Optimal algebraic Breadth-First Search for sparse graphs

Paul Burkhardt [*]

October 4, 2019

Abstract

There has been a rise in the popularity of algebraic methods for graph algorithms given the development of the GraphBLAS library and other sparse matrix methods. An exemplar for these approaches is Breadth-First Search (BFS). The algebraic BFS algorithm is simply a recursion of matrix-vector multiplications with the $n \times n$ adjacency matrix, but the many redundant operations over nonzeros ultimately lead to suboptimal performance. Therefore an optimal algebraic BFS should be of keen interest especially if it is easily integrated with existing matrix methods.

Current methods, notably in the GraphBLAS, use a Sparse Matrix Sparse Vector (SpMSpV) multiplication in which the input vector is kept in a sparse representation in each step of the BFS. But simply applying SpMSpV in BFS does not lead to optimal runtime. Each nonzero in the vector must be masked in subsequent steps. This has been an area of recent research in GraphBLAS and other libraries. While in theory these masking methods are asymptotically optimal on sparse graphs, many add work that leads to suboptimal runtime. We give a new optimal, algebraic BFS for sparse graphs that is also a constant factor faster than theoretically optimal SpMSpV methods, closing a gap in the literature.

Our method multiplies progressively smaller submatrices of the adjacency matrix at each step, taking $O(m)$ algebraic operations on a sparse graph of $O(m)$ edges as opposed to $O(mn)$ operations of other sparse matrix approaches. Thus for sparse graphs it matches the bounds of the best-known sequential algorithm and on a Parallel Random Access Machine (PRAM) it is work-optimal. Compared to a leading Graph-BLAS library our method achieves up to 24x faster sequential time and for parallel computation it can be 17x faster on large graphs and 12x faster on large-diameter graphs.

Keywords: breadth-first search, graph algorithm, sparse matrix, linear algebra

1 Introduction

Breadth-First Search (BFS) is a principal search algorithm and fundamental primitive for many graph algorithms such as computing reachability and shortest paths. The sequential combinatorial algorithm is well-known [7] and attributed to the 1959 discovery by Moore [18]. The algorithm proceeds iteratively where in each step it finds the neighbors of vertices from the previous step such that these neighbors are also unique to the current step. Each step constructs a set of vertices that are not in other steps. These sets are the levels of the BFS tree and so the traversal is called level-synchronous. The algorithm takes $O(m+n)$ time due

[*]Research Directorate, National Security Agency, Fort Meade, MD 20755. Email: pburkha@nsa.gov

to referencing $O(n)$ vertices and testing $O(m)$ edges. To avoid cycles the algorithm must proceed one level at a time. Since the graph diameter is $D \leq n$ then there are D levels, hence the search is inherently sequential. As of yet, there is no sublinear-time, parallel algorithm for BFS that achieves $O(m + n)$ work.

Not long after Moore's discovery, Floyd and Warshall used matrix multiplication to solve transitive closure and shortest-path problems [12, 21] which are generalizations of Breadth-First Search. By labeling vertices $1..n$, a symmetric $n \times n$ adjacency matrix, A, can be constructed so that every nonzero element of the matrix denotes an edge, hence each column or row vector of this matrix describes the adjacency or neighborhood of a vertex. The linear algebraic BFS algorithm is simply a recursion of matrix-vector multiplications with this adjacency matrix and the previous multiplication product. It solves the $\vec{x}_{k+1} = A\vec{x}_k$ relation, where each matrix-vector product captures the next level in the search. Observe that this recurrence can be iterated to give $\vec{x}_{n+1} = A^n\vec{x}_1$ and therefore matrix exponentiation of A by repeated squaring [1] leads to a sublinear-time, parallel BFS, but requires $\Omega(n^3 \log n)$ work.

Suppose that A is a sparse matrix, then computing an algebraic BFS by $\vec{x}_{k+1} = A\vec{x}_k$ takes takes $O(mn)$ time because all $O(m)$ nonzeros in A are multiplied in all $O(n)$ steps. This Sparse Matrix-Vector (SpMV) approach is clearly wasteful because nonzeros in A will be multiplied by zeros in $\vec{x}$. Despite this, the appeal of the algebraic approach is due in part to the availability of highly optimized matrix libraries that are finely tuned to the computer architectures. These libraries take advantage of the memory subsystem and it is this low-level interaction with hardware that enables the algebraic BFS to be faster in practice than the theoretically optimal combinatorial algorithm. Practical implementations have provided measurable speedup over the combinatorial algorithm [2, 4, 5, 16]. These practical implementations rely on the level-synchronous sequential algorithm where the focus is on parallelizing the work within a level of the BFS tree.

Newer approaches employ a Sparse Matrix Sparse Vector (SpMSpV) multiplication where both the input vector and the matrix are in sparse format [1, 5, 22, 23]. Applying SpMSpV in BFS can still take $\Omega(mD)$ time because nonzeros reappear in $\vec{x}$, meaning every vertex is visited again, and so $\vec{x}$ becomes dense. This is readily observed given a long path connected to a clique where the search starts with a vertex in this clique. Thus it is not enough to treat both A and $\vec{x}$ as sparse, and consequently a straightforward SpMSpV method for BFS is not optimal.

Let $d(v)$ denote the degree or number of neighbors of vertex v. Then the column of A indexed by v has exactly $d(v)$ nonzeros, and on average there are $\bar{d} = 2m/n$ nonzeros in each column of A. In a single SpMSpV multiplication there are f nonzeros in the sparse vector and thus $\Omega(\bar{d}f)$ operations on average. Hiding or masking nonzeros over all steps in a BFS such that only f unique columns from A are accessed will then lead to $O(m)$ runtime, making it optimal for sparse graphs. But a theoretically optimal SpMSpV method for BFS will multiply each vertex v in A by $d(v)$ times, specifically the row for v, since every vertex will be the successor of its neighbors. Hence it makes $\sum_v d(v) = O(m)$ additional operations. Moreover, many of the new SpMSpV methods for BFS add work that degrade the performance to $O(mn)$ time.

We give an algebraic BFS that is optimal for sparse graphs. Our method multiplies progressively smaller submatrices of the adjacency matrix at each step, taking $O(m)$ algebraic operations as opposed to $O(mn)$ operations of other sparse matrix approaches. It is a constant factor faster than theoretically optimal SpMSpV methods. Our solution is

[1]Ex. $x^{17} = x \times x^{16} = x \times x^{4^{4}} = x \times x^{2^{2^{2^{2}}}}$

quite simple so it is surprising that it has been overlooked.[2] Our new method can be easily integrated with existing matrix methods, and may benefit the masking techniques in the GraphBLAS library [6, 9, 22], so we expect it would provide substantial value in practical settings.

We summarize our contribution in Section 3 and review the current SpMSpV approaches in Section 4. In Section 5 we define a new algebraic BFS by submatrix multiplication and analyze the asymptotic bounds on operations. Then in Section 6 we derive a linear transformation that demonstrates the relationship between our new method and the conventional matrix recursion for BFS. In Section 7 we describe our main algorithm and show that its performance matches the combinatorial algorithm and on a Parallel Random Access Machine (PRAM) it is work-optimal. We use the popular Compressed Sparse Row (CSR) format to demonstrate the theoretical contribution and the ease with which it can be integrated with existing sparse matrix methods. Experimental results of this are given in Section 10 where we demonstrate faster performance than the leading GraphBLAS library, achieving up to 24x faster sequential time and for parallel computation it can be 17x faster on large graphs and 12x faster on large-diameter graphs.

2 Notation

All following descriptions are for simple undirected graphs denoted by $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. The number of neighbors of a vertex is given by its degree $d(v) = |\{u|(u, v) \in E\}|$. Let $A \in \{0, 1\}^{n\times n}$ be the symmetric adjacency matrix for G. The vertices in G are labeled $[n] = \{1, 2, 3...n\}$. We denote $A[\alpha, \beta]$ as the submatrix of A with respect to subscripts in α, β, thus $A[(1, 2), (3, 4)]$ is the submatrix given by the rows $1, 2$ and columns $3, 4$ in A.

Correctness in the algebraic BFS requires only the distinction between zero and nonzero values and this also holds in our method. In addition to the Arithmetic semiring it is safe to use the Boolean (OR for addition, AND for multiplication) or Tropical min-plus semiring, both of which also avoid bit-complexity concerns. When appropriate, we'll denote the addition and multiplication operators by the conventional symbols, $\oplus$ and $\otimes$, respectively. To keep our discussion simple, we'll assume that any algebraic operation takes $O(1)$ time.

3 Our contribution

We give a new algebraic BFS by submatrix multiplication and show how it eliminates redundant operations so any element in the adjacency matrix is operated upon no more than once. We denote our algebraic BFS by the recursion $\vec{x}_{k+1} = A[V_k, V_k]\vec{x}_k$ where $A[V_k, V_k]$ is the submatrix of A with row and column indices given by the set V_k containing vertices not yet found in the search as of step k. The main theoretical result is then given by our first theorem.

Theorem 1. *Breadth-First Search can be computed by $\vec{x}_{k+1} = A[V_k, V_k]\vec{x}_k$ for $k = 1, \ldots, O(n)$ steps.*

We also show there is a linear transformation on the conventional algebraic recursion that produces our output. Given $\vec{y}_k = A\vec{y}_{k-1}$ for the conventional BFS recursion we then derive

[2]This author presented initial findings at an internal conference in 2015.

$\vec{x}_{k+1} = A[V_k, V_k]\vec{y}_k$ and ultimately $\vec{x}_{k+1} = A[V_k, V_k]A^{k-1}\vec{x}_1$. This linear transformation demonstrates the equality between our method and the conventional recursion and is given by Lemma 2.

Lemma 2. *There is a linear transformation on $\vec{y}_k = A\vec{y}_{k-1}$ that gives $\vec{x}_{k+1} = A[V_k, V_k]\vec{x}_k$, specifically $\vec{x}_{k+1} = A_k\vec{y}_k$ and subsequently $\vec{x}_{k+1} = A_k A^{k-1}\vec{x}_1$.*

We introduce a new algebraic BFS in Algorithm 1 that is optimal on sparse graphs. The adjacency matrix remains unchanged, rather we are masking the rows and columns in the matrix that corresponds to previously visited vertices. Hence the transpose element for a frontier vertex is not multiplied thereby reducing the references to half the number of nonzeros. The input vector in each step is also effectively masked so it is a sparse vector. Thus our method multiplies a sparse submatrix by a sparse vector in decreasing size each step. This leads to an asymptotic speedup over the conventional algebraic method. This holds for both sequential and parallel computation, thus it is work-optimal on a PRAM. Our main algorithmic results are the following.

Theorem 3. *Algorithm 1 computes an algebraic Breadth-First Search in $O(m+n)$ time for sparse G.*

Theorem 4. *Algorithm 1 computes an algebraic Breadth-First Search over t steps in $O(t)$ time and $O(m)$ work using $O(m/t)$ PRAM processors for sparse G.*

Our algebraic BFS algorithm on sparse graphs is deterministic and asymptotically optimal for any ordering of matrix and vector indices. The current state-of-the-art SpMSpV approaches are only optimal in BFS if the vector indices are unordered [1, 22]. It also appears that other recent SpMSpV methods take $O(mn)$ time overall for BFS because their masking method requires an elementwise multiplication with a dense vector or explicitly testing every vertex in each step [5, 22, 23]. Masking only the sparse vector requires more algebraic operations than our method since every vertex will be referenced by each of its successors, ostensibly the nonzero column indices in the row for that vertex in the sparse matrix, adding $O(m)$ operations to the runtime. We avoid this because we mask both the matrix and the vector, and so our submatrix multiplication method is a constant factor faster than theoretically optimal SpMSpV for BFS.

4 Related work

Using a SpMSpV for each step in a BFS is theoretically optimal for sparse graphs if only $O(n)$ unique columns from A are referenced for the search, and therefore $O(m)$ nonzeros are multiplied. But many of the current approaches add more work that degrades the runtime. In the following review of current SpMSpV algorithms for BFS, we ignore any work related to initialization, parallelization, or other overhead that do not affect the asymptotic complexity.

In an algebraic BFS on sparse graphs, the nonzeros from the multiplication must be written to a new sparse output vector. Using SpMSpV for the algebraic BFS then requires a multi-way merge due to the linear combination of either rows or columns of A that are projected by the nonzeros in the sparse vector in the multiplication. Strategies for efficient merging include using a priority queue (heap) or sorting, but this results in $\Omega(n \log n)$ runtime for BFS [23]. Another popular method is to employ a sparse accumulator (SPA) [1, 13] which is comprised of a dense vector of values, a dense vector of true/false flags, and an unordered list of indices to nonzeros in the dense vector. But it is stated in [1] that there is no known algorithm for SpMSpV that attains the lower-bound of $\Omega(\bar{d}f)$ if the indices in the sparse vector must be sorted. This is because the list of row indices in the SPA must be sorted if the sparse matrix was stored with ordered indices and the multiplication algorithm requires that ordering [13]. The SpMSpV methods using a SPA then take $\Omega(n \log n)$ time if the output vector needs sorted indices, making their use in a BFS non-optimal.

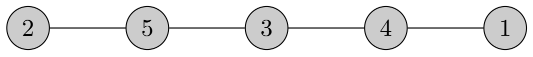
The focus of these new SpMSpV methods is in efficiently reading and writing the sparse vector. But there is an analysis gap on the asymptotic cost of preventing previous frontier vertices in the BFS from reappearing in the sparse vector. Masking out these frontier nonzeros was analyzed in [22] and it appears to require an elementwise multiplication with a dense masking vector which must be $O(n)$ size to accommodate all vertices. This suggests these SpMSpV methods with masking take $O(mn)$ time for BFS. In [5] an elementwise multiplication with a dense predecessor array is performed in each step of the BFS to mask the old frontier, leading to $O(mn)$ runtime. The SpMSpV method in [5] also required sorted output so either method of a priority queue or SPA leads to suboptimal time. The SpMSpV algorithm for BFS in [23] tests all vertices in each step and zeros out those in the output vector that have already been reached, leading to $\Omega(mn)$ time. A masked, column-based matrix-vector method for BFS that relies on radix sorting is given in [22] but takes $\Omega(m \log n)$ time. The authors allow unsorted indices to avoid the $\Omega(\log n)$ factor but elementwise multiplication with the dense masking vector results in $O(mn)$ time. Sorted vectors are also used in [1], thereby taking $\Omega(m \log n)$ time for BFS. A version with unsorted indices is given in [1] but the authors do not describe how visited vertices are avoided or masked.

5 Submatrix multiplication

Consider a path connected to a clique and suppose BFS starts with a vertex in this clique. There are $\Theta(D)$ steps in the search and $m > n$. Applying SpMSpV each step ignores zeros in $\vec{x}$ but observe that every new nonzero in the vector remains in each subsequent step. This leads to $\Omega(mD)$ runtime as follows. In the matrix-vector multiplication the $d(v)$ nonzeros for column v in A are accessed and multiplied. For all vertices this leads to $D \sum_{v \in V} d(v) = \Omega(mD)$ time. Since $D = \Theta(n)$ then SpMSpV for BFS over graphs with large diameter take $\Omega(mn)$ time. We illustrate repeating nonzeros in SpMSpV using a single matrix-vector multiplication. Recall that matrix-vector multiplication by the outer-product is the linear combination of the column vectors in the matrix scaled by the entries in the input vector as follows.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

We denote $A_{*,i}$ and $A_{i,*}$ as the i^{th} column and row vectors of A, respectively. The neighbors of vertex i are the nonzero elements in $A_{*,i}$. Here we show the dense matrix and vector for illustration only, so then all zeros can be ignored including those in the vectors. The linear combination of the $A_{*,2}$ and $A_{*,3}$ columns result in the nonzeros at $1, 2, 3, 4$ indices of the product vector. In BFS this corresponds to finding the neighbors $1, 3, 4$ of



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 & 1 \\ 4 & 1 & 0 & 1 & 0 & 0 \\ 5 & 0 & 1 & 1 & 0 & 0 \end{array} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{Step 1}$$

$$\begin{array}{c|cccc} & 1 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 1 & 1 \\ 4 & 1 & 1 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 \end{array} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{Step 2}$$

$$\begin{array}{c|ccc} & 1 & 3 & 4 \\ \hline 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 4 & 1 & 1 & 0 \end{array} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{Step 3}$$

$$\begin{array}{c|cc} & 1 & 4 \\ \hline 1 & 0 & 1 \\ 4 & 1 & 0 \end{array} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{Step 4}$$

Figure 1: Optimal algebraic BFS starting from vertex 2.

vertex 2 and neighbors $1, 2$ of vertex 3. The search continues by multiplying the matrix with this new product vector. The reader should note that in the next step the $A_{*,2}$ and $A_{*,3}$ columns are projected again resulting in redundant operations that do not add new vertices to the search. This will result in each vertex being revisited leading to $O(mn)$ time. Masking nonzeros in $\vec{x}$ prevents their recurrence and leads to $O(m)$ optimal time. But masking the vector alone still incurs twice as many algebraic operations than theoretically needed. For example, merely ignoring the $2, 3$ elements in this next input vector does not eliminate their recurrence because the columns $A_{*,1}, A_{*,4}$ will give $2, 3$ again in the following output vector. A theoretically optimal SpMSpV method for BFS will make $\sum_v d(v) = O(m)$ additional operations.

We notice that the computation can be performed over progressively smaller submatrices of A so any element in A is operated upon at most once, and that vertices are visited only once. Any new vertex i discovered in the search is because there exists $A(i, j) = A(j, i) \neq 0$. After the next step, we prevent i from appearing in subsequent matrix products by simply eliminating $A_{i,*}, A_{*,i}$ and the i^{th} element in $\vec{x}$ from all remaining matrix-vector products. This effectively removes i from the graph as illustrated in Figure 1. Then at each step the algebraic Breadth-First Search matrix multiplication is over the submatrix of A in which the row and column indices that remain are those not used in preceding steps. Namely the $A_{j,*}, A_{*,j}$ indexed by $x_k(j)$ nonzeros can be ignored for all remaining steps. Consequently the number of algebraic operations are reduced. We emphasize that A can be left unchanged, only the appropriate submatrices of A are needed in the computation. For sparse graphs the algebraic method will now be as efficient as the combinatorial BFS.

Definition 1. *Let V_k be the set of column indices in A at step k that does not collectively* *include indices to nonzeros in all $\vec{x}$ from previous steps. Thus V_k is given by $V_k = V_{k-1} \setminus support(\vec{x}_{k-1})$, where V_0 contains $\{1, 2, \ldots, n\}$ and $\vec{x}_0 = \vec{0}$.*

Theorem 1. *Breadth-First Search can be computed by $\vec{x}_{k+1} = A[V_k, V_k]\vec{x}_k$ for $k = 1, \ldots, O(n)$ steps.*

Proof. Recall the matrix-vector outer-product is a linear combination of column vectors in A,

$$\vec{x}_{k+1} = \bigoplus_j x_k(j) \otimes A_{*,j}.$$

Only nonzeros in $\vec{x}_k$ can produce nonzeros in the resultant vector $\vec{x}_{k+1}$ because any nonzero $x_{k+1}(i)$ vector element is due to a nonzero $x_k(j) \otimes A(i,j)$ product. Also observe that a $A_{*,j}$ column vector can only produce a nonzero $x_{k+1}(i)$ if $A(i,j)$ is nonzero. Thus, subsequent operations on the $A_{*,j}$ column vector do not produce new $x_{k+1}(i)$ nonzeros. For Breadth-First Search this does not update a new level. Then for each j in $support(\vec{x}_k)$ at step k, the $A_{j,*}, A_{*,j}$ can be ignored in all remaining steps leading to $\vec{x}_{k+1} = A[V_k, V_k]\vec{x}_k$ as claimed. $\square$

Lemma 1. *In computing Breadth-First Search by Theorem 1, each column vector in A is multiplied at most once and subsequently each element of A is operated upon no more than once.*

Proof. Only nonzeros are required in Breath-First Search so the matrix-vector product in Theorem 1 is computed from only the j indices in V_k by

$$x_{k+1}(i) = \bigoplus_{j \in V_k \cap support(\vec{x}_k)} x_k(j) \otimes A(i,j).$$

Now for all remaining steps since V_k does not contain indices from V_{k-1} then the multiplication is over the submatrix $A[V_k, V_k]$ that does not include $A_{*,j}, A_{j,*}$. Then there cannot be a vertex i at some later step that produces j by $x_{k+1}(j) = x_k(i) \otimes A(j,i)$ because all $A_{j,*}$ are prohibited. Thus each $A_{*,j}$ column vector can be multiplied only once and subsequently any element in A is accessed no more than once. $\square$

At each step, masking $A_{*,j}$ for all j from the previous frontier prohibits rediscovery because it prunes (or hides) the (i,j) edges that would lead to discovering j vertices again when the i neighbors are the frontier vertices. Likewise masking $A_{j,*}$ hides the (j,i) edges that have already led to the i neighbors of j. Thus only half the nonzeros in A are referenced in the overall search and each is accessed at most once.

Computing BFS by Theorem 1 reduces the algebraic operations compared to simply computing the recurrence over $A\vec{x}$ because there will be at least one nonzero in $\vec{x}$ at each step until completion. Hence the size of the submatrices $A[V_k, V_k]$ strictly decreases with n. The path graph in Figure 1 is a worst-case example where only one (row, column) pair of A is ignored at each subsequent step, yet this is still an asymptotic improvement over multiplying A in full over all steps. For most graphs the reduction in operations is far more dramatic since there can be many nonzeros in $\vec{x}$ at each step.

When G is a sparse graph the matrix product can be accomplished using sparse matrix representations so that matrix-vector multiplication operates on just the nonzero elements in the matrix. Since there are two algebraic operations for each of the $2m$ nonzeros, it takes

$4km = O(mn)$ algebraic operations for the conventional algebraic BFS. This can be reduced by our method.

Theorem 2. *The algorithm of Theorem 1 computes an algebraic Breadth-First Search on a sparse graph G in $2m$ algebraic operations.*

Proof. In sparse matrix multiplication only nonzero entries are operated upon. Lemma 1 establishes that each entry in A is operated upon at most once. Moreover, since a column vector in A and its transpose are masked, then only half the nonzeros in A can be referenced. Hence there are m multiplications and m additions, for a total of $2m$ algebraic operations. $\square$

This is $\Omega(n)$ improvement over the conventional method. It isn't difficult to see there will also be an asymptotic improvement over the classic dense matrix-vector approach.

6 Linear transformation

Before we describe an algorithm for Theorem 1 we will show that it is possible to get our output vector at step $k+1$ by a linear transformation of the k step output vector from the conventional algebraic recursion. Namely, we can derive $\vec{x}_{k+1} = A[V_k, V_k]\vec{y}_k$ where we use $\vec{y}_k = A\vec{y}_{k-1}$ to denote the conventional BFS recursion. This leads to $\vec{x}_{k+1} = A[V_k, V_k]A^{k-1}\vec{x}_1$. Thus we can show the equality between our method and the conventional recursion. We begin with the following definitions and claims.

Definition 2. *Let $A_k \in \{0,1\}^{n\times n}$ be a Boolean symmetric matrix such that $A_k(i,j) = 1$ if $A(i,j) = 1$ and $i,j \in V_k$, and is zero otherwise.*

The size of A_k does not change at each step, only the $A_{*,i}, A_{i,*}$ are zeroed out for those i not in V_k. Then A_1 is the adjacency matrix A.

Definition 3. *Let $S_k \in \{0,1\}^{n\times n}$ be a* selection matrix *that is a Boolean diagonal matrix such that $S_k(i,j) = 1$ if $i = j$ and $i,j \in V_k$, and is zero otherwise. Since it is diagonal with some elements being zero, S_k is therefore an idempotent matrix.*

A *selection matrix* is a Boolean diagonal matrix that is used to mask or zero out rows/columns of some other matrix. A selection matrix S that is not the Identity will have $\vec{0}$ for some row and column vectors. Then by the usual rules of matrix multiplication, multiplying a matrix A on the right by S will inherit the $\vec{0}$ column vectors in S, and multiplying A on the left by S inherits the $\vec{0}$ rows in S. A symmetric selection on a matrix A is then given by SAS, which returns a new matrix with the same dimensions of A containing only the $A_{*,i}, A_{i,*}$ corresponding to nonzero $S_k(i,i)$ diagonal elements, and all other rows/columns are zeroed. For example, if all diagonal elements in S were one except for $S(2,2)$, then SAS returns A with $A_{*,2}$ and $A_{2,*}$ as the $\vec{0}$ zero vector.

Claim 1. *The equality $S_{k+1}S_k = S_{k+1}$ holds for $k \geq 1$.*

Proof. By Definition 3 the nonzeros in S_{k+1} must be in S_k and so these rows are identical. Then multiplying S_k on the left by S_{k+1} annihilates the rows in S_k indexed by $\vec{0}$ row vectors in S_{k+1}, hence the product $S_{k+1}S_k$ must return S_{k+1}. $\square$

Claim 2. *The equality $A_{k+1} = S_{k+1}A_kS_{k+1}$ holds for $k \geq 1$.*

Proof. We prove this by induction. In the base step $A_2 = S_2 A_1 S_2$ holds because $S_2(i,i)$ is zero for the source vertex i and hence symmetric selection annihilates $A_{*,i}$ and $A_{i,*}$ to give $A[V_2, V_2] = A_2$ which satisfies Definition 2. Then $S_3 A_2 S_3$ gives $A[V_3, V_3] = A_3$ by the same definition.

Now assume $A_k = S_k A_{k-1} S_k$ is true for all steps $1..k$. Since S_{k+1} masks out the nonzeros from $\vec{x}_{k+1}$ and all previous $\vec{x}_k$ have already been masked in A_k, then $S_{k+1} A_k S_{k+1}$ gives $A_{k+1} = A[V_{k+1}, V_{k+1}]$. □

Claim 3. *The equality $A_k = S_k A S_k$ holds for $k \geq 1$.*

Proof. We prove this by induction. In the base step the claim follows trivially for $A_1 = S_1 A S_1$ since S_1 is the Identity. Now in the inductive step, assume $A_k = S_k A S_k$ holds. Then applying Claims 1 and 2 gives $A_{k+1} = S_{k+1} A S_{k+1}$ as follows.

$$\begin{aligned} A_{k+1} &= S_{k+1} A_k S_{k+1} && \text{(Claim 2)} \\ &= S_{k+1}(S_k A S_k) S_{k+1} && \text{(Induction)} \\ &= S_{k+1} A S_{k+1} && \text{(Claim 1)} \end{aligned}$$

□

Lemma 2. *There is a linear transformation on $\vec{y}_k = A\vec{y}_{k-1}$ that gives $\vec{x}_{k+1} = A[V_k, V_k]\vec{x}_k$, specifically $\vec{x}_{k+1} = A_k \vec{y}_k$ and subsequently $\vec{x}_{k+1} = A_k A^{k-1} \vec{x}_1$.*

Proof. We first show that $S_k \vec{x}_k$ is equal to $S_k \vec{y}_k$. Here $\vec{y}_k$ and $\vec{x}_k$ contain nonzeros that have not been produced by previous steps. It follows from Theorem 1 that $\vec{x}_k$ contains only such nonzeros. Now $S_k \vec{y}_k$ annihilates the nonzeros in $\vec{y}_k$ that are not in V_k, hence $S_k \vec{y}_k$ is equal to $\vec{x}_k$. Since S_k is idempotent then $S_k \vec{x}_k = S_k(S_k \vec{y}_k) = S_k \vec{y}_k$. Using this equality and Claim 3 we can show that $A_k \vec{y}_k$ is equal to $A_k \vec{x}_k$.

$$\begin{aligned} A_k \vec{y}_k &= S_k A S_k \vec{y}_k && \text{(Claim 3)} \\ &= S_k A S_k \vec{x}_k \\ &= A_k \vec{x}_k && \text{(Claim 3)} \end{aligned}$$

This leads to $\vec{x}_{k+1} = A_k \vec{x}_k = A_k \vec{y}_k$. Iteration on $\vec{y}_k = A\vec{y}_{k-1}$ yields $\vec{y}_k = A^{k-1}\vec{y}_1$. Since the source vector is the same for this conventional recursion and that of Theorem 1, then $\vec{x}_1 = \vec{y}_1$, giving the result $\vec{x}_{k+1} = A_k \vec{y}_k$ and $\vec{x}_{k+1} = A_k A^{k-1} \vec{x}_1$ as claimed. □

We emphasize that the result of Lemma 2 supposes that a chosen semiring is applied consistently. If the arithmetic semiring was used to produce $\vec{x}_{k+1}$ then it must be used to compute $A_k A^{k-1} \vec{x}_1$.

7 Optimal algorithm

We now give our main algorithm for sparse graphs that employs the new method given by Theorem 1. Our new Algorithm 1 does not specify a sparse matrix format to be as general as possible.

Algorithm 1

```
Require: $\Gamma(i)$    ▷ array for each vertex $i$ holding subscripts $j$ such that $A(i,j)$ is nonzero
    Initialize $V_1$ with $1, 2, ..n$
    Initialize $\vec{x}_1$ with the source vertex
1: for $k = 1, 2, \ldots$ until end of component do
2:     for all $i \in support(\vec{x}_k)$ do
3:         for all $j \in \Gamma(i)$ and $j \in V_k$ do
4:             $x_{k+1}(j) \leftarrow x_{k+1}(j) \oplus x_k(i) \otimes A(i,j)$
5:             mark $i, j$ so these will not be in $V_{k+1}$    ▷ Achieves $V_{k+1} = V_k \setminus support(\vec{x}_k)$
```

Theorem 3. *Algorithm 1 computes an algebraic Breadth-First Search in $O(m+n)$ time for sparse G.*

Proof. The algorithm computes Breadth-First Search by Theorem 1. At each step k only the nonzero indices in $\vec{x}_k$ are involved in the multiplication with the submatrix $A[V_k, V_k]$. The submatrix is given by those j indices in each $\Gamma(i)$ that were not removed from V_k. These j are the column indices in A. Each new j is then marked so it will be ignored in subsequent steps. Each i index for nonzeros in $\vec{x}_k$ is also marked and will not be used in subsequent calculations. It then follows from Lemma 1 that each column in A is multiplied once. Since there are $O(n)$ possible values for j, then $O(n)$ rows of A are accessed for all steps and subsequently $O(m)$ nonzero entries are involved in the computation. Hence in accordance with Theorem 2 there are $O(m)$ algebraic operations.

The total time is then as follows. There are $O(n)$ entries in total added to all $\vec{x}_k$, taking $O(n)$ time. This is possible by storing each new j from the inner loop in a separate array that can be iterated over in each step. There are $O(m)$ tests overall to determine the j that are in V_k. This takes $O(m)$ time, possibly by storing each j in another array of size n so in $O(1)$ time it can be determined if a j has been previously used. Iterating over all nonzeros takes $O(m)$ time because each row in A is accessed at most once. Each algebraic operation takes $O(1)$ time so all algebraic operations takes $O(m)$ time. Therefore in total it takes $O(m+n)$ time. ☐

Algorithm 1 is also work-optimal on a PRAM under the Work-Depth (WD) model [3, 15, 19]. In this model a **for all** construct denotes a parallel region in which instructions are performed concurrently. All other statements outside this construct are sequential. The scheduling of processors is handled implicitly and an arbitrary number of simultaneous operations can be performed each step. The work is the sum of actual operations performed and the number of processors is not a parameter in the WD model. The depth is the longest chain of dependencies, often the number of computation steps, and is denoted by D. The work-complexity, W, denotes the total number of operations. A parallel algorithm in WD can be simulated on a p-processor PRAM in $O(W/p + D)$ time and using $p = W/D$ processors achieves optimal work on the PRAM.

Theorem 4. *Algorithm 1 computes an algebraic Breadth-First Search over t steps in $O(t)$ time and $O(m)$ work using $O(m/t)$ PRAM processors for sparse G.*

Proof. The algorithm computes Breadth-First Search by Theorem 1. A processor is assigned to a nonzero in the sparse matrix representation, so there are $O(m)$ processors for the edges. At each step k every processor reads its $x_k(i)$ value and performs the algebraic operations for the matrix product and writes out the new $x_{k+1}(j)$ value, specifically the processor performs,

$$x_{k+1}(j) \leftarrow x_{k+1}(j) \oplus x_k(i) \otimes A(i,j).$$

A processor writing the $x_{k+1}(j)$ value also updates $V_{k+1}[j]$ so it can be masked out of the calculation in subsequent steps. Each i index for nonzeros in $\vec{x}_k$ is also marked. Hence only the submatrix $A[V_k, V_k]$ is used in the computation at each step k. Observe that concurrent writes to the same j index in x_{k+1} do not change the result and so arbitrary write resolution suffices. The same holds for updating i, j in V_{k+1}.

Under the Work-Depth (WD) model there are as many processors as needed to compute a single step k in $O(1)$ time. Here the processors perform just the necessary calculations to produce the next submatrix each step, hence the work is the sum of actual operations that are performed over all the steps. By Lemma 1 each column in A is multiplied once. Since there are $O(n)$ possible values for j, then $O(n)$ rows of A are accessed for all steps and subsequently $O(m)$ nonzero entries are involved in the computation overall. Theorem 2 establishes there are $O(m)$ algebraic operations overall. The work is then $O(m)$ and it follows from Brent's Theorem that it takes $O(m/p + t)$ total time and $O(m + pt)$ work on a PRAM. Therefore it takes $O(t)$ time and $O(m)$ work using $O(m/t)$ processors on a PRAM. $\square$

Our Algorithm 1 is an optimal algebraic BFS algorithm for sparse graphs that is deterministic and does not depend on ordering or lack of ordering in the matrix and vector indices. Although this algorithm fills a gap in the study of BFS, it offers no theoretical advantage over the simple combinatorial algorithm. However, we believe it offers a practical advantage because of optimized matrix multiplication methods. It is also a constant factor faster than a theoretically optimal SpMSpV method. Our technique of hiding or "masking" portions of A and the input vector could benefit new algebraic graph libraries that already feature "masked" sparse linear algebra operations [6, 9, 22], specifically for SpMSpV. Next we'll demonstrate how to easily integrate our method in the popular Compressed Sparse Row (CSR) format which is used in many sparse matrix libraries [11, 14, 20].

8 Practical optimal sequential algorithm

The CSR format is a well-known sparse matrix representation that utilizes three arrays, *nz*, *col*, and *row*, to identify the nonzero elements. The *nz* array holds the nonzero values in row-major order. The *col* array contains the column indices for nonzeros in the same row-major order of *nz*, and *row* is an array of *col* indices for the first nonzero in each row of the matrix. The last value in *row* must be one more than the last *col* index. A matrix-vector multiplication in CSR iterates over the gap between successive values of the *row* array to access each nonzero in a row. We give the basic CSR matrix-vector multiplication in Listing 1.

Listing 1 Matrix-vector multiplication in CSR

```
Require: $nz$,$col$,$row$                         ▷ CSR data structures
Require: $\vec{x}, \vec{y}$                       ▷ input and output vectors
  for $i = 1..n$ do
      for $j = row[i]..row[i+1]-1$ do
          $y(i) \leftarrow y(i) \oplus nz[j] \otimes x(col[j])$
```

Our Algorithm 2 requires a simple modification to CSR matrix-vector multiplication. We add an array, T, to store nonzero indices for vertices that have been visited. This is used to limit the multiplication over the appropriate submatrix of A every step. At each step we iterate over only nonzeros in $\vec{x}$, the indices of which are stored in another array L. We could have used a sparse vector representation for $\vec{x}$ each step, but for simplicity we just increment a pointer in L.

Algorithm 2

```
Require: $nz$,$col$,$row$                         ▷ CSR data structures
Require: $\vec{x}, \vec{y}$                       ▷ input and output vectors
Require: $T, L$                                   ▷ arrays of size n
    Initialize $\vec{x}$ and $L$ with source vertex
 1: set $start := 0$ and $end := 1$ and $z := end$
 2: for $k = 1, 2, \ldots$ until end of component do
 3:     for $i = L[start]..L[end]$ do
 4:         for $j = row[i]..row[i+1]-1$ do
 5:             if $T[col[j]]$ is 0 then
 6:                 $y(col[j]) \leftarrow y(col[j]) \oplus nz[j] \otimes x(i)$
 7:                 set $L[z] := col[j]$ and $T[col[j]] := 1$
 8:                 set $z := z + 1$
 9:         set $T[i] := 1$ and $x(i) := 0$
10:     set $start := end$ and $end := z$
11:     exchange pointers between $\vec{x}$ and $\vec{y}$
```

Each column index $col[j]$ from $row[i]$ up to $row[i+1]$ is tested if it refers to a vertex that has already been visited. If not, the CSR matrix-vector product is performed. The $col[j]$ are added to L and marked as visited in T. Each new entry in the product vector $\vec{y}$ must be nonzero because only nonzero operands are used in the multiplication.

Observe that each $col[j]$ is immediately marked *visited*. This avoids redundantly operating over it in the same step because a nonzero in a column can appear in more than one row of the adjacency matrix. It also ensures there are n unique entries in the array L. The total number of algebraic operations is $2(n-1)$ rather than $2m$. If desired, all $col[j]$ in a step can contribute to the output by simply adding a test if $T[i]$ is zero before the inner loop over the row array, then removing the update to $T[col[j]]$ and allowing L to take $O(m)$ values.

We have previously expressed matrix-vector multiplication as a linear combination of column vectors. To align this with the row-oriented CSR computation we just take the linear combination over the row vectors since A is symmetric. Hence we are computing the transpose matrix-vector multiplication at each step.

Since Algorithm 2 is based on Algorithm 1 and does not add any new operations, then Theorem 2 follows immediately.

Theorem 5. *Algorithm 2 computes an algebraic Breadth-First Search in $O(m+n)$ time for sparse G.*

It isn't difficult to see that Algorithm 2 is very similar to the combinatorial algorithm. Our intent here is to demonstrate that existing sparse matrix methods require only a simple adaptation to achieve the optimality of the combinatorial algorithm while maintaining their practical advantages. We expect that our method could benefit new graph libraries such as GraphBLAS that already support masking sparse linear algebraic operations [6, 9, 22]. Our main algorithmic result is given in Algorithm 1 since it captures the general concept given in Theorem 1 and therefore is amenable to many forms of sparse matrix and sparse vector implementations.

9 Practical work-optimal parallel algorithm

We give parallel, work-optimal CSR algorithm in Algorithm 3 that is both simple and practical. Aside from the use of **for all** over the L and *row* arrays, a significant difference between this algorithm and Algorithm 2 is that we have to avoid adding duplicates to the L array. During the search two processors at level k can find the same new vertex $u = col[j]$ at level $k+1$. Concurrent write to $y(col[j])$ at line 6 does not pose a problem because the same value is written by any processor. But we have to avoid adding u to L more than once. This can be handled with a new parent array P so then processors concurrently write to $P[u]$ their frontier vertex v that reached u. Each processor then reads $P[u]$ and the processor that has a matching parent vertex for u adds u to L. Since only one processor can win the write the duplicates are avoided.

Algorithm 3

```
Require: $nz, col, row$                                   ▷ CSR data structures
Require: $\vec{x}, \vec{y}$                                ▷ input and output vectors
Require: $T, L, P$                                        ▷ arrays of size n
    Initialize $\vec{x}$ and $L$ with source vertex
 1: set $start := 0$ and $end := 1$ and $z := end$
 2: for $k = 1, 2, \ldots$ until end of component do
 3:     for all $i = L[start]..L[end]$ do
 4:         for all $j = row[i]..row[i+1]-1$ do
 5:             if $T[col[j]]$ is 0 then
 6:                 $y(col[j]) \leftarrow y(col[j]) \oplus nz[j] \otimes x(i)$
 7:                 set $P[col[j]] := i$
 8:                 if $P[col[j]]$ is $i$ then
 9:                     set $L[z] := col[j]$ and $T[col[j]] := 1$
10:                     set $z := z + 1$
11:         set $T[i] := 1$ and $x(i) := 0$
12:     set $start := end$ and $end := z$
13:     exchange pointers between $\vec{x}$ and $\vec{y}$
```

Algorithm 3 has the same work and depth as Algorithm 1 so Theorem 3 follows.

Table 1: Test Graphs

	n (vertices)	m (edges)	D (diameter)
roadNet-TX	1,379,917	1,921,660	1057
roadNet-CA	1,965,206	2,766,607	854
roadNet-PA	1,088,092	1,541,898	787
com-Amazon	334,863	925,872	44
com-Youtube	1,134,890	2,987,624	20
com-LiveJournal	3,997,962	34,681,189	17
com-Orkut	3,072,441	117,185,083	9

Theorem 6. *Algorithm 3 computes an algebraic Breadth-First Search in t steps and $O(m)$ work using $O(m/t)$ processors for sparse G.*

10 Experiments

We show that Theorem 1 leads to significant savings in algebraic operations on real-world graphs. We compare Algorithm 2 to simple CSR implementations for BFS using dense vector (SpMV), sparse vector (SpMSpV), and masked sparse vector (SpMmSpV). In each of the BFS implementations in our tests, the visited vertices in a search are tracked in a similar manner as in Algorithm 2. In the next descriptions we denote these test implementations as follows. Let SpMV-BFS denote the dense vector method, SpMSpV-BFS for the sparse vector method, and finally SpMmSpV-BFS for a "masked" sparse vector method. In SpMV-BFS all nonzeros in A are multiplied each step. In SpMSpV-BFS we only count algebraic operations due to nonzeros in the sparse vector. The SpMmSpV-BFS is nearly identical to Algorithm 2 but tests visited frontier vertices after the algebraic operations, and is therefore an asymptotically optimal "masked" sparse vector method.

Listed in Table 1 are graphs from the Stanford Network Analysis Project (SNAP) [17] used in the experiments.[3] We chose a source vertex for each graph such that BFS is run for the entire reported diameter.[4] We count two algebraic operations for the $(\oplus, \otimes)$-semiring operations in the inner loop of the CSR multiplication. A comparison of the methods is illustrated by a log-scale histogram in Figure 2 where it is clear that Algorithm 2 requires orders of magnitude fewer algebraic operations than the other methods with the exception of the masked sparse vector approach.

It follows from Lemma 1 and Theorem 2 that an optimal algebraic BFS should take $2m$ algebraic operations on a sparse graph because rows and columns in A are masked, hence the transpose element in A for a frontier vertex is not multiplied. This is expected for our Algorithm 2. Then SpMmSpV-BFS should take $4m$ operations as we claimed in Sections 1 and 5. At worst, SpMV-BFS multiplies all $2m$ nonzeros in A every step leading to $4mK$ operations in K steps. This prediction bears out in the experimental results listed in Table 2. But recall that Algorithm 2 takes $2(n-1)$ algebraic operations in total because it ignores repeated nonzeros from the same row. A simple modification was described in Section 8 to include all nonzeros in a row. We tested this modification in the experiments and verified that it leads to $2m$ algebraic operations on each of the graphs; therefore Algorithm 2 at worst takes half the number of operations as SpMmSpV-BFS.

[3]Diameters may differ from SNAP due to random sampling.
[4]The number of BFS steps is one more than the diameter.

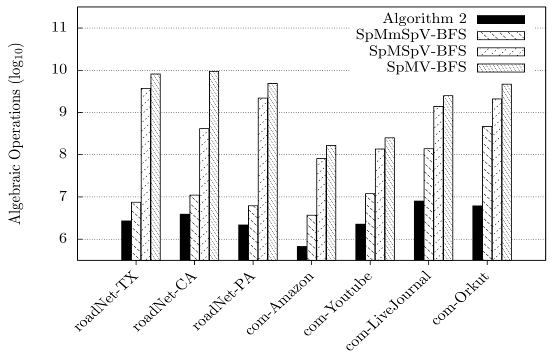


Figure 2: Comparison of total algebraic operations.

Table 2: Total Algebraic Operations

	Algorithm 2	SpMmSpV-BFS	SpMSpV-BFS	SpMV-BFS
roadNet-TX	2,702,272	7,516,804	3,739,129,896	8,132,465,120
roadNet-CA	3,914,052	11,041,552	4,134,833,458	9,461,795,940
roadNet-PA	2,175,122	6,166,056	2,192,455,678	4,860,062,496
com-Amazon	669,724	3,703,488	80,934,254	166,656,960
com-Youtube	2,269,778	11,950,496	136,126,648	250,960,416
com-LiveJournal	7,995,922	138,724,756	1,391,162,196	2,497,045,608
com-Orkut	6,144,880	468,740,332	2,093,464,344	4,687,403,320

The savings in algebraic operations is most pronounced in large diameter graphs. For the roadNet-TX graph the SpMV-BFS method over $D + 1$ steps takes $4m(D + 1) = 8, 132, 465, 120$ algebraic operations, which is 3000× more operations than our method. The operations by SpMSpV-BFS also grow linearly with D, taking about half as many operations in comparison to SpMV-BFS as evident in Table 2. Thus as we had claimed, using a sparse vector alone takes $\Omega(mD)$ time. Both SpMmSpV-BFS and our method are efficient but SpMmSpV-BFS makes twice as many operations as our algorithm, and hence our method is significantly more practical. Next we'll compare the sequential and parallel runtime performance of our approach with that of the GraphBLAS library.

We compared our algorithm implementations against the SuiteSparse GraphBLAS library [8–10]. This is considered a complete reference implementation and for convenience we will refer to it simply as GraphBLAS. This library implements SpMmSpV-BFS using a masked sparse vector [10, c.f. UserGuide pg. 191]. We use version 2.2.2 and 3.0.1 of GraphBLAS for the sequential and parallel tests, respectively. The parallel BFS implementations of both our method and GraphBLAS use OpenMP multithreading. The experiments were run on a single workstation with over 100 GB of RAM and 28 Intel Xeon E5-2680 cores. Each BFS starts at vertex 0 so the absolute diameter is less than those reported in Table 1. We use T_{GB} to denote the runtime for GraphBLAS and T_{Alg} for the runtime of either Algorithm 2 or Algorithm 3. In the plots to follow the performance comparison is given as the ratio T_{GB}/T_{Alg} so if T_{Alg} is faster then the ratio is greater than one and will appear higher on the y-axis. In all experiments our sequential and parallel implementations were faster than GraphBLAS, often by over an order of magnitude.

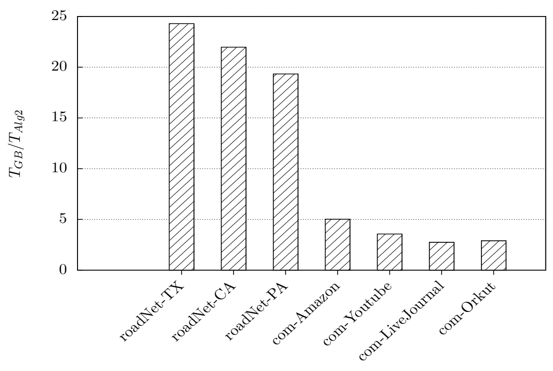


Figure 3: Comparison of sequential runtime.

We remark that the implementation of Algorithm 3 using OpenMP avoids duplicates by keeping a local array for each thread where a thread can store the neighbors of its subset of frontier vertices. Each thread iterates over its local array and atomically updates the T array and removes any of its local vertices that had been visited. The threads then update L in batch by computing individual offsets in L atomically. There can be only $O(m)$ duplicates overall so our OpenMP implementation remains work-optimal asymptotically, but performs more algebraic operations than the sequential implementation of Algorithm 2 and this may account for the gap in performance with respect to the experimental results for the sequential algorithm.

A comparison of sequential runtime is given in Figure 3. Our BFS is 19-24x faster than GraphBLAS on the road network graphs, and on average it is 22x faster. These graphs have large diameter so inefficiencies are compounded with each step. The gains for the lower diameter graphs are more modest but on average we are still more than 3.5x faster. In these graphs the work is more concentrated within each level of the BFS rather than distributed over the total number of levels.

The parallel runtime is given in Figure 4 where it is also evident that our method is considerably faster than GraphBLAS. In two-thirds of the tests our algorithm completed in less than one-tenth of a second, and took over one second for just one test where two threads were used on the largest graph, com-Orkut, which took 1.59 seconds. At 64 threads on the two largest graphs, com-Orkut and com-LiveJournal, our BFS completes in 0.153 and 0.082 seconds resulting in 11x and 17x faster time than GraphBLAS, respectively. For com-Orkut we achieve over 1.5 GTEPS (Giga Traversed Edges Per Second). Also notable on these large graphs is that our performance with respect to GraphBLAS scales linearly. For the large diameter graphs our performance peaks with fewer threads, again because the per level work is very low so the overhead of adding more threads becomes significant. But we are still about 9-12x faster than GraphBLAS at our peak for these graphs.

With appropriately tuned libraries, an optimal algebraic BFS could out-perform the combinatorial BFS in practice. Moreover, our approach saves a factor of two in algebraic operations over a theoretically optimal SpMmSpV-BFS. Thus even for relatively low-diameter graphs, the constant-factor speedup is significant for graphs with many edges. We believe our approach can be easily integrated with existing matrix methods, and benefit those that support masking operations.

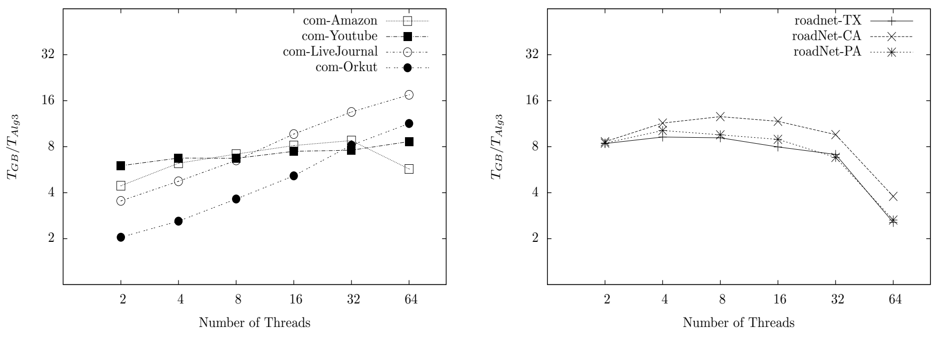


Figure 4: Comparison of parallel runtime.

Acknowledgments

The author thanks David G. Harris and Christopher H. Long for their helpful comments.

References

[1] A. Azad and A. Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium*, IPDPS'17, pages 688–697, May 2017.

[2] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. SlimSell: A vectorizable graph representation for breadth-first search. In *2017 IEEE International Parallel and Distributed Processing Symposium*, IPDPS'17, pages 32–41, 2017.

[3] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[4] H. M. Bücker and C. Sohr. Reformulating a breadth-first search algorithm on an undirected graph in the language of linear algebra. In *2014 International Conference on Mathematics and Computers in Sciences and in Industry*, pages 33–35, 2014.

[5] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, pages 65:1–65:12, 2011.

[6] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *Parallel and Distributed Processing Symposium Workshops*, GABB17. IEEE International, 2017.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[8] T. A. Davis. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. In *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference*, HPEC'18, pages 1–6, 2018.

[9] T. A. Davis. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the languge of sparse linear algebra. *ACM Transactions on Mathematical Software*, 2019. To appear.

[10] T. A. Davis. SuiteSparse:GraphBLAS. `http:faculty.cse.tamu.edu/davis/GraphBLAS.html`, 2019.

[11] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. Technical report, University of Tennessee, 1994.

[12] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345–, 1962.

[13] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal of Matrix Analysis and Applications*, 13(1):333–356, 1992.

[14] Intel. *Intel Math Kernel Library. Reference Manual.* Intel Corporation, Santa Clara, USA, 2009. ISBN 630813-054US.

[15] J. JaJa. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[16] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, 2011.

[17] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[18] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[19] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel Max-Flow algorithm. *Journal of Algorithms*, 3:128–146, 1982.

[20] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(11):521, 2005.

[21] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[22] C. Yang, A. Buluç, and J. D. Owens. Implementing push-pull efficiently in GraphBLAS. In *Proceedings of the International Conference on Parallel Processing*, ICPP 2018, pages 89:1–89:11, 2018.

[23] C. Yang, Y. Wang, and J. D. Owens. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW'15, pages 841–847, 2015.