GraphBLAS: Handling performance concerns in large graph analytics – *invited paper*

Manoj Kumar, José E. Moreira, Pratap Pattnaik
IBM Thomas J. Watson Research Center
Yorktown Heights, NY
{manoj1,jmoreira,pratap}@us.ibm.com

ABSTRACT

Emerging applications in health-care, social media analytics, cyber-security, homeland security, and marketing require large graph analytics. Attaining good performance on these applications on modern day hardware is challenging because of the complex pipelines and deep memory hierarchy of these machines. In this paper, we review the linear algebra formulation of graph-analytics and show that it effectively handles the separation of performance concerns, best handled by system developers, from application logic concerns.

The linear algebra formulation leverages the community experience in optimizing both hardware and software for applications that have a substantial linear algebra component. We review the GraphBLAS API, a compact C API for linear algebra formulation of graph algorithms. The core semiring operations are described first, followed by the rest of the API. We then illustrate how commonly used graph algorithms are implemented using the main GraphBLAS API calls. Executing these algorithms on a highly optimized linear algebra run-time validates that the time spent in execution of the algorithm is indeed almost entirely in the library, thus delegating the performance concerns solely to the library developer. Furthermore, the linear algebra formulation consistently outperforms the text-book version of these algorithms by a factor of two to five. Vector and matrix multiplications consume the majority of the computational time, particularly as problem size increases, putting them in the cross hairs for performance optimization.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Software and its engineering → Modules / packages**;

KEYWORDS

Graph analytics, linear algebra, GraphBLAS

ACM Reference Format:
Manoj Kumar, José E. Moreira, Pratap Pattnaik. 2018. GraphBLAS: Handling performance concerns in large graph analytics – *invited paper*. In *CF '18: CF '18: Computing Frontiers Conference, May 8–10, 2018, Ischia, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3203217.3205342

CF '18, May 8–10, 2018, Ischia, Italy

ACM ISBN 978-1-4503-5761-6/18/05…$15.00
https://doi.org/10.1145/3203217.3205342

1 INTRODUCTION

Emerging applications in health-care, social network analytics, and financial fraud prevention represent data as large graphs for analysis [14]. These graph analytics applications differ from the more conventional high performance computing and transaction processing, and the emerging machine intelligence and deep learning applications in three important respects. First of all, the size of graph data is very large. Social graphs are already approaching billions of vertices with several hundred attributes per vertex or edge, requiring algorithms to be crafted to carefully minimize latency for data access across the cache/memory hierarchy. Secondly, the access patterns are highly irregular. They lack predictable strides or locality in reference. This makes the task of managing the cache/memory hierarchy substantially harder. Finally, the graph data is highly non-uniform in terms of in/out degree distribution of vertices and the presence of community structures. This complicates the exploitation of parallelism on modern multi-core parallel systems [15, 17] from the perspective of load balancing, minimization of synchronization overhead, and minimization of inter-task communication.

In addition to the idiosyncrasies of the graph analytics problem, the modern multi-core processors too have their own complexities that need to be factored in application programs to minimize the execution time of applications. Programmers have to undertake complex program restructuring to optimize performance of graph analytics on modern processors [1, 3, 12]. Cache line sizes, cache capacities, page sizes and limited translation entries are a few examples. Suffices to say that developing useful and innovative graph analytics applications, and getting best performance when executing them, require two complementary and highly developed skills. Accordingly, computer science research has responded to this challenge by developing graph analytics frameworks that separate the concerns of developing the applications from the concern of their optimal execution, some of the notable ones being [6, 8, 13, 16].

Graph algorithms can be formulated in a variety of ways, which fall into two large classes. *Local formulations* are described in terms of operations on one vertex or one edge of the graph, which are then repeated over the entire graph [13, 16]. *Global formulations* operate on the entire set of vertices or edges of a graph at a time, typically producing intermediate results and continuously repeated until that result converges [6, 8]. Local and global by itself is not a measure of parallelism or performance. Local operations can be repeated many times to yield plenty of parallelism, whereas global operations can have inter- and intra-operation dependencies that hurt parallelism.

For our goal of speeding up large scale graph analytics for business applications on POWER processors [17], we decided to focus

on global formulations. In particular, we decided on the *linear algebra formulation*. This formulation better leverages the experience and effort of the community in optimizing hardware and software for HPC applications that have substantial linear algebra component. High performance libraries for this formulation have been demonstrated [11].

In Section 2 we introduce the basic matrix-vector and matrix-matrix operations on adjacency matrices of graphs and vertex-sets, and draw the parallel between the linear algebra operations on sparse matrices and operations on graphs. In this section we also introduce the semiring algebraic structure for these operations, an abstraction essential for ensuring the compactness of the GraphBLAS API.

In section 3, we summarize the function calls and the C API of the GraphBLAS library and discuss its three important elements, mask, descriptor, and the accumulation operator. These constructs minimize the number of GraphBLAS calls needed to carry out an application task. More importantly, they enable the library implementer to minimize the data movement between the processor and the memory, mitigating chip I/O bandwidth constraints that arise frequently in large graph analytics. These linear algebra operations, implemented as a library invoked from application programs, can then have hardware optimized implementations that take hardware specific parameters and capabilities into account. Transformations to improve the locality in memory access for better cache performance [12], and to address memory bandwidth constraints when necessary for performance, are examples of hardware specific optimizations.

Next we give three examples of graph analytics applications implemented with the GraphBLAS API in section 4. The linear algebra formulation of page rank and betweenness centrality is adapted from [9]. Computations in training and inferencing of Deep Neural Networks (DNNs), resemble graph analytics computations [10] and can be handled effectively in GraphBLAS, as illustrated in this section.

The betweenness centrality and page rank algorithms are run on various sized scale-free graphs, generated with the Recursive Matrix (R-MAT) algorithm [4], to identify the GraphBLAS functions that have highest usage. Similarly the DNN forward propagation step was also run on varying size sparse weight matrices, the weights generated using an uniform distribution. The results of these experiments are discussed in section 5, the key observation being that matrix and vector multiplication operations consume most of the execution cycles.

The examples in section 4 show that the GraphBLAS API effectively handles the separation of application logic concerns from the performance concerns, delegating the task of attaining good performance to the developers of the graph analytics library, who are knowledgeable in hardware specific optimizations. Furthermore, the measurements in section 5 show that that almost all the computation of these examples is contained within the main operations of the GraphBLAS library, where a well written library can provide a factor of two to five better performance than the application programmer written code [11].

2 GRAPHS AND LINEAR ALGEBRA

A graph $G = \langle V, E\rangle$ is a tuple consisting of a set $V = \{v_i\}, i = 0, \ldots N-1$ of vertices and a set $E = \{e_k\}, k = 0, \ldots M-1$ of edges. The edges themselves are tuples $e = \langle v_s, v_t\rangle$ consisting of a source vertex v_s and a target vertex v_t. Vertices and edges have properties called *attributes*. We denote the attributes of a vertex v and an edge e by $\|v\|$ and $\|e\|$, respectively.

A multitude of theorems and algorithms have been derived from this simple definition. We are mostly interested in algorithms, particularly ones in which the attributes of an edge can be described by a single scalar quantity that we shall call the *weight* of the edge and denote by $w_{ij} = \|\langle v_i, v_j\rangle\|$. These algorithms manipulate the set of edges of a graph to find important properties of the graph. Examples include the now ubiquitous *Page Rank* and *Betweenness Centrality* algorithms, but go far beyond that. In fact, as we shall see later, even deep learning computation can be expressed with graph linear algebra operations.

If the only attribute of an edge $\langle v_i, v_j\rangle$ of a graph $G = \langle V, E\rangle$ is its weight w_{ij}, then it is natural to represent the graph by an $N \times N$ matrix $\mathbf{A}$, called the *adjacency matrix* of the graph, such that element (i, j) of the matrix is defined as $\mathbf{A}(i, j) = w_{ij}$. We do not define the value of element $\mathbf{A}(i, j)$ unless there is an edge, of some weight, connecting vertices v_i and v_j.

The reader will notice a similarity with the concept of *sparse matrices* in linear algebra, for which only some elements are explicitly represented and the remaining elements are implicitly assumed to have the value zero (0). Our approach is slightly different, in that elements are either explicitly defined, with a specific weight, or nonexistent. The GraphBLAS specification describes in great detail how to handle these nonexistent elements in computations. This is at the heart of the formulation with semirings, that we shall see soon.

It is also convenient, when writing graph algorithms in a linear algebra formulation, to make use of vectors. A vector is the one-dimensional equivalent of a matrix and, therefore, is described by a size parameter N. Like matrices, the elements of a vector are either explicitly defined or nonexistent.

It is possible to assign a value to the nonexisting (undefined) elements of a matrix (or vector), and to perform operations on that matrix (or vector), through the introduction a *semiring* algebraic structure. A semiring $S = \langle D, \oplus, \otimes, \mathbf{0}\rangle$ is defined by a domain D, a commutative and associative binary operation $\oplus : D \times D \to D$, called the *addition* of the semiring, a binary operation $\otimes : D \times D \to D$, called the *multiplication* of the semiring, and an element $\mathbf{0} \in D$ that is both the *additive identity* and the *multiplicative annihilator* of the semiring. That is, $e \oplus \mathbf{0} = \mathbf{0} \oplus e = e$ and $e \otimes \mathbf{0} = \mathbf{0} \otimes e = \mathbf{0}, \forall e \in D$.

Given a matrix $\mathbf{A}$ of elements in the domain D and a semiring S, also with domain D, we can now interpret the undefined elements of $\mathbf{A}$ as having the value of the additive identity of S. By associating $\mathbf{A}$ with a different S we automatically change the values of the undefined elements and also the $\oplus$ and $\otimes$ operations on the elements of that matrix.

Going a little further, given two matrices $\mathbf{A}$ and $\mathbf{B}$ (of domain D) and shapes $M \times P$ and $P \times N$ respectively, and a semiring $S = \langle D, \oplus, \otimes, \mathbf{0}\rangle$ we can compute a matrix $\mathbf{C}$ (also of domain D and of shape $M \times N$) using the standard matrix multiplication definition

Table 1: Summary of operations in a linear algebra formulation of graph algorithms. A, B and C are matrices. x, y and z are vectors. Summation over repeated indices is implicit.

Semiring	Operation	Definition
$S = \langle D, \oplus, \otimes, \mathbf{0} \rangle$	$\mathbf{C} = \mathbf{A} \overset{S}{\underset{\otimes \cdot \oplus}{}} \mathbf{B}$	$\mathbf{C}(i,j) = \bigoplus(\mathbf{A}(i,k) \otimes \mathbf{B}(k,j))$
	$\mathbf{y} = \mathbf{x} \overset{S}{\underset{\otimes \cdot \oplus}{}} \mathbf{A}$	$\mathbf{y}(j) = \bigoplus(\mathbf{y}(k) \otimes \mathbf{A}(k,j))$
	$\mathbf{y} = \mathbf{A} \overset{S}{\underset{\otimes \cdot \oplus}{}} \mathbf{x}$	$\mathbf{y}(i) = \bigoplus(\mathbf{A}(i,k) \otimes \mathbf{x}(k))$
	$\mathbf{C} = \mathbf{A} \overset{S}{\underset{\oplus}{}} \mathbf{B}$	$\mathbf{C}(i,j) = (\mathbf{A}(i,k) \oplus \mathbf{B}(i,j))$
	$\mathbf{C} = \mathbf{A} \overset{S}{\underset{\otimes}{}} \mathbf{B}$	$\mathbf{C}(i,j) = (\mathbf{A}(i,k) \otimes \mathbf{B}(i,j))$
	$\mathbf{z} = \mathbf{x} \overset{S}{\underset{\oplus}{}} \mathbf{y}$	$\mathbf{z}(i) = (\mathbf{x}(i) \oplus \mathbf{y}(i))$
	$\mathbf{z} = \mathbf{x} \overset{S}{\underset{\otimes}{}} \mathbf{y}$	$\mathbf{z}(i) = (\mathbf{x}(i) \otimes \mathbf{y}(i))$

$\mathbf{C} = \mathbf{A} \overset{S}{\underset{\otimes \cdot \oplus}{}} \mathbf{B}$, where

$$\mathbf{C}(i,j) = \bigoplus_{k=0}^{P-1} (\mathbf{A}(i,k) \otimes \mathbf{B}(k,j)), \begin{cases} i=0,\ldots,M-1 \\ j=0,\ldots,N-1 \end{cases},$$

where $\otimes$ means semiring multiplication of two elements and $\bigoplus_{k=0}^{P-1}$ means summation over the products using semiring addition. Element $\mathbf{C}(i,j)$ is computed only if the i-th row of $\mathbf{A}$ and j-th column of $\mathbf{B}$ have at least one element defined with the same k. (Otherwise, $\mathbf{C}(i,j)$ is nonexistent and treated as the additive identity.)

We can also define element-wise operations for matrices, $\mathbf{C} = \mathbf{A} \overset{S}{\underset{\oplus}{}} \mathbf{B}$ or $\mathbf{C} = \mathbf{A} \overset{S}{\underset{\otimes}{}} \mathbf{B}$, which are described, together with their vector variants, in Table 1. Matrix-matrix multiplication, combined with its vector-matrix and matrix-vector variants, and element-wise operations form the basis for writing graph algorithms in the linear algebra formulation.

To illustrate this, let $G = \langle V, E \rangle$ be a graph. Consider the subset of the vertices that are d hops away from some source vertex s, and let this subset be represented by a vector $\mathbf{x}$ in the domain $D = \{\mathbf{false}, \mathbf{true}\}$ such that $\mathbf{x}(i) = \mathbf{true}$ if and only if vertex v_i is in the subset ($\mathbf{x}(i)$ is undefined otherwise). Also, let $\mathbf{A}$ be a matrix, in the same domain, such that $\mathbf{A}(i,j) = \mathbf{true}$ if and only if $\langle v_i, v_j \rangle \in E$. We call $\mathbf{x}$ the d-distance *frontier* in a *breadth-first search* (BFS) of graph G from origin vertex s.

Given a semiring $S = \langle D, \vee, \wedge, \mathbf{false} \rangle$ where $\vee$ is logical or and $\wedge$ is logical and, we can compute $\mathbf{y} = \mathbf{x} \overset{S}{\underset{\otimes \cdot \oplus}{}} \mathbf{A}$, which is the set of vertices that can be reached with one edge traversal (hop) from any of the vertices in frontier $\mathbf{x}$. That is, $\mathbf{y}$ is the $d+1$-distance frontier in a breadth-first search of graph G from origin vertex s. In other words, vector-matrix multiplication implements one step of a BFS traversal algorithm. (Strictly speaking, we have to remove from $\mathbf{y}$ those vertices that have already been visited. This is easily done with an element-wise operation.)

Furthermore, if we have a matrix $\mathbf{X}$ of P rows, each row representing a different d-distance frontier (from a different source vertex), then $\mathbf{Y} = \mathbf{X} \overset{S}{\underset{\otimes \cdot \oplus}{}} \mathbf{A}$ represents the next frontier (at distance $d+1$) from each of the source vertices. Matrix-matrix multiplication can be used to perform a step of BFS from multiple source vertices at the same time.

We now start to see the benefits of the linear algebra formulation of graph algorithms from a separation of concerns aspect. If algorithms can be written using the aforementioned few building blocks, and a few others we will cover in the next section, most of the processing time is in these building blocks. (The time spent in the various building blocks is quantified in Section 5). Thus we have a small set of operations that need to be optimized for a given platform. Furthermore, the optimization and parallelization of each of these building blocks are backed by many years of work by the community.

3 OPERATIONS IN GRAPHBLAS C API

Version 1.1.0 of the The GraphBLAS C API Specification (the most recent at the time of this writing) is 190 pages long. Its essence can be captured in Table 2. Each row lists one of the key operations of GraphBLAS, together with its mathematical definition and the corresponding API call that encodes it in the C bindings of GraphBLAS.

Before further describing each operation, it is useful to describe the overall form of the operations, in which a value (of a certain shape) computed in the right-hand side of an assignment ($\langle$**rhs**$\rangle$) is copied into a left-hand side target ($\langle$**lhs**$\rangle$) of the same shape,

$$\langle \mathbf{lhs} \rangle \leftarrow \langle \mathbf{rhs} \rangle$$

and, in particular, the form of the left-hand side deserves some detailed discussion. The discussion is valid for both matrix and vector targets, but let us focus on matrices for brevity.

After a result is computed in the right-hand side of the assignment, it has to be stored in the target matrix **C**. In the simplest form, the entire results is stored in **C**, replacing whatever elements were there before, and that is the end of the assignment ($\mathbf{C} \leftarrow \ldots$).

We can optionally use another matrix **M** as a mask ($\mathbf{C}\langle\mathbf{M}\rangle \leftarrow \ldots$). The mask matrix must have the same shape (dimensions) of the target matrix **C**. Each element $\mathbf{M}(i,j)$ of **M** is interpreted as a Boolean value: **true** or **false**. (Undefined elements are always treated as **false**.) If the value of $\mathbf{M}(i,j)$ is **true**, then $\mathbf{C}(i,j)$ is written with the value of the corresponding element of the results (or annihilated, if there is no corresponding result element). If the value of $\mathbf{M}(i,j)$ is **false**, then $\mathbf{C}(i,j)$ is left unchanged.

Sometimes it is desirable to annihilate from **C** those elements for which the corresponding mask element is **false**. This is indicated with a ‡ modified to the left-hand side ($\mathbf{C}\langle\mathbf{M}\rangle‡ \leftarrow \ldots$). One may also want to complement the mask. That is, **true** elements become **false** and vice-versa. This is indicated with a negation symbol in front of the mask ($\mathbf{C}\langle\neg\mathbf{M}\rangle \leftarrow \ldots$ or $\mathbf{C}\langle\neg\mathbf{M}\rangle‡ \leftarrow \ldots$).

Using square brackets to indicate optional constructs, we end up with the common syntax for all left-hand side of the operations in Table 2:

$$\mathbf{C}\,[\langle[\neg]\mathbf{M}\rangle[‡]]$$

or, for vectors

$$\mathbf{c}\,[\langle[\neg]\mathbf{m}\rangle[‡]].$$

The right-hand side of the operations also has some common features across the various operations. First, there is the optional

Table 2: GraphBLAS operations and API. A and B are input matrices, while a and b are input vectors. C and c are the target (output) matrix and vector, respectively. $S = \langle \mathbb{R}, \oplus, \otimes, \mathbf{0} \rangle$ is a semiring. $\odot$ is an element-wise operation. M and m are matrix and vector masks, respectively. I and J are arrays of indices. f is a scalar function and Δ is a GraphBLAS descriptor.

Operation	Semantics	API
mxm	$\mathbf{C}\,[\langle[\neg]\mathbf{M}\rangle[\ddagger]] \leftarrow [\mathbf{C}\odot]\mathbf{A}^{[T]} \overset{S}{\underset{\otimes\cdot\oplus}{}} \mathbf{B}^{[T]}$	`GrB_mxm(`$\mathbf{C},\mathbf{M},\odot,S,\mathbf{A},\mathbf{B},\Delta$`)`
vxm	$\mathbf{c}\,[\langle[\neg]\mathbf{m}\rangle[\ddagger]] \leftarrow [\mathbf{c}\odot]\mathbf{a} \overset{S}{\underset{\otimes\cdot\oplus}{}} \mathbf{B}^{[T]}$	`GrB_vxm(`$\mathbf{c},\mathbf{m},\odot,S,\mathbf{a},\mathbf{B},\Delta$`)`
mxv	$\mathbf{c}\,[\langle[\neg]\mathbf{m}\rangle[\ddagger]] \leftarrow [\mathbf{c}\odot]\mathbf{A}^{[T]} \overset{S}{\underset{\otimes\cdot\oplus}{}} \mathbf{b}$	`GrB_mxv(`$\mathbf{c},\mathbf{m},\odot,S,\mathbf{A},\mathbf{b},\Delta$`)`
ewise ×	$\mathbf{C}\,[\langle[\neg]\mathbf{M}\rangle[\ddagger]] \leftarrow [\mathbf{C}\odot]\mathbf{A}^{[T]} \overset{S}{\underset{\otimes}{}} \mathbf{B}^{[T]}$	`GrB_eWiseMult(`$\mathbf{C},\mathbf{M},\odot,S,\mathbf{A},\mathbf{B},\Delta$`)`
ewise +	$\mathbf{C}\,[\langle[\neg]\mathbf{M}\rangle[\ddagger]] \leftarrow [\mathbf{C}\odot]\mathbf{A}^{[T]} \overset{S}{\underset{\oplus}{}} \mathbf{B}^{[T]}$	`GrB_eWiseAdd(`$\mathbf{C},\mathbf{M},\odot,S,\mathbf{A},\mathbf{B},\Delta$`)`
extract	$\mathbf{C}\,[\langle[\neg]\mathbf{M}\rangle[\ddagger]] \leftarrow [\mathbf{C}\odot]\mathbf{A}^{[T]}(I,J)$	`GrB_extract(`$\mathbf{C},\mathbf{M},\odot,\mathbf{A},I,J,\Delta$`)`
assign	$\mathbf{C}\,[\langle[\neg]\mathbf{M}\rangle[\ddagger]] \leftarrow \mathbf{C}\vert I,J\vert[\mathbf{C}(I,J)\odot]\mathbf{A}^{[T]}$	`GrB_assign(`$\mathbf{C},\mathbf{M},\odot,\mathbf{A},I,J,\Delta$`)`
apply	$\mathbf{C}\,[\langle[\neg]\mathbf{M}\rangle[\ddagger]] \leftarrow [\mathbf{C}\odot]f(\mathbf{A}^{[T]})$	`GrB_apply(`$\mathbf{C},\mathbf{M},\odot,f,\mathbf{A},\Delta$`)`
reduce	$\mathbf{c}\,[\langle[\neg]\mathbf{m}\rangle[\ddagger]] \leftarrow [\mathbf{c}\odot](\overset{S}{\underset{\oplus}{}}\mathbf{A}^{[T]})$	`GrB_reduce(`$\mathbf{c},\mathbf{m},\odot,\langle\mathbb{R},\oplus,\mathbf{0}\rangle,\mathbf{A},\Delta$`)`

accumulation of the result of the computation with the target matrix **C**. This is denoted by $[\mathbf{C}\odot]\ldots$ (or $[\mathbf{c}\odot]\ldots$ in case of vector) construct. The $\odot$ is a binary operation that is applied element-wise between the target matrix **C** and the computation results. Missing elements (on either side of $\odot$) are treated as identity for the operation. (The semantics of accumulation are slightly different for the GraphBLAS assign operation. We will cover that separately.) Second, when an input to an operation is a matrix, it can be optionally transposed before the computation is performed. This is indicated in the semantics by $\mathbf{A}^{[T]}$ or $\mathbf{B}^{[T]}$.

Of the nine GraphBLAS operations in Table 2, the matrix and vector multiplications and the element-wise operations were already described in the previous sections. They can all be augmented with an optional accumulation with the target and possibly transposition of the input matrices. The final writeback of the results to the target can be controlled with a mask.

The *apply* operation applies a scalar function to every defined element of input matrix **A** (or $\mathbf{A}^T$) to compute the intermediate result, which can then be optionally accumulated with the target matrix **C** and written under control of a mask.

The *reduce* operation computes a result vector of size M out of an $M \times N$ matrix **A** (or $\mathbf{A}^T$) by adding across elements of each row of the matrix (using the addition operation of a semiring). Again, the compute vector can be optionally accumulated with the target vector **c** and written under control of a mask.

The *extract* operation selects a rectangular subsection of input matrix **A** (or $\mathbf{A}^T$) consisting of the rows and columns of **A** identified by the index arrays I and J, respectively. This intermediate results can be optionally accumulated and written, under the control of a mask, to target matrix **C**.

Finally, the *assign* operation builds a result matrix by inserting the elements of an input matrix **A** (or $\mathbf{A}^T$) into the rows and columns of target matrix **C** as defined by the index arrays I and J respectively. The other elements of **C** are preserved in this intermediate result. If an optional accumulation with the target **C** is included, the accumulation, different from all other cases, is performed *before* the insertion operation, using only section $\mathbf{C}(I,J)$ of matrix **C**. This final result is then written to the target **C** under optional control of a mask.

There is a clear one-to-one correspondence in Table 2 between the more mathematical semantics and the C API, and the translation between the two is straightforward. The reader will notice a last argument Δ in the API calls. This is what GraphBLAS calls a *descriptor* and it is used for passing additional modifiers to the operation, such as transposing the input matrices, complementing the mask, or zeroing those elements of the target that are not updated by this operation.

Table 2 omits several details of the API. In particular, there are vector versions of the element-wise, extract, assign, apply and reduce operations. Furthermore, in case of element-wise addition or multiplication, it is also possible to use a monoid or binary operation in place of the semiring. The semantics are the same as if using a semiring with that particular operation as addition (for element-wise add) or multiplication (for element-wise multiply). Rest of the API, not covered in Table 2, pertains to importing and exporting data into and from GraphBLAS matrix and vector objects, defining semirings and other algebraic structures, and building user-defined types.

Nevertheless, the main message of this paper still remains. Not only can we write our graph algorithms using the GraphBLAS operations, but the vast majority of execution time is captured by a few of those operations. The validation of this statement is the topic of the next two sections.

4 EXAMPLE ALGORITHMS

We illustrate the use of GraphBLAS in three different computations. The first is the feed-forward pass of a deep neural network. Although this may not be considered a traditional graph algorithm, it is a linear algebra computation and a natural fit for GraphBLAS.

The second and third examples are more conventional graph computations, involving the evaluation of vertex properties known as *betweenness centrality* and *page rank*.

4.1 Deep Neural Network

Our first example of implementing an algorithm in GraphBLAS is for the computation of a feed-forward pass in a deep neural network (DNN). The formulation of this computation in GraphBLAS was first developed by Kepner et al [10]. The algorithm is shown in Algorithm 1. The corresponding C code, using the GraphBLAS C API, is shown in Figure 1

Algorithm 1 Algorithm for feed-forward DNN.

1: **procedure** DNN($\mathbf{Y}_{0:L}, \mathbf{W}_{0:L-1}, \mathbf{B}_{0:L-1}$)
2: $S_1 = \langle \mathbb{R}, +, \times, 0 \rangle$ ▷ arithmetic semiring
3: $S_2 = \langle \{-\infty \cup \mathbb{R}\}, \max, +, -\infty \rangle$ ▷ max-plus semiring
4: **for** $k = 0, \ldots, L-1$ **do**
5: $\mathbf{Y}_{k+1} \leftarrow \mathbf{W}_k \overset{S_1}{\otimes_{\cdot\oplus}} \mathbf{Y}_k$ ▷ multiply input by weights
6: $\mathbf{Y}_{k+1} \leftarrow \mathbf{Y}_{k+1} \overset{S_2}{\otimes} \mathbf{B}_k$ ▷ add bias
7: $\mathbf{Y}_{k+1} \leftarrow \mathbf{Y}_{k+1} \overset{S_2}{\oplus} \mathbf{0}$ ▷ apply ReLU
8: **end for**
9: **return** Y_L ▷ return last layer output
10: **end procedure**

The algorithm makes use of two semirings. The first (S_1 in line 2 of Algorithm 1) is a conventional arithmetic semiring over the real numbers. The second (S_2 in line 3 of Algorithm 1 is usually called a *max-plus* semiring, also over the real numbers.

The deep neural network contains L layers, numbered $0, \ldots, L-1$. Y_k is the input to layer k and Y_{k+1} is the output of layer k (which is the input to layer $k+1$). For each layer k, the algorithm first multiplies the weight matrix W_k of that layer by the input Y_k, using the arithmetic semiring (line 5 of Algorithm 1). The bias matrix B_k for the layer is added to that first result using element-wise multiplication with the max-plus semiring, as shown in line 6 of Algorithm 1. (Addition is the multiplicative operation of a max-plus semiring.) Finally, this second result is compared with the constant 0 to implement a *rectifier linear unit* (ReLU) using element-wise addition with the max-plus semiring, as shown in line 7 of Algorithm 1. (Maximum is the additive operation of a max-plus semiring.)

The C code of Figure 1 is a direct translation of Algorithm 1. Line 12-15 and 17-20 of Figure 1 create the two semirings (arithmetic and max-plus), while lines 22-26 create the 0 matrix to be used in the ReLU implementation. In practice, these semirings and 0 matrices (possibly of different size for each layer) would be created once in an initialization phase. The actual feed-forward phase happens in the loop body of lines 30-32, in direct correspondence with lines 4-6 of Algorithm 1.

4.2 Betweenness Centrality (BC)

The algorithm for computing the betweenness centrality (BC) of the vertices of a graph G, written with GraphBLAS operations, is shown in Algorithm 2. That figure illustrates the brevity of linear algebra formulation of betweenness centrality algorithm by Brandes [2].

All operations are performed using the standard arithmetic semiring over real numbers (S in line 2 of Algorithm 2). The first while loop (lines 4-9) is the forward pass of the algorithm which implements a breadth first search to compute the number of shortest paths from a chosen starting vertex s to every other vertex. In that loop, $\mathbf{f}$ is the current frontier, which is initialized with just a starting vertex s. In each iteration, the loop accumulates the frontier at the current level in a vector $\mathbf{p}$ and saves the history of frontiers in matrix $\mathbf{F}$ (lines 6 and 7). Most important, it computes the next frontier by performing a vector-matrix multiply of the current frontier $\mathbf{f}$ and the adjacency matrix $\mathbf{A}$ of the graph. Already visited vertices are excluded from the next frontier by using complement of already visited vertices as a mask (line 8 of Algorithm 2).

Because we are using an arithmetic semiring, the frontier $\mathbf{f}$ consists of the number of distinct shortest paths from the source vertex s to each of the vertices in the frontier at that level. Matrix $\mathbf{F}$ however is treated as a Boolean matrix, with $\mathbf{F}(d, v) = 1$ if vertex v is visited in level d of the breadth first search and 0 otherwise. At the end of the first while loop, $\mathbf{p}$ has the number of distinct shortest paths from s to each vertex in the graph.

The computation of the betweenness centrality values happens in the second while loop (lines 11-20) of Algorithm 2, known as the *backward pass* of the algorithm. The goal is to compute the vector δ of the contributions to the betweenness centrality of each vertex from the paths starting in source vertex s. Those contributions are then added to the overall betweenness centrality vector $\mathbf{b}$. (A brute force algorithm would then repeat the procedure for every vertex s of the graph. Sampling has been shown to work well and significantly reduce the total amount of computation.)

Algorithm 2 Algorithm for Betweenness Centrality.

1: **procedure** BC($\mathbf{A}, \mathbf{b}, s$)
2: $S = \langle \mathbb{R}, +, \times, 0 \rangle$
3: $d \leftarrow 0$, $\mathbf{p} \leftarrow 0$, $\mathbf{f} \leftarrow 0$, $\mathbf{f}(s) \leftarrow 1$
4: **while** ($|\mathbf{f}| \neq 0$) **do**
5: $d \leftarrow d + 1$
6: $\mathbf{p} \leftarrow \mathbf{p} \overset{S}{\oplus} \mathbf{f}$
7: $\mathbf{F}(d, :) \leftarrow \mathbf{f}$
8: $\mathbf{f}\langle \neg \mathbf{p} \rangle \ddagger \leftarrow \mathbf{f} \overset{S}{\otimes_{\cdot\oplus}} \mathbf{A}$
9: **end while**
10: $d \leftarrow d - 1$, $\delta\langle \mathbf{F}(d, :) \rangle \leftarrow 0$
11: **while** ($d \geq 2$) **do**
12: // $\mathbf{w}$ is contribution to upper layer δ from lower layers
13: $\mathbf{w}\langle \mathbf{F}(d, :) \rangle \ddagger \leftarrow \delta \overset{S}{\oplus} \mathbf{1}$
14: $\mathbf{w}\langle \mathbf{F}(d, :) \rangle \ddagger \leftarrow \mathbf{w} \overset{S}{\otimes} (1/\mathbf{p})$
15: // Aggregation of lower layer contributions
16: $\mathbf{t}\langle \mathbf{F}(d-1, :) \rangle \ddagger \leftarrow 0$
17: $\mathbf{t}\langle \mathbf{F}(d-1, :) \rangle \leftarrow \mathbf{t} + \mathbf{A} \overset{S}{\otimes_{\cdot\oplus}} \mathbf{w}$
18: $\delta\langle \mathbf{F}(d-1, :) \rangle \leftarrow \delta + (\mathbf{t} \overset{S}{\otimes} \mathbf{p})$
19: $d \leftarrow d - 1$
20: **end while**
21: $\mathbf{b} \leftarrow \mathbf{b} \overset{S}{\oplus} \mathbf{u}$
22: **return** $\mathbf{b}$
23: **end procedure**

Figure 1: C implementation of a DNN inference computation using GraphBLAS.

```
1  #include <math.h>
2  #include <GraphBLAS.h>
3
4  GrB_Info dnn(GrB_Matrix *Y, GrB_Matrix *W, GrB_Matrix *B, GrB_Index L)
5  /*
6   * L         - Number of layers
7   * W[0:L-1] - Array of m x m weight matrices
8   * B[0:L-1] - Array of m x n bias matrices
9   * Y[0:L]   - Array of m x n layer-input/output matrices
10  */
11 {
12    GrB_Monoid FP32Add;                                          // Monoid <float,+,0.0>
13    GrB_Monoid_new(&FP32Add, GrB_FP32, GrB_PLUS_FP32, 0.0f);
14    GrB_Semiring FP32AddMul;                                     // Semiring <float,+,*,0.0>
15    GrB_Semiring_new(&FP32AddMul, FP32Add, GrB_TIMES_FP32);
16
17    GrB_Monoid FP32Max;                                          // Monoid <float,max,-inf>
18    GrB_Monoid_new(&FP32Max, GrB_FP32, GrB_MAX_FP32, -INFINITY);
19    GrB_Semiring FP32MaxPlus;                                    // Semiring <float,max,+,-inf>
20    GrB_Semiring_new(&FP32MaxPlus, FP32Max, GrB_PLUS_FP32);
21
22    GrB_Index m, n;
23    GrB_Matrix_nrows(&m,Y[0]); GrB_Matrix_ncols(&n,Y[0]);
24    GrB_Matrix Zero;                                             // Zero = 0.0
25    GrB_Matrix_new(&Zero, GrB_FP32, m, n);
26    GrB_assign(Zero, GrB_NULL, GrB_NULL, 0.0, GrB_ALL, m, GrB_ALL, n, GrB_NULL);
27
28    for(int k=0; k<L; k++)
29    {
30      GrB_mxm(Y[k+1], GrB_NULL, GrB_NULL, FP32AddMul, W[k], Y[k], GrB_NULL);          // Y[k+1] = W[k]*Y[k]
31      GrB_eWiseMult(Y[k+1], GrB_NULL, GrB_NULL, FP32MaxPlus, Y[k+1], B[k], GrB_NULL); // Y[k+1] = W[k]*Y[k] + B[k]
32      GrB_eWiseAdd (Y[k+1], GrB_NULL, GrB_NULL, FP32MaxPlus, Y[k+1], Zero, GrB_NULL); // Y[k+1] = max(W[k]*Y[k] + B[k],0)
33    }
34
35    GrB_free(&Zero);
36    GrB_free(&FP32Add);     GrB_free(&FP32Max);
37    GrB_free(&FP32AddMul); GrB_free(&FP32MaxPlus);
38
39    return GrB_SUCCESS;
40 }
```

The OpenMP C implementation of the algorithm in SSCA [7] is two to three hundred lines of code. In addition to being compact, the linear algebra formulation performs better than the SSCA2 reference implementation, as discussed later in section 5. Furthermore, the linear algebra formulation has no code that is specific to a particular data layout, no code to to allocate/deallocate memory, and no explicitly parallel constructs. All these tasks, often containing hardware platform- or distributed computing environment-specific constructs, have been subsumed in the GraphBLAS library implementations, thus making the application code more portable.

4.3 Page Rank (PR)

The GraphBLAS implementation of an algorithm for computing the page rank, $\mathbf{p}_{new}$, of the vertices of an n vertex graph with adjacency matrix $\mathbf{A}$ is shown in Algorithm 3. The input α is the probability with which a user on a page with **od** hyperlinks navigates to one of the outgoing hyperlinks chosen equiprobably. In addition to these navigational jumps, from a page with hyperlinks, the user randomly jumps to any of the n pages with probability $(1-\alpha)$, the target page chosen equiprobably. From pages with no hyperlinks, the random jump is equiprobable to any of the n pages.

The algorithm iterates on the equation $\mathbf{p}_{new} = \mathbf{p}_{old} \times \mathbf{G}$ until convergence, where $\mathbf{G}$ is defined as

$$\mathbf{G} = \alpha \times \mathbf{A} \times diag(1/\mathbf{od}) + \mathbf{b} \times \mathbf{1}^{\mathrm{T}}$$

and $\mathbf{b}$ is a n-vector of entries $(1-\alpha)$ or 1 corresponding to pages with and without hyperlinks. The matrix $\mathbf{G}$ is not materialized in Algorithm 3, as the right hand term corresponding to random jump contributions is a dense matrix of identical columns. It is computed more efficiently by using vector operations and replicating the result.

In our notation, assignment of a scalar, shown in this *italicized* font, to a vector, shown in this **bold** font, implies replication of the scalar. An $\oplus$ to the right of an "$v \leftarrow v$" construct, as shown in statement 17, indicates the use of the accumulate operator discussed in section 3. Statements 6, 7 and 15 illustrate the use of mask, 6 uses a complemented value of the mask, per descriptor supplied with the function call. The ‡ on the left hand side of statements 7 and 15 indicates that the result vector elements corresponding to the **false** positions of the mask are annihilated, i.e., they do not exist in the sparse representation. The unary $\overset{S}{\oplus}$ notation in statements 5, 11 and 17 denote the GraphBLAS reduce operation defined in Table Table 2.

The multiplication with constant vector **1** in statements 5, 7 and 17 is notational. GraphBLAS specifies reduction operations that are implemented more efficiently than general multiplication in GraphBLAS implementations. The workhorse of this algorithm, as we shall see later, is the vector-matrix multiplication in statement 14, used to compute the contribution of hyperlink navigation to page rank. The random jump contribution to page rank, t computed in statement 11, is identical for all vertices.

Algorithm 3 Algorithm for Page Rank.

```
1:  procedure PageRank(A, n, α, ε)
2:      S = ⟨ℝ, +, ×, 0⟩                          ▷ Arithmetic semiring
3:      p_new ← 1/n, d ← α
4:      err = 1/n                                  ▷ L2 norm of page rank change
5:      od ← ⊕^S A                                 ▷ Out degree of pages
6:      b⟨od⟩ ← (1 − α)/n, b⟨¬od⟩ ← 1/n            ▷ Random jumps
7:      od_inv⟨od⟩‡ ← 1/od
8:      while (err > ε) do
9:          // Contribution from random jumps
10:         p_old ← p_new
11:         temp ← b ⊗^S p_old
12:         t ← ⊕^S temp
13:         p_new ← t
14:         // Contribution from hyperlink navigation
15:         temp⟨od⟩‡ ← p_old ⊗^S od_inv
16:         temp ← temp ⊗·⊕^S A
17:         p_new ← p_new ⊕ (temp ⊗^S d)
18:         // Test for convergence
19:         temp ← p_new − p_old, temp ← temp ⊗^S temp
20:         err ← ⊕^S temp
21:     end while
22:     return p_new
23: end procedure
```

5 EXPERIMENTS

We coded each of the algorithms discussed in Section 4 using GPI (Graph Processing Interface) [6]. GPI is a GraphBLAS like interface that predates GraphBLAS, but has a mature implementation. At the time of this writing performance of GPI implementation is about an order of magnitude faster than the nascent GraphBLAS specification-compatible implementation [5]. We profiled the execution of the algorithms for different graph sizes using the Linux `perf` command to quantify the time spent in various GraphBLAS (actually their GPI variant) calls. The measurements are shown in Figure 2.

All experiments were performed on an IBM Power S824L server configured with 24 POWER8 cores running at 3.325GHz. Each core is capable of running from 1 single-thread (ST) to 8 simultaneous threads of execution (SMT8). The server is configured with 512 GiB of memory, accessible through a total of 16 memory channels. Total bandwidth from processing cores to memory is over 400GB/s, with two-thirds of that for reading from memory and one-third for writing to memory. The operating system installed is Ubuntu 16.04 distribution of Linux for PowerPC Little Endian. All code is compiled with GNU 5.4 compilers (gcc/g++).

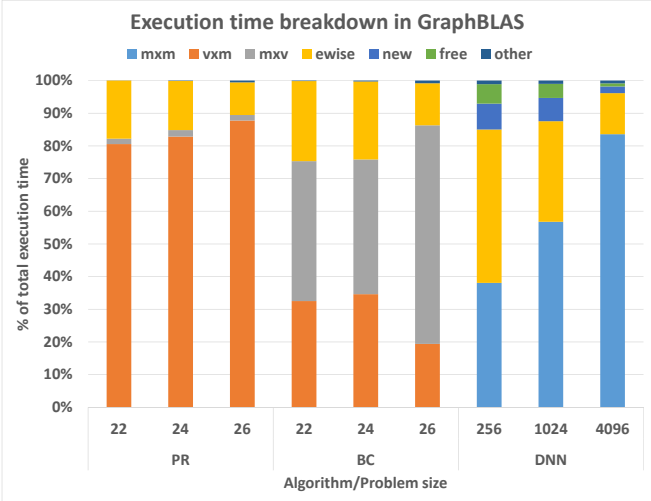

Figure 2: Execution time percentage for different GraphBLAS operations, as a percentage of total algorithm time.

The x-axis in Figure 2 shows the three algorithms and the input graph sizes. For page rank and betweenness centrality, abbreviated PR and BC respectively, the x-axis label is the scale of the graph. That is, the graphs have 2^{22}, 2^{24} and 2^{26} vertices respectively. The graph is undirected with an average of 16 edges per vertex. These R-MAT graph were generated with parameters A=0.55, B=C=0.1 and D=(1.0-A-B-C)=0.25. For DNN, the x-axis label is the size of the weight matrix. The weight matrices are sparse square matrices of single-precision (32-bit) floating-point numbers, on average one in sixty four entries being non-zero. The location of nonzero entries is determined by using independent Bernoulli distributions, and the value at those locations is taken from a $U[-1, 3)$ distribution.

The following conclusions can be drawn from the analysis of the performance data. First, it is clear that for all three applications, most of the computation happens in the matrix-matrix or matrix-vector routines. (The single exception being the small DNN problem size.) As expected, these are the performance bottlenecks. The balance of the execution time can be mostly accounted for by the element-wise operations. Everything else represents a relatively minor contribution to the total execution time.

The fraction of time spend in the three matrix/vector multiplication operations increases with the problem size, the jump from scale 24 to 26 being more significant. For page rank and betweenness centrality, the number of nonzero entries in the matrices increases linearly with the problem size, since the number of entries per row remains constant at 32 (sixteen undirected edges are represented as twice as many directed edges in the adjacency matrix). The increase in fraction of time spent in mxv/vxm as the problem size gets larger is due to lower cycles per instruction (CPI), resulting from a higher rate of on-chip cache misses. For DNNs the number of elements in the weight matrices increases quadratically with matrix size. The matrices being relatively small (when compared to the PR or BC cases), the instruction execution rate actually increases as the problem size gets larger. The higher time spend in mxm operation

n DNNs is strictly due to larger fraction of instructions in mxm. The variations across applications also provide interesting insights. Betweenness centrality uses mxv operation to aggregate the contribution to centrality values from vertices to their parents (deltas) in the BFS tree. Currently, in GPI, mxv is implemented as a vxm on the transpose of the matrix. The cost of transpose makes the mxv operation more expensive for scale 26 matrix. Betweenness centrality has a higher fraction of element-wise operations compared to page rank because the calculations of deltas are mostly element-wise operations on vectors.

It is clear that the matrix/vector multiplication variants should be the first target for any optimization effort. For single-threaded execution, they may very well represent most if not all of the optimization effort. However, when we considers the task of parallelizing a GraphBLAS implementation (meaning, each individual operation exploits multi-threaded parallelism), then the other operations can gain importance as well. Even though matrix-vector and matrix-matrix operations accounts for 90% of the execution time of an algorithm, we can get a speedup of no more than 10 from parallelizing only these operations. In a world of machines with tens, hundreds and even thousands (in the case of GPUs) threads per processor, any parallelization effort has to consider the broader set of operations, particularly the element-wise operations.

6 SUMMARY AND CONCLUSIONS

It is well known that linear algebra formulations enable succinct representation of graph analytics algorithms [9]. Efficient implementations of linear algebra formulations give a factor of two to five performance improvement over textbook approaches [11]. In this paper we extended this body of knowledge with the evidence that the nine main operations of the GraphBLAS API, to which the authors have made significant contributions, are sufficient to capture the bulk of graph analytics computation. We also demonstrated through examples that this API effectively unburdens the application programmer from the performance concerns such as efficient data structures to represent graph data, exploitation of parallelism, and control constructs to manage sets of vertices and edges. This is particularly valuable because modern day hardware requires platform specific optimizations to attain good performance. With the GraphBLAS API, the performance concerns can be handled entirely by the GraphBLAS library developer.

We also discussed the two important elements of the API, 1) the polymorphism enabled by separating the data containers (vectors and matrices) from the semirings that operate on that data and 2) the use of descriptors, masks and accumulate operation in the the various GraphBLAS operations. Both are essential to the compactness of the GraphBLAS API. The 11 built-in arithmetic and 6 built-in comparison operations permit construction of 960 unique semirings and 44 unique monoids that can be used by three vector/matrix × matrix multiplications and 4 element-wise operations listed in Table 1. Additional semirings and monoids can be created with user-defined types and operations. The descriptors, masks and accumulate operator are important to mitigate the memory bandwidth bottlenecks, which are more severe in graph analytics applications, and particularly more so with the linear algebra formulation.

REFERENCES

[1] A. Azad and A. BuluÃğ. 2017. A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, New York, NY, 688–697. https://doi.org/10.1109/IPDPS.2017.76

[2] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[3] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. 2016. Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 37, 12 pages. https://doi.org/10.1145/2925426.2926278

[4] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. *SDM* 4 (2004), 442–446.

[5] T. Davis. 2017. SuiteSparse:GraphBLAS: graph algorithms via sparse matrix operations on semirings. (October 2017).

[6] K. Ekanadham, W. P. Horn, Manoj Kumar, Joefon Jann, José Moreira, Pratap Pattnaik, Mauricio Serrano, Gabriel Tanase, and Hao Yu. 2016. Graph Programming Interface (GPI): A Linear Algebra Programming Model for Large Scale Graph Computations. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/2903150.2903164

[7] David A. Bader et.al. Released 5 September 2007. *HPCS Scalable Synthetic Compact Applications #2 Graph Analysis* (v2.2 ed.). Technical Report. GraphAnalysis.org. arXiv:http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.2.pdf

[8] J. Kepner, P. Aaltonen, D. Bader, A. BuluÃğ, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, New York, NY, 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[9] Jeremy Kepner and John Gilbert (Eds.). 2011. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/1.9780898719918 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9780898719918

[10] J. Kepner, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, and H. Tufo. 2017. Enabling massive deep neural networks with the GraphBLAS. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, New York, NY, 1–10. https://doi.org/10.1109/HPEC.2017.8091098

[11] M. Kumar, W.P. Horn, J. Moreira, and P. Pattnaik. 2018. IBM POWER9 and cognitive computing. *IBM Journal of Research and Development* 62, 4/5 (July/August 2018), *(accepted for publication)*.

[12] M. Kumar, M. Serrano, J. Moreira, P. Pattnaik, W. P. Horn, J. Jann, and G. Tanase. 2016. Efficient implementation of scatter-gather operations for large scale graph analytics. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, New York, NY, 1–7.

[13] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[14] MIT Lincoln Laboratory 2013. Special Issue: Graphs and Networks. *MIT Lincoln Laboratory Journal* 20, 1 (2013).

[15] D. Mulnix. 2017. *IntelÂő XeonÂő Processor Scalable Family Technical Overview*. Technical Report. Intel Corporation. arXiv:https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview

[16] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[17] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (Jan 2015), 2:1–2:21. https://doi.org/10.1147/JRD.2014.2376112