process continues until the graph does not change. Since more than one iteration is typically required, computing the 4-truss takes more time than simply counting the triangles in the graph.

We tested both variants: $\mathbf{C}\langle\mathbf{C}\rangle = \mathbf{C}^2$, which uses the masked Gustavson method, and and $\mathbf{C}\langle\mathbf{C}\rangle = \mathbf{C}\mathbf{C}^{\mathrm{T}}$, which uses the masked dot product. Table 2 reports the results of the fastest method found.

4.3 Breadth-First Search The breadth-first search is a traversal of all nodes of the undirected graph that the sparse matrix represents. The specific algorithm is a direction-optimized push/pull breadth first search, which finds all nodes reachable for a single starting node. [1,17] In our experiment, we used node zero. This reaches most or all the graph, for 5 of the 6 graphs in our tests (for one graph, only a few nodes are reached and we thus do not report the results for this graph). The output vector v contains the level k of each vertex i as it is discovered during the search. The search tree can also optionally be returned, but our results presented here do not compute the tree.

The basic operation of this algorithm computes $\mathbf{A}^{\mathrm{T}}\mathbf{q}$ where $\mathbf{q}$ is the *queue* of nodes in the current level. This can be done with `GrB_vxm(q,A)` $= (\mathbf{q}^{\mathrm{T}}\mathbf{A})^{\mathrm{T}} = \mathbf{A}^{\mathrm{T}}\mathbf{q}$, or by `GrB_mxv(B,q)` $= \mathbf{B}\mathbf{q} = \mathbf{A}^{\mathrm{T}}\mathbf{q}$, where $\mathbf{B} = \mathbf{A}^{\mathrm{T}}$ is the explicit transpose of $\mathbf{A}$. Both steps compute the same thing, just in a different way; the first is a push step and the second is a pull step, assuming

the matrices are in CSR format. In GraphBLAS, unlike MATLAB, a `GrB_Vector` is simultaneously a row and column vector, so $\mathbf{q}$ and $\mathbf{q}^{\mathrm{T}}$ are interchangeable.

4.4 Bellman-Ford Shortest Path For a given source vertex s, the Bellman-Ford algorithm finds the shortest path to all other $n-1$ vertices in the directed weighted graph $\mathbf{A}$, which is assumed to have no self-edges (the adjacency matrix has an explicit zero diagonal). The Bellman-Ford algorithm performs *relaxation* for $n-1$ iterations. Each relaxation updates the approximate distance to each vertex with the minimum of its old value and a newly found distance, represented as $\boldsymbol{d}_v = \min\{\boldsymbol{d}_v, \boldsymbol{d}_u + \boldsymbol{A}_{u,v}\}$, where $\boldsymbol{d}$ is the distance vector of length n and $\boldsymbol{A}$ is the adjacency matrix of the graph [11]. As with our breadth-first search, we used $s = 0$ as the source node which reaches most or all the graph, except in the case of graph datagen-9_2-zf, where insufficient exploration due to multiple connected components required us to exclude it from this experiment.

Similar to breadth-first search, the traversal is done with `GrB_vxm(d,A)`, or by `GrB_mxv(B,d)`, where $\mathbf{B} = \mathbf{A}^{\mathrm{T}}$ is the explicit transpose of $\mathbf{A}$. Previous test results indicate that `GrB_vxm(d,A)` is faster than `GrB_mxv(B,d)` when d is sparse, while `GrB_mxv(B,d)` is faster when d is dense. More specifically, `GrB_mxv(B,d)` tends to be faster when the saxpy-based Gustavson method is used for `GrB_vxm(d,A)`. Furthermore, when the sparse vector d becomes too dense, the performance of both `GrB_vxm(d,A)` and `GrB_mxv(B,d)` can be improved by treating d as a dense vector. Therefore, to accelerate the Bellman-Ford algorithm, the vector d is initially sparse and converted to dense when $nnz(d) > n/2$ if only $\mathbf{A}$ is available or when $nnz(d) > \sqrt{n}$ if only $\mathbf{B}$ is available. Otherwise, when both $\mathbf{A}$ and $\mathbf{B}$ are available, `GrB_vxm(d,A)` is used by default, and replaced by `GrB_mxv(B,d)` when the saxpy-based Gustavson method is used.

4.5 Local Clustering Coefficient For a given vertex v, LCC(v) can be determined as the number of triangles containing v divided by the number of possible edges between the neighbors of v. The algorithm uses the adjacency matrix $\mathbf{A}$ and a Boolean matrix $\mathbf{C}$ describing neighborhood relations, which considers vertices on both incoming and outgoing edges. If the adjacency matrix $\mathbf{A}$ is symmetric, $\mathbf{C} = \mathbf{A}$, else $\mathbf{C} = \mathbf{A} \vee \mathbf{A}^{\mathrm{T}}$.

The triangles are enumerated using $\mathbf{T}\langle\mathbf{C}\rangle = \mathbf{CA}$ or `T=(C*A).*C` in MATLAB notation. This is computed with `GrB_mxm` using a masked dot-product matrix multiply. The number of triangles for each vertex are summed using a row-wise `GrB_reduce` operation on $\mathbf{T}$ with the operator `GrB_PLUS_FP64`, resulting in vector $\mathbf{t}$.

```
#include "GraphBLAS.h"
void ymax_fp32 (float *z, const float *x)
{
    (*z) = fminf ((*x), (float) 32.0) ;
}
void dnn        // solve a sparse deep neural network
(
    GrB_Matrix *Yhandle, // Y, created on output
    GrB_Matrix *W,       // W [0..nlayers-1]
    GrB_Matrix *Bias,    // Bias [0..nlayers-1]
    int nlayers,         // # of layers
    GrB_Matrix Y0        // nfeatures-by-nneurons
)
{
    GrB_Matrix Y = NULL ;
    GrB_UnaryOp ymax = NULL ;
    GrB_Index nfeatures, nneurons ;
    GrB_Matrix_nrows (&nfeatures, Y0) ;
    GrB_Matrix_ncols (&nneurons,  Y0) ;
    GrB_Matrix_new (&Y, type, nfeatures, nneurons) ;
    GrB_UnaryOp_new (&ymax, ymax_fp32, GrB_FP32, GrB_FP32) ;
    // propagate the features through the neuron layers
    for (int layer = 0 ; layer < nlayers ; layer++)
    {
        // Y = Y * W [layer]
        GrB_mxm (Y, NULL, NULL, GxB_PLUS_TIMES_FP32,
            ((layer == 0) ? Y0 : Y), W [layer], NULL) ;
        // Y(i,j) += Bias [layer] (j,j) for each Y(i,j)
        GrB_mxm (Y, NULL, NULL, GxB_PLUS_PLUS_FP32,
            Y, Bias [layer], NULL) ;
        // delete entries; keep only those > 0
        GxB_select (Y, NULL, NULL, GxB_GT_ZERO, Y,
            NULL, NULL) ;
        // threshold maximum values: Y (Y > 32) = 32
        GrB_apply (Y, NULL, NULL, ymax, Y, NULL) ;
    }
    GrB_free (&ymax) ;  // free the unary operator
    (*Yhandle) = Y ;    // return result
}
```

Figure 1: A complete solution to the Sparse Deep Neural Network in GraphBLAS.

To get the number of neighbors for each vertex, the next step computes vector $\mathbf{w}$ from matrix $\mathbf{C}$, again using a row-wise `GrB_reduce` with `GrB_PLUS_FP64`. The maximum possible number of edges between w neighbors is $w \cdot (w-1)$, which is captured as a `GrB_UnaryOp` operator and is applied on $\mathbf{w}$ using `GrB_apply`, resulting in vector $\mathbf{p}$. Finally, $\mathbf{lcc} = \mathbf{t}/\mathbf{p}$ gives the LCC values.

4.6 Benchmark Matrices To demonstrate GraphBLAS on the aforementioned algorithms, we chose six large matrices as input (see Table 3). The first two matrices are generated from the LDBC Datagen component configured to produce undirected simple graphs for LDBC Graphalytics [9], while the latter four are from the HPEC '18 Graph Challenge (`https://graphchallenge.mit.edu/`).

4.7 Sparse Deep Neural Network Forward Propagation The sparse deep neural network problem involves computing an output vector (Y_n) based on an input vector (Y_0) and a deep neural network

Matrix Name	Nodes	Edges	Description
datagen-8_9-fb	10,572,901	848,681,908	LDBC Datagen
datagen-9_2-zf	434,943,376	1,042,340,732	LDBC Datagen
cit-Patents	3,774,768	16,518,947	Citation network among US patents
g-1073643522-268435456	268,435,456	1,073,643,522	Synthetic image grid benchmark matrix
graph500-scale25-ef16	17,043,780	523,467,448	Synthetic graph500 network of scale 25
MAWI/201512020330	226,196,185	240,023,949	Packet trace from WIDE internet backbone

Table 3: The six matrices used for all benchmarks except sparse deep neural network.

consisting of n layers of a fixed number of neurons in each layer. The connections between each layer of neurons are sparse, where the connections have zero weight [12, 13, 15]. The values of each layer are computed using a rectified linear unit (ReLU) computation. The Sparse Deep Neural Network Graph Challenge describes several instances of the sparse deep neural network problem, which we solved using GraphBLAS in a prior paper, [10].

Figure 1 presents the entire algorithm in GraphBLAS. A prototype of the code, without the `ymax` step, took only 20 minutes to write, and compiled and ran correctly the first time it was compiled. It is almost as elegant as the MATLAB reference implementation; even more so for the step that applies the bias. Applying the bias adds a negative value to each nonzero column of `Y`, leaving zeros unchanged. In the MATLAB reference, the line is `Y = Z + (double(logical(Z)) .* bias{layer})`. In GraphBLAS, this is a simple diagonal scaling, `Y=Y*Bias[layer]`, using the `PLUS_PLUS` semiring.

On the same system used for the other experiments reported in this paper, the smallest problem took 179 seconds in MATLAB R2018a. Using a single thread, GraphBLAS took 21 seconds, and 1.5 seconds with 40 threads, a speedup of over 100x. The largest problem required exactly 3 days for MATLAB, but only a little over an hour for GraphBLAS, with 40 threads, a speedup of 70x. This gain in performance came at no extra cost in development time, since writing the code in Figure 1 is just as fast as writing the corresponding MATLAB code.

5 Performance results

Experiments were performed on an Intel® Xeon® E5-2698 v4 CPU system with 20 cores running at 2.2 Ghz, 256 GB of RAM and the Intel® `icc` compiler (19.0.3.199). Two-way simultaneous multithreading (SMT) was enabled so the system had a total of 40 hardware-threads.

Computational results for solving the various problems are tabulated in Table 2. GraphBLAS uses the OpenMP-based parallelism described in Section 3.

This parallel implementation of GraphBLAS generally scales well, but performance is dependent on the specific algorithm and matrix.

Results for matrix datagen-9_2-zf are excluded for breadth-first search and the Bellman-Ford algorithm as it has many disconnected components (very little of the graph is explored). 4-truss resulted in a timeout for the graph500-scale25-ef16 matrix when using a single thread, so speedup results are not reported. The edge computation rate for 40 threads is low for this particular matrix; we are studying our algorithms to determine why this is the case.

For the sparse deep neural network forward propagation challenge, GraphBLAS provided speedups ranging from 10.1 to 15.3 using 40 threads relative to 1 thread, depending on the size and structure of the networks. The edge computation rate ranged from 60.1 million to 275.4 million edges per second when using 40 threads [7]. The bulk of the work for this algorithm is spent in a parallel `GrB_mxm`.

The parallel speedups and raw performance of triangle counting and k-truss are mostly very good, except for the graph500-scale25-ef16 matrix. Both of these algorithms rely on a parallel matrix-matrix multiply (`GrB_mxm`). Breadth-first search shows modest parallelism; it relies on a matrix-vector or vector-matrix multiply, which is harder to parallelize. The LCC algorithm also relies on a highly-parallel `GrB_mxm`, but it does other work as well so its parallelism is not always as good as the simpler triangle counting or k-truss methods. We are developing a revision of this method for which we hope to obtain better parallelism.

6 Conclusions

These results demonstrate that GraphBLAS can be an efficient library that allows end users to write simple yet fast code. Most of the algorithms are fully scalable. We plan to include a hash-based matrix-multiply [14] to improve scalabilty, by reducing our reliance on the Gustvason-based method. All codes used in this paper appear at `http://suitesparse.com` or `https://github.com/GraphBLAS/LAGraph`.

References

[1] S. Beamer, K. Asanović, and D. Patterson, *Direction-optimizing breadth-first search*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, Los Alamitos, CA, USA, 2012, IEEE Computer Society Press, pp. 12:1–12:10, http://dl.acm.org/citation.cfm?id=2388996.2389013.

[2] A. Buluç and J. R. Gilbert, *On the representation and multiplication of hypersparse matrices*, in IPDPS08: the IEEE International Symposium on Parallel & Distributed Processing, IEEE Computer Society, 2008, pp. 1–11.

[3] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, *The GraphBLAS C API specification*, tech. report, http://graphblas.org/, 2017.

[4] J. Cohen, *Graph twiddling in a map-reduce world*, Computing in Science and Eng., 11 (2009), pp. 29–41.

[5] T. A. Davis, *Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and k-truss*, in 2018 IEEE High Performance extreme Computing Conference (HPEC), Sep. 2018, pp. 1–6, https://doi.org/10.1109/HPEC.2018.8547538.

[6] T. A. Davis, *Algorithm 1000: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra*, ACM Trans. Math. Softw., to appear (2019).

[7] T. A. Davis, M. Aznaveh, and S. Kolodziej, *Write quick, run fast: Sparse deep neural network in 20 minutes of development time via SuiteSparse:GraphBLAS*, in 2019 IEEE High Performance extreme Computing Conference (HPEC), 2019 (submitted), pp. –.

[8] F. G. Gustavson, *Two fast algorithms for sparse matrices: Multiplication and permuted transposition*, ACM Trans. Math. Softw., 4 (1978), pp. 250–269, https://doi.org/10.1145/355791.355796.

[9] A. Iosup et al., *LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms*, VLDB, 9 (2016), pp. 1317–1328.

[10] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, *Sparse deep neural network graph challenge*, tech. report, MIT Lincoln Laboratory Supercomputing Center, 2019. http://graphchallenge.mit.edu/sites/default/files/documents/SparseDNN-GraphChallenge-2019-06-13-DRAFT.pdf.

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

[12] X. Liu, J. Pool, S. Han, and W. J. Dally, *Efficient sparse-winograd convolutional neural networks*, arXiv preprint arXiv:1802.06367, (2018).

[13] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, *Exploring the regularity of sparse structure in convolutional neural networks*, arXiv preprint arXiv:1705.08922, (2017).

[14] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, *High-performance sparse matrix-matrix products on Intel KNL and multicore architectures*, CoRR, Distributed, Parallel, and Cluster Computing In 47th International Conference on Parallel Processing Workshops (ICPPW), 2018, abs/1804.01698 (2018), http://arxiv.org/abs/1804.01698, https://arxiv.org/abs/1804.01698.

[15] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, *SCNN: An accelerator for compressed-sparse convolutional neural networks*, in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2017, pp. 27–40.

[16] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, *Fast linear algebra-based triangle counting with KokkosKernels*, in 2017 IEEE High Performance Extreme Computing Conference (HPEC), Sept 2017, pp. 1–7, https://doi.org/10.1109/HPEC.2017.8091043.

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