# On the Representation and Multiplication of Hypersparse Matrices *

Aydın Buluç
Department of Computer Science
University of California, Santa Barbara
aydin@cs.ucsb.edu

John R. Gilbert
Department of Computer Science
University of California, Santa Barbara
gilbert@cs.ucsb.edu

## Abstract

*Multicore processors are marking the beginning of a new era of computing where massive parallelism is available and necessary. Slightly slower but easy to parallelize kernels are becoming more valuable than sequentially faster kernels that are unscalable when parallelized. In this paper, we focus on the multiplication of sparse matrices (SpGEMM). We first present the issues with existing sparse matrix representations and multiplication algorithms that make them unscalable to thousands of processors. Then, we develop and analyze two new algorithms that overcome these limitations. We consider our algorithms first as the sequential kernel of a scalable parallel sparse matrix multiplication algorithm and second as part of a polyalgorithm for SpGEMM that would execute different kernels depending on the sparsity of the input matrices. Such a sequential kernel requires a new data structure that exploits the hypersparsity of the individual submatrices owned by a single processor after the 2D partitioning. We experimentally evaluate the performance and characteristics of our algorithms and show that they scale significantly better than existing kernels.*

## 1. Introduction

Development and implementation of large-scale parallel graph algorithms poses numerous challenges [17, 27]. A promising research direction concerns the array-based linear algebra formulations of graph algorithms [1, 18, 26]. By exploiting the duality between matrices and graphs, array-based algorithms aim to apply the existing knowledge on parallel matrix algorithms to parallel graph algorithms. One of the key primitives in array-based graph algorithms is computing the product of two sparse matrices (SpGEMM) over a semiring. Most interesting graphs, such as the WWW graph, finite element meshes, planar graphs, and trees, are sparse. In this work, we consider a graph to be sparse if $nnz = O(n)$, where $nnz$ is the number of edges and $n$ is the number of vertices. Using a dense matrix multiplication algorithm for SpGEMM is overkill since the current fastest matrix multiplication algorithm has complexity $O(n^{2.38})$ [6, 24]. Furthermore, fast dense matrix multiplication algorithms operate on a ring instead of a semiring, which makes them unsuitable for most of the graph algorithms. Shortest-path computations [30], matching algorithms [22], cycle detection [28], and parsing context-free languages [21] are example application areas of a fast SpGEMM over a semiring.

There has been relatively little research on sparse matrix multiplication in the three decades since Gustavson's 1978 paper [12]. This inactivity was partly because of the diverging goals of the fields of numerical analysis and discrete algorithms. From a numerical analysis viewpoint, SpGEMM was not considered as important as the sparse matrix times dense vector (SpMV) operation, which serves as the building block of iterative linear solvers. Operations on two sparse matrices were completely left out of the Sparse BLAS [8], mainly because they were found too complicated to be supported by a low-level kernel like Sparse BLAS even though the authors acknowledge the fact that operations on two sparse matrices are required for some applications. From an algorithms viewpoint, using the adjacency matrix representation of a graph was generally considered overkill due to the absence of underlying primitives for sparse matrices. The adjacency list representation of graphs, therefore, was more popular among algorithm designers. Currently, the role of sparse matrices in graph algorithms is receiving increasing attention due to the implementation and performance challenges posed by parallel graph algorithms. The goal of this new research focus is to utilize the existing infrastructure for parallel matrix computations instead of redesigning and reimplementing the algorithms in parallel.

*Flops* ($\mathrm{flops}(AB)$ or $\mathrm{flops}$ in short) is defined as the number of nonzero arithmetic operations required to compute the output matrix $C$. The computation complexity of a sparse matrix algorithm should ideally depend only on

*This research was supported in part by the Department of Energy under award number DE-FG02-04ER25632.

flops. This is called "the time is proportional to flops" rule in Matlab [11]. Matlab's original design allows the complexity to be dependent on $nnz$, the number of nonzero elements in the matrix, and even the matrix dimensions $m$ or $n$. However, we claim that the dependency on the matrix dimensions $m$ or $n$ is too much in some cases, as explained in Section 2.1, and should be avoided.

The SpGEMM problem was recently reconsidered by Yuster and Zwick [29] over a ring, where the authors use a fast dense matrix multiplication such as arithmetic progression [6] as a subroutine. Their algorithm uses $O(nnz^{0.7}n^{1.2} + n^{2+o(1)})$ arithmetic operations , which is theoretically close to optimal only if we assume that the number of nonzeros in the resulting matrix $C$, $nnz(C)$, is $\Theta(n^2)$. This assumption rarely holds in reality, and instead we use the optimality criteria of a sparse matrix multiplication in terms of flops, following Gustavson's convention. This makes our algorithms work/output sensitive and leads to work/output sensitive algorithms for other problems that can be reduced to sparse matrix multiplication, for example colored intersection searching [14].

Some practical algorithms for SpGEMM have been proposed by various researchers over the years [20, 25]. Although they achieve reasonable performance on some classes of matrices, their computational complexity is rather high compared with an industrial-strength general purpose algorithm as the one used in Matlab [11]. The algorithm proposed by Sulatycke and Ghose [25] examines all possible $(i, j)$ positions of the input matrix $A$ in the outermost loop and tests whether they are nonzero. Therefore, their algorithm has $O(\text{flops} + n^2)$ complexity, performing unnecessary operations when $\text{flops} < n^2$. On the other hand, the current algorithm used in Matlab, which we will call the M-Algorithm from now on, has time complexity $O(\text{flops}+nnz(B)+n+m)$. Davis recently implemented a row-by-row dialect of the M-Algorithm in his CSparse software [7], which achieves slightly different running times but has the same performance characteristics as the M-Algorithm.

In this paper, we present two novel algorithms for sparse matrix multiplication. The first one is based on the outer-product formulation with time complexity $O(nzc(A) + nzr(B) + \text{flops} \lg ni)$, where $nzc(A)$ is the number of columns of $A$ that contain at least one nonzero, $nzr(B)$ is the number of rows of $B$ that contain at least one nonzero, and $ni$ is the number of indices $i$ for which $A(:, i) \neq \emptyset$ and $B(i, :) \neq \emptyset$. We adopt the Matlab notation for sparse matrix indexing (`subsref`), where $A(:, i)$ denotes the $i^{\text{th}}$ column, $A(i, :)$ denotes the $i^{\text{th}}$ row, and $A(i, j)$ denotes the element at the $(i, j)^{\text{th}}$ position of matrix $A$. The overall space complexity of our algorithm is only $O(nnz(A) + nnz(B) + nnz(C))$. Note that the time complexity of our algorithm does not depend on $n$, and the space complexity does not depend on flops. The second algorithm is an ordered variant of the column-by-column formulation, and has better expected time complexity for random matrices, but worse worst-case time complexity in general. We give an experimental evaluation of our algorithms using matrices from Erdős-Rényi random graphs [19], synthetically generated real-world graphs [16], and 3D geometric graphs [10]. Our experiments are sequential simulations where we decouple the cost of submatrix multiplications from other costs such as updates (submatrix additions) and parallelization overheads because the main contributions of our work are the sequential hypersparse matrix multiplication algorithms.

## 2 Problem Definition

The sparse matrix multiplication problem is to compute $C = AB$, where the input matrices $A$ and $B$, having dimensions $m \times k$ and $k \times n$ respectively, are both sparse. The input matrices are represented in some space efficient format and the output matrix $C$ should also be in the same format as $A$ and $B$. The data structure for storing sparse matrices in most sparse matrix packages, including Matlab, is CSC, which is explained in Section 3 in more detail.

The M-Algorithm has time complexity $O(\text{flops} + nnz(B)+n+m)$ and uses an auxiliary data structure called the sparse accumulator (SPA) in order to allow fast random access to the current active column. SPA is composed of three components: a dense vector that holds the real values for the active column of $C$, another dense boolean vector that holds the "occupied" flags, and a sparse list that holds the indices of nonzero elements of the current active column so far. The SPA itself uses space $O(m)$ and its initialization consequently takes $O(m)$ time, contributing to the $m$ term in the complexity of the M-Algorithm. As the method of multiplication in the M-Algorithm is column-by-column, there is an $n$ term in the complexity as well.

One goal is to come up with an algorithm (and a new data structure to hold the matrices if necessary) that uses $O(nnz(A)+nnz(B)+nnz(C))$ space and takes $O(\text{flops})$ time. However, a worst-case $O(\text{flops})$ algorithm is impossible to obtain in general, since for certain pairs of matrices $A$ and $B$, the product $C$ may be composed of all zeros, which means that $\text{flops} = 0$. A more realistic goal is to include the size of the input parts that are used nontrivially during the computation, which in our case means an algorithm that runs in time $O(nnz(A) + nnz(B) + \text{flops})$.

### 2.1 Need for Hypersparse Algorithms

The M-Algorithm is asymptotically suboptimal when the matrices are what we call *hypersparse*. A matrix is hypersparse if $nnz < n$. Such matrices are fairly rare in numerical linear algebra (indeed, a nonsingular square matrix

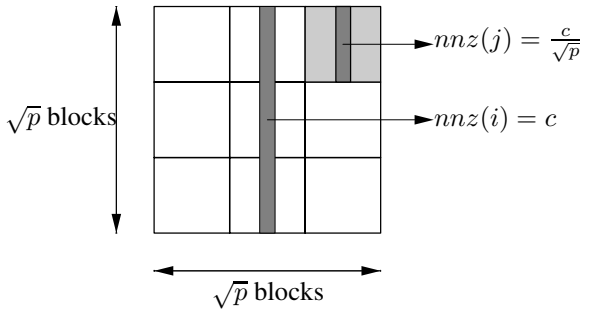


**Figure 1. 2D Sparse Matrix Decomposition**

must have $nnz \geq n$), but they occur frequently in computations on graphs, particularly in parallel graph computations. For example, the WWW graph [5] contains many rows and columns that are completely zero.

Our motivation for considering hypersparse matrices comes from parallel processing, since hypersparse matrices arise after the 2-dimensional block data decomposition of matrices for parallel processing, as in the SUMMA algorithm [9]. Assume, on the average, there are $c$ nonzero elements in each column of the matrix. After the 2D decomposition, each processor has a submatrix of size $(n/\sqrt{p}) \times (n/\sqrt{p})$. Storing each of those submatrices in CSC format has $O(n\sqrt{p}+nnz)$ space complexity, while storing the whole matrix in CSC format would only take $O(n + nnz)$ space on a single processor. This method clearly does not provide storage scalability with increasing number of processors.

A different view of the storage problem is depicted in Figure 1. The number of nonzeros in a single column of the submatrices, $nnz(j)$, goes to zero as $p$ increases. This inefficiency of CSC leads to a more fundamental problem: any algorithm that utilizes CSC and scans over all the columns becomes unscalable when matrices become hypersparse. Even without any communication at all, such an algorithm will be unscalable for $n\sqrt{p} \geq max\{\text{flops}, nnz\}$, i.e. it will eventually hit a wall as $p$ increases. To see why, consider the parallel efficiency of the M-algorithm assuming no parallel overheads (zero communication cost, no idle time due to synchronization, etc.) :

$$E = \frac{n + nnz + \text{flops}}{p(\frac{n}{\sqrt{p}} + \frac{nnz}{p} + \frac{\text{flops}}{p})} = \frac{n + nnz + \text{flops}}{n\sqrt{p} + nnz + \text{flops}}$$

Note that after the point $n\sqrt{p} \geq max\{\text{flops}, nnz\}$, no matter how much we increase our problem size we can never keep efficiency constant as the number of processors increase.

In conclusion, the M-algorithm is optimal only if flops is both $\Omega(nnz(B))$ and $\Omega(n)$.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| JC | = | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 4 |
| | | ↓ | | | | | | ↓ | ↓ | | |
| IR | = | 5 | 7 | | | | | 3 | 1 | | |
| NUM | = | 0.1 | 0.2 | | | | | 0.3 | 0.4 | | |

**Figure 2. Matrix $A$ in CSC format**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| JC($old$) | = | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 4 | |
| D | = | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | |
| CP | = | 0 | | | | | | 2 | 3 | | 4 | |
| | | ↑ | | | | | | ↑ | | | ↑ | ↑ |
| AUX | = | 0 | | | | | | 1 | | | 3 | 3 |

**Figure 3. DCSC construction from CSC**

## 3 DCSC Data Structure

The compressed sparse column (CSC) format is a common general storage format for sparse matrices. It is composed of three arrays, which we call JC, IR and NUM. They correspond to *column pointers*, *row indices* and *numerical values* respectively. The total space complexity of CSC is $O(n + nnz)$, where $nnz$ is the number of nonzeros in the whole matrix. This is because the JC array is of size $n + 1$ while the IR and NUM arrays are of size $nnz$. For an example, think about an 9-by-9 matrix A with 4 non-zeros which has the following representation in triplets format:

$$A = \{(5, 0, 0.1), (7, 0, 0.2)(3, 6, 0.3), (1, 7, 0.4)\}$$

Note that the indices start from zero. The CSC storage of matrix $A$ is shown in Figure 2.

The CSC data structure allows fast access to columns of a matrix but it is extremely slow for accessing rows. Simultaneously storing the compressed row format (CSR) along with the CSC is a theoretical remedy to this problem, but it is rarely used in practice since it doubles the storage. We notice that the JC array contains unnecessary repetitions for hypersparse matrices. Looking at the difference matrix $\mathsf{D}(i) = \mathsf{JC}(i + 1) - \mathsf{JC}(i)$, shown in Figure 3, we see that it is mostly composed of zeros.

The information theoretical solution is to compress the JC array. We define $nzc$ ($nzr$) as the number of columns (rows) containing at least one nonzero element. If we compress JC at a rate of $cf = (n+1)/nzc$, we get a compressed array of size $nzc$. Thus, by removing all the repetitions in JC (columns that are completely zero), we can get rid of the $n$ term in the storage complexity.

Removing the zero columns introduces an indexing problem, since we can no longer access the first element

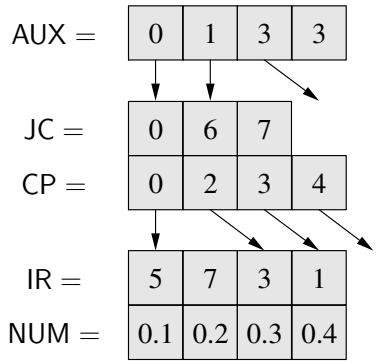


**Figure 4. Matrix $A$ in DCSC format**

in a column in constant time. To remedy this problem, we introduce an AUX array of size $(n+1)/\lceil cf \rceil \approx nzc$ that contains pointers to nonzero columns (columns that contain at least one nonzero element). That practically means chopping the JC array into different chunks, each having size $\lceil cf \rceil$. Therefore, AUX contains exactly one element per chunk that is pointing to the first nonzero column in that chunk, and an extra element at the very end to serve as the null pointer, as shown in Figure 3. Note that, on the average, there will be only one nonzero column in each chunk, although in practice the distribution may be skewed and there may be up to $\lceil cf \rceil$ nonzero columns in a single chunk.

In order to allow for efficient searching of columns within a chunk, we will store the column indices in the JC array and the pointers to the IR array will be held in a new array called CP (column pointers). We call this new format DCSC (doubly compressed sparse column). Matrix $A$ is illustrated in Figure 4 in DCSC format. The storage requirement of DCSC is clearly $O(nnz)$ since $|\mathsf{NUM}| = |\mathsf{IR}| = nnz$, $|\mathsf{JC}| = nzc$, $|\mathsf{CP}| = nzc + 1$, and $|\mathsf{AUX}| \approx nzc$.

## 4 Indexing in DCSC

Suppose that we need to access the element at the $(i,j)^{\text{th}}$ position of matrix $A$. Our indexing scheme should return the element $A(i,j)$ if it exists or *zero* if it does not. Assume that the size of each chunk, $chunk = \lceil cf \rceil$, is available to the program. The full scheme is given in Algorithm 4.1. Since both the AUX array and the row indices within a given column are sorted, both searches can be performed as binary searches.

As an example, assume that we need to access $A(3,7)$. That belongs to the second chunk since $\lfloor 7/4 \rfloor = 1$ (remember that chunk indices also start from zero) and that chunk has two non-empty columns. The algorithm then searches for 7 inside the subarray JC[1..2]. Since there is indeed an element with column index 7, the search is successful and returns an index to the CP array, namely it returns $p = 2$. Finally, the algorithm searches for the row index 3 inside the subarray IR[CP[2]..CP[3] $-$ 1] $=$ IR[3..3] and returns *zero* since the only nonzero element in the $7^{\text{th}}$ column has a row index of 1.

```
1  start ← AUX[⌊j/chunk⌋] ;
2  end ← AUX[⌊j/chunk⌋ + 1] ;
3  pos ← Search (j, JC[start... (end-1)]) ;
4  if pos = null then
5      return 0 ;
6  else
7      startc ← CP[pos] ;
8      endc ← CP[pos + 1] ;
9      posc ← Search (i, IR[startc... (endc-1) ]) ;
10     if posc = null then
11         return 0 ;
12     else
13         return NUM[posc] ;
14     end
15 end
```

**Algorithm 4.1**: **Indexing for** $A(i,j)$

Similarly, one can perform the column-wise indexing $A(:,j)$ by stopping the procedure at line 8 and returning the submatrix with row indices IR[$startc...endc-1$], column indices all $j$'s, and numerical values NUM[$startc...endc-1$].

The expected cost of column-wise indexing is constant assuming that nonzero columns (columns that contain at least one nonzero) are distributed evenly. Worst case performance of column-wise indexing is $\lg \lceil cf \rceil$ since we can use binary search when the distribution of nonzero columns becomes skewed. Binary search can also be used for element-wise indexing, giving an expected cost of $\lg nnz(A(:,j))$ and worst case cost of $\lg \lceil cf \rceil + \lg nnz(A(:,j))$.

## 5 Multiplication Algorithms

### 5.1 Algorithm I

Algorithm I is based on outer product multiplication. It assumes that each input matrix is represented in DCSC and also in DCSR (doubly compressed sparse row), which doubles the storage but does not change the asymptotic space complexity. DCSR is the row-based dialect of DCSC where the columns and rows are interchanged. For a matrix $A$, $A.dcsc$ and $A.dcsr$ represent $A$ in DCSC format and $A$ in DCSR format. Our goal is to compute $C = AB$, where $A$ and $B$ are represented in both DCSC-DCSR formats; thus, the output $C$ will also be represented in both formats.

| Index | SizeA | SizeB | StartA | StartB |
|---|---|---|---|---|
| 0 | 2 | 2 | 0 | 0 |
| 6 | 1 | 3 | 2 | 3 |

**Table 1. Intersection Table (Isect)**

The first observation about DCSC is that the JC array is already sorted. Therefore, $A.dcsc.\mathsf{JC}$ is the sorted indices of the columns that contain at least one nonzero and similarly $B.dcsr.\mathsf{IR}$ is the sorted indices of the rows that contain at least one nonzero. The idea behind the algorithm is to use the outer product formulation of matrix multiplication efficiently. In this formulation, the $i^{\text{th}}$ column of $A$ and the $i^{\text{th}}$ row of $B$ are multiplied to form a rank-1 matrix. The naive algorithm does the same procedure for all values of $i$ and gets $n$ different rank-1 matrices. Finally it adds them to get the resulting matrix $C$. Our algorithm has a preprocessing step where the intersection $\text{Isect} = A.dcsc.\mathsf{JC} \cap B.dcsr.\mathsf{IR}$ is found, which gives us the exact set of indices that we need to do the outer product.

For an example, assume

$$A = \{(5, 0, 0.1), (7, 0, 0.2), (3, 6, 0.3), (1, 7, 0.4)\},$$

$$B = \{(0, 2, 1.1), (0, 8, 1.2), (3, 4, 1.3), (6, 2, 1.4),$$
$$(6, 4, 1.5), (6, 5, 1.6), (8, 8, 1.7)\},$$

where $A$ is the same as in Section 2.1. The output of the intersection (Isect) is shown in Table 1. Here, index column shows the indices that occur in both $A.dcsc.\mathsf{JC}$ and $B.dcsr.\mathsf{IR}$. SizeA and SizeB stand for the number of elements with that particular row/column index in the corresponding matrix. StartA and StartB are initial pointers to the CP arrays. It is easy to see that the preprocessing takes $O(nzc(A) + nzr(B))$ time due to intersection. From now on, we will focus mainly on constructing $C.dcsr$ from $A.dcsc$ and $B.dcsr$. Constructing $C.dcsc$ will be almost identical except columns being replaced by rows and vice-versa. Note that we never need $A.dcsr$ and $B.dcsc$ for constructing the resulting matrix $C$. Therefore, in our pseudocode, we omit the middle identifiers for clarity (e.g., $A.\mathsf{CP}$ stands for $A.dcsc.\mathsf{CP}$).

The crucial observation that helps us develop phase 1 of our algorithm is that we can think of each row in Table 1 as a fictitious list of size $nnz(A(:, i))\, nnz(B(i, :))$. Combining this with the fact that all the elements within a given column/row $i$ are sorted according to their row/column indices (i.e. $\mathsf{IR}[\mathsf{JC}[i]]...\mathsf{IR}[\mathsf{JC}[i] + 1]$ is a sorted range), we conclude that the problem is similar to multiway merging [15]. The only difference is that we will never explicitly construct the lists; we will compute their elements one-by-one on demand. Figure 5 shows a graphical representation of phase 1. Here, the algorithm does not touch the shaded elements, since they do not contribute to the output.

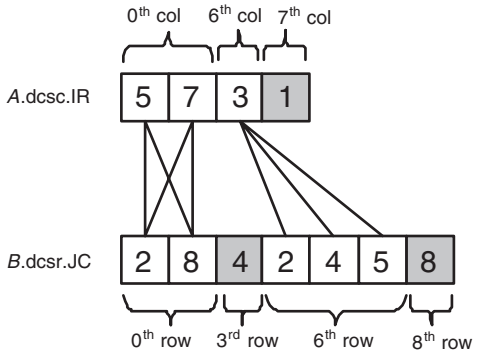


**Figure 5. Graphical Illustration of Phase-1**

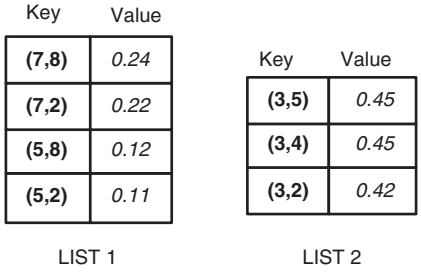


**Figure 6. Multiway Merging Analogy**

The merging algorithm uses a heap (priority queue) of size $ni = |\text{Isect}|$, where the value of a heap entry is its NUM value and the key of a heap entry is $(i, j)$ with lexicographical (row major) order. The idea is to repeatedly extract-min from the heap and insert another element from the list that the currently extracted element originally belonged to. If there are multiple elements in the lists with the same key, then their values are summed on the fly. If we were to explicitly create $ni$ lists instead of doing the computation on the fly, we would get the lists shown in Figure 6, which are sorted from bottom to top. Note that this example is specifically for constructing $C.dcsr$. For constructing $C.dcsc$ part, all we need to do is to create the heap keys in column major order instead of row major (note that the order in the fictitious lists will be different then). For further details of multiway merging, consult Knuth [15].

The time complexity of phase 1 is then $O(\text{flops} \lg ni)$ where the space complexity is $O(nnz(C) + ni)$. The output of phase 1 is two stacks of NUM values ordered lexicographically (row major for constructing $C.dcsr$ and column major for constructing $C.dcsc$). Note that $nnz(C)$ term in the space complexity comes from the output and flops term

in time complexity comes from the observation that

$$\sum_{i \in \text{Isect}} nnz(A(:,i))\, nnz(B(i,:)) = \text{flops}.$$

Phase 2 of the algorithm constructs the DCSC and DCSR structures from these lexicographically ordered stacks. This can be done in $O(nnz(C))$ time and space as long as the final data structure is $O(nnz(C))$ as in the case of DCSC and DCSR.

The overall time complexity of our algorithm is $O(nzc(A)+nzr(B)+\text{flops} \lg ni)$. Note that $nnz(C)$ does not appear in this formula since $nnz(C) \leq \text{flops}$. Overall space complexity is $O(nnz(A)+nnz(B)+nnz(C))$ only. The time complexity does not depend on $n$, and space complexity does not depend on flops.

The full pseudocode for the whole algorithm is given below. It uses two subprocedures: `MultInsI` generates the next element from the $ti^{\text{th}}$ fictitious list and inserts it to the heap PQ, `IncrementList` increments the pointers of the $ti^{\text{th}}$ fictitious list or deletes the list from the intersection table if it is depleted.

**Procedure** `IncrementList` (Isect, *ti*)

```
1  if Isect[ti].curb < Isect[ti].sizeb then
2  |  Isect[ti].curb ← Isect[ti].curb + 1 ;
3  else
4  |  Isect[ti].curb ← Isect[ti].startb ;
5  |  if Isect[ti].cura < Isect[ti].sizea then
6  |  |  Isect[ti].cura ← Isect[ti].cura + 1 ;
7  |  else
8  |  |  Isect.Delete (ti) ;
9  |  end
10 end
```

**Procedure** `MultInsI` (*A*, *B*, PQ, Isect, *ti*)

```
1  ptra ← A.CP[Isect[ti].cura] ;
2  ptrb ← B.CP[Isect[ti].curb] ;
3  product ← A.NUM[ptra] * B.NUM[ptrb] ;
4  key ← Pair (A.IR[ptra], B.JC[ptrb]) ;
5  value ← Pair (product, ti) ;
6  PQ. Insert (key,value) ;
```

A final note is for the curious about the constants in the algorithm and the data structure. The core of the algorithm is actually computed twice: once to construct DCSR and once again to construct DCSC, therefore doubling the work. Apart from that, the requirement that matrices should be represented in both DCSR and DCSC increases the storage to $4\,nnz+6\,nzc$, which may be unacceptable for some practical applications. We propose a variant of Algorithm I to remedy those problems. In this variant, the input matrices are represented only in DCSC format. Before the multiplication $C = AB$ is performed, $B$ is transposed so that we get the DCSR representation of $B$. Transposition in this case is the lexicographical sorting with $nnz(B)\ \lg nnz(B)$ cost. Consequently, we halve the constant in flops $\lg ni$ term and we reduce the size of the data structure to $2\,nnz + 3\,nzc$, at the price of increasing the overall complexity to $O(nzc(A)+nnz(B)\lg nnz(B)+\text{flops}\ \lg ni)$. We experimentally evaluate the performance of this DCSC-only variant in Section 6.

```
   /* Preprocessing                          */
1  Isect ← Intersection (A.JC, B.IR) ;
   /* Phase 1                                */
2  forall ti ∈ Isect do
3  |  MultInsI (A,B,PQ, Isect, ti) ;
4  |  IncrementList (Isect, ti) ;
5  end
6  while Isect.IsNotFinished do
7  |  (key,value) ← PQ.ExtractMin ;
8  |  (product, ti) ← Unpair (value) ;
9  |  if key ≠ Q.Top then
10 |  |  Q.Insert (key, product) ;
11 |  else
12 |  |  Q.UpdateTop (key, product) ;
13 |  end
14 |  if Isect[ti].IsNotEmpty then
15 |  |  MultInsI (A,B,PQ, Isect, ti) ;
16 |  |  IncrementList (Isect, ti) ;
17 |  end
18 end
   /* Phase 2                                */
19 ConstructDcsr (Q) ;
```

**Pseudocode for Algorithm I**

## 5.2 Algorithm II

Our second algorithm is based on the column-by-column formulation of the multiplication as the M-algorithm. For Algorithm II, it is sufficient for matrices to be represented in only one format, either DCSC or DCSR. We assume that the inputs as well as the outputs are represented in DCSC. Similar to the M-algorithm, we will be computing a whole column of $C$ in one step by examining the same column of $B$. In other words, $C(:,j)$ will be a linear combination of the columns $A(:,i)$ for which $B(i,j) \neq 0$. Algorithm II's complexity, however, is independent of $n$ and $m$ since it neither scans all the columns of matrix $B$, nor it uses an SPA.

For the construction of $C(:,j)$, we use a heap of size $nnz(B(:,j))$ to help the merging process. As in the case of Algorithm I, we require the row indices within a given column to be sorted. Luckily, this requirement is already sat-

isfied by DCSC. The idea of merging columns using a heap has been employed before, within the Ordered-SPA [13] data structure. However, it was never used to suppress the $m$ factor in the algorithm because Ordered-SPA is still an $\Theta(m)$ data structure. Furthermore, the Ordered-SPA uses a heap of size $nnz(C(:,j))$, which can be much bigger than $nnz(B(:,j))$.

Let us illustrate the execution of the algorithm through the example inputs $A$ and $B$ given below:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 3 & 0 & 4 \\ 6 & 0 & 0 & 0 \\ 0 & 5 & 5 & 5 \end{pmatrix}, B = \begin{pmatrix} 7 & 0 & 2 & 0 \\ 3 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{pmatrix}$$

By looking at the first column of $B$, $B(:,0) = \begin{bmatrix} 7 & 3 & 0 & 0 \end{bmatrix}^T$, we know the set of column indices of $A$ that will be required during the construction of $C(:,0)$ (in this case they are the $1^{\text{st}}$ and $2^{\text{nd}}$ columns of $A$). We can now do a multiway merge, with a heap of size $B(:,0) = 2$ as follows: Initially, we insert $(key, value) = (2, 7 * 6)$ and $(key, value) = (1, 3 * 3)$. Then we start a pop/push sequence. After each pop operation, we will insert another element from the column that the popped elements originally belonged to. In our example we first pop $(1, 3 * 3)$ and then insert $(3, 3 * 5)$. Finally when the columns are depleted, we pop all the remaining elements from the heap. The row index alone is sufficient as the key because Algorithm II constructs one column at a time and all the elements in that column has the same column index. The high level pseudocode for the algorithm is given below, which utilizes a slightly different `MultInsII` subroutine.

**Procedure** `MultInsII` (PQ, Lists, *i*, bval)

```
1 product ← Lists[i].val * bval ;
2 value ← Pair (product, i) ;
3 key ← Lists[i].ind ;
4 PQ.Insert (key, value) ;
5 Advance (Lists[i]) ;
```

The complexity of the algorithm depends on the distribution of nonzeros in matrix $B$. From Section 4, we know we can access a given column of matrix $B$ in expected constant time assuming that nonzero columns of $B$ (columns that contain at least one nonzero) are distributed evenly. When this is the case, Algorithm II has an expected cost of

$$\sum_{j=0}^{nzc(B)} \text{flops}(C(:,j)) \lg nnz(B(:,j)).$$

Here, $\text{flops}(C(:,j))$ is the number of nonzero multiplications required to generate the $j^{\text{th}}$ column of $C$. When

```
1  for j ← 0 to nzc(B) do
2      foreach B(i,j) ∈ B(:,j) do
3          Lists[i] ← SparseList (A(:,i)) ;
4          MultInsII (PQ, Lists, i, B(i,j)) ;
5      end
6      while Lists.IsNotFinished do
7          (key,value) ← PQ.ExtractMin ;
8          (product, i) ← Unpair (value) ;
9          if key ≠ Q.Top then
10             Q.Insert (key, product) ;
11         else
12             Q.UpdateTop (key, product) ;
13         end
14         if Lists[i].IsNotEmpty then
15             MultInsII (PQ, Lists, i, B(i,j)) ;
16         else Lists.Delete (i) ;
17     end
18     C(:,j) ← Output (Q) ;
19     Reset (Q) ;
20 end
```

**Pseudocode for Algorithm II**

the inputs are random matrices, matrices representing structured graphs (such as 3D geometric graphs) or permutation matrices, $\lg nnz(B(:,j))$ is constant (in expected sense for random matrices, and in exact sense for others). Furthermore, nonzero columns are distributed evenly for those matrices. Let the number of nonzeros in any column of $B$ be $c$. Then, for those families of matrices, the expected cost of Algorithm II is

$$\lg c \sum_{j=0}^{nzc(B)} \text{flops}(C(:,j)) = O(\text{flops} \lg c + nzc(B)).$$

It is worth mentioning that $C(:,j)$ could have been constructed using the SPA [11] data structure instead of a heap. This decision is reminiscent to choosing buckets or heaps/looser trees for merging sorted lists. An SPA-like data structure makes sense when it is not advantageous to use our hypersparse algorithms, i.e. when the matrices are not sparse enough. This gives us another subalgorithm based on DCSC data structure that we can incorporate into the polyalgorithm. Due to the time spent in the initialization of SPA, such an algorithm has expected time complexity of $O(n + \text{flops})$ for all inputs.

## 6 Implementation and Experimentation

We have implemented our data structures and multiplication algorithms in C++. The code is compiled using the GNU Compiler Collection (GCC) Version 4.1, with the

$$A = \begin{pmatrix} A_{11} & ... & A_{1\sqrt{p}} \\ ... & ... & ... \\ A_{\sqrt{p}1} & ... & A_{\sqrt{p}\sqrt{p}} \end{pmatrix}$$

**Figure 7.** Square Grid Decomposition

flags `-O6`. We compare the performance of our implementation with Matlab R2007A (64-bit version). Our self contained library is generic in two ways. First, each entry in the NUM array is templated; it may be composed of reals, integers, and particularly submatrices. Second, multiplication and addition operators are passed as function objects, allowing sparse matrix multiplication to be performed on arbitrary semirings. Throughout the experiments, however, we have chosen the NUM array to be composed of doubles and multiplication/addition operators to be regular multiplication/addition operators on the field of real numbers, to allow a fair comparison with Matlab.

Sparse matrix multiplication is a built-in function in Matlab, so there are no interpretation overheads associated with it. We are simply comparing our C++ code with the underlying precompiled C code used in Matlab. We have incorporated Peter Sander's Sequence Heaps [23] for all the priority queues used by our algorithms.

All of our experiments are performed on a single core of Opteron 2.2 Ghz with 64 GB main memory, where we simulate the execution of a parallel SpGEMM. The simulation is done by dividing the input matrices of size $n \times n$ into $p$ submatrices of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ using the 2D block decomposition. In a real parallel SpGEMM, executing on $p$ processors that forms a $\sqrt{p} \times \sqrt{p}$ grid, the $(i,j)^{\text{th}}$ processor would be denoted by $P_{ij}$, and submatrices $A_{ij}$ and $B_{ij}$ would be assigned to $P_{ij}$. Such a decomposition of matrix $A$ is depicted in Figure 7.

We performed sequential simulations instead of experimenting with a parallel implementation because the main contributions of our work are the sequential hypersparse matrix multiplication algorithms. This way, we were able to decouple the cost of submatrix multiplications from other costs such as updates and parallelization overheads. This choice was also motivated by the fact that there are few parallel implementations of SpGEMM to compare our work with. Our parallel code is being tuned and we will report on it elsewhere.

Expressing the matrix multiplication as algebraic operations on submatrices instead of individual elements, we see that each submatrix of the product is computed using $C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik}\, B_{kj}$. Since we are primarily concerned with the sequential sparse matrix multiplication kernel, we will exclude the cost of submatrix additions and other parallel overheads. That is to say, we will only time the submatrix multiplications, exactly plotting

$$time(p, A, B) = \sum_{i=1}^{\sqrt{p}} \sum_{j=1}^{\sqrt{p}} \sum_{k=1}^{\sqrt{p}} time(A_{ik}\, B_{kj}),$$

which is equal to the amount of work done by a parallel matrix multiplication algorithm such as SUMMA [9].

Increasing $p$ in this case does not mean we use more processors to compute the product. Instead, it means we use smaller and smaller blocks while computing the product on a single processor. Therefore, a perfectly scalable algorithm would yield flat timing curves as $p$ increases. We expect our algorithms to outperform the M-Algorithm as $p$ increases due to reasons explained in Section 2.1. We label the original Algorithm I *Alg 1A* and the DCSC-only variant *Alg 1B*. Algorithm II is labeled *Alg 2*, and the M-Algorithm is denoted by *Matlab*. In Alg 1B, each submatrix $A_{ij}$ is transposed only once instead of $\sqrt{p}$ times, because this is what would happen in a smart parallel implementation.

We originally included Matlab Tensor Toolbox [2] in our tests, which supports sparse tensors and sparse tensor multiplication. However, it was consistently 4-5 times slower than any of the other algorithms, probably because it was optimized for tensors with higher dimensions than two. Therefore, we excluded the results from Tensor Toolbox from our plots.

In all experiments, the input matrices have dimensions $2^{23} \times 2^{23}$, i.e. the input graphs have around 8 million vertices. Our test matrices comes from several graph families that are described in detail in the following subsections.

### 6.1 Models of Real-World Graphs

Our first set of experiments are conducted on matrices that represent the adjacency structure of Kronecker graphs, scale-free graphs that are generated using repeated Knonecker products (KronMat). KronMat models the behavior of several real-world graphs such as the WWW graph, small world graphs, and citation graphs [16]. We have used an implementation based on Kepner's vectorized code [3], which generates directed graphs with an average of degree of 8, meaning that there will be approximately $8\,n$ nonzeros in the adjacency matrix. Thus, KronMat graphs we have used in our experiments have a total of $2^{26}$ edges each. We ran two main sets of multiplication experiments with real world graphs, one where both input matrices are KronMat, and one where $A$ is a KronMat matrix and $B$ is a permutation matrix. The results are shown in Figures 8 and 9.

In the case of KronMat $\times$ KronMat, the M-Algorithm is initially faster than both variants of Algorithm I and has about the same speed as Algorithm II. Up to $p = 16$, the ranking stays roughly the same, but the column-by-column

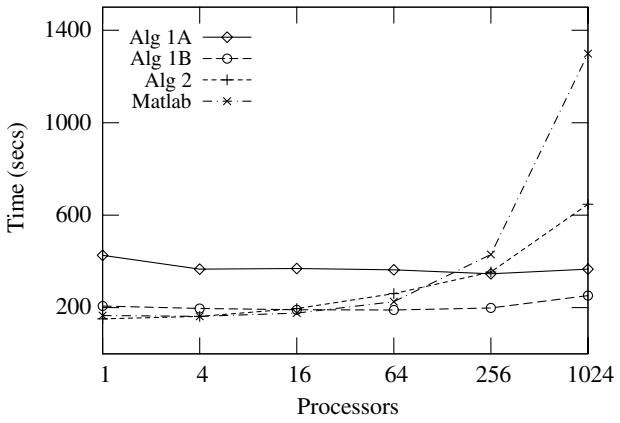


**Figure 8. Scalability of SpGEMM kernels for multiplying two matrices from real-world graphs (KronMat $\times$ KronMat)**

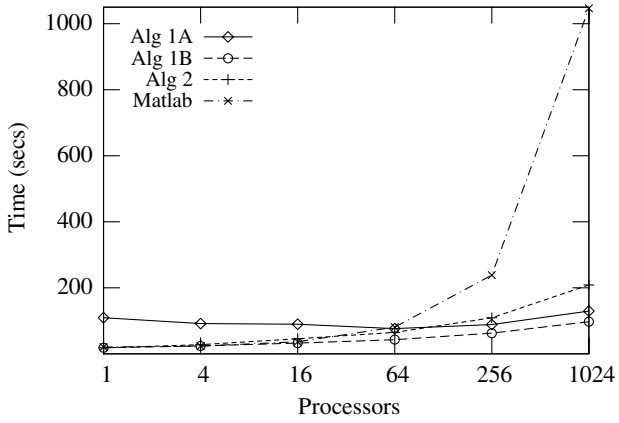


**Figure 9. Scalability of SpGEMM kernels for multiplying a real-world graph matrix with a permutation matrix (KronMat $\times$ Perm)**

algorithms (Matlab and Alg 2) show slight increases in overall execution times. For $p > 64$, however, the M-Algorithm starts performing poorly because submatrices start getting hypersparse. To see why, consider the ratio of $nnz$ to $n$ for each submatrix:

$$\frac{nnz(A_{ij})}{n/\sqrt{p}} = \frac{8\,n/p}{n/\sqrt{p}} = \frac{8}{\sqrt{p}}$$

This ratio is smaller than 1 for $p > 64$, making submatrices hypersparse.

While all of our algorithms are more scalable than the M-Algorithm, the variants of Algorithm I that are based on outer product formulation scale particularly well, showing almost flat curves. For $p = 1024$, Alg 1B performs more than 5 times faster than the M-Algorithm. Also, a polyalgorithm using the DCSC data structure should choose Alg 2 for $p < 16$ and Alg 1B for $p \geq 16$.

In the case of multiplying a KronMat matrix with a permutation matrix (KronMat $\times$ Perm), M-Algorithm's poor scalability is more apparent. All of our algorithms outperform the M-Algorithm for $p \geq 64$ with Alg 1B starting to outperform for as low as $p > 4$. The break-even point after which our algorithms dominate the M-algorithm is lower in this case because permutation matrices are more sparse with only 1 nonzero per column/row. This time, Alg 1B is faster than Alg 2 for almost all values of $p$, making it the algorithm of choice in a future polyalgorithm that executes on matrices represented only in DCSC format.

## 6.2 Erdős-Rényi Random Graphs

In the Erdős-Rényi random graph model, each possible edge in the graph exists with fixed probability $p$. In this set of experiments, we have generated matrices with approximately $7\,n$ nonzeros, using the `sprand` function of Matlab. We have conducted a single set of experiments where we multiply two matrices representing Erdős-Rényi random graphs.

Looking at the timings shown in Figure 10, we see that the M-Algorithm is dominated by our new algorithms for most values of $p$. Up to 16 processors, Alg 2 is the fastest; while for $p > 64$, Alg 1B turns out to be the fastest. More importantly, when we reach thousands of processors, our algorithms show their scalability for this type of inputs also. In particular, Alg 1B is more than 4 times faster than the M-Algorithm for 1024 processors when multiplying random matrices.

## 6.3 Regular 3D Grids

For our last set of experiments, we have used matrices arising from graphs representing the 3D 7-point finite difference mesh (`grid3d`). These input matrices, which are generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [10], are highly structured block diagonal matrices. Such band matrices are unsuitable for 2D block decomposition since the off-diagonal processors sit idle without storing any nonzeros and performing any computation. To remedy this problem, we perform random permutations of vertices on both inputs before performing the multiplication. In other words, instead of computing $C = AB$, we will compute $C' = A'B' = (PAP^T)(PBP^T) = PAIBP^T = PCP^T$ where $I$ is the identity matrix.

Even after applying random symmetric permutations, imbalances remain. Submatrices in the diagonal are ex-

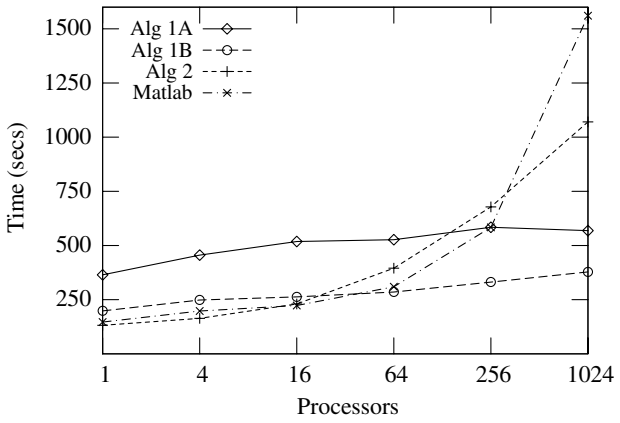


**Figure 10. Scalability of SpGEMM kernels for multiplying two matrices from Erdős-Rényi graphs (Rand $\times$ Rand)**

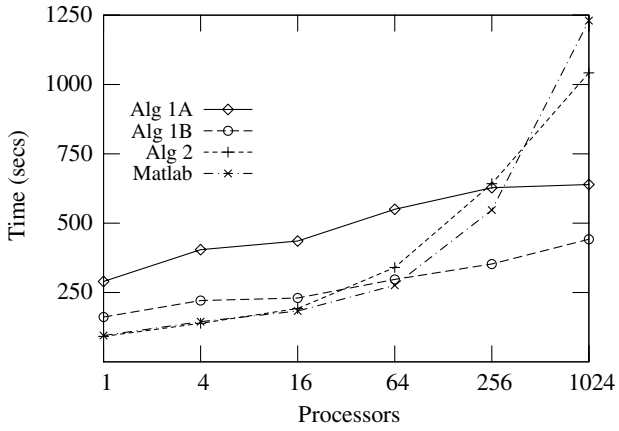


**Figure 11. Scalability of SpGEMM kernels for multiplying two matrices from geometric graphs (Grid3D $\times$ Grid3D )**

pected to have more nonzeros than others. This is because symmetric permutations essentially relabel the vertices of the underlying graph, so they are unable to scatter the nonzeros in the diagonal. Multiplications among diagonal blocks favor Matlab because diagonal blocks can never become hypersparse no matter how much $p$ increases. Multiplication among off-diagonal blocks are more suitable for our hypersparse algorithms. More technically, our observation means

$$\text{flops}(A_{ii}\, B_{ii}) > \text{flops}(A_{ii}\, B_{ij}) > \text{flops}(A_{ik}\, B_{kj}).$$

Therefore, the variances in timings of submatrix multiplications are large compared with other sets of test matrices.

Asymptotic behavior of the algorithms is also slightly different in this case as it can be seen in Figure 11. Yet, Alg 1B is around 4 times faster than the M-Algorithm for $p = 1024$ and Alg 2 is competitive with M-Algorithm for almost all values of $p$.

## 7 Conclusions and Future Work

We considered the problem of sparse matrix multiplication (SpGEMM). We introduced the notion of hypersparsity for matrices that arise after the 2D data decomposition of matrices for parallel processing. We presented two new algorithms to be used as sequential SpGEMM kernels, and we experimentally demonstrated that our algorithms are faster than existing algorithms when hypersparsity is present. Variants of the algorithms presented here work on the same DCSC data structure, which allows them to be used in a polyalgorithm for SpGEMM that executes different kernels depending on the sparsity of the input matrices. This is a crucial feature since switching data structures on the fly is rarely practical. For the polyalgorithm, we guarantee a time complexity of $O(min\{nzc(A)+ nnz(B) \lg nnz(B) + \text{flops} \lg ni, \text{flops} + n\})$

A limitation of our work is the absence of a lower bound for SpGEMM. Therefore, we do not know whether our algorithms are optimal. Achieving optimality in the output/work sensitive setting is harder than the general setting. In the general setting, one can come up with pathological inputs which inherently requires as much work as the algorithm does in the worst case, providing a lower bound for the problem and hence making the algorithm optimal. However, an output/work sensitive algorithm should perform optimally well for all pairs of inputs to be considered optimal. Whether or not such an optimal algorithm for SpGEMM exists in the output/work sensitive setting is an open question.

We have not considered the cache efficiency of our algorithms in this paper. As the memory hierarchies became dominant in computer architectures, the cache complexity of a given algorithm became as important as its RAM complexity. We intend to perform future work on the development of cache-aware and cache-oblivious algorithms for SpGEMM. In the cache-aware and cache-oblivious settings, optimal algorithms exists for the problem of SpMV [4], but we have not encountered any theoretically optimal algorithms for SpGEMM although there has been some empirical work on the subject [25].

Since parallelism is the main motivation of our work, we are also experimenting with parallel algorithms based on the sequential kernels we have introduced in this paper. Further research is required for finding the best way to perform updates on the sparse matrices in the case of parallelism. We

intend to publish our results with parallel SpGEMM in a future paper.

## References


[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[2] B. W. Bader and T. G. Kolda. Efficient Matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[3] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2. version 1.1.

[4] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures*, pages 61–70, New York, NY, USA, 2007. ACM Press.

[5] T. Bennouas and F. de Montgolfier. Random web crawls. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 451–460, New York, NY, USA, 2007. ACM Press.

[6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.

[7] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[8] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transactions on Mathematical Software*, 28(2):239–267, 2002.

[9] R. A. V. D. Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[10] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[11] 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.

[12] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.

[13] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 249–256, London, UK, 1997. Springer-Verlag.

[14] H. Kaplan, M. Sharir, and E. Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *SCG '06: Proceedings of the twenty-second annual symposium on computational geometry*, pages 52–60, New York, NY, USA, 2006. ACM Press.

[15] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): Fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[16] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *PKDD*, pages 133–145, 2005.

[17] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[18] B. M. Maggs and S. A. Poltkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, 1988.

[19] P. os and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6(1):290–297, 1959.

[20] S. C. Park, J. P. Draayer, and S.-Q. Zheng. Fast sparse matrix multiplication. *Computer Physics Communications*, 70:557–568, July 1992.

[21] G. Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354(1):72–81, 2006.

[22] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4):557–567, 1989.

[23] P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics*, 5:7, 2000.

[24] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[25] P. Sulatycke and K. Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium*, page 117, Washington, DC, USA, 1998. IEEE Computer Society.

[26] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

[27] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/L. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2005. IEEE Computer Society.

[28] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on discrete algorithms*, pages 254–260, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[29] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.

[30] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.