GraphBLAS Programmability: Python and MATLAB Interfaces

1st Timothy G. Mattson
Parallel Computing Lab
Intel Corp.
Ilwaco, WA, USA
timothy.g.mattson@intel.com

2nd Michel Pelletier
CEO
Graphegon, Inc.
Portland, OR, USA
pelletier.michel@gmail.com

3rd Timothy A. Davis
Computer Science and Engineering
Texas A&M University
College Station, TX, USA
davis@tamu.edu

***Abstract*—A graph can be represented by a sparse array. In that representation, the basic operations used to construct graph algorithms are linear algebra operations over algebraic semirings. The GraphBLAS forum was established in 2013 to define a set of Basic Linear Algebra Subprograms for Graph Algorithms. They started with a mathematical definition of the GraphBLAS followed by a binding to the C Programming language. The result was the GraphBLAS C specification currently at version 1.3.1. A robust implementation of the GraphBLAS C specification is available as the SuiteSparse GraphBLAS library.**

The primary motivation of the GraphBLAS was to provide a high performance foundation for graph algorithms. Many members of the GraphBLAS forum were also motivated by the simplicity of graph algorithms expressed in terms of linear algebra. This simplicity, however, is difficult to fully appreciate when the programming interface is the C programming language. To see the full expressive power of the GraphBLAS, a high level interface is needed so the elegance of the mathematical underpinnings of the GraphBLAS is apparent.

In this paper we present python (pygraphblas) and MATLAB (`@GrB`) interfaces to the SuiteSparse implementation of the GraphBLAS. We use a subset of the GAP benchmark suite to demonstrate the simplicity of these GraphBLAS interfaces and show that in most cases, little performance is sacrificed by moving to the more productive Python and MATLAB interfaces.

***Index Terms*—Graph Algorithms, Linear Algebra, GraphBLAS**

I. Introduction

The GraphBLAS have come a long way since the effort started in 2013 with a manifesto [1] calling for standard building blocks for graph algorithms expressed in the language of linear algebra. The GraphBLAS mathematics [2] have been formally defined, a binding of the math to the C programming language [3] was released, and a high quality implementation of the C GraphBLAS specification exists with the SuiteSparse GraphBLAS library [4] . The group continues to enhance the GraphBLAS specification and is working to create a library of high level algorithms on top of the GraphBLAS (LAGraph).

Much of the focus in the GraphBLAS community has been on defining core specifications and delivering high performance. It is not surprising that so much attention is on performance. We can measure run times and compare against a cost model for a system. Performance can be quantified, hence it is all too often how we compare different systems for expressing graph algorithms.

To most programmers, however, performance just needs to be "good enough". As long as an algorithm scales to handle interesting graphs and runs in a reasonable amount of time, performance is not the most important criteria. Time-to-solution is far more important and for all but the largest problems, the time needed to solve a problem is dominated by the time to write and debug the software. In other words, as long as the performance is reasonably good, the most important feature of a framework for expressing graph algorithms is *programmability*.

Software researchers don't often discuss programmability. It is hard to quantify and therefore difficult to turn into simple numbers that can be compared between alternative systems. We submit, however, that programmability is vitally important and a feature of graph algorithm systems that deserves much more attention.

In this paper, we will explore the programmability of the GraphBLAS. It is hard to see the benefits of the GraphBLAS as an abstraction for writing high quality software when it is approached through the C programming language. If expressed in terms of productivity languages such as Python or MATLAB, the programmability benefits of the GraphBLAS become clear. In particular, the mathematics of an algorithm are clearly apparent in the source code. The source code is sparse without the clutter required of the C GraphBLAS API. Finally, the code is easy to read and comprehend.

In this paper, we explore Python and MATLAB interfaces to the SuiteSparse GraphBLAS library. We use a subset of the GAP benchmark suite to measure the abstraction overhead of the high level interfaces relative to calling the C code directly. We show the code for both interfaces and invite the reader to notice how clear the underlying algorithm is in the code. This is the major contribution of this paper. Not the performance numbers or additional benchmark results, but the conversation about frameworks for expressing graph algorithms and programmability. We believe we should all spend more time looking at code and less time talking about run times for different programs.

With support from NSF 1514406 and 1835499, Intel, NVIDIA, MIT Lincoln Lab, Redis Labs, and IBM.

II. The GraphBLAS

Consider a graph with n vertices. We can represent this graph as an n-by-n adjacency matrix $\mathbf{A}$. The rows and columns of $\mathbf{A}$ correspond to the vertices while the non zero elements of $\mathbf{A}$ represent the edges with a_{ij} as the weight of the edge from vertex i to vertex j. Most graphs are sparsely connected, hence the adjacency matrix for a graph is usually sparse.

Consider a second k-by-n matrix $\mathbf{B}$. We can use this matrix to select a subset of k vertices from the graph. The elements of $\mathbf{B}$ are 0 accept for those selecting a vertex with b_{ji} equal to 1 to select the ith vertex as the jth element of the vertex subset. The familiar matrix product over real arithmetic $\mathbf{B} \times \mathbf{A}$ returns the cost (using the edge weights from $\mathbf{A}$) of reaching the set of vertices adjacent to the vertices selected by $\mathbf{B}$. This fundamental operation can be used to construct a wide range of graph algorithms.

The expressive power of the GraphBLAS is greatly enhanced through the use of algebraic semirings. The pattern of operations from the basic linear algebra operations are used, but the operators and interpretation of values in matrices and vectors (the *domain*) can be changed using a semiring.

In a semiring, the add and multiply operators in conventional matrix multiplication are replaced with an additive monoid (an associative and commutative operator with an identity value) and a multiplicative operator. The conventional semiring is PLUS-TIMES. Determining the next level of nodes in a breadth-first search (BFS) can be written as a matrix-vector multiplication using the Boolean semiring (AND-OR), or if the least-numbered parent of a node is needed, the MIN-FIRST semiring can be used, where the FIRST operator is $f(x, y) = x$. The FIRST operator is handy as a multiplicative operator when computing the vector-matrix multiply $\mathbf{y} = \mathbf{x}'\mathbf{A}$, when the sparse adjacency matrix $\mathbf{A}$ is unweighted (or has weights that should be ignored).

TABLE I: GraphBLAS Operations

function name	description	GraphBLAS notation
`GrB_mxm`	matrix-matrix mult.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix mult.	$\mathbf{w}'\langle\mathbf{m}'\rangle = \mathbf{w}' \odot \mathbf{u}'\mathbf{A}$
`GrB_mxv`	matrix-vector mult.	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise, set-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	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i}, \mathbf{j}) = \mathbf{C}(\mathbf{i}, \mathbf{j}) \odot \mathbf{A}$ $\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$ $\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
`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	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$
`GrB_kronecker`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{kron}(\mathbf{A}, \mathbf{B})$

Table I lists all GraphBLAS operations, 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 $\odot$ is a binary accumulator operator.

The *mask* is a fundamental part of GraphBLAS, written as $\mathbf{C}\langle\mathbf{M}\rangle = ...$, allowing only selected parts of $\mathbf{C}$ to be updated. For example, when traversing the nodes in a BFS, nodes that have already been visited should be excluded. If $\mathbf{q}$ is a sparse Boolean vector holding the set of nodes in the current level, then all the nodes adjacent to any node in $\mathbf{q}$ is given by $\mathbf{q}'\mathbf{A}$. To exclude nodes already seen, a mask can be used. Suppose $\mathbf{v}$ is a vector of size n where $v_i = 0$ if node i has not been visited. If the mask m_i is true, then this means the ith entry of the result can be modified. In this case, the vector $\mathbf{v}$ can be used as a mask, but it must be complemented, as $\mathbf{q}'\langle\neg\mathbf{v}\rangle = \mathbf{q}'\mathbf{A}$.

III. GAP Benchmark Suite

To understand the performance of graph algorithm frameworks, a representative subset of algorithms over multiple graphs should be considered. In this paper, we use the GAP benchmark suite [5]. This benchmark suite uses five input graphs selected for topological diversity and five graph kernels that cover the more common use cases from the literature. The GAP benchmark suite also includes well-optimized reference codes multithreaded with OpenMP to provide a high-performance baseline for assessing performance.

A common result from the study of graph algorithms is that the graph processing is data-driven. The topology of a graph can have a greater impact on the performance of a workload than the algorithm. Hence, the GAP benchmark suite uses the five input graphs listed in Table II. These graphs represent both real-world data (Road, Twitter, Web) and synthetic generators (Kron & Urand).

For the five kernels used in the GAP benchmark, specific algorithms are not specified. The solutions for each kernel, however, are specified with enough clarity to avoid ambiguity. The kernels contain an interesting mix of traits, and are sufficiently scalable to run on large graphs. In this paper, we use two of the five kernels: PageRank (PR) and Betweenness Centrality (BC).

A. Betweenness-centrality

The *node-betweenness-centrality* of a node i in a graph is a weighted sum of the number of shortest paths that pass through node v. If σ_{st} is the number of unique shortest paths from node s to node t, and $\sigma_{st}(i)$ is the number of those paths that go through node i, then the betweeness centrality of node i is $\sum_{s,t \neq i}(\sigma_{st}(i)/\sigma_{st})$. It is costly to compute for all pairs s and t, so practical methods rely on sampling, where a small number of source nodes s are selected at random.

The Brandes algorithm [12] for computing the node betweenness-centrality starts with a set of source nodes, and performs a forward breadth-first search pass to find and count the shortest paths through all nodes. A backward pass from the last level back to the source nodes then computes the centrality of each node.

B. PageRank

The *PageRank* kernel (PR) computes a measure of the importance (rank) of each vertex in a graph based on the

Name	Description	# Vertices (M)	# Edges (M)	Directed	Degree	Degree Distribution	Approx. Diameter	References
Urand	Uniform Random Graph	134.2	2,147.5	N	16.0	normal	7	[6]
Kron	Kronecker Synthetic Graph	134.2	2,111.6	N	15.7	power	6	[7], [8]
Twitter	Twitter Follow Links	61.6	1,468.4	Y	23.8	power	14	[9]
Web	Web Crawl of .sk Domain	50.6	1,930.3	Y	38.1	power	135	[10]
Road	Roads of USA	23.9	57.7	Y	2.4	bounded	6,304	[11]

TABLE II: Graphs used for evaluation

rank of the nodes connected to the vertex. For a graph with n vertices, the PageRank algorithm is iterative. It starts with the PageRank vector r of length n initialized to a small positive number (usually $1/n$) and then updates the vector according to the recurrence relation:

$$r_k(v) = (1-d)/n + d\sum_{u \neq v}(r_{k-1}(u)/d_{out}(u))$$

where v and u are vertices, d is a damping factor set equal to 0.85 and d_{out} is a vector for the out degree for each vertex. This is an iterative algorithm. The GAP benchmark suite requires an implementation to iterate until the sum of the absolute values of the differences of two successive iterations is less than 10^{-4}.

IV. PyGraphBLAS

The Python programming language emphasizes simplicity and readability, which aligns well with the high level usability goals of the GraphBLAS. Like MATLAB, pygraphblas code tends to look very similar to the linear algebraic description of an algorithm, and they both share a spirit of clarity and fast experimentation in exchange for some runtime overhead, although this overhead is demonstrated to be negligible for large graphs.

Python has evolved heavily into a data analysis and manipulation language, where users express high level concepts with high level tools like numpy and pandas, but still have the performance gained with low level optimizations applied. Much like numpy uses BLAS to accomplish this goal, pygraphblas uses the GraphBLAS library to do the actual work. Python's job is merely to schedule the work in a clear way. By standing on the shoulders of performance optimizers who refine these core libraries, we can see even further with less effort.

A key difference between the Python and MATLAB implementations is Python allows functions to modify their input parameters, so this gives Python an edge as shown in the benchmark results runtimes for Betweenness Centrality on the road data set. For the most part, Python benchmarking times were equal to the LAGraph C runtimes within the bounds of error due to timing jitter, but some MATLAB runtimes were slower due to the nature of this limitation in MATLAB.

Another important feature currently available only to Python and C are user-defined types. These are custom matrix element (graph node) types that have custom semiring operations. pygraphblas uses the numba JIT compiler to compile Python functions into user-defined type operations. An example of using this feature can be found in the pygraphblas demo source as a Jupyter Notebook [13].

Figures 1 and 2 describe the the PageRank and Betweenness Centrality algorithms from the GAP benchmark used to generate the performance results presented in Section VI.

V. MATLAB Interface to GraphBLAS

SuiteSparse:GraphBLAS [4] includes a MATLAB interface, in which the GraphBLAS matrix becomes a `@GrB` MATLAB object. All operators and many built-in functions have been overloaded. If `A` and `B` are MATLAB and/or GraphBLAS sparse matrices, then the MATLAB statement `C=A*B` is matrix multiplication over the standard semiring. A GraphBLAS sparse matrix can have many more types than a MATLAB sparse matrix; MATLAB has `double`, `logical`, and `double complex`; GraphBLAS adds 8 kinds of sparse integer matrices (signed and unsigned, in 8, 16, 32, and 64 bits), and `single complex`. The C API of GraphBLAS allows for arbitrary user-defined types, but this feature is not available through the MATLAB interface.

In the MATLAB interface, a semiring is specified as a string of the form `add.mult.type`, where `add` is the additive monoid, `mult` is the multiplicative operator, and `type` is the type of the inputs x and y for the multiplicative operator. In this notation, the expression `C=A*B` uses the conventional semirings: `+.*.double` for real matrices, or `+.*.double complex` for complex ones. GraphBLAS allows for thousands of semirings, combining many monoids (`'+'`, `'*'`, `'min'`, `'max'`, and `'any'` for most types; logical monoids `'|'`, `'&'`, `'xor'`, `'xnor'` for the MATLAB `'logical'` type, bitwise monoids `'bitand'`, `'bitor'`, `'bitxor'`, `'bitxnor'` for integers), with dozens of binary multiplicative operators (`'1st'`, `'2nd'`, `'+'`, `'*'`, `'/'`, and so on).

A. PageRank in GraphBLAS+MATLAB

At each iteration, the `pagerank` method computes an update to the current rank vector `r`, with the PLUS-SECOND semiring (`'+.2nd'` in MATLAB). The SECOND function is $f(x, y) = y$, and it allows the edge weights of $\mathbf{A}$ to be ignored when computing $\mathtt{r} += \mathbf{A}'\mathtt{t}$ (line 20, right side of Figure 1). It performs its computations in 32-bit single precision, which MATLAB cannot do with its sparse matrices, but which GraphBLAS supports.

B. Betweenness centrality in GraphBLAS+MATLAB

The MATLAB method for computing the centrality is illustrated on the right side of Figure 2. The forward pass is a bulk BFS (with `ns` source nodes), so the `frontier` matrix is `ns`-by-`n`. The frontier is updated much like the BFS, but with a different semiring. The additive monoid is PLUS, not

```python
def pagerank(A, d, damp=0.85, itermax=100):
    # A: input graph, d: vector of out-degree for A
    # damp: damping factor, itermax: max iterations
    n = A.nrows
    # initial PageRank: all nodes have rank 1/n
    r = Vector.dense(FP32, n, 1.0 / n)
    t = Vector.sparse(FP32, n)
    # scale the out-degree vector by the damping factor
    d.assign_scalar(damp, accum=FP32.DIV)
    tfactor = (1 - damp) / n
    for i in range(itermax):
        # swap t and r every iteration
        temp = t ; t = r ; r = temp
        w = t / d
        r[:] = tfactor
        # r += A' * (t./d)
        A.mxv(w, out=r, accum=FP32.PLUS,
              semiring=FP32.PLUS_SECOND, desc=TransposeA)
        t -= r
        # halt if norm (r-t,1) < 1e-4
        t.apply(FP32.ABS, out=t)
        rdiff = t.reduce_float()
        if rdiff < 1e-4:
            break
    return r
```

```
function r = pagerank (A, d, damp, itermax)
% r = PageRank of graph A, where A is n-by-n,
% d: vector of out degrees of A.
% damp: damping factor, itermax: max iterations
if (nargin < 3) damp = 0.85 ; end
if (nargin < 4) itermax = 100 ; end
n = size (A, 1) ;
% initial PageRank: all nodes have rank 1/n
r = GrB.ones (n, 1, 'single') / n ;
% scale the out-degree vector by the damping factor
d = d / damp ;
% teleport factor:
tfactor = cast ((1 - damp) / n, 'single') ;
desc.in0 = 'transpose' ;
for iter = 1:itermax
    t = r ;
    % r (:) = tfactor ;
    r = GrB.expand (tfactor, r) ;
    % r += A' * (t./d)
    r = GrB.mxm (r, '+', A, '+.2nd', t./d, desc) ;
    % halt if norm (r-t,1) < 1e-4
    if (GrB.normdiff (r, t, 1) < 1e-4)
        break ;
    end
end
```

Fig. 1: PageRank in Python and MATLAB

MIN or OR, so that the number of paths can be counted (see lines 28 and 44) with the `+.1st` semiring.

The parameter `drc` is a GraphBLAS *descriptor* that modifies the operation. In this case, `drc` states that the mask is complemented, and that the output (`frontier`) should be cleared of all entries after it is used in the matrix multiply, but before it is written to by the assignment.

Another example of a GraphBLAS operation is element-wise "multiplication" (a Hadamard product). This is `A.*B` in MATLAB notation, except that in GraphBLAS, any binary operator can be used. The result is the set intersection of the entries in the two operands. This is used with an accumulator (`'+'`), on line 46, to compute `bc+=W.*paths`.

C. Limitations of GraphBLAS+MATLAB

With one exception (`subsasgn`, discussed below), user-written MATLAB m-files cannot modify their inputs. However, many GraphBLAS operations perform small incremental modifications to their input/output matrix (the matrix $\mathbf{C}$ or vector $\mathbf{w}$ in Table I). This is particularly true if a very sparse mask $\mathbf{M}$ is used. With $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$, most of $\mathbf{C}$ does not change, and it can be modified in place in many use cases of the C API for GraphBLAS. In MATLAB, we currently do:

```
Cout = GrB.mxm (Cin, M, accum, semiring, A, B, desc) ;
```

In this case, the input matrix `Cin` is not incrementally modified to produce the output matrix `Cout`. Instead, an entirely new output matrix `Cout` is constructed. As will be seen in Section VI, this can have serious performance issues when the updates are very sparse in the 4-source BFS in betweenness-centrality for high-diameter graphs (the Road graph in particular, as seen in Table III below).

A similar case occurs when computing `C(I,J)=A`. With GraphBLAS matrices, this can be orders of magnitude faster than the MATLAB built-in expression, particularly when the matrix `A` is large, but it could be faster still. In its C API, GraphBLAS provides a non-blocking mode where the updates to `C` are left pending, and done in bulk later on, but this cannot be exploited via its MATLAB interface.

Starting with `A=spalloc(n,n,e)` (an empty MATLAB sparse matrix with space for `e` entries), or `A=GrB(n,n)` (an empty GraphBLAS sparse matrix) constructing a matrix with e entries one entry at a time takes $O(e^2)$ time, if the entries are not provided in column-major order. Both the MATLAB documentation and MATLAB Code Analyzer recommend against the following [14]:

```
A = GrB (n,n) ; % or spalloc(n,n,e) or sparse(n,n)
for k = 1:e
    i = some row index
    j = some column index
    x = some scalar value
    A (i,j) = x ;
end
```

GraphBLAS provides an optimal solution to this issue, in its C API, where the following takes only $O(e \log e)$ time if the entries are in arbitrary order, or $O(e)$ time if the entries are inserted in row- or column-major order:

```
GrB_Matrix_new (&A, GrB_FP64, n, n) ;
for (int64_t k = 0 ; k < e ; k++)
{
    GrB_Index i = some scalar value
    GrB_Index j = some scalar value
    double x = some scalar value
    GrB_Matrix_setElement (A, x, i, j) ;
}
```

```python
def betweenness(sources, A, AT):
    # input: set of source nodes; matrix A and transpose AT.
    # output is n-by-1: centrality(i) = score for node i.
    # initialize frontier and path count matrix (paths):
    n = A.nrows
    ns = len(sources)
    paths      = Matrix.dense (FP32, ns, n, 0)
    frontier   = Matrix.sparse(FP32, ns, n)
    for i, s in enumerate(sources):
        paths[   i, sources[i]] = 1
        frontier[i, sources[i]] = 1



    # S[k] is the pattern of the frontier at level k:
    S = []
    # first frontier:  frontier<!paths> = frontier*A
    frontier.mxm(A, out=frontier, mask=paths,
        semiring=FP32.PLUS_FIRST, desc=descriptor.oocr)

    # breadth-first search stage:
    for depth in range(n):
        # S[depth] = pattern of frontier, as a bool matrix
        s = Matrix.sparse(BOOL, ns, n)
        frontier.apply(BOOL.ONE, out=s)
        S.append(s)
        # accumulate path counts: paths += frontier
        paths.assign_matrix(frontier, accum=FP32.PLUS)
        # update frontier: frontier<!paths> = frontier*A
        frontier.mxm(A, out=frontier, mask=paths,
            semiring=FP32.PLUS_FIRST, desc=descriptor.oocr)
        if frontier.nvals == 0:
            break

    # betweenness centrality computation phase:
    bc = Matrix.dense(FP32, ns, n, 1)
    W = Matrix.sparse(FP32, ns, n)
    for i in range(depth - 1, 0, -1):
        # update from successors, mask with kth frontier:
        # W<S[k]> = bc ./ path
        bc.emult(paths, FP32.DIV, out=W,
                 mask=S[i], desc=Replace)
        # W<S[k-1]> = W*A'
        W.mxm(AT, out=W, mask=S[i-1],
              semiring=FP32.PLUS_FIRST, desc=Replace)
        # bc += W .* paths
        W.emult(paths, FP32.TIMES, out=bc, accum=FP32.PLUS)

    # centrality (i) = sum (bc(:,i)) - ns, for all nodes i:
    centrality = Vector.dense (FP32, n, -ns)
    bc.reduce_vector(accum=FP32.PLUS, out=centrality,
                     desc=TransposeA)
    return centrality
```

```python
function centrality = betweenness (sources, A, AT)
% input: set of source nodes; matrix A and transpose AT.
% output is n-by-1: centrality(i) = score for node i.
% initialize frontier and path count matrix (paths):
n = size (A,1) ;
ns = length (sources) ;
paths = GrB (ns, n, 'single', 'by row') ; paths (:,:) = 0 ;
frontier = GrB (ns, n, 'single', 'by row') ;
for i = 1:ns
    paths (i, sources (i)) = 1 ;
    frontier (i, sources (i)) = 1 ;
end
drc = struct ('out', 'replace', 'mask', 'complement') ;
drs = struct ('out', 'replace', 'mask', 'structural') ;
% S{k} is the pattern of the frontier at level k:
S = cell (1, n) ;
% first frontier:  frontier<!paths> = frontier*A
frontier = GrB.mxm (frontier, paths, '+.1st', ...
    frontier, A, drc) ;

% breadth-first search stage:
for depth = 1:n
    % S {depth} = pattern of frontier, as a logical matrix
    S {depth} = spones (frontier, 'logical') ;
    % accumulate path counts: paths += frontier
    paths = GrB.assign (paths, '+', frontier) ;
    % update frontier: frontier<!paths> = frontier*A
    frontier = GrB.mxm (frontier, paths, '+.1st', ...
        frontier, A, drc) ;
    if (GrB.entries (frontier) == 0)
        break ;
    end
end

% betweenness centrality computation phase:
bc = GrB (ns, n, 'single', 'by row') ;
bc (:,:) = 1 ;
W = GrB (ns, n, 'single', 'by row') ;
for k = depth:-1:2
    % update from successors, mask with kth frontier:
    % W<S{k}> = bc ./ path
    W = GrB.emult (W, S{k}, '/', bc, paths, drs) ;
    % W<S{k-1}> = W*A'
    W = GrB.mxm (W, S{k-1}, '+.1st', W, AT, drs) ;
    % bc += W .* paths
    bc = GrB.emult (bc, '+', W, '*', paths) ;
end

% centrality (i) = sum (bc(:,i)) - ns, for all nodes i:
centrality = GrB (n, 1, 'single', 'by col') ;
centrality (:) = -ns ;
centrality = GrB.vreduce (centrality, '+', '+', bc,
    struct ('in0', 'transpose') ;
```

Fig. 2: Betweenness-Centrality in Python and MATLAB

The updates are placed in an unordered list and `A` is constructed once the work is done, via the GraphBLAS non-blocking mode. However, in MATLAB, the resulting matrix `A` cannot be modified if passed as an input to another GraphBLAS m-file in MATLAB, and thus `A` must be finalized after each assignment. This leads to $O(e^2)$ time if this construction is performed in MATLAB as in the first example above, with `A(i,j)=x`. The MATLAB method takes $O(e)$ time in only one case, where the entries are inserted in column-major order in a MATLAB sparse matrix `A` initialized with `A=spalloc(n,n,e)`.

The inability to fully exploit non-blocking mode is currently the most critical limiting factor in using GraphBLAS via MATLAB.

There is one strategy that we are exploring that will mitigate against this issue, in one important case: `C(I,J)=A` or `C(M)=A` when `C` is dense and `I` and `J` are very small or `M` is very sparse.

Our MATLAB interface in SuiteSparse:GraphBLAS overloads the `C=subsasgn(C,S,A)` method, so that the syntax `C(I,J)=A` and `C(M)=A` can be used for `@GrB` objects in MATLAB (the input `S` is a cell array containing the index vectors `I` and `J`, or the mask matrix `M`). The `subsasgn` method is the single exception in MATLAB where `C` can be modified in-place, but we have not yet been able to exploit this feature.

The difficulty is that the `subsasgn` m-file must first extract the contents of the `C` object, pass it to a C `mexFunction` interface to the C GraphBLAS library, which must then modify the matrix `C` in place, and the resulting content must be inserted back into the `@GrB` object `C`. This should be feasible, but we have not yet been able to exploit this technique in our MATLAB interface to GraphBLAS. If it can be done, it would drastically reduce the time to compute the betweenness centrality on the Road graph.

Even if this technique is exploited, however, it will not address the repeated insertion of individual entries in arbitrary order via the syntax `A(i,j)=x`. That approach will always be slower in the MATLAB interface to GraphBLAS when the entries are inserted out of order, in contrast to `GrB_Matrix_setElement(A,x,i,j)` in the C interface to GraphBLAS. There are better ways to build a MATLAB or GraphBLAS sparse matrix, by passing all the entries to a single constructor function: `A=sparse(I,J,X)` to build a MATLAB sparse matrix, or `A=GrB.build(I,J,X)` to build the same matrix as a `@GrB` object in MATLAB.

Thus, in spite of the inability of most MATLAB m-files and `mexFunctions` to modify their inputs, MATLAB remains a powerful and expressive tool for writing graph algorithms via GraphBLAS. As presented in Section VI, graph algorithms are typically able to achieve comparable performance as the same algorithm written in the C interface to GraphBLAS, while being much easier to write.

VI. Performance Results

Table III compares the performance of four different implementations of the PageRank and Betweenness-Centrality algorithms, on an Intel® Xeon® E5-2698 v4 CPU with 20 hardware cores. The same compiler (gcc 5.4.0) was used, and 40 threads were used for each method (the value returned by `omp_get_max_threads`), with hyperthreading enabled. SuiteSparse:GraphBLAS v3.3.0 [4] was used.

- the GAP benchmark code by Scott Beamer [5].
- LAGraph [15], using the C API of GraphBLAS.
- pygraphblas with Python 3.8, presented in Section IV.
- `@GrB` with MATLAB R2018b, presented in Section V.

Benchmark	Urand	Kron	Twitter	Web	Road
PR: GAP benchmark	25.3	19.8	15.2	5.3	1.0
PR: LAGraph (in C)	27.7	22.1	17.9	8.9	1.5
PR: pygraphblas	28.2	22.8	18.3	10.3	2.3
PR: MATLAB `@GrB`	60.3	56.2	52.6	39.5	13.8
BC: GAP benchmark	46.4	31.5	10.8	3.0	1.5
BC: LAGraph (in C)	62.2	40.6	14.1	7.9	61.2
BC: pygraphblas	61.9	40.4	13.6	7.8	61.1
BC: MATLAB `@GrB`	76.2	53.5	24.0	31.4	1773.1

TABLE III: Performance results (run time in seconds)

The Python results show that pygraphblas adds very little interpretation overhead to GraphBLAS algorithms, as compared with the C code in LAGraph, since the bulk of the work is performed by GraphBLAS. Python's job is merely to schedule the work of matrix operations in an optimal order. Because Python uses the non-blocking mode SuiteSparse:GraphBLAS, where operations can return in lazy fashion before being completed, the Python interpreter can quickly schedule many operations and let the implementation pick the best approach for deferring work until the last possible moment.

The only significant difference between the Python and C benchmark times were for the Road graph, which has a very high diameter but is otherwise a small graph. The number of iterations is high but the actual amount of parallelizable work per iteration is low. This forces to the surface a very small but measurable per iteration serial overhead of the interpreter's inner loop. For larger graphs that can take full advantage of multi-core systems, like the rest of the datasets measured, the Python interpreter overhead is negligible.

The MATLAB implementation uses an object-oriented interface `@GrB`, and a set of C `mexFunction` interfaces. The implementation is elegant and expressive. For many of the larger problems, the performance is competitive. It is limited by the inability of a MATLAB `mexFunction` to modify its inputs. This is particularly true of the Road graph for the BC problem. The graph has a very high diameter, and each iteration makes a small change to a dense matrix of size 4-by-n. The entire matrix must be reconstructed at each iteration, leading to a high run time. As a result, each iteration takes $\Omega(n)$ time, while doing as little as $\Omega(1)$ useful work.

Subjectively speaking, the small cost in runtime associated with using Python, and in most cases MATLAB, is traded off with a vast increase in code readability, expressibility, and rapid development time. Users can gain near-optimal performance, within a few percent of state of the art hand-crafted C code, with no compilation complexity or highly specific knowledge of platform optimization. For example, we wrote a high-performance deep neural network inference algorithm in MATLAB and C in a mere 20 minutes; the code worked the first time we tried it, and it was 100x faster than the reference implementation [16].

This benefit also carries from one platform to the next. A GraphBLAS algorithm can be developed on a laptop, and then copied verbatim to a GPU cluster with no changes needed to take advantage of a completely different architecture. A GPU implementation of SuiteSparse:GraphBLAS is in progress, in collaboration with NVIDIA, as is an implementation that exploits optimized kernels in the Intel Math Kernel Library, in collaboration with Intel.

VII. Conclusion

We have shown that programmers can use Python and MATLAB interfaces to the GraphBLAS without unduly sacrificing performance. These interfaces realize a critical motivation for many of us working on the GraphBLAS; namely, that graph algorithms, when expressed in the language of linear algebra, are quick to develop and easy to understand.

References

[1] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker,

S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conf. (HPEC)*. IEEE, 2013.

[2] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson, "Mathematical foundations of the GraphBLAS," in *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[3] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *Graph Algorithms Building Blocks workshop at IPDPS (GABB)*. IEEE, 2017.

[4] T. A. Davis, "Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, vol. 45, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3322125

[5] S. Beamer, K. Asanovic, and D. Patterson, "The GAP benchmark suite," arXiv:1508.03619, 2015.

[6] P. Erdős and A. Rényi, "On random graphs. I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[7] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," in *Cray User's Group (CUG)*. CUG, 2010.

[8] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication," *European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2005.

[9] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" *International World Wide Web Conference (WWW)*, 2010.

[10] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," *International World Wide Web Conference (WWW)*, pp. 595–601, 2004.

[11] "9th DIMACS implementation challenge - shortest paths." `http://www.dis.uniroma1.it/challenge9/`, 2006.

[12] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001. [Online]. Available: https://doi.org/10.1080/0022250X.2001.9990249

[13] "pygraphblas user defined types," https://github.com/michelp/pygraphblas/blob/master/pygraphblas/demo/User-Defined-Types.ipynb.

[14] T. A. Davis, "Creating sparse finite-element matrices in MATLAB," in *Loren on the Art of MATLAB*. MathWorks, Inc., Mar. 2007. [Online]. Available: https://blogs.mathworks.com/loren/2007/03/01/creating-sparse-finite-element-matrices-in-matlab/

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

[16] 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, pp. 1–6.