The GraphBLAS in Julia and Python: the PageRank and Triangle Centralities

Michel Pelletier
CEO
Graphegon, Inc.
Corvallis, OR, USA
michel@graphegon.com

Will Kimmerer
Julia Lab
MIT
Lexington, KY, USA
kimmerer@mit.edu

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

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

***Abstract*—The GraphBLAS is a standard API for expressing Graphs in the language of linear algebra. The goal is to provide high performance while exploiting the fundamental simplicity of Graph algorithms in terms of a common set of "Basic Linear Algebra Subprograms". A robust parallel implementation of the GraphBLAS C specification is available as the SuiteSparse GraphBLAS library [1]. The simplicity of the GraphBLAS, so apparent "in the math", is diminished when expressed in terms of a low level language such as C. To see the full expressive power of the GraphBLAS, a high level interface is needed so that the elegance of the mathematical underpinnings of the GraphBLAS is clearly apparent in the code. In this paper we introduce the Julia interface to the SuiteSparse:GraphBLAS library and compare it to the Python interface [2]. We implement the PageRank and Triangle Centrality algorithms with remarkably little code and show that no significant performance is sacrificed by moving from C to the more productive Python and Julia interfaces.**

***Index Terms*—Graph Algorithms, Linear Algebra, GraphBLAS, Programmability, Python, Julia**

I. INTRODUCTION

The GraphBLAS are a community [3] driven effort to define a framework for implementing graph algorithms. They are defined by two documents; a mathematical definition [4] and the C language binding [5]. With the GraphBLAS as well as the wider community of graph algorithm researchers, much of the focus is on performance. Performance can be quantified. Hence when comparing different systems for expressing graph algorithms, runtimes and other performance metrics serve as the basis of comparisons.

The GraphBLAS, however, are not motivated purely by performance. They are also motivated by the mathematical elegance of Linear algebra and the concise and sometimes simple representation of complex algorithms [6]. This elegance can be described as its *programmability*. Software researchers don't often discuss programmability. It is hard to quantify and therefore difficult to turn into simple metrics 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 explore the programmability of the GraphBLAS through Julia and Python interfaces to the SuiteSparse GraphBLAS [1] library. We implement and benchmark two centrality algorithms: the well studied PageRank Centrality and a novel algorithm for Triangle Centrality [7] against a high-performance C implementation using large social graphs available from the SuiteSparse Matrix Collection [8]. We show the code for both languages side by side and invite the reader to notice how clear the underlying algorithm is in the code while retaining the performance of the underlying parallel C implementation.

While much has been written about the GraphBLAS and the Python Interface [2], [9], this paper is the first introduction to the Julia Interface. Given the rapid growth in adoption of the Julia programming language [10], this interface has the potential to become an important tool in graph algorithm research. Another key contribution of this paper is the Triangle centrality algorithm. As far as we know, this is the first paper to show how to implement this algorithm using the GraphBLAS.

II. THE GRAPHBLAS

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 result is a duality between matrix multiplication and graph traversal. This duality is shown in Figure 1 (source [4]), where matrix vector multiplication takes a single step in a Breadth First Search across a graph.

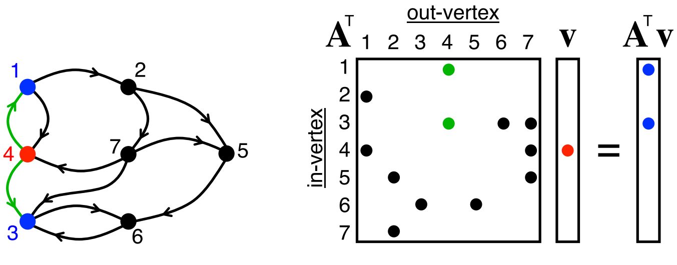

Fig. 1. Matrix Vector Multiplication is a single step in Breadth First Search

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 except 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	**Description**	**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	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
	reduce to scalar	$s = s \odot [\oplus_{ij} \mathbf{A}(i, j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$
`GrB_kronecker`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathrm{kron}(\mathbf{A}, \mathbf{B})$

Table I lists all GraphBLAS operations [5], 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. PYTHON AND JULIA GRAPHBLAS BINDINGS

Each GraphBLAS Operation is implemented in a language-specific way in both Python and Julia. When it comes to exposing a library to a language, a series of aesthetic decisions are made to address the ergonomics of using functionality from the perspective of a programmer that does not know the details of the underlying library language.

In the case of SuiteSparse, which is written in C, decisions were made as to how to best expose all the features of the library in a way that comports with the expectations of Python and Julia programmers. This consideration is often referred to making code look “Pythonic” in Python and “Julian” in Julia.

For example, the C language has no optional function arguments, so unused arguments must be passed as `NULL`. Optional arguments are supported in both Python and Julia, and in both cases reasonable defaults are chosen for programmers so they do not need to consider which arguments should be `NULL` or not.

Another consideration for programmers are choices of default operators for particular operations. The GraphBLAS C API does not dictate which operations are the default for a given operation and the operator argument is always required when working with the C API. However this burden is typically not expected to be carried in high level languages, so binding authors make default choices.

Some choices are obvious, for example the default semiring for Matrix Multiplication (`GrB_mxm()` and friends) is `plus_times`, which mirrors the behavior found in dense library like `numpy`. Another default choice is to use `plus_monoid` when using `GrB_reduce()` functions.

In Table II we show most of the GraphBLAS operations in both Python and Julia. In many of those cases, there is either a functional API or an operator equivalent API for both Python and Julia. This shows how different styles of expression can be used by different programmers to suite their specific cases and tastes.

A. Python

The Python programming language emphasizes simplicity and readability, which aligns well with the high level usability goals of the GraphBLAS. Like Julia, Python 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. Python provides this clarity and ease in exchange for some run-time overhead, although this overhead is demonstrated to be negligible for large graphs.

The core operations of the GraphBLAS express Linear Algebra, and thus the algebraic operators of Python can be

TABLE II
GRAPHBLAS OPERATIONS IN PYTHON AND JULIA

Python Function Syntax	**Python Operator Syntax**	**Julia Function Syntax**	**Julia Operator Syntax**

`A.min_plus(B,out=C)`	`with FP32.min_plus:`
	`C=A@B`
		`C = mul(A,B,(min, plus))`	`C = ⋆(min,+)(A,B)`

`A.eadd(B,out=C,mask=M)`	`with Mask(M):`
	`C=A|B`
		`C = eadd(A,B,mask=M)`	`C = ⊕(A,B,mask=M)`

`A.emult(B,out=C,accum=FP32.min)`	`with Accum(FP32.min):`
	`C=A&B`
		`emul!(C,A,B,accum=/)`	`C./=A .⋆ B`

`A.assign(B,slice(1,2),slice(3,4)`	`A[1:2,3:4]=B`	`setindex!(A,1:2,3:4,B)`	`A[1:2,3:4]=B`
`A.assign(B,mask=C)`	`A[C]=B`	`setindex!(A,B,mask=C)`	`A[mask=C]=v`
`A.assign_row(1,v)`	`C[1]=v`	`setindex!(A,1,:,v)`	`A[1,:]=v`
`A.assign_col(1,v)`	`C[:,1]=v`	`setindex!(A,:,1,v)`	`A[:,1]=v`
`A.extract_row(1,out=v)`	`v=C[1]`	`v = getindex(A,1,:)`	`v = A[1,:]`
`A.extract_col(1,out=v)`	`v=C[:,1]`	`v = getindex(A,:,1)`	`v = A[:,1]`
`A.assign_scalar(42,slice(1,2))`	`A[1:2]=42`	`setindex!(A,1:2,:,42)`	`A[1:2,:] = 42`
`A.apply(FP64.abs)`	`A.abs()`	`map(abs,A)`	`abs.(A)`
`A.reduce_vector(FP64.min)`	N/A	`reduce(min,A,dims=1)`	N/A
`A.reduce_float(FP64.min)`	N/A	`reduce(min,A)`	N/A

used to express mathematical formula naturally in code. For those who do not prefer operator-heavy code, there is also a "functional" API that eschews syntactic operators, trading off a bit more typing for less cryptic code.

Of course, there is no accounting for taste, and one person's elegant expression is another's inscrutable mess. For this reason Python (and Julia) provide multiple ways to express the same GraphBLAS operation as shown in Table II. This table and the code examples below it highlight how remarkably similar the two languages are at a syntactic level, and how they can both provide multiple ways to express algorithms using both functional and operational paradigms.

Some common criticisms of the Python programming language is that it is slow, and in the most literal sense this is true. Many languages, including Julia, handily beat Python in common code benchmarks because Python lacks some of the advanced features of these other languages, like native JIT compilation, that can generate machine code many times faster than the interpreted bytecode of Python.

However, Python often makes the pragmatic decision to defer work to lower level functions written in faster "bare metal" languages like C. The extremely popular numpy library for example, provides high performance dense matrix functionality by exploiting existing high performance BLAS libraries. In this sense, Python is simply used as a job scheduler for underlying work-efficient libraries.

This approach is exactly how pygraphblas is implemented: it utilizes the SuiteSparse:GraphBLAS library [1], [11] to provide a high performance, parallel asynchronous implementation of the underlying functionality. As graphs get larger, the amount of time spent "in Python" vanishes towards insignificant noise, as the bulk of the execution time is spent in kernels inside SuiteSparse.

Research and development of the GraphBLAS in the Python community is very active, and there are now two libraries that utilize SuiteSparse:GraphBLAS with differing syntax and semantics. This paper covers the pygraphblas library, but there is also a library called grblas [12] whose syntactic variations are not described in this paper, but for which all GraphBLAS related concepts will still apply. It's worth noting that the authors of both bindings collaborate on a shared low-level dependency pacakage to SuiteSparse:GraphBLAS [9].

B. Julia

The Julia programming language provides the high-performance of statically-typed languages with the expressiveness of dynamically-typed scripting languages. Julia is well positioned as a GraphBLAS interface for multiple reasons. From the perspective of a library author calling out to C and Fortran libraries can be done with no additional glue code and low overhead. From a users perspective Julia's Unicode support, syntactic macros, and multiple-dispatch paradigm help present a GraphBLAS interface which is readable, discoverable, expressive, and composable.

Multiple dispatch is the core enabling feature of the GraphBLAS interface. There are significant performance implications of multiple dispatch, allowing function calls whose argument types can be statically inferred to be efficiently JIT compiled by LLVM. In this case, the real benefits of multiple dispatch are not in performance, which is handled entirely by SuiteSparse:GraphBLAS, but are instead be found in the user facing interface.

We will illustrate these benefits by example with the Julia `Base.map` function. The first function signature is quite simple, `Base.map(f, A)` which takes two untyped arguments:

an operation of some sort, and something on which to apply that operation. A generic fallback method like this one is a common idiom in Julia. We can define other methods for this function, such as `Base.map(f, A::AbstractArray)`. When `map` is called and `A` is a subtype of `AbstractArray` Julia will dispatch to that specific definition, or a more specific one.

This allows us to define `map` methods which call the correct GraphBLAS `GrB_apply` functions. For instance, `Base.map(op::BinaryOp, x, A::GBMatrix)`, `Base.map(op::BinaryOp, A::GBMatrix, x)`, and `Base.map(op::UnaryOp, v::GBVector)` illustrate the use of multiple dispatch to extend existing functions onto new types. Extending existing types with new functions is achieved in the same fashion, by defining a generic function, which can optionally have type specific behaviors.

For an end-user, the first noticeable benefit is discoverability. The Julia standard library `Base` provides a powerful but relatively small set of verbs, including functions like `reduce`, and `map` as well as operators like `*` and `/`. A good example of this is the `to_numpy` function from the Python interface, used to construct a NumPy array from a GraphBLAS matrix. In Julia the equivalent would be `Matrix(A::GBMatrix)` since we are free to extend the constructors of `Matrix`.

Multiple dispatch, and other Julia language features like broadcasting, make simple operations succinct and readable. For instance, `C .+= abs.(A' .- B)` additively accumulates $|A^T - B|$ into C. More complex expressions involving masks can be constructed using operators in their prefix form such as `-(A, B, mask=M)`.

For more complex expressions Julia has first-class support for macros, which enable an experimental interface virtually identical to the GraphBLAS notation. `@gb C(M) = C + A max.+ B` executes the equivalent Julia code with virtually no overhead while matching the algebraic notation up to parentheses instead of brackets.

The final, and most key benefit of the Julia package is composability. `GBMatrix` is a subtype of `AbstractArray`. That means it will work with any function that accepts `AbstractArray` arguments, as well as any which accept Julia's existing `SparseMatrixCSC`. This includes virtually the entire Julia ecosystem, including sparse solvers from SuiteSparse and MKL Pardiso.

The Julia package `SuiteSparseGraphBLAS.jl` contains all basic functionality illustrated in this paper while `GraphBLAS.jl` will accumulate algorithms in a similar spirit to LAGraph as well as close integration with other packages. `GraphBLAS.jl` is in active iteration, while `SuiteSparseGraphBLAS.jl` is converging on a stable release with the potential to become a recommended default sparse package for Julia.

IV. PAGERANK

PageRank is a link analysis algorithm that ranks vertices in a graph according to the rank and number of their incoming links. This creates an iterative recursive definition, where ranks in a vector are multiplied against a graph until they converge to within a certain limiting threshold of rank change. While many implementations exist that do not use linear algebra, mathematically PageRank has a natural Linear Algebra formulation and this is exploited by the GraphBLAS to make a simple algorithm that competes well against even hand-tuned parallel C code [13].

The *PageRank* kernel (PR) computes a measure of the importance (rank) of each vertex in a graph based on the rank of the nodes connected to the vertex. For a graph with n vertices 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. 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}.

In general, PageRank is a kind of Centrality Algorithm, as it ranks vertices according to its specific notion of importance. There are many other types of centrality algorithms, but few of them are studied quite as deeply as PageRank. By contrast, there is a new kind of centrality algorithm developed by P. Burkhardt [7] called Triangle Centrality that we will explore next.

V. TRIANGLE CENTRALITY

Centrality is a way to measure the importance of vertices in a graph. While PageRank is one of the best known centrality measures, there many other ways to quantify centrality. A relatively new centrality measure is the *Triangle Centrality* [7]. It defines the importance of a vertex based on the concentration of triangles in the immediate neighborhood of the vertex. More specifically, it measures the sum of triangle-counts for a vertex and its neighbors, normalized by the total number of triangles in a graph. Unlike Betweeness Centrality, it does not require all the shortest paths (or a large sample of shortest paths) and, unlike PageRank, it is a direct rather than iterative method.

Triangle Centrality is defined in [7] as follows. Given an undirected graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges, where $N(v)$ is the neighborhood set of v, $N_\Delta(v)$ is the set of neighbors that are in triangles with v, and $N^+_\Delta(v)$ is the closed set that includes v, then the triangle centrality for v, where $\Delta(v)$ and $\Delta(G)$ denote the respective triangle counts of v and G, is:

$$TC(\nu) = \frac{\frac{1}{3}\sum_{u\in N^+_\Delta(\nu)}\Delta(u) + \sum_{w\in(N(\nu)/N_{\Delta(\nu)})}\Delta(w)}{\Delta(G)}$$

This equation can be cast into a Linear Algebraic formulation based on the adjacency matrix, $\mathbf{A}$, of G. For an undirected graph such as G, the adjacency matrix is symmetric. The matrix product $A \times A$ returns a matrix with the two-hop

```python
def PR(A, dout, damping=0.85, itermax=100, tol=1e-4)
    # A: transpose of adjacency matrix
    # dout: vector of out-degree of A
    n = A.nrows
    t = Vector.sparse(FP64, n)
    r = Vector.dense(FP64, n, fill=1.0 / n)
    d = dout / damping
    dmin = Vector.dense(FP64, n, fill=1.0 / damping)
    d.eadd(dmin, FP64.max, out=d)
    teleport = (1 - damping) / n
    for i in range(1, itermax):
        t, r = r, t
        w = t / d
        r[:] = teleport
        A.plus_second(w, out=r, accum=FP64.plus)
        t -= r
        t.abs(out=t)
        if t.reduce_float() <= tol:
            break
    return r
```

```julia
function PR(A, d, damping=0.85, itermax=100, tol=1e-4)
    # A: transpose of adjacency matrix
    # d: vector of out-degree for A
    n = size(A, 1)
    t = GBVector{Float64}(n)
    r = GBVector(n, 1.0 / n)
    d = d ./ damping
    dmin = GBVector(n, 1.0 / n)
    eadd!(d, d, dmin, max)
    teleport = (1 - damping) / n
    for j in 1:itermax
        t, r = r, t
        w = t ./ d
        r[:] = teleport
        mul!(r, A, w, (+, second), accum=+)
        eadd!(t, t, r, -)
        map!(abs, t)
        if reduce(+, t) <= tol
            break
        end
    end
    return r
end
```

Fig. 2. PageRank in Python and Julia

paths in the graph (the wedges in the graph). If you take the intersection of the two-hop matrix) with the adjacency matrix (the one-hop paths) you get the matrix of triangles $\mathbf{T} = \mathbf{A}^2 \circ \mathbf{A}$ where $\circ$ is the Hadamard product. If $\mathbf{I}$ is the identity matrix, $\mathbf{1}$ is a dense vector of ones, and $\check{\mathbf{T}}$ is the Boolean analog of $\mathbf{T}$ (all zeros other than defined elements which are equal to 1), we compute the vector of triangle centrality measures for each vertex.

$$\mathbf{c} = \frac{(3\mathbf{A} - 2\check{\mathbf{T}} + \mathbf{I})\mathbf{T}\mathbf{1}}{\mathbf{1}^T\mathbf{T}\mathbf{1}}$$

From this equation, we have a straightforward algorithm that can be implemented using the GraphBLAS (Algorithm 3 in [7]).

Algorithm 1 Triangle Centrality

```
procedure TC($A$)
    $T \leftarrow A^2 \circ A$
    $\hat{T} \leftarrow$ binary matrix of $T$
    $X \leftarrow 3A - 2\hat{T} - I$
    $\boldsymbol{y} \leftarrow T1$
    $k \leftarrow \mathbf{1}^{\mathsf{T}}\boldsymbol{y}$
    $\boldsymbol{c} \leftarrow \frac{1}{k}X\boldsymbol{y}$
    return $\boldsymbol{c}$
```

Algorithm 1 is expressed in both Python and Julia in Figure 3. Two versions of the algorithm are provided, one is a straightforward translation of the above formula and the other uses SuiteSparse specific optimizations to improve performance. In Section VI we discuss these optimizations, which complicate the algorithm slightly but provide improved performance, particularly for large graphs.

VI. SuiteSparse Optimizations

Mapping the mathematical formulation of Linear Algebra to the GraphBLAS is straightforward. It does not always give SuiteSparse enough information to deliver maximum performance. A simple implementation of an algorithm can be written quickly, but to achieve the best results it often helps to consider variations that make algorithms more work efficient.

A particular optimization applied when an operation on a matrix is structural (i.e., the existence of an edge, not its weight, is relevant) is to never load its value, saving many memory loading cycles. The work of structural operations becomes only that "of the indexes" and SuiteSparse can leverage this to skip work and short-circuit computational paths.

In Figure 3 we show an updated version of the Triangle Centrality algorithm using these SuiteSparse optimizations. An example is the use of the `plus_pair` and `plus_second` semirings instead of the default semiring `plus_times`. Because the multiplication operation is not necessary for triangle counting, only the sum, the `times` operator can be replaced with the more efficient `pair` and `second`.

The `pair` operator returns 1 when an edge is present in both the left and right operands of a matrix multiplication. Since only knowledge of the existence of the "pair" is necessary to count the triangular edge, there is no need to load or multiply the edge weights. For the operator `second`, the value to be summed is taken from the *second operand* of the matrix multiplication. Often this is useful when the left operation is a binary, or structural, and it is being used to match up with values in the right operand, which in the case of triangle counting contains the current count for an edge.

In addition, it is not necessary to compute all of $\mathbf{T}$. Only the lower triangular part is needed.

```python
def triangle_centrality1(A):
    T = A.mxm(A, mask=A, desc=ST1)
    y = T.reduce_vector()
    k = y.reduce_float()
    return (3 * (A@y) - 2 * (T.one()@y) + y) / k


def triangle_centrality2(A):
    M = A.tril(-1)
    T = A.plus_pair(A, mask=M, desc=ST1)
    y = (T.reduce_vector() +
         T.reduce_vector(desc=T0))
    k = y.reduce_float()
    T2 = T.plus_second(y) +
         T.plus_second(y,desc=T0)
    r = (3 * A.plus_second(y)) + ((-2) * T2) + y
    return r / k
```

```julia
function triangle_centrality1(A)
    T = mul(A, A', mask=A, desc=S)
    y = reduce(+, T, dims=2)
    k = reduce(+, y)
    return (3 * (A*y)) - 2 *(one.(T)*y) + y) ./ k
end

function triangle_centrality2(A)
    M = tril(A, -1)
    T = mul(A, A', (+, pair), mask=M, desc=S)
    y = reduce(+, T, dims=2) .+
        reduce(+, T', dims=2)
    k = reduce(+, y)
    T2 = *(+, second)(T, y) .+
         *(+, second)(T', y)
    r = (3 .* *(+, second)(A, y)) + (-2 .* T2) + y
    return r ./ k
end
```

Fig. 3. TriangleCentrality in Python and Julia with (bottom) and without (top) Optimizations

VII. PERFORMANCE RESULTS

Table III compares the performance of three different implementations of the PageRank (PR) and Triangle Centrality (TC) algorithms, on an Intel® Xeon® E5-2698 v4 CPU with 20 hardware cores. The same compiler (gcc 10.2) was used, and 40 threads were used for each method (the value returned by `omp_get_max_threads`), with hyperthreading enabled. SuiteSparse:GraphBLAS v5.1.5 [1], [11] was used.

For Triangle Centrality there are two versions of the algorithm. TC1 is a straightforward translation of the mathematical definition of Triangle Centrality using no SuiteSparse specific optimizations. TC2 is optimized, computing just the lower triangular part of $\mathbf{T}$, and using a SuiteSparse specific optimized semiring (`plus_pair`). These optimizations are discussed in Section VI.

- LAGraph [14], using the C API of GraphBLAS.
- pygraphblas with Python 3.8, presented in Section III-A.
- SuiteSparseGraphBLAS.jl with Julia 1.6, presented in Section III-B.

The results show that both the Julia and Python interfaces add little overhead compared to C code from LAGraph [14], since the bulk of the work is performed by GraphBLAS. The Python and Julia code just schedule the work of matrix operations in an optimal order. Since they use the non-blocking mode of SuiteSparse:GraphBLAS, where operations can return in lazy fashion before being completed, they quickly schedule operations and let SuiteSparse pick the best approach for deferred work until the last possible moment.

The small cost associated with Python or Julia is balanced by an increase in code readability, expressibility, and rapid development. Users 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.

This benefit should carry from one platform to the next. A GPU implementation of SuiteSparse:GraphBLAS is in progress, in collaboration with NVIDIA. Once completed, a GraphBLAS algorithm can be developed on a laptop and then copied verbatim to a GPU with no changes needed to take advantage of a completely different architecture.

TABLE III
PERFORMANCE RESULTS (RUN TIME IN SECONDS)

Benchmark	**Skitter**	**LiveJournal**	**Orkut**	**Friendster**
Vertices	1,696,415	3,997,962	3,072,441	65,608,366
Edges	11,095,298	34,681,189	117,185,083	1,806,067,135
Triangles	28,769,868	177,820,130	627,584,181	4,173,724,142
PR:LAGraph	0.229	0.381	1.067	72.073
PR:Python	0.281	0.502	1.226	84.009
PR:Julia	0.293	0.499	1.215	84.50
TC1:LAGraph	0.520	2.072	18.765	424.425
TC1:Python	0.593	2.270	19.633	438.422
TC1:Julia	0.593	2.242	19.058	426.819
TC2:LAGraph	0.320	1.075	9.126	208.354
TC2:Python	0.367	1.172	9.385	213.996
TC2:Julia	0.351	1.138	9.271	213.622

VIII. CONCLUSION

In this paper we have introduced the Julia interface to the GraphBLAS. This matches the expressiveness noted in our earlier paper about the Python interface to the GraphBLAS [2]. We have shown that programmers can use Python and Julia interfaces to the GraphBLAS without unduly sacrificing performance, making graph algorithms expressed in the language of linear algebra quicker to develop and easier to understand.

This work was supported by NVIDIA, Intel, MIT Lincoln Lab, IBM, the National Science Foundation (1514406), and Julia Computing.

REFERENCES

[1] 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

[2] T. G. Mattson, M. Pelletier, and T. A. Davis, "Graphblas programmability: Python and MATLAB interfaces," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020.

[3] "The GraphBLAS Forum," http://graphblas.org/.

[4] 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.

[5] A. Buluç, B. Brock, T. Mattson, S. McMillan, and J. Moreira, "The GraphBLAS C API specification," graphblas.org/, 2019.

[6] 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.

[7] P. Burkhardt, "Triangle centrality," https://arxiv.org/pdf/2105.00110.pdf, April 2021.

[8] "Suitesparse matrix collection," https://sparse.tamu.edu/.

[9] M. Pelletier, "pygraphblas: Python SuiteSparse:GraphBLAS binding," https://github.com/Graphegon/pygraphblas.

[10] "The Julia programming language," https://https://julialang.org/.

[11] T. A. Davis, "Algorithm 10xx: SuiteSparse:GraphBLAS: Parallel graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, 2021, (submitted).

[12] E. Welch and J. Kitchen, "grblas python wrapper around GraphBLAS," https://github.com/metagraph-dev/grblas.

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

[14] 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.