## Title

A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm

Authors
Azad, Ariful
Buluç, Aydin

Publication Date
2017-05-01

DOI
10.1109/ipdps.2017.76

Peer reviewed

# A work-efficient parallel sparse matrix-sparse vector multiplication algorithm

Ariful Azad, Aydın Buluç
{azad,abuluc}@lbl.gov
Computational Research Division
Lawrence Berkeley National Laboratory

***Abstract*—We design and develop a work-efficient multithreaded algorithm for sparse matrix-sparse vector multiplication (SpMSpV) where the matrix, the input vector, and the output vector are all sparse. SpMSpV is an important primitive in the emerging GraphBLAS standard and is the workhorse of many graph algorithms including breadth-first search, bipartite graph matching, and maximal independent set. As thread counts increase, existing multithreaded SpMSpV algorithms can spend more time accessing the sparse matrix data structure than doing arithmetic. Our shared-memory parallel SpMSpV algorithm is work efficient in the sense that its total work is proportional to the number of arithmetic operations required. The key insight is to avoid each thread individually scan the list of matrix columns.**

**Our algorithm is simple to implement and operates on existing column-based sparse matrix formats. It performs well on diverse matrices and vectors with heterogeneous sparsity patterns. A high-performance implementation of the algorithm attains up to 15x speedup on a 24-core Intel Ivy Bridge processor and up to 49x speedup on a 64-core Intel KNL manycore processor. In contrast to implementations of existing algorithms, the performance of our algorithm is sustained on a variety of different input types include matrices representing scale-free and high-diameter graphs.**

## I. Introduction

Sparse matrix-sparse vector multiplication (SpMSpV) is an important computational primitive with many applications in graph algorithms and machine learning. The SpMSpV operation can be formalized as $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ where a sparse matrix $\mathbf{A}$ is multiplied by a sparse vector $\mathbf{x}$ to produce a (potentially sparse) vector $\mathbf{y}$. Due to lack of applications in traditional scientific computing, the research community has not paid much attention to computing SpMSpV efficiently. It is possible to interpret SpMSpV as a special case of sparse matrix-matrix multiplication where the second matrix has dimensions $n \times 1$. While this formulation can be relatively efficient for computing SpMSpV sequentially, for example by using Gustavson's SpGEMM algorithms [1], it is not a good fit for computing SpMSpV in parallel. This is because there is often little work in each SpMSpV operation, necessitating novel approaches in order to scale to increasing thread counts.

The computational pattern in many graph algorithms involves transforming a set of active vertices (often called "the current frontier") to a new set of active vertices (often called the "next frontier"). Such graph algorithms, which are often called "data-driven' algorithms" [2], are harder to parallelize because the work changes dynamically as the algorithm proceeds and there is often very little work per transformation. This "frontier expansion" pattern is neatly captured by the SpMSpV primitive: the current frontier is represented with the input vector $\mathbf{x}$, the graph is represented by the matrix $\mathbf{A}$ and the next frontier is represented by $\mathbf{y}$. For this reason, SpMSpV is the workhorse of many graph algorithms that are implemented using matrix primitives, such as breadth-first search [3], maximal independent sets [4], connected components [5], and bipartite graph matching [6]. This makes SpMSpV one of the most important primitives in the upcoming GraphBLAS [7] standard (http://graphblas.org).

Even seemingly more regular graph algorithms, such as PageRank, are better implemented in a data-driven way using the SpMSpV primitive as opposed to using sparse matrix-dense vector multiplication. This is because SpMSpV allows marking vertices "inactive" using the sparsity of the input vector, as soon as its value converges (i.e. stops changing). Finally, local graph clustering methods such as those based on the Spielman-Teng algorithm [8] or the more practical Andersen-Chung-Lang (ACL) algorithm [9] essentially perform one SpMSpV at each step.

In the area of supervised learning, SpMSpV becomes the workhorse of many support-vector machine (SVM) implementations that use the sequential minimal optimization (SMO) approach [10]. In this formulation, the working set is represented by the sparse matrix $\mathbf{A}$ and the sample data is represented by the sparse input vector $\mathbf{x}$. SpMSpV is also the primitive used for solving logistic regression problems in dual form [11].

For a given problem, the minimum amount of work that needs to be performed by any algorithm is called a *lower bound*, and the algorithms that match the lower bound within a constant factor are called *optimal*. Parallel algorithms for which the total work performed by all processors is within a constant factor of the state-of-the-art serial algorithm are called *work-efficient*. A parallel algorithm is said to have a *data race* whenever multiple threads access the same part of the memory and at least one of those accesses is a write operation. Whenever there is a data race among threads, the algorithm needs a *synchronization* mechanism to avoid inconsistencies.

In this work, we show that existing shared-memory parallel SpMSpV algorithms are not work-efficient because they start spending more time accessing the sparse matrix data structure than doing arithmetic as parallelism increases. We present a new shared-memory parallel SpMSpV algorithm. When the input and output vectors are not sorted, the algorithm is *optimal* for matrices with at least one nonzero per column on average. The key insight is to avoid each thread individually scan the list of matrix columns, which is unscalable even if the columns are stored in a sparse format. We also implement and evaluate a variation of our algorithm where the input and output vectors are sorted, as it shows better performance in practice due to increased cache efficiency. Both variations of the algorithm avoid unnecessary *synchronization*. We experimentally evaluate our algorithm on an Intel Ivy Bridge multicore processor as well as the new Intel Knight's Landing processor on a variety of real-world matrices with varying nonzero structures and topologies.

## II. Background

### A. Notation

Sparse matrix-sparse vector multiplication is the operation $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ where a sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is multiplied by a sparse vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$ to produce a sparse vector $\mathbf{y} \in \mathbb{R}^{m \times 1}$. Intuitively, a matrix (vector) is said to be sparse when it is computationally advantageous to treat it differently from a dense matrix (vector). In this paper, we only consider this "right multiplication" with the column vector case because the "left multiplication" $\mathbf{y}' \leftarrow \mathbf{x}'\mathbf{A}$ by the row vector $\mathbf{x}'$ is symmetric and the algorithms we present can be trivially adopted to the "left multiplication" case.

The $nnz()$ function computes the number of nonzeros in its input, e.g., $nnz(\mathbf{x})$ returns the number of nonzeros in $\mathbf{x}$. The $nzc()$ function, which is only applicable to matrices, computes the number of nonempty columns of its input. When the object is clear from the context, we sometimes drop the input and simply write $nnz$ and $nzc$. We follow the Matlab colon notation: $\mathbf{A}(:, i)$ denotes the $i$th column, $\mathbf{A}(i, :)$ denotes the $i$th row, and $\mathbf{A}(i, j)$ denotes the element at the $(i, j)$th position of matrix $\mathbf{A}$.

Our SpMSpV algorithm works for all inputs with different sparsity structures as evidenced by our experimental results, but we will analyze its computational complexity on Erdős-Rényi random graphs for simplicity. In the Erdős-Rényi random graph model $G(n, p)$, each edge is present with probability $p$ independently from each other. For $p = d/n$ where $d \ll n$, in expectation $d$ nonzeros are uniformly distributed in each column. We use $f$ as shorthand of $nnz(\mathbf{x})$ in our analysis.

### B. Classes of SpMSpV algorithms

SpMSpV algorithms can be broadly classified into two classes: *vector-driven* and *matrix-driven* algorithms. In a vector-driven algorithm, computation and data access patterns are determined by the nonzero structure of the input vector $\mathbf{x}$. By contrast, the nonzeros of the matrix $\mathbf{A}$ drive the computation of matrix-driven algorithms. In some sense, vector-driven algorithms can be classified as pull-based since the entries of the matrix are selectively pulled depending on the location of the nonzeros in the input vector. Following the same logic, matrix-driven algorithms can be classified as push-based. In the vector-driven formulation, the SpMSpV problem becomes reminiscent of merging multiple lists (i.e., scaled columns of $\mathbf{A}(:, i)$ for which $\mathbf{x}(i) \neq 0$).

### C. Sparse matrix and vector data structures

There is no shortage of sparse matrix formats, most of which were exclusively invented for the sparse matrix-dense vector multiplication (SpMV) operation. A recent paper includes an up-to-date list of sparse matrix storage formats [12]. The SpMV operation can be implemented by sequentially iterating over the nonzeros of the sparse matrix, hence does not require fast random access to the columns of a matrix. In SpMSpV, however, only those columns $\mathbf{A}(:, i)$ for which $\mathbf{x}(i) \neq 0$ needs to be accessed. Consequently, we only consider the storage formats that allow fast random access to columns.

The Compressed Sparse Columns (CSC) format is perhaps the most widely used sparse matrix storage format, together with its row analog; the Compressed Sparse Rows. CSC has three arrays: $colptrs$ is an integer array of length $n + 1$ that effectively stores pointers to the start and end positions of the nonzeros for each column, $rowids$ is an integer array of length $nnz$ that stores the row ids for nonzeros, and $values$ is an array of length $nnz$ that stores the numerical values for nonzeros. CSC supports random access to the start of a column in constant time. Some implementations of CSC keep the row ids of nonzeros within each column sorted, e.g. the range $rowids(colptrs(i) \ldots colptrs(i + 1))$ is sorted for all $i$, but this is not a universal requirement.

The Double-Compressed Sparse Column (DCSC) format [13] further compresses CSC by removing repetitions in the $colptrs$ array, which arise from empty columns. In DCSC, only columns that have at least one nonzero are represented, together with their column indices. DCSC requires $O(nzc + nnz)$ space compared to CSC's $O(n + nnz)$. DCSC can be augmented to support fast column indexing by building an auxiliary index array that enables random access to the start of a column in expected constant time. This additional array does not increase the asymptotic storage.

There are two commonly utilized methods to store sparse vectors. The *list format* simply stores the vector compactly as a list of (index,value) pairs. The list can be sorted or unsorted. In contrast to its name, the actual data structure is often an array of pairs for maximizing cache performance. This format is space efficient, requiring only $O(nnz)$ space. It is often the format of choice for vector-driven algorithms but inefficient for matrix-driven algorithms because it does not support constant time random access for a given index. The alternative *bitvector format* [14] is composed of a $O(n)$-length bitmap that signals whether or not a particular index is nonzero, and an $O(nnz)$ list of values.

We require our algorithm to produce the output vector $\mathbf{y}$ in the same format that it received the input vector $\mathbf{x}$. For example, if the input is presented in sorted list format, then the output should also be in sorted list format. This is necessary to ensure the usability of our algorithms as part of a larger computational process where the output of one SpMSpV can then be reused as the input of another SpMSpV.

### *D. A Lower Bound for SpMSpV*

We present a simple lower bound for multiplying an $n$-by-$n$ matrix that represents the Erdős-Rényi graph $G(n, d/n)$ by a sparse $n$-by-1 vector $\mathbf{x}$ with $f$ nonzeros. Any SpMSpV algorithm needs to access the nonzero entries from $f$ columns of $\mathbf{A}$ corresponding to the nonzeros of $\mathbf{x}$. Since each column of $\mathbf{A}$ has $d$ nonzeros in expectation, the asymptotic lower bound of SpMSpV is $\Omega(df)$. This lower bound assumes no restrictions for storing the matrix and the vectors. The algorithm we present in this paper attains this lower bound using unsorted vectors. No known algorithm attains this lower bound if we require the vectors to be sorted.

### *E. Prior work on parallel SpMSpV algorithms*

A summary of existing SpMSpV algorithms are shown in Table I, where we cite the first appearances of algorithms in the literature. Combinatorial BLAS (CombBLAS) [16] includes implementations of a variety of vector-driven algorithms. The algorithms that use the DCSC format have been first used in the context of parallel breadth-first search (BFS) [3]. For shared-memory parallelization, the BFS work advocated splitting the matrix row-wise to $t$ (number of threads) pieces. Each thread local $m/t$-by-$n$ submatrix was then stored in the DCSC format. The authors experimented with multiple data structures for merging scaled columns of $\mathbf{A}$: a sparse accumulator (SPA) and a priority queue (heap). The SPA [17] is an abstract data type that (at minimum) consists of a dense vector of numerical values and a list of indices that refer to nonzero entries in the dense vector. CombBLAS later extended its support to CSC.

GraphMat [14] supports a matrix-driven SpMSpV algorithm. In GraphMat, the matrix is represented in the DCSC format and the vector is stored using the bitvector format. GraphMat also splits matrix row-wise. Nurvitadhi et al. [18] present a hardware accelarator for a vector-driven SpMSpV algorithm. Their algorithm description makes random accesses to vector $\mathbf{y}$, without any reference to its sparsity. Yang et al. [15] present a vector-driven SpMSpV implementation on the GPU, using sorted input/output vectors.

### *F. Requirements for a parallel work-efficient SpMSpV algorithm*

- **To attain the lower bound, an SpMSpV algorithm must be vector driven.** A matrix-driven algorithm needs $O(n)$ time to iterate over the columns of a matrix in CSC format. For a very sparse input vector, $O(n)$ dominates the total runtime. By contrast, a vector-driven algorithm can efficiently access $df$ entries of the matrix.
- **To attain the lower bound, a SPA-based SpMSpV algorithm should not initialize the entire SPA.** Since SPA is a dense vector of size $m$, initializing the entire SPA requires $O(m)$ time. By contrast, an algorithm that only initializes entries of SPA to be accessed in the multiplication requires $O(nnz(\mathbf{y}))$ initialization time; hence can be work efficient.
- **A parallel SpMSpV algorithm that splits the matrix row-wise is not work efficient for sufficiently large number of threads.** Consider an algorithm that splits $\mathbf{A}$ row-wise to $t$ pieces and multiplies each of the $m/t$-by-$n$ submatrices independently with $\mathbf{x}$ in parallel by $t$ threads to produce $1/t$ piece of $\mathbf{y}$. Here, each thread needs to access the entire $\mathbf{x}$, requiring $O(f)$ time per thread. The total time to access $\mathbf{x}$ over all threads is $O(tf)$, making it work inefficient when $t{>}d$. Even when $t{<}d$, the row-split algorithm does not scale well with increasing thread counts because of the additional work the algorithm needs to perform in scanning $\mathbf{x}$. The row-split algorithm does not require synchronization because each thread writes to a separate part of the output vector.
- **A parallel SpMSpV algorithm that splits the matrix column-wise needs synchronization among threads.** Consider an algorithm that splits $\mathbf{A}$ column-wise to $t$ pieces and multiplies each of the $m$-by-$n/t$ submatrices with $1/t$ piece of $\mathbf{x}$ in parallel by $t$ threads to produce $\mathbf{y}$. This algorithm is work efficient because the nonzero entries of $\mathbf{x}$ and $\mathbf{A}$ are accessed only once. However, synchronization is required among threads in the column split case because each thread writes to the same output vector $\mathbf{y}$ via a shared SPA.
- **A parallel SpMSpV algorithm that employs 2-D partitioning of the matrix is not work efficient.** Consider an algorithm that partitions $\mathbf{A}$ into $\sqrt{t} \times \sqrt{t}$ grids and multiplies each of the $m/\sqrt{t}$-by-$n/\sqrt{t}$ submatrices with $1/\sqrt{t}$ piece of $\mathbf{x}$ to generate partial $1/\sqrt{t}$ piece of $\mathbf{y}$. Since each submatrix in a column of the grid needs to access the same $1/\sqrt{t}$ piece of $\mathbf{x}$, the input vector is accessed $\sqrt{t}$ times across all threads, making the algorithm work inefficient when $t{>}d^2$. Futhermore, threads processing submatrices in a row of the grid need to update the same part of the output vector $\mathbf{y}$, requiring synchronization among threads. This algorithm mimics the concepts of distributed-memory SpMSpV algorithms in CombBLAS and GraphMat.

We summarize the properties SPA-based sequential and parallel SpMSpV algorithms in Table II. Based on this table, an asymptotically optimal SpMSpV algorithm that attains the lower bound should be vector-driven and initializes only necessary entries of SPA. A desirable parallel algorithm should be work-efficient and should perform as little synchronization as possible (synchronization-avoiding). However, none of the parallelization scheme described in Table II is both work-efficient and synchronization-free at the same time. This observation motivates us to develop a new parallel algorithm incorporating the advantages of both row- and column-split schemes to make the newly-developed algorithm both work efficient and synchronization-avoiding. In contrast to CombBLAS and GraphMat that split the matrix row-wise beforehand, our algorithm, called SpMSpV-bucket, splits the necessary columns of the matrix on the fly using a list of buckets. This approach can address the need of each multiplication independently and has been shown to be very effective in sparse matrix-dense vector multiplication (SpMV) [19]. We describe the SpMSpV-bucket

TABLE I: Classification of parallel SpMSpV algorithms. $t$ denotes the number of threads. SpMSpV-bucket is presented in this paper.

| Class | Algorithms | Data structures | | Merging | Sequential | Parallelization | Parallel |
|---|---|---|---|---|---|---|---|
| | | matrix | vector | strategy | complexity | strategy | complexity |
| matrix-driven | GraphMat [14] | DCSC | bitvector | SPA | $O(nzc + df)$ | row-split matrix and private SPA | $O(nzc + df/t)$ |
| vector-driven | CombBLAS-SPA [3] | DCSC | list | SPA | $O(df)$ | row-split matrix and private SPA | $O(f + df/t)$ |
| vector-driven | CombBLAS-heap [3] | DCSC | list | heap | $O(df \lg f)$ | row-split matrix and private heap | $O(df/t \lg f)$ |
| vector-driven | SpMSpV-sort [15] | CSC | list | sorting | $O(df \lg df)$ | concatenate, sort and prune | $--$ |
| vector-driven | SpMSpV-bucket | CSC | list | buckets | $O(df)$ | 2-step merging and private SPA | $O(df/t)$ |

TABLE II: Characteristics of SPA-based sequential and parallel SpMSpV algorithms. [1] In column-split and 2-D partitioning based algorithms, private SPA is not considered because it requires $O(tm)$ memory for $t$ threads.

| | Algorithm aspects | Attain lower bound? | Work efficient? | Synch. needed? |
|---|---|---|---|---|
| Sequential | matrix driven | × | | |
| | vector driven | ✓ | | |
| | SPA full init | × | | |
| | SPA partial init | ✓ | | |
| Parallel | row-split (private SPA) | | × | × |
| | column-split (shared SPA[1]) | | ✓ | ✓ |
| | 2-D (shared SPA[1]) | | × | ✓ |

algorithm in the next section.

## III. The SpMSpV-bucket algorithm

Algorithm 1 describes the steps of the SpMSpV-bucket algorithm that takes a dense vector *SPA* of size $m$ and a list of $nb$ buckets along with $\mathbf{A}$ and $\mathbf{x}$ as inputs. The matrix is stored in the CSC format and the vector is stored in the list format. The buckets are uninitialized space to be used by threads to temporarily store (row index, scaled value) pairs from the selected columns of the matrix. Each bucket corresponds to a subset of consecutive rows of the matrix. The $i$th location of SPA corresponds to the $i$th row of the matrix and is accessed by a single thread only. The SpMSpV-bucket algorithm then performs the following three steps for the multiplication.

**Step 1: Accumulate columns of A into buckets (lines 2-7 of Algorithm 1).** In this step, the columns $\mathbf{A}(:,i)$ for which $\mathbf{x}(i) \neq 0$ are extracted, the values of the extracted columns are multiplied by the nonzero values of $\mathbf{x}$, and the scaled values paired with their row indices are stored in buckets. The bucket in which a scaled matrix entry is placed is determined by its row index. More specifically, values in the $i$th row are stored in ($\lfloor (i \times nb)/m \rfloor$)-th bucket where $nb$ is the number of buckets. This step is depicted in Step 1 of Figure 1 where the second, fifth and seventh columns of $\mathbf{A}$ corresponding to nonzero indices of $\mathbf{x}$ are extracted and stored in four buckets B1, B2, B3, and B4. This step is similar to the column-split variant of SpMSpV algorithms and ensures the work-efficiency of our parallel algorithm.

In the parallel algorithm, each thread processes a subset of nonzero entries of $\mathbf{x}$ and stores the scaled entries of the corresponding columns of $\mathbf{A}$ in their designated buckets.

**Algorithm 1** Parallel SpMSpV algorithm. **Input:** An $m \times n$ sparse matrix $\mathbf{A}$ stored in CSC format, the input sparse vector $\mathbf{x}$, a dense vector *SPA* of size $m$, and a list of $nb$ buckets *Buckets*. **Output:** the output sparse vector $\mathbf{y}$.

```
 1: procedure SpMSpV($\mathbf{A}$, $\mathbf{x}$, $SPA$, $Buckets$)
 2:     ▷ Step1: Gather necessary columns of $\mathbf{A}$ in $t$ buckets (each
        bucket corresponds to a subset of consecutive rows of the matrix)
 3:     for every nonzero entry $(j, x(j))$ in $\mathbf{x}$ do in parallel
 4:         for every nonzero $\mathbf{A}(i,j)$ in $\mathbf{A}(:,j)$ do
 5:             $k \leftarrow \lfloor (i \times nb)/m \rfloor$                ▷ the destination bucket
 6:             ▷ Lock-free insertion, see text for details
 7:             $B_k \leftarrow B_k \cup (i, \text{MULT}(\mathbf{x}(j), \mathbf{A}(i,j)))$
 8:     for each bucket $B_k$ in $Buckets$ do in parallel
 9:         $uind_k \leftarrow \phi$          ▷ unique indices found in this bucket
10:         ▷ Step2: Merge entries in each bucket
11:         for every $(ind, val)$ pair in $B_k$ do
12:             $SPA[ind] \leftarrow \infty$
13:         for every $(ind, val)$ pair in $B_k$ do
14:             if $SPA[ind] = \infty$ then
15:                 $uind_k \leftarrow uind_k \cup ind$       ▷ save unique indices
16:                 $SPA[ind] \leftarrow val$
17:             else
18:                 $SPA[ind] \leftarrow \text{ADD}(SPA[ind], val)$
19:         ▷ Step3: Construct $\mathbf{y}$ by concatenating buckets using SPA
20:         $offset_k \leftarrow \sum_{l=0}^{k-1} |uind_l|$       ▷ using prefix sum on the
        master thread
21:         $i \leftarrow 0$
22:         for each $ind$ in $uind_k$ do
23:             $y[offset_k + i] \leftarrow (ind, SPA[ind])$
24:             $i \leftarrow i + 1$
```

**Algorithm 2** Preprocessing step of parallel SpMSpV algorithm needed to avoid synchronization among threads when inserting entries to buckets. **Input:** see Algorithm 1. **Output:** An $t$-by-$nb$ array *Boffset* where *Boffset*$[i][j]$ stores the number of entries that the $i$th thread will insert to the $j$th bucket in Step 1 of Algorithm 1. Here, $t$ denotes the number of threads.

```
1: procedure Estimate-Buckets($\mathbf{A}$, $\mathbf{x}$, $Buckets$)
2:     for $k$ in $1..t$ do in parallel
3:         $Boffset[k] \leftarrow 0$                      ▷ initialize to zero
4:         $\mathbf{x}_k \leftarrow 1/t$ piece of $\mathbf{x}$ processed by the $k$-th thread
5:         for every nonzero entry $(j, \mathbf{x}_k(j))$ in $\mathbf{x}_k$ do
6:             for every nonzero $\mathbf{A}(i,j)$ in $\mathbf{A}(:,j)$ do
7:                 $b \leftarrow \lfloor (i \times nb)/m \rfloor$          ▷ destination bucket
8:                 $Boffset[k][b] \leftarrow Boffset[k][b] + 1$
```

Writing to buckets requires synchronization among threads because multiple threads could write simultaneously to the

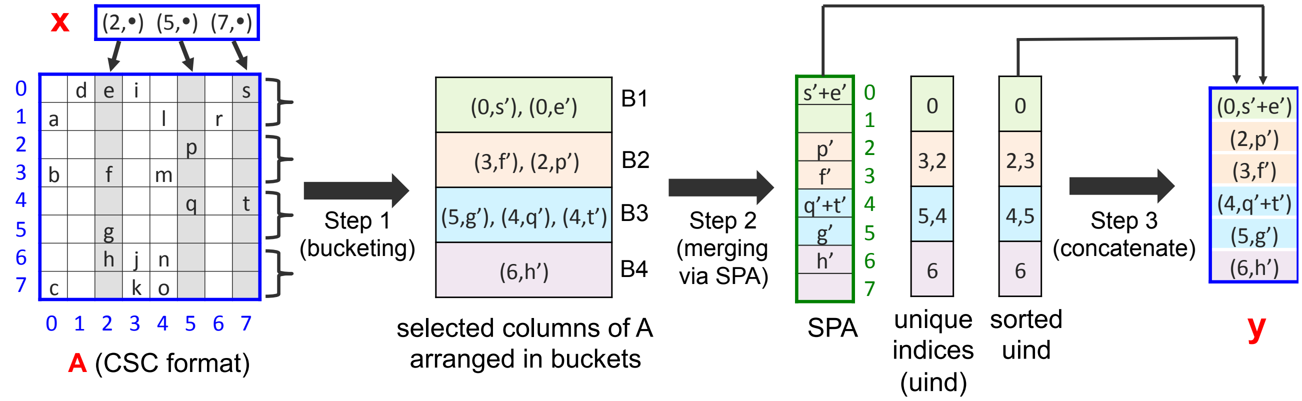

Fig. 1: Three steps of the SpMSpV algorithm. In the first step, nonzero entries of the selected columns of $\mathbf{A}$ are multiplied by the corresponding elements of $\mathbf{x}$. The multiplied values (denoted with prime symbols) coupled with their row indices are stored in four buckets. The bucket where an entry is stored is determined by its row index. Data structures possessed or touched by four buckets are shown in four different colors. In the second step, entries in each bucket are merged independently by using a sparse accumulator. In each bucket, unique indices ($uind$) are identified and sorted (sorting is an optional step and is only performed when sorted output is required or to improve cache locality). In the third step, the output vector $\mathbf{y}$ is created by concatenating $uind$ from all buckets and fetching the corresponding values from the SPA.

same bucket when they extract entries from the same row $\mathbf{A}$. To avoid expensive synchronizations, we pass over the columns of $\mathbf{A}(:, i)$ for which $\mathbf{x}(i) \neq 0$ in a preprocessing step and count how many scaled entries each thread will write to a bucket in Step 1 of Algorithm 1. The preprocessing step is described in Algorithm 2 where $Boffset[i][j]$ stores the number of entries that the $i$th thread will insert to the $j$th bucket in Step 1 of Algorithm 1. We use $Boffset$ to precisely compute where each thread will insert in each bucket. Using this approach, threads can insert to buckets (line 7 of Algorithm 1) without any synchronization.

**Step 2: merge entries in each bucket (lines 10-18 of Algorithm 1).** At this point, the algorithm behaves like a row-split algorithm where the buckets store scaled entries split row-wise among the buckets. Since there is no data dependency among buckets after they are populated, a bucket can be merged independently by a thread. At the beginning of this step, each thread initializes only those locations of the SPA to be used in merging entries in the current bucket. Next, entries in a bucket are merged using a part of SPA dedicated only for this bucket. In this process, the algorithm retrieves unique indices from the $k$th bucket and stores them in $uind_k$. This step is depicted in Step 2 of Figure 1 where each of the four buckets independently merges its entries by adding values with the same row indices.

**Step 3: Construct $\mathbf{y}$ by concatenating buckets using SPA (lines 19-24 of Algorithm 1).** In the final step, unique indices identified in a bucket are coupled with the corresponding values of the SPA and the (index, value) pairs are inserted to the result vector. To make this step synchronization free, unique indices in a bucket are mapped to indices of $\mathbf{y}$ using a prefix sum operation described in line 20 of Algorithm 1. This step is depicted in Step 3 of Figure 1 where six unique indices are coupled with the computed values of SPA and saved in the output vector $\mathbf{y}$. In Figure 1, we also showed the situation when indices in $\mathbf{y}$ are required to be sorted.

So far, we have not addressed the sortedness of the vectors. The algorithm works as-is for unsorted vectors. The sortedness of the input $\mathbf{y}$ does not affect the correctness of the algorithm. However, in order to return a sorted output, the algorithm requires a modification to add a sorting step at the very end.

### A. *Performance optimizations*

**Load balancing.** In order to balance work among threads, we create more buckets than the available number of threads. In our experiments, we use $4t$ buckets when using $t$ threads and employ dynamic scheduling of threads over the buckets. Using more buckets tends to improve the scalability of the SpMSpV-bucket algorithm except when the input vector is extremely sparse (see the discussion in Section IV-F).

**Cache efficiency.** To improve the cache locality of Step 1 in Algorithm 1, we allocate a small private buffer for each thread. A thread first fills its private buffer as it accesses the columns of $\mathbf{A}$ and copies data from the private buffer to buckets when the local buffer is full. The thread-private buffer is small enough to fit in L1 or L2 cache. Sorting the input vector $\mathbf{x}$ beforehand (if it is not sorted) improves the cache locality of the bucketing step when $\mathbf{x}$ is denser. This is due to the fact that when $\mathbf{x}$ is denser, the probability of accessing consecutive columns of $\mathbf{A}$ increases significantly.

**Memory allocation.** The memory allocation time for buckets and SPA can be expensive, especially when we run SpMSpV many times in an iterative algorithm such as the BFS. Hence, we allocate enough memory for all buckets and for the SPA in advance and pass them to the SpMSpV-bucket algorithm. The number of entries inserted in all buckets is at most $O(nnz(\mathbf{A}))$. Hence, preallocating the buckets does not increase the total memory requirement of our algorithm.

### *B. Time and space complexity*

**Serial complexity.** The preprocessing step described in Algorithm 2 and Step 1 in Algorithm 1 both access $df$ nonzero entries from $f$ columns of $\mathbf{A}$. Hence these steps require $O(df)$ time. The initialization of SPA and merging entries in all buckets require another $O(df)$ time in the second step. The total number of entries in $uind_k$ across all buckets is $nnz(\mathbf{y})$. Since $nnz(\mathbf{y}) \leq df$, the overall complexity of the algorithm is $O(df)$. If $\mathbf{y}$ is needed to be sorted by nonzero indices, another $O(nnz(\mathbf{y}) \log nnz(\mathbf{y}))$ time is required for sorting. However, sorting is very efficient in SpMSpV-Bucket algorithm because only unique indices in each buckets are needed to be sorted. Hence, each thread can run a sequential integer sorting function on its local indices using efficient sorting algorithms such as the radix sort.

**Parallel complexity.** In this analysis, we assume that the SpMSpV-Bucket algorithm employs at most $f$ threads to perform the multiplication (i.e., the number of threads $t$ is less than or equal to $f$). In the first step, $f$ nonzero entries of the input vector are evenly distributed among $t$ threads. Hence, each thread accesses $fd/t$ nonzero entries of the matrix. Since the nonzero entries of the matrix are evenly distributed among rows in the Erdős-Rényi model, each bucket will have $fd/t$ entries in expectation when $t$ buckets are used. Hence the parallel complexity of the SpMSpV-Bucket algorithm is $O(fd/t)$.

The worst case span (critical path) of our algorithm, as it is presented, can be higher in theory for skewed matrices than it is for matrices representing Erdős-Rényi graphs. However, two simple modifications to our algorithm would suffice to avoid this high span case. First, we would modify the assignment of work to threads in ESTIMATE-BUCKETS (in Algorithm 2, line 4) to be based on nonzeros, as opposed to rows, of $\mathbf{x}$. Second, we would perform line 4 of Algorithm 1 in parallel.

**Space complexity.** The total space required for all buckets is no more than $O(nnz(\mathbf{A}))$. Hence total space requirement of our algorithm is $O(m + nnz(\mathbf{A}))$.

### *C. Comparison with the binning-based SpMV algorithm*

The SpMSpV-bucket algorithm is significantly different than the binning-based SpMV algorithm [19]. Since SpMV accesses every nonzero of the matrix, the destination buckets are trivially defined. By contrast, SpMSpV accesses only a fraction of columns of the matrix guided by the sparsity of the input vector, making the bucketing step more involved and requiring a preprocessing step to estimate buckets as described in Algorithm 2. Dense output vector in SpMV greatly simplifies the multiplication in each bucket, whereas SpMSpV requires a SPA to perform the multiplication efficiently. The computational load of SpMSpV is highly dynamic as well because it is often determined by the sparsity of the input vector. Therefore, achieving good performance in SpMSpV is more challenging than SpMV.

| | **Cori (Intel KNL)** | **Edison (Intel Ivy Bridge)** |
|---|---|---|
| **Core** | | |
| Clock (GHz) | 1.4 | 2.4 |
| L1 Cache (KB) | 32 | 32 |
| L2 Cache (KB) | 1024[1] | 256 |
| DP GFlop/s/core | 44 | 19.2 |
| **Node Arch.** | | |
| Sockets/node | 1 | 2 |
| Cores per socket | 64 | 12 |
| STREAM BW[2] | 102 GB/s | 104 GB/s |
| Memory per node | 96 GB | 64 GB |
| **Prog. Environment** | | |
| Compiler | gcc 5.3.0 | gcc 5.3.0 |
| Optimization | -O3 | -O3 |

TABLE III: Overview of Evaluated Platforms. [1]Shared between 2 cores in a tile. [2]Memory bandwidth is measured using the STREAM copy benchmark per node.

## IV. RESULTS

### *A. Experimental Setup*

We evaluate the performance of SpMSpV algorithms on Edison, a Cray XC30 supercomputer at NERSC and on a KNL manycore porcessor that will be integrated with NERSC/Cori. These two systems are described in Table III. We used OpenMP for multithreaded execution in our code.

Table IV describes a set of real matrices from the University of Florida sparse matrix collection [20] used in our experiments. We selected the low-diameter scale-free graphs and high-diameter graphs arising in various scientific domains.

### *B. Impact of sorted input and output vectors on the performance of the SpMSpV-bucket algorithm*

We implemented two variants of the SpMSpV-bucket algorithm based on the sortedness of the input and output vectors: in one variant both $\mathbf{x}$ and $\mathbf{y}$ are kept sorted by their indices, while the second variant works on unsorted vectors. Figure 2 shows the impact of sorted vectors on the performance of the SpMSpV-bucket algorithm for $\mathbf{x}$ with 10K and 2.5M nonzeros. When the vector is relatively dense, keeping the vectors sorted improves the performance of our algorithm as can be seen in the right subfigure in Figure 2. This is due to the fact that when $\mathbf{x}$ is denser, the probability of accessing consecutive columns of $\mathbf{A}$ increases, making the bucketing step (Step 1 in Algorithm 1) more cache efficient. By contrast, columns of $\mathbf{A}$ are often accessed inconsecutively when $\mathbf{x}$ is sparser (e.g., when $nnz(\mathbf{x})$ is less than $1\%$ of $n$). Since the unsorted version never seems to outperform the sorted version in practice, we only present results with sorted vectors in the remainder of the results section. Sorted vectors also ensure fairness when comparing our algorithm with existing algorithms (GraphMat and CombBLAS) that keep their vectors ordered.

### *C. Relative performance of SpMSpV algorithms*

We compare the performance SpMSpV-bucket with three other SpMSpV algorithms: CombBLAS-SPA, CombBLAS-heap, and GraphMat. These algorithms are already discussed in Section II-E. At first, we study the impact of $nnz(\mathbf{x})$

TABLE IV: Test problems from the University of Florida sparse matrix collection [20].

| Class | Graph | #vertices ($\times 10^6$) | #edges ($\times 10^6$) | pseudo diameter | Description |
|---|---|---|---|---|---|
| low-diameter graphs | amazon0312 | 0.40 | 3.20 | 21 | Amazon product co-purchasing network |
| | web-Google | 0.92 | 5.11 | 16 | Webgraph from the Google prog. contest, 2002 |
| | wikipedia-20070206 | 3.56 | 45.03 | 14 | Wikipedia page links |
| | ljournal-2008 | 5.36 | 79.02 | 34 | LiveJournal social network |
| | wb-edu | 9.85 | 57.16 | 38 | Web crawl on .edu domain |
| | dielFilterV3real | 1.10 | 89.31 | 84 | High-order vector finite element method in EM |
| high-diameter graphs | G3_circuit | 1.56 | 7.66 | 514 | circuit simulation problem |
| | hugetric-00020 | 7.12 | 21.36 | 3,662 | undirected graph |
| | hugetrace-00020 | 16.00 | 48.00 | 5,633 | Frames from 2D Dynamic Simulations |
| | delaunay_n24 | 16.77 | 100.66 | 1,718 | Delaunay triangulations of random points |
| | rgg_n24_s0 | 16.77 | 165.1 | 3,069 | Random geometric graph |

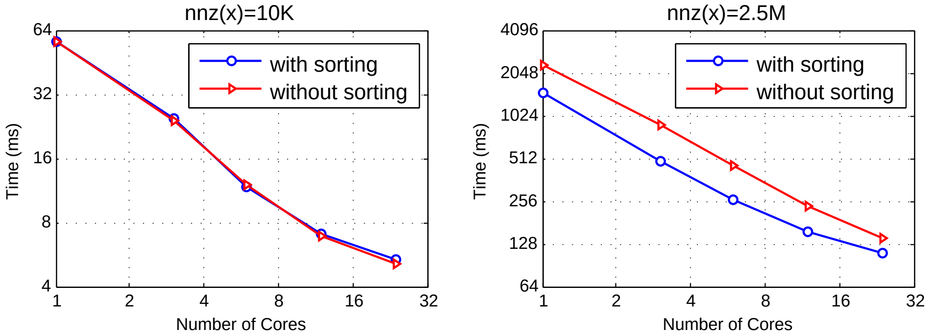

Fig. 2: Runtime of the SpMSpV-bucket algorithm with or without sorted input and output vectors. Here, the adjacency matrix of `ljournal-2008` is multiplied by sparse vectors with (a) 10K and (b) 2.5M nonzeros. The experiment was run on Edison.

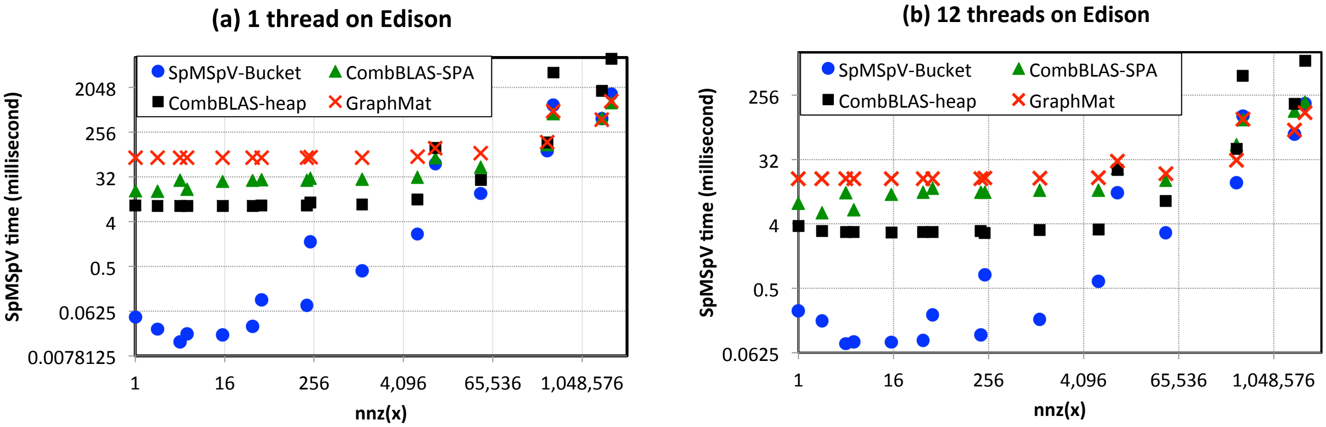

Fig. 3: Runtime of four SpMSpV algorithms when the adjacency matrix of `ljournal-2008` is multiplied by sparse vectors with different number of nonzero entries using (a) 1 thread and (b) 12 threads on Edison. The sparse vectors represent frontiers in a BFS starting from the first vertex of `ljournal-2008`.

on the performance of SpMSpV algorithms. Figure 3 shows the runtime of four algorithms when the adjacency matrix of `ljournal-2008` is multiplied by $\mathbf{x}$ with different sparsity patterns using (a) 1 thread and (b) 12 threads on Edison. When $\mathbf{x}$ is very sparse (i.e., $nnz(\mathbf{x})$ less than 50K), the runtime of GraphMat remains constant for a fixed thread count. This is a property of any matrix-driven algorithm whose runtime is dominated by the $O(nzc)$ term needed to iterate over all nonzero columns of the matrix, especially when the vector is very sparse. For very sparse vectors, CombBLAS-SPA is also inefficient because of its strategy of initializing the whole SPA. By contrast, SpMSpV-bucket does not have any extra overhead when the vector is very sparse; hence it outperforms its competitors by several orders of magnitude. For example, when $nnz(\mathbf{x}) = 50$, SpMSpV-bucket is $200\times$, $81\times$, and $744\times$ faster than CombBLAS-SPA, CombBLAS-heap, and GraphMat, respectively on a single thread. When $nnz(\mathbf{x}) = 1100$, SpMSpV-bucket is 68×, 21×, and 191× faster than CombBLAS-SPA, CombBLAS-heap, and GraphMat, respectively on a single thread. This huge gap in performance shrinks as the input vector becomes denser, where the multiplication cost is able to hide overheads of CombBLAS-SPA and GraphMat. For example, when $nnz(\mathbf{x}) = 1.9M$, SpMSpV-bucket, CombBLAS-SPA, and GraphMat all perform similarly and run 3.5× faster than CombBLAS-heap because of the logarithm term in the latter algorithm. The story remains more or less similar on higher concurrency as can be seen in Figure 3(b).

### *D. Performance of SpMSpV algorithms when used in BFS*

BFS is arguably the most common customer of SpMSpV where the product of the adjacency matrix of the graph and the sparse vector representation of the current frontier provides the next frontier of the BFS. This approach has been successfully used in parallel BFS targeting GPU and the shared- and distributed-memory platforms [3], [14], [15]. Here we compare the performance of four SpMSpV algorithms when they are used in BFS.

Figure 4 shows the performance of four shared-memory SpMSpV algorithms on eleven real world matrices from Table IV on a single node of Edison. To ensure the fairness in comparing algorithms, the same source vertex is used to start the BFS by all four algorithms and only the runtime of SpMSpVs in all iterations are considered. For all problems in Figure 4, SpMSpV-bucket runs the fastest on all concurrencies. The performance improvement is more dramatic on high-diameter graphs where SpMSpV-bucket runs 3× to 10× faster than GraphMat as can be seen in the bottom row of Figure 4. According to the discussion in Section IV-C, this performance gap is expected for high-diameter graphs where BFS executes many SpMSpVs with very sparse vectors – a territory where matrix-driven algorithms are inefficient. On scale-free graphs, SpMSpV-bucket still performs the best, but the gaps among the algorithms are narrower. This is due to the fact that BFS on a scale-free graph is usually dominated by few iterations with dense frontiers where matrix-driven algorithms usually perform their best.

On average, SpMSpV-bucket achieves 11× (max: 14×, min: 9×), CombBLAS-SPA achieves 6× (max: 7×, min: 5×), CombBLAS-heap achieves 12× (max: 17×, min: 4×), and GraphMat achieves, 11× (max: 15×, min: 9×) speedups, when going from 1 thread to 24 threads on Edison. GraphMat attains better scalability, even when $\mathbf{x}$ is very sparse, because it always performs $O(nzc)$ work to iterate over nonzero columns of the matrix. By contrast, our work-efficient algorithm might not scale well when the vector is very sparse (e.g., when $nnz(\mathbf{x})$ is less than the number of threads) due to the scarcity of work for all threads. The parallel efficiency of CombBLAS-SPA decreases with increasing concurrency because the total amount of work performed by all threads increases as each thread scans the entire input vector. Poor serial runtime often contributes to the high speedups of the CombBLAS-heap algorithm.

### *E. Performance of SpMSpV algorithms on the Intel KNL processor*

Figure 5 shows the performance of three SpMSpV algorithms on the Intel KNL processor equipped with 64 cores. We were unable to run GraphMat on KNL. On average, SpMSpV-bucket achieves 32× (max: 49×, min: 20×), CombBLAS-SPA achieves 12× (max: 14×, min: 10×), and CombBLAS-heap achieves 20× (max: 30×, min: 12×) speed-up when going from 1 thread to 64 threads on KNL. As before, the serial performance of CombBLAS-SPA is similar to or slightly better than SpMSpV-bucket on scale-free graphs. However, scalability of CombBLAS-SPA suffers with increasing number of threads because of its work inefficiency. By contrast, SpMSpV-bucket scales well up to 64 cores of KNL for diverse classes of matrices. We did not observe any benefit of using multiple threads per core on KNL.

### *F. Performance breakdown of the SpMSpV-bucket algorithm*

The SpMSpV-bucket algorithm has four distinct steps (including the preprocessing step) that are described in Section III. Here we show how these steps contribute to the total runtime of SpMSpV and how they scale with increased thread count. Figure 6 shows the strong scaling of the components of the SpMSpV-bucket algorithm when the adjacency matrix of `ljournal-2008` is multiplied by $\mathbf{x}$ with different sparsity patterns. SPA-based merging is the most expensive step of the sequential SpMSpV-bucket algorithm for all sparsity patterns of the input vector. As $\mathbf{x}$ becomes denser, bucketing becomes as expensive as merging on a single thread. For example, in Figure 6, SPA-based merging takes 73%, 62%, and 46% of the total sequential runtime when $nnz(\mathbf{x})$ is 200, 10K and 2.5M, respectively. By contrast, the bucketing steps takes 10%, 17%, and 35% of the serial runtime when $nnz(\mathbf{x})$ is 200, 10K and 2.5M, respectively.

SPA-based merging has the best scalability than other steps of the SpMSpV-bucket algorithm for all sparsity levels of $\mathbf{x}$ because each thread independently performs the merging on its private bucket. For example, when we go from 1 core to 24 cores in Figure 6, SPA-based merging achieves 11×, 19×, and 22× speedups when $nnz(\mathbf{x})$ is 200, 10K and 2.5M, respectively. By contrast, the bucketing step achieves 6×, and 10× speedups when $nnz(\mathbf{x})$ is 10K and 2.5M, respectively, when we go from 1 core to 24 cores on Edison. This step slows down by a factor of 2 when $nnz(\mathbf{x})$ is 200 because the overhead of managing 96 buckets (24 threads multiplied by 4) becomes more expensive than performing the per-bucket merging operations. To understand the limited scalability of the bucketing step, notice that each thread enjoys good spatial locality when reading the matrix column-by-column. However, matrix entries are written into buckets in an irregular fashion determined by the sparsity pattern of the matrix. These irregular writes to buckets essentially limit the scalability of the bucketing step. Consequently, bucketing

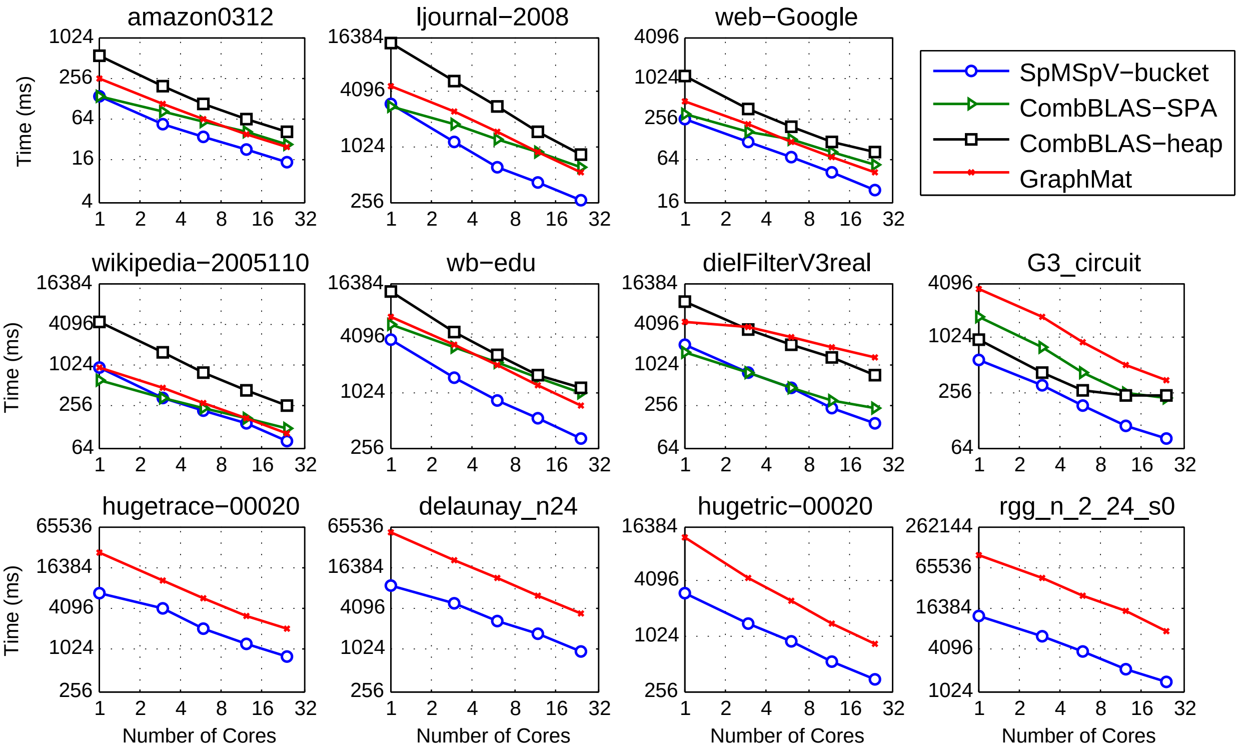

Fig. 4: Strong scaling of four shared-memory SpMSpV algorithms when they are used in BFS. The experiments were run on a single node of Edison. For each graph, the same source vertex is used to start the BFS by all four algorithms. We only report the runtime of SpMSpVs in all iterations omitting other costs of the BFS. For the high-diameter graphs in the bottom row, CombBLAS-DCSC and heap-merge algorithms were not competitive, hence we omit them for these graphs.

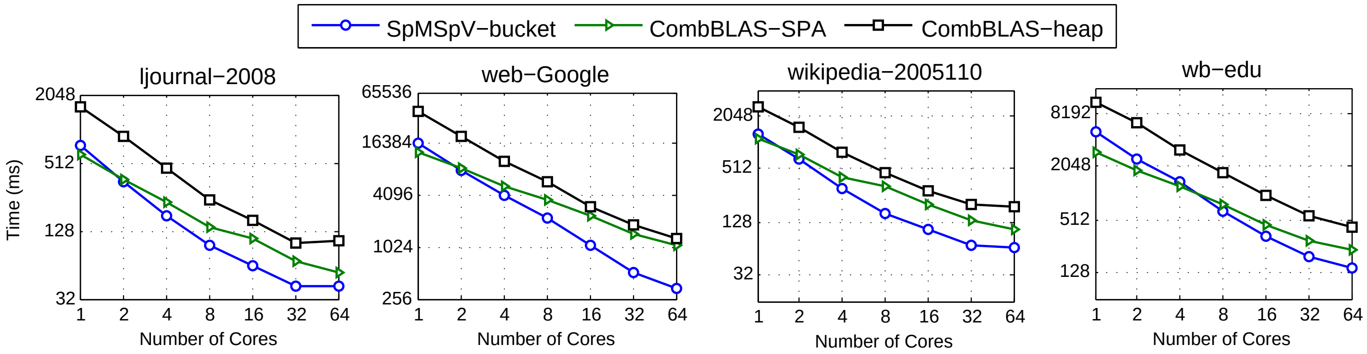

Fig. 5: Strong scaling of three shared-memory SpMSpV algorithms when they are used in BFS on KNL. For each graph, the same source vertex is used to start the BFS by all four algorithms. We only report the runtime of SpMSpVs in all iterations omitting other costs of the BFS. We were unable to run GraphMat on KNL.

step starts to dominate the runtime of the SpMSpV-bucket algorithm on high concurrency. The “output” step is often the least expensive step, and its scalability is negatively affected by the non-consecutive access of SPA. The scalability of all components improves as the input vector becomes denser, as expected.

## V. Conclusions and Future Work

We presented a work-efficient parallel algorithm for the sparse matrix-sparse vector multiplication (SpMSpV) problem. We carefully characterized different potential ways to organize the SpMSpV computation and identified the requirements for a high-performance algorithm that is work-efficient and one that also avoids unnecessary synchronization.

Our algorithm avoids synchronization by performing a row-

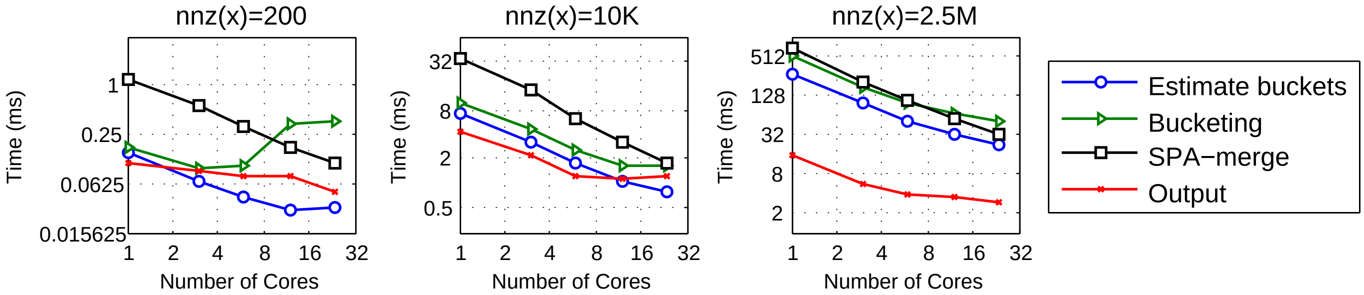

Fig. 6: Strong scaling of four components of SpMSpV-bucket algorithm when the adjacency matrix of `ljournal-2008` is multiplied by sparse vectors with different number of nonzeros on Edison.

wise partitioning of the input matrix on the fly, and attains work efficiency by employing the common computational pattern of the column-wise algorithms. Our algorithm achieves high-performance for a wide range of vector sparsity levels thanks to its vector-driven nature. The implementation of our algorithm on the Intel Ivy Bridge and the Intel KNL processors significantly outperforms existing approaches when the input vector is very sparse, and performs competitively when the input vector gets denser Matrix-driven algorithms are only competitive when the input vector gets relatively dense. As future work, we will investigate when and if it is beneficial to switch to a matrix-driven algorithm.

Further refinements of the SpMSpV problem arise in different contexts. Some SVM implementations shrink the working set periodically, hence requiring a data structure that is more friendly for row deletions. This could effect the tradeoffs involved in choosing the right SpMSpV algorithm, depending on the frequency of the shrinking. In addition, GraphBLAS effort is in the process of defining masked operations, including SpMSpV. This could also effect the algorithmic tradeoffs involved. Studying those effects are subject to future work.

## Acknowledgments

This work is supported by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

We thank the anonymous reviewers for correctly pointing out to corner cases in our analysis and making the final paper better.

## References

[1] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[2] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Communications of the ACM*, vol. 59, no. 5, pp. 78–87, 2016.

[3] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011.

[4] A. Buluç, E. Duriakova, A. Fox, J. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams, "High-productivity and high-performance analysis of filtered semantic graphs," in *Proceedings of the IPDPS*. IEEE Computer Society, 2013.

[5] K. Ekanadham, W. Horn, M. Kumar, J. Jann, J. Moreira, P. Pattnaik, M. Serrano, G. Tanase, and H. Yu, "Graph programming interface (GPI): a linear algebra programming model for large scale graph computations," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016, pp. 72–81.

[6] A. Azad and A. Buluç, "Distributed-memory algorithms for maximum cardinality matching in bipartite graphs," in *Proceedings of the IPDPS*. IEEE, 2016.

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

[8] D. A. Spielman and S.-H. Teng, "A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning," *SIAM Journal on Computing*, vol. 42, no. 1, pp. 1–26, 2013.

[9] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 2006, pp. 475–486.

[10] C.-C. Chang and C.-J. Lin, "LIBSVM: a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[11] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *The Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[12] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 2, pp. 428–440, 2016.

[13] A. Buluç and J. R. Gilbert, "On the Representation and Multiplication of Hypersparse Matrices," in *Proceedings of the IPDPS*, April 2008.

[14] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[15] C. Yang, Y. Wang, and J. D. Owens, "Fast sparse matrix and sparse vector multiplication algorithm on the GPU," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2015, pp. 841–847.

[16] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *IJHPCA*, vol. 25, no. 4, 2011.

[17] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[18] E. Nurvitadhi, A. Mishra, Y. Wang, G. Venkatesh, and D. Marr, "Hardware accelerator for analytics of sparse data," in *Europe Conference in Design, Automation, Test & Exhibition (DATE)*. IEEE, 2016, pp. 1616–1621.

[19] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 37.

[20] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 1, 2011.