Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Chapter 13

Implementing Sparse Matrices for Graph Algorithms

Aydın Buluç[*], *John Gilbert*[†], *and Viral B. Shah*[†]

Abstract

Sparse matrices are a key data structure for implementing graph algorithms using linear algebra. This chapter reviews and evaluates storage formats for sparse matrices and their impact on primitive operations. We present complexity results of these operations on different sparse storage formats both in the random access memory (RAM) model and in the input/output (I/O) model. RAM complexity results were known except for the analysis of sparse matrix indexing. On the other hand, most of the I/O complexity results presented are new. The chapter focuses on different variations of the triples (coordinates) format and the widely used compressed sparse row (CSR) and compressed sparse column (CSC) formats. For most primitives, we provide detailed pseudocodes for implementing them on triples and CSR/CSC.

13.1 Introduction

The choice of data structure is one of the most important steps in algorithm design and implementation. Sparse matrix algorithms are no exception. The representation of a sparse matrix not only determines the efficiency of the algorithm, but also influences the algorithm design process. Given this bidirectional relationship,

[*]High Performance Computing Research, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720 (`abuluc@lbl.gov`).

[†]Computer Science Department, University of California, Santa Barbara, CA 93106-5110 (`gilbert@cs.ucsb.edu`, `viral@mayin.org`).

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

this chapter reviews and evaluates sparse matrix data structures with key primitive operations in mind. In the case of array-based graph algorithms, these primitives are sparse matrix vector multiplication (SpMV), sparse general matrix matrix multiplication (SpGEMM), sparse matrix reference/assignment (SpRef/SpAsgn), and sparse matrix addition (SpAdd). The administrative overheads of different sparse matrix data structures, both in terms of storage and processing, are also important and are exposed throughout the chapter.

Let $\mathbf{A} \in \mathbb{S}^{M \times N}$ be a sparse rectangular matrix of elements from an arbitrary semiring $\mathbb{S}$. We use $nnz(\mathbf{A})$ to denote the number of nonzero elements in $\mathbf{A}$. When the matrix is clear from context, we drop the parentheses and simply use nnz. For sparse matrix indexing, we use the convenient MATLAB colon notation, where $\mathbf{A}(:, i)$ denotes the ith column, $\mathbf{A}(i, :)$ denotes the ith row, and $\mathbf{A}(i, j)$ denotes the element at the (i, j)th position of matrix $\mathbf{A}$. For one-dimensional arrays, $\mathbf{a}(i)$ denotes the ith component of the array. Indices are 1-based throughout the chapter. We use flops($\mathbf{A}\, op\, \mathbf{B}$) to denote the number of nonzero arithmetic operations required by the operation $\mathbf{A}\, op\, \mathbf{B}$. Again, when the operation and the operands are clear from the context, we simply use flops. To reduce notational overhead, we take each operation's complexity to be at least one, i.e., we say $O(\cdot)$ instead of $O(\max(\cdot, 1))$.

One of the traditional ways to analyze the computational complexity of a sparse matrix operation is by counting the number of floating-point operations performed. This is similar to analyzing algorithms according to their RAM complexities (see [Aho et al. 1974]). As memory hierarchies became dominant in computer architectures, the I/O complexity (also called the cache complexity) of a given algorithm became as important as its RAM complexity. Cache performance is especially important for sparse matrix computations because of their irregular nature and low ratio of flops to memory access. One approach to hiding the memory-processor speed gap is to use massively multithreaded architectures [Feo et al. 2005]. However, these architectures have limited availability at present.

In the I/O model, only two levels of memory are considered for simplicity: a fast memory and a slow memory. The fast memory is called cache and the slow memory is called disk, but the analysis is valid at different levels of memory hierarchy with appropriate parameter values. Both levels of memories are partitioned into blocks of size L, usually called the cache line size. The size of the fast memory is denoted by Z. If data needed by the CPU is not found in the fast memory, a *cache miss* occurs, and the memory block containing the needed data is fetched from the slow memory. The I/O complexity of an algorithm can be roughly defined as the number of memory transfers it makes between the fast and slow memories [Aggarwal & Vitter 1988]. The number of memory transfers does not necessarily mean the number of words moved between fast and slow memories, since the memory transfers happen in blocks of size L. For example, scanning an array of size N has I/O complexity N/L. In this chapter, $scan(\mathbf{A}) = nnz(\mathbf{A})/L$ is used as an abbreviation for the I/O complexity of examining all the nonzeros of matrix $\mathbf{A}$ in the order that they are stored. Figure 13.1 shows a simple memory hierarchy with some typical latency values as of today. Meyer, Sanders, and Sibeyn provided a contemporary treatment of algorithmic implications of memory hierarchies [Meyer et al. 2002].

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

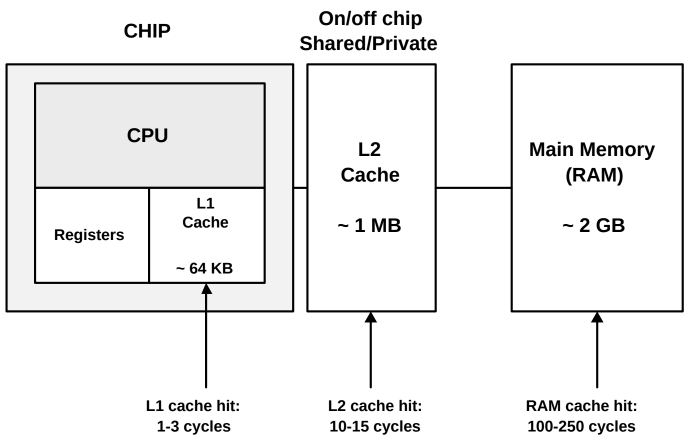

Figure 13.1. Typical memory hierarchy.
Approximate values of memory size and latency (partially adapted from Hennessy and Patterson [Hennessy & Patterson 2006]) assuming a 2 GHz processing core.

We present the computational complexities of algorithms in the RAM model as well as the I/O model. However, instead of trying to come up with the most I/O efficient implementations, we analyze the I/O complexities of the most widely used implementations, which are usually motivated by the RAM model. There are two reasons for this approach. First, I/O optimality is still an open problem for some of the key primitives presented in this chapter. Second, I/O efficient implementations of some key primitives turn out to be suboptimal in the RAM model with respect to the amount of work they do.

Now, we state two crucial assumptions that are used throughout this chapter.

Assumption 1. *A sparse matrix with dimensions $M \times N$ has $nnz \geq M, N$. More formally, $nnz = \Omega(N, M)$.*

Assumption 1 simplifies the asymptotic analysis of the algorithms presented in this chapter. It implies that when both the order of the matrix and its number of nonzeros are included as terms in the asymptotic complexity, only nnz is pronounced. While this assumption is common in numerical linear algebra (it is one of the requirements for nonsingularity), in some parallel graph computations it may not hold. In this chapter, however, we use this assumption in our analysis. In contrast, Buluç and Gilbert gave an SpGEMM algorithm specifically designed for *hypersparse* matrices (matrices with $nnz < N, M$) [Buluç & Gilbert 2008b].

Assumption 2. *The fast memory is not big enough to hold data structures of $O(N)$ size, where N is the matrix dimension.*

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

We argue that Assumption 2 is justified when the fast memory under consideration is either the L1 or L2 cache. Out-of-order CPUs can generally hide memory latencies from L1 cache misses, but not L2 cache misses [Hennessy & Patterson 2006]. Therefore, it is more reasonable to treat the L2 cache as the fast memory and RAM (main memory) as the slow memory. The largest sparse matrix that fills the whole machine RAM (assuming the triples representation that occupies 16 bytes per nonzero, has $2^{30}/16 = 2^{26}$ nonzeros per GB of RAM). A square sparse matrix, with an average of eight nonzeros per column, has dimensions $2^{23} \times 2^{23}$ per GB of RAM. A single dense vector of double-precision floating-point numbers with 2^{23} elements would require 64 MB of L2 cache memory per GB of RAM, which is typically much larger than the typical size of the L2 cache.

The goal of this chapter is to explain sparse matrix data structures progressively, starting from the least structured and most simple format (unordered triples) and ending with the most structured formats: compressed sparse row (CSR) and compressed sparse column (CSC). This way, we provide motivation on why the experts prefer to use CSR/CSC formats by comparing and contrasting them with simpler formats. For example, CSR, a dense collection of sparse row arrays, can also be viewed as an extension of the triples format enhanced with row indexing capabilities. Furthermore, many ideas and intermediate data structures that are used to implement key primitives on triples are also widely used with implementations on CSR/CSC formats.

A vast amount of literature exists on sparse matrix storage schemes. Some additional other specialized data structures that are worth mentioning include

- Blocked compressed stripe formats (BCSR and BCSC) use less bandwidth to accelerate bandwidth limited computations such as SpMV.

- Knuth storage allows fast access to both rows and columns at the same time, and it makes dynamic changes to the matrix possible. Therefore, it is very suitable for all kinds of SpRef and SpAsgn operations. Its drawback is its excessive memory usage ($5\ nnz + 2M$) and high cache miss ratio.

- Hierarchical storage schemes such as quadtrees [Samet 1984, Wise & Franco 1990] are theoretically attractive, but achieving good performance in practice requires careful algorithm engineering to avoid high cache miss ratios that would result from straightforward pointer-based implementations.

The rest of this chapter is organized as follows. Section 13.2 describes the key sparse matrix primitives. Section 13.3 reviews the triples/coordinates representation, which is natural and easy to understand. The triples representation generalizes to higher dimensions [Bader & Kolda 2007]. Its resemblance to database tables will help us expose some interesting connections between databases and sparse matrices. Section 13.4 reviews the most commonly used compressed storage formats for general sparse matrices, namely CSR and CSC. Section 13.5 presents a case on how sparse matrices are represented in the Star-P programming environment. Section 13.6 concludes the chapter.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

13.2 Key primitives

Most of the sparse matrix operations have been motivated by numerical linear algebra. Some of them are also useful for graph algorithms:

1. Sparse matrix indexing and assignment (SpRef/SpAsgn): Corresponds to subgraph selection.

2. Sparse matrix-dense vector multiplication (SpMV): Corresponds to breadth-first or depth-first search.

3. Sparse matrix addition and other pointwise operations (SpAdd): Corresponds to graph merging.

4. Sparse matrix-sparse matrix multiplication (SpGEMM): Corresponds to breadth-first or depth-first search to/from multiply vertices simultaneously.

SpRef is the operation of storing a submatrix of a sparse matrix in another sparse matrix ($\mathbf{B} = \mathbf{A}(\mathbf{p}, \mathbf{q})$), and SpAsgn is the operation of assigning a sparse matrix to a submatrix of another sparse matrix ($\mathbf{B}(\mathbf{p}, \mathbf{q}) = \mathbf{A}$). It is worth noting that SpAsgn is the only key primitive that mutates its sparse matrix operand in the general case.* Sparse matrix indexing can be quite powerful and complex if we allow $\mathbf{p}$ and $\mathbf{q}$ to be arbitrary vectors of indices. Therefore, this chapter limits itself to row wise ($\mathbf{A}(i,:)$), column wise ($\mathbf{A}(:,i)$), and element-wise ($\mathbf{A}(i,j)$) indexing, as they find more widespread use in graph algorithms. SpAsgn also requires the matrix dimensions to match, e.g., if $\mathbf{B}(:,i) = \mathbf{A}$ where $\mathbf{B} \in \mathbb{S}^{M\times N}$, then $\mathbf{A} \in \mathbb{S}^{1\times N}$.

SpMV is the most widely used sparse matrix kernel since it is the workhorse of iterative linear equation solvers and eigenvalue computations. A sparse matrix can be multiplied by a dense vector either on the right ($\mathbf{y} = \mathbf{A}\mathbf{x}$) or on the left ($\mathbf{y}' = \mathbf{x}'\mathbf{A}$). This chapter concentrates on the multiplication on the right. It is generally straightforward to reformulate algorithms that use multiplication on the left so that they use multiplication on the right. Some representative graph computations that use SpMV are page ranking (an eigenvalue computation), breadth-first search, the Bellman–Ford shortest paths algorithm, and Prim's minimum spanning tree algorithm.

SpAdd, $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$, computes the sum of two sparse matrices of dimensions $M \times N$. SpAdd is an abstraction that is not limited to any summation operator. In general, any pointwise binary scalar operation between two sparse matrices falls into this primitive. Examples include MIN operator that returns the minimum of its operands, logical AND, logical OR, ordinary addition, and subtraction.

SpGEMM computes the sparse product $\mathbf{C} = \mathbf{AB}$, where the input matrices $\mathbf{A} \in \mathbb{S}^{M\times K}$ and $\mathbf{B} \in \mathbb{S}^{K\times N}$ are both sparse. It is a common operation for operating on large graphs, used in graph contraction, peer pressure clustering, recursive

*While $\mathbf{A} = \mathbf{A} \oplus \mathbf{B}$ or $\mathbf{A} = \mathbf{AB}$ may also be considered as mutator operations, these are just special cases when the output is the same as one of the inputs.

formulations of all-pairs shortest path algorithms, and breadth-first search from multiple source vertices.

The computation for matrix multiplication can be organized in several ways, leading to different formulations. One common formulation is the inner product formulation, as shown in Algorithm 13.1. In this case, every element of the product $\mathbf{C}(i, j)$ is computed as the dot product of a row i in $\mathbf{A}$ and a column j in $\mathbf{B}$. Another formulation of matrix multiplication is the outer product formulation (Algorithm 13.2). The product is computed as a sum of n rank-one matrices. Each rank-one matrix is computed as the outer product of column k of $\mathbf{A}$ and row k of $\mathbf{B}$.

Algorithm 13.1. Inner product matrix multiply.
Inner product formulation of matrix multiplication.

```
$\mathbf{C} : \mathbb{R}^{S(M\times N)} = \textsc{InnerProduct-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(M\times K)}, \mathbf{B} : \mathbb{R}^{S(K\times N)})$
1  for $i = 1$ to $M$
2      do for $j = 1$ to $N$
3          do $\mathbf{C}(i, j) = \mathbf{A}(i, :) \cdot \mathbf{B}(:, j)$
```

Algorithm 13.2. Outer product matrix multiply.
Outer product formulation of matrix multiplication.

```
$\mathbf{C} : \mathbb{R}^{S(M\times N)} = \textsc{OuterProduct-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(M\times K)}, \mathbf{B} : \mathbb{R}^{S(K\times N)})$
1  $\mathbf{C} = 0$
2  for $k = 1$ to $K$
3      do $\mathbf{C} = \mathbf{C} + \mathbf{A}(:, k) \cdot \mathbf{B}(k, :)$
```

SpGEMM can also be set up so that $\mathbf{A}$ and $\mathbf{B}$ are accessed by rows or columns, computing one row/column of the product $\mathbf{C}$ at a time. Algorithm 13.3 shows the column wise formulation where column j of $\mathbf{C}$ is computed as a linear combination of the columns of $\mathbf{A}$ as specified by the nonzeros in column j of $\mathbf{B}$. Figure 13.2 shows the same concept graphically. Similarly, for the row wise formulation, each row i of $\mathbf{C}$ is computed as a linear combination of the rows of $\mathbf{B}$ specified by nonzeros in row i of $\mathbf{A}$ as shown in Algorithm 13.4.

Algorithm 13.3. Column wise matrix multiplication.
Column wise formulation of matrix multiplication.

```
$\mathbf{C} : \mathbb{R}^{S(M\times N)} = \textsc{ColumnWise-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(M\times K)}, \mathbf{B} : \mathbb{R}^{S(K\times N)})$
1  for $j = 1$ to $N$
2      do for $k$ where $\mathbf{B}(k, j) \neq 0$
3          do $\mathbf{C}(:, j) = \mathbf{C}(:, j) + \mathbf{A}(:, k) \cdot \mathbf{B}(k, j)$
```

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

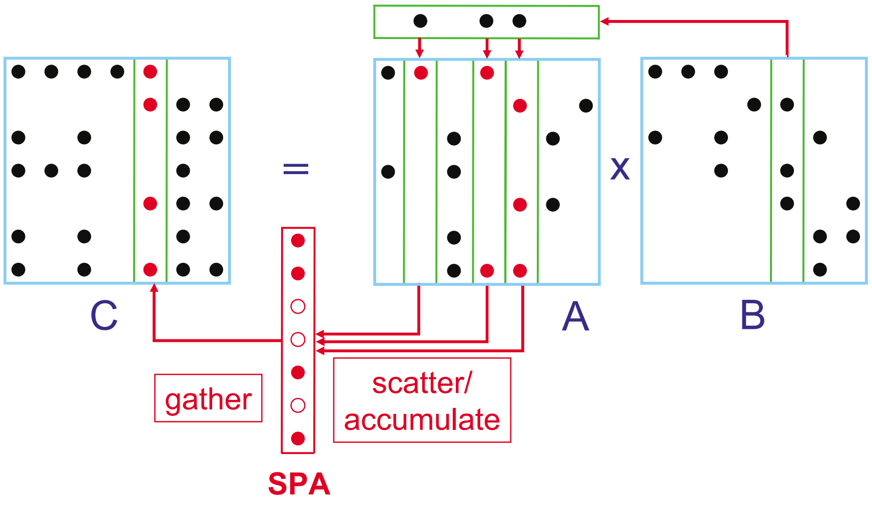

Figure 13.2. Multiply sparse matrices column by column. Multiplication of sparse matrices stored by columns. Columns of $\mathbf{A}$ are accumulated as specified by the nonzero entries in a column of $\mathbf{B}$ using a sparse accumulator (SPA). The contents of the SPA are stored in a column of $\mathbf{C}$ once all required columns are accumulated.

Algorithm 13.4. Row wise matrix multiply.
Row wise formulation of matrix multiplication.

```
$\mathbf{C} : \mathbb{R}^{S(M\times N)} = \text{RowWise-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(M\times K)}, \mathbf{B} : \mathbb{R}^{S(K\times N)})$
1  for $i = 1$ to $M$
2      do for $k$ where $\mathbf{A}(i,k) \neq 0$
3          do $\mathbf{C}(i,:) = \mathbf{C}(i,:) + \mathbf{A}(i,k) \cdot \mathbf{B}(k,:)$
```

13.3 Triples

The simplest way to represent a sparse matrix is the *triples* (or *coordinates*) format. For each $\mathbf{A}(i,j) \neq 0$, the triple $(i, j, \mathbf{A}(i,j))$ is stored in memory. Each entry in the triple is usually stored in a different array and the whole matrix $\mathbf{A}$ is represented as three arrays $\mathbf{A}$.I (row indices), $\mathbf{A}$.J (column indices), and $\mathbf{A}$.V (numerical values), as illustrated in Figure 13.3. These separate arrays are called “parallel arrays” by Duff and Reid (see [Duff & Reid 1979]) but we reserve “parallel” for parallel algorithms. Using 8-byte integers for row and column indices, storage cost is $8 + 8 + 8 = 24$ bytes per nonzero.

Modern programming languages offer easier ways of representing an array of tuples than using three separate arrays. An alternative implementation might choose to represent the set of triples as an array of records (or structs). Such an implementation might improve cache performance, especially if the algorithm accesses

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

$$A = \begin{pmatrix} 19 & 0 & 11 & 0 \\ 0 & 43 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 27 & 0 & 35 \end{pmatrix}$$

A.I	**A**.J	**A**.V
1	1	19
4	2	27
2	2	43
4	4	35
1	3	11

Figure 13.3. Triples representation.
Matrix **A** (left) and its unordered triples representation (right).

elements of the same index from different arrays. This type of cache optimization is known as array merging [Kowarschik & Weiß 2002]. Some programming languages, such as Python and Haskell, even support tuples as built-in types, and C++ includes tuples support in its standard library [Becker 2006]. In this section, we use the established notation of three separate arrays (**A**.I, **A**.J, **A**.V) for simplicity, but an implementer should keep in mind the other options.

This section evaluates the triples format under different levels of ordering. Unordered triples representation imposes no ordering constraints on the triples. Row ordered triples keep nonzeros ordered with respect to their row indices only. Nonzeros within the same row are stored arbitrarily, irrespective of their column indices. Finally, row-major order keeps nonzeros ordered lexicographically first according to their row indices and then according to their column indices to break ties. It is also possible to order with respect to columns instead of rows, but we analyze the row-based versions. Column-ordered and column-major ordered triples are similar. RAM and I/O complexities of key primitives for unordered and row ordered triples are listed in Tables 13.1 and 13.2.

A theoretically attractive fourth option is to use hashing and store triples in a hash table. In the case of SpGEMM and SpAdd, dynamically managing the output matrix is computationally expensive since dynamic perfect hashing does not yield high performance in practice [Mehlhorn & Näher 1999] and requires $35N$ space [Dietzfelbinger et al. 1994]. A recently proposed dynamic hashing method called Cuckoo hashing is promising. It supports queries in worst-case constant time and updates in amortized expected constant time, while using only $2N$ space [Pagh & Rodler 2004]. Experiments show that it is substantially faster than existing hashing schemes on modern architectures like Pentium 4 and IBM Cell [Ross 2007]. Although hash-based schemes seem attractive, especially for SpAsgn and SpRef primitives [Aspnäs et al. 2006], further research is required to test their efficiency for sparse matrix storage.

13.3.1 Unordered triples

The administrative overhead of the triples representation is low, especially if the triples are not sorted in any order. With unsorted triples, however, there is no

Table 13.1. Unordered and row ordered RAM complexities. Memory access complexities of key primitives on unordered and row ordered triples.

	Unordered	**Row Ordered**
SpRef	$O(nnz(\mathbf{A}))$	$O(\lg nnz(\mathbf{A})+nnz(\mathbf{A}(i,:)))\begin{cases}\mathbf{A}(i,j)\\ \mathbf{A}(i,:)\end{cases}$ $O(nnz(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(nnz(\mathbf{A}))+O(nnz(\mathbf{B}))$	$O(nnz(\mathbf{A}))+O(nnz(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(nnz(\mathbf{A})+nnz(\mathbf{B}))$	$O(nnz(\mathbf{A})+nnz(\mathbf{B}))$
SpGEMM	$O(nnz(\mathbf{A})+nnz(\mathbf{B})+\text{flops})$	$O(nnz(\mathbf{A})+\text{flops})$

Table 13.2. Unordered and row ordered I/O complexities. Input/Output access complexities of key primitives on unordered and row ordered triples.

	Unordered	**Row Ordered**
SpRef	$O(scan(\mathbf{A}))$	$O(\lg nnz(\mathbf{A})+scan(\mathbf{A}(i,:))\begin{cases}\mathbf{A}(i,j)\\ \mathbf{A}(i,:)\end{cases}$ $O(scan(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(scan(\mathbf{A})+scan(\mathbf{B}))$	$O(scan(\mathbf{A})+scan(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(nnz(\mathbf{A})+nnz(\mathbf{B}))$	$O(nnz(\mathbf{A})+nnz(\mathbf{B}))$
SpGEMM	$O(nnz(\mathbf{A})+nnz(\mathbf{B})+\text{flops})$	$O(\min\{nnz(\mathbf{A})+\text{flops},$ $scan(\mathbf{A})\ \lg(nnz(\mathbf{B}))+\text{flops}\})$

spatial locality when accessing nonzeros of a given row or column.[*] In the worst case, all indexing operations might require a complete scan of the data structure. Therefore, SpRef has $O(nnz(\mathbf{A}))$ RAM complexity and $O(scan(\mathbf{A}))$ I/O complexity.

SpAsgn is no faster, even though insertions take only constant time per element. In addition to accessing all the elements of the right-hand side matrix $\mathbf{A}$, SpAsgn also invalidates the existing nonzeros that need to be changed in the left-hand side matrix $\mathbf{B}$. Just finding those triples takes time proportional to the number of nonzeros in $\mathbf{B}$ with unordered triples. Thus, RAM complexity of SpAsgn is $O(nnz(\mathbf{A})+nnz(\mathbf{B}))$ and its I/O complexity is $O(scan(\mathbf{A})+scan(\mathbf{B}))$. A simple implementation achieving these bounds performs a single scan of $\mathbf{B}$, outputs only

[*]A procedure exploits spatial locality if data that are stored in nearby memory locations are likely to be referenced close in time.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

the nonassigned triples (e.g., for $\mathbf{B}(:,k) = \mathbf{A}$, those are the triples $(i, j, \mathbf{B}(i, j))$ where $j \neq k$), and finally concatenates the nonzeros in $\mathbf{A}$ to the output.

SpMV has full spatial locality when accessing the elements of $\mathbf{A}$ because the algorithm scans all the nonzeros of $\mathbf{A}$ in the exact order that they are stored. Therefore, $O(scan(\mathbf{A}))$ cache misses are taken for granted as compulsory misses.[*] Although SpMV is optimal in the RAM model without any ordering constraints, its cache performance suffers, as the algorithm cannot exploit any spatial locality when accessing vectors $\mathbf{x}$ and $\mathbf{y}$.

In considering the cache misses involved, for each triple $(i, j, \mathbf{A}(i, j))$, a random access to the jth component of $\mathbf{x}$ is required, and the result of the element-wise multiplication $\mathbf{A}(i, j) \cdot \mathbf{x}(j)$ must be written to the random location $\mathbf{y}(i)$. Assumption 2 implies that the fast memory is not big enough to hold the dense arrays $\mathbf{x}$ and $\mathbf{y}$. Thus, we make up to two extra cache misses per flop. These indirect memory accesses can be clearly seen in the Triples-SpMV code shown in Algorithm 13.5, where the values of $\mathbf{A}.\mathsf{I}(k)$ and $\mathbf{A}.\mathsf{J}(k)$ may change in every iteration. Consequently, I/O complexity of SpMV on unordered triples is

$$nnz(\mathbf{A})/L + 2\ nnz(\mathbf{A}) = O(nnz(\mathbf{A})) \tag{13.1}$$

Algorithm 13.5. Triples matrix vector multiply.
Operation $\mathbf{y} = \mathbf{A}\mathbf{x}$ using triples.

```
y : ℝ^M = Triples-SpMV(A : ℝ^{S(M×N)}, x : ℝ^N)
1   y = 0
2   for k = 1 to nnz(A)
3          do y(A.I(k)) = y(A.I(k)) + A.V(k) · x(A.J(k))
```

The SpAdd algorithm needs to identify all (i, j) pairs such that $\mathbf{A}(i, j) \neq 0$ and $\mathbf{B}(i, j) \neq 0$, and to add their values to create a single entry in the resulting matrix. This step can be accomplished by first sorting the nonzeros of the input matrices and then performing a simultaneous scan of sorted nonzeros to sum matching triples. Using linear time counting sort, SpAdd is fast in the RAM model with $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$ complexity.

Counting sort, in its naive form, has poor cache utilization because the total size of the counting array is likely to be bigger than the size of the fast memory. Sorting the nonzeros of a sparse matrix translates into one cache miss per nonzero in the worst case. Therefore, the complexity of SpAdd in the I/O model becomes $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$. The number of cache misses can be decreased by using cache optimal sorting algorithms (see [Aggarwal & Vitter 1988]), but such algorithms are comparison based. They do $O(n \lg n)$ work as opposed to linear work. Rahman and Raman [Rahman & Raman 2000] gave a counting sort algorithm that has better cache utilization in practice than the naive algorithm, and still does linear work.

SpGEMM needs fast access to columns, rows, or a given particular element, depending on the algorithm. One can also think of $\mathbf{A}$ as a table of i's and k's

[*] Assuming that no explicit data prefetching mechanism is used.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

and $\mathbf{B}$ as a table of k's and j's; then $\mathbf{C}$ is their join on k. This database analogy [Kotlyar et al. 1997] may lead to alternative SpGEMM implementations based on ideas from databases. An outer product formulation of SpGEMM on unordered triples has three basic steps (a similar algorithm for general sparse tensor multiplication is given by Bader and Kolda [Bader & Kolda 2007]):

1. For each $k \in \{1, \dots, K\}$, identify the set of triples that belongs to the kth column of $\mathbf{A}$ and the kth row of $\mathbf{B}$. Formally, find $\mathbf{A}(:, k)$ and $\mathbf{B}(k, :)$.
2. For each $k \in \{1, \dots, K\}$, compute the Cartesian product of the row indices of $\mathbf{A}(:, k)$ and the column indices of $\mathbf{B}(k, :)$. Formally, compute the sets $\mathbf{C}_k = \{\mathbf{A}(:, k).\mathsf{I}\} \times \{\mathbf{B}(k, :).\mathsf{J}\}$.
3. Find the union of all Cartesian products, summing up duplicates during set union: $\mathbf{C} = \bigcup_{k \in \{1,\dots,K\}} \mathbf{C}_k$.

Step 1 of the algorithm can be efficiently implemented by sorting the triples of $\mathbf{A}$ according to their column indices and the triples of $\mathbf{B}$ according to their row indices. Computing the Cartesian products in step 2 takes time

$$\sum_{k=1}^{K} nnz(\mathbf{A}(:, k)) \cdot nnz(\mathbf{B}(k, :)) = \text{flops} \tag{13.2}$$

Finally, summing up duplicates can be done by lexicographically sorting the elements from sets $\mathbf{C}_k$, which has a total running time of

$$O(sort(nnz(\mathbf{A})) + sort(nnz(\mathbf{B})) + \text{flops} + sort(\text{flops})) \tag{13.3}$$

As long as the number of nonzeros is more than the dimensions of the matrices (Assumption 1), it is advantageous to use a linear time sorting algorithm instead of a comparison-based sort. Since a lexicographical sort is not required for finding $\mathbf{A}(:, k)$ or $\mathbf{B}(k, :)$ in step 1, a single pass of linear time counting sort [Cormen et al. 2001] suffices for each input matrix. However, two passes of linear time counting sort are required in step 3 to produce a lexicographically sorted output. RAM complexity of this implementation turns out to be

$$nnz(\mathbf{A}) + nnz(\mathbf{B}) + 3 \cdot \text{flops} = O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + \text{flops}) \tag{13.4}$$

However, due to the cache-inefficient nature of counting sort, this algorithm makes $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + \text{flops})$ cache misses in the worst case.

Another way to implement SpGEMM on unordered triples is to iterate through the triples of $\mathbf{A}$. For each $(i, j, \mathbf{A}(i, j))$, we find $\mathbf{B}(:, j)$ and multiply $\mathbf{A}(i, j)$ with each nonzero in $\mathbf{B}(:, j)$. The duplicate summation step is left intact. The time this implementation takes is

$$nnz(\mathbf{A}) \cdot nnz(\mathbf{B}) + 3 \cdot \text{flops} = O(nnz(\mathbf{A}) \cdot nnz(\mathbf{B})) \tag{13.5}$$

The term flops is dominated by the term $nnz(\mathbf{A}) \cdot nnz(\mathbf{B})$ according to Theorem 13.1. Therefore, the performance is worse than for the previous implementation that sorts the input matrices first.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Theorem 13.1. *For all matrices* $\mathbf{A}$ *and* $\mathbf{B}$*,* $\text{flops}(\mathbf{AB}) \leq nnz(\mathbf{A}) \cdot nnz(\mathbf{B})$

Proof. Let the vector of column counts of $\mathbf{A}$ be

$$\mathbf{a} = (a_1, a_2, \ldots, a_k) = (nnz(\mathbf{A}(:,1), nnz(\mathbf{A}(:,2)), \ldots, nnz(\mathbf{A}(:,k))$$

and the vector of row counts of $\mathbf{B}$

$$\mathbf{b} = (b_1, b_2, \ldots, b_k) = (nnz(\mathbf{B}(1,:), nnz(\mathbf{B}(2,:)), \ldots, nnz(\mathbf{B}(k,:)).$$

Note that flops $= \mathbf{a}^\mathsf{T}\mathbf{b} = \sum_{i=j} a_i b_j$, and $\sum_{i \neq j} a_i b_j \geq 0$ as $\mathbf{a}$ and $\mathbf{b}$ are nonnegative. Consequently,

$$\begin{aligned} nnz(\mathbf{A}) \cdot nnz(\mathbf{B}) &= \left(\sum_{k=1}^{K} a_k\right) \cdot \left(\sum_{k=1}^{K} b_k\right) = \left(\sum_{i=j} a_i b_j\right) + \left(\sum_{i \neq j} a_i b_j\right) \\ &\geq \sum_{i=j} a_i b_j = \mathbf{a}^\mathsf{T}\mathbf{b} = \text{flops} \quad \square \end{aligned}$$

It is worth noting that both implementations of SpGEMM using unordered triples have $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + \text{flops})$ space complexity, due to the intermediate triples that are all present in the memory after step 2. Ideally, the space complexity of SpGEMM should be $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + nnz(\mathbf{C}))$, which is independent of flops.

13.3.2 Row ordered triples

The second option is to keep the triples sorted according to their rows or columns only. We analyze the row ordered version; column order is symmetric. This section is divided into three subsections. The first one is on indexing and SpMV. The second one is on a fundamental abstract data type that is used frequently in sparse matrix algorithms, namely the sparse accumulator (SPA). The SPA is used for implementing some of the SpAdd and SpGEMM algorithms throughout the rest of this chapter. Finally, the last subsection is on SpAdd and SpGEMM algorithms.

Indexing and SpMV with row ordered triples

Using row ordered triples, indexing still turns out to be inefficient. In practice, even a fast row access cannot be accomplished since there is no efficient way of spotting the beginning of the ith row without using an index.[*] Row wise referencing can be done by performing binary search on the whole matrix to identify a nonzero belonging to the referenced row, and then by scanning in both directions to find the rest of the nonzeros belonging to that row. Therefore, SpRef for $\mathbf{A}(i,:)$ has $O(\lg nnz(\mathbf{A}) + nnz(\mathbf{A}(i,:))$ RAM complexity and $O(\lg nnz(\mathbf{A}) + scan(\mathbf{A}(i,:)))$ I/O

[*]That is a drawback of the triples representation in general. The compressed sparse storage formats described in Section 13.4 provide efficient indexing mechanisms for either rows or columns.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

complexity. Element-wise referencing also has the same cost, in both models. Column wise referencing, on the other hand, is as slow as it was with unordered triples, requiring a complete scan of the triples.

SpAsgn might incur excessive data movement, as the number of nonzeros in the left-hand side matrix $\mathbf{B}$ might change during the operation. For a concrete example, consider the operation $\mathbf{B}(i,:) = \mathbf{A}$ where $nnz(\mathbf{B}(i,:)) \neq nnz(\mathbf{A})$ before the operation. Since the data structure needs to keep nonzeros with increasing row indices, all triples with row indices bigger than i need to be shifted by distance $|\,nnz(\mathbf{A}) - nnz(\mathbf{B}(i,:))|$.

SpAsgn has RAM complexity $O(nnz(\mathbf{A})) + nnz(\mathbf{B}))$ and I/O complexity $O(scan(\mathbf{A}) + scan(\mathbf{B}))$, where $\mathbf{B}$ is the left-hand side matrix before the operation. While implementations of row wise and element-wise referencing are straightforward, column wise referencing ($\mathbf{B}(:,i) = \mathbf{A}$) seems harder as it reduces to a restricted case of SpAdd. The restriction is that $\mathbf{B} \in \mathbb{S}^{1\times N}$ has, at most, one nonzero in a given row. Therefore, a similar scanning-based implementation suffices.

Row ordered triples format allows an SpMV implementation that makes, at most, one extra cache miss per flop. The reason is that references to vector $\mathbf{y}$ show good spatial locality: they are ordered with monotonically increasing values of $\mathbf{A}.\mathsf{I}(k)$, avoiding scattered memory referencing on vector $\mathbf{y}$. However, accesses to vector $\mathbf{x}$ are still irregular as the memory stride when accessing $\mathbf{x}$ ($|\mathbf{A}.\mathsf{J}(k+1) - \mathbf{A}.\mathsf{J}(k)|$) might be as big as the matrix dimension N. Memory strides can be reduced by clustering the nonzeros in every row. More formally, this clustering corresponds to reducing the bandwidth of the matrix, which is defined as $\beta(\mathbf{A}) = \max\{|i-j| : \mathbf{A}(i,j) \neq 0\}$. Toledo experimentally studied different methods of reordering the matrix to reduce its bandwidth, along with other optimizations like blocking and prefetching, to improve the memory performance of SpMV [Toledo 1997]. Overall, row ordering does not improve the asymptotic I/O complexity of SpMV over unordered triples, although it cuts the cache misses by nearly half. Its I/O complexity becomes

$$nnz(\mathbf{A})/L + N/L + nnz(\mathbf{A}) = O(nnz(\mathbf{A})) \tag{13.6}$$

The sparse accumulator

Most operations that output a sparse matrix generate it one row at a time. The current active row is stored temporarily on a special structure called the sparse accumulator (SPA) [Gilbert et al. 1992] (or expanded real accumulator [Pissanetsky 1984]). The SPA helps merging unordered lists in linear time.

There are different ways of implementing the SPA as it is an abstract data type not a concrete data structure. In our SPA implementation, $\mathbf{w}$ is the dense vector of values, $\mathbf{b}$ is the boolean dense vector that contains "occupied" flags, and LS is the list that keeps an unordered list of indices, as Gilbert, Moler and Schreiber described [Gilbert et al. 1992].

`Scatter-SPA` function, given in Algorithm 13.6, adds a scalar ($value$) to a specific position (pos) of the SPA. Scattering is a constant time operation. Gathering the SPA's nonzeros to the output matrix $\mathbf{C}$ takes $O(nnz(\mathsf{SPA}))$ time. The

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

pseudocode for the `Gather-SPA` is given in Algorithm 13.7. It is crucial to initialize SPA only once at the beginning as this takes $O(N)$ time. Resetting it later for the next active row takes only $O(nnz(\mathsf{SPA}))$ time by using LS to reach all the nonzero elements and resetting only those indices of $\mathbf{w}$ and $\mathbf{b}$.

Algorithm 13.6. Scatter SPA.
Scatters/accumulates the nonzeros in the SPA.

```
SCATTER-SPA(SPA, $value$, $pos$)
1  if (SPA.$\mathbf{b}(pos) = 0$)
2      then SPA.$\mathbf{w}(pos) \leftarrow value$
3           SPA.$\mathbf{b}(pos) \leftarrow 1$
4           INSERT(SPA.LS, $pos$)
5      else SPA.$\mathbf{w}(pos) \leftarrow$ SPA.$\mathbf{w}(pos) + value$
```

Algorithm 13.7. Gather SPA.
Gathers/outputs the nonzeros in the SPA.

```
$nzi$ = GATHER-SPA(SPA, $\mathbf{val}$, $\mathbf{col}$, $nzcur$)
1  $cptr \leftarrow head$(SPA.LS)
2  $nzi \leftarrow 0$                          ▷ number of nonzeros in the $i$th row of $\mathbf{C}$
3  while $cptr \neq$ NIL
4      do
5          $\mathbf{col}(nzcur + n) \leftarrow element(cptr)$            ▷ Set column index
6          $\mathbf{val}(nzcur + n) \leftarrow$ SPA.$\mathbf{w}(element(cptr))$          ▷ Set value
7          $nzi \leftarrow nzi + 1$
8          ADVANCE($cptr$)
```

The cost of resetting the SPA can be completely avoided by using the *multiple switch* technique (also called the *phase counter* technique) described by Gustavson (see [Gustavson 1976, Gustavson 1997]). Here, $\mathbf{b}$ becomes a dense *switch vector* of integers instead of a dense boolean vector. For computing each row, we use a different switch value. Every time a nonzero is introduced to position *pos* of the SPA, we set the switch to the current active row index (SPA.$\mathbf{b}(pos) = i$). During the computation of subsequent rows $j = \{i+1, \ldots, M\}$, the switch value being less than the current active row index (SPA.$\mathbf{b}(pos) \leq j$) means that the position *pos* of the SPA is "free." Therefore, the need to reset $\mathbf{b}$ for each row is avoided.

SpAdd and SpGEMM with row ordered triples

Using the SPA, we can implement SpAdd with $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$ RAM complexity. The full procedure is given in Algorithm 13.8. The I/O complexity of this SpAdd implementation is also $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$ because for each nonzero scanned from inputs, the algorithm checks and updates an arbitrary position of the SPA. From Assumption 2, these arbitrary accesses are likely to incur cache misses every time.

Algorithm 13.8. Row ordered matrix add.
Operation $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$ using row ordered triples.

```
C : $\mathbb{R}^{S(M\times N)}$ = RowTriples-SpAdd($\mathbf{A} : \mathbb{R}^{S(M\times N)}, \mathbf{B} : \mathbb{R}^{S(M\times N)}$)
1   Set-SPA(SPA)               ▷ Set $\mathbf{w} = 0$, $\mathbf{b} = 0$ and create empty list LS
2   $ka \leftarrow kb \leftarrow kc \leftarrow 1$           ▷ Initialize current indices to one
3   for $i \leftarrow 1$ to $M$
4         do while ($ka \le nnz(A)$ and $\mathbf{A}$.I($ka$) $= i$)
5               do Scatter-SPA(SPA, $\mathbf{A}$.V($ka$), $\mathbf{A}$.J($ka$))
6                  $ka \leftarrow ka + 1$
7           while ($kb \le nnz(B)$ and $\mathbf{B}$.I($kb$) $= i$)
8               do Scatter-SPA(SPA, $\mathbf{B}$.V($kb$), $\mathbf{B}$.J($kb$))
9                  $kb \leftarrow kb + 1$
10          $nznew \leftarrow$ Gather-SPA(SPA, $\mathbf{C}$.V, $\mathbf{C}$.J, $kc$)
11          for $j \leftarrow 0$ to $nznew - 1$
12              do $\mathbf{C}$.I($kc + j$) $\leftarrow i$      ▷ Set row index
13          $kc \leftarrow kc + nznew$
14          Reset-SPA(SPA)              ▷ Reset $\mathbf{w} = 0$, $\mathbf{b} = 0$ and empty LS
```

It is possible to implement SpGEMM by using the same outer product formulation described in Section 13.3.1, with a slightly better asymptotic RAM complexity of $O(nnz(\mathbf{A}) + \text{flops})$, as the triples of $\mathbf{B}$ are already sorted according to their row indices. Instead, we describe a row wise implementation, similar to the CSR-based algorithm described in Section 13.4. Because of inefficient row wise indexing support of row ordered triples, however, the operation count is higher than the CSR version. A SPA of size N is used to accumulate the nonzero structure of the current active row of $\mathbf{C}$. A direct scan of the nonzeros of $\mathbf{A}$ allows enumeration of nonzeros in $\mathbf{A}(i,:)$ for increasing values of $i \in \{1,\ldots,M\}$. Then, for each triple $(i,k,\mathbf{A}(i,k))$ in the ith row of $\mathbf{A}$, the matching triples $(k,j,\mathbf{B}(k,j))$ of the kth row of $\mathbf{B}$ need to be found using the SpRef primitive. This way, the nonzeros in $\mathbf{C}(i,:)$ are accumulated. The whole procedure is given in Algorithm 13.9. Its RAM complexity is

$$\sum_{\mathbf{A}(i,k)\neq 0} \big(nnz(\mathbf{B}(i,:))\ \lg(nnz(\mathbf{B}))\big) + \text{flops} = O(nnz(\mathbf{A})\ \lg(nnz(\mathbf{B})) + \text{flops}) \quad (13.7)$$

where the $\lg(nnz(\mathbf{B}))$ factor per each nonzero in $\mathbf{A}$ comes from the row wise SpRef operation in line 5. Its I/O complexity is

$$O(scan(\mathbf{A})\ \lg(nnz(\mathbf{B})) + \text{flops}) \quad (13.8)$$

While the complexity of row wise implementation is asymptotically worse than the outer product implementation in the RAM model, it has the advantage of using only $O(nnz(\mathbf{C}))$ space as opposed to the $O(\text{flops})$ space used by the outer product implementation. On the other hand, the I/O complexities of the outer product version and the row wise version are not directly comparable. Which one is faster depends on the cache line size and the number of nonzeros in $\mathbf{B}$.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Algorithm 13.9. Row ordered matrix multiply.
Operation $\mathbf{C} = \mathbf{AB}$ using row ordered triples.

```
C : $\mathbb{R}^{S(M\times N)}$ = RowTriples-SpGEMM($\mathbf{A} : \mathbb{R}^{S(M\times K)}, \mathbf{B} : \mathbb{R}^{S(K\times N)}$)
1   Set-SPA(SPA)                    ▷ Set $\mathbf{w} = 0$, $\mathbf{b} = 0$ and create empty list LS
2   $ka \leftarrow kc \leftarrow 1$                  ▷ Initialize current indices to one
3   for $i \leftarrow 1$ to $M$
4       do while ($ka \le nnz(A)$ and $\mathbf{A}.\mathsf{I}(ka) = i$)
5               do $\mathbf{BR} \leftarrow \mathbf{B}(\mathbf{A}.\mathsf{J}(ka), :)$           ▷ Using SpRef
6                  for $kb \leftarrow 1$ to $nnz(\mathbf{BR})$
7                      do $value \leftarrow \mathbf{A}.\mathsf{NUM}(ka) \cdot \mathbf{BR}.\mathsf{NUM}(kb)$
8                         Scatter-SPA(SPA, $value$, $\mathbf{BR}.\mathsf{J}(kb)$)
9                  $ka \leftarrow ka + 1$
10          $nznew \leftarrow$ Gather-SPA(SPA, $\mathbf{C}.\mathsf{V}$, $\mathbf{C}.\mathsf{J}$, $kc$)
11          for $j \leftarrow 0$ to $nznew - 1$
12              do $\mathbf{C}.\mathsf{I}(kc + j) \leftarrow i$        ▷ Set row index
13          $kc \leftarrow kc + nznew$
14          Reset-SPA(SPA)                ▷ Reset $\mathbf{w} = 0$, $\mathbf{b} = 0$ and empty LS
```

13.3.3 Row-major ordered triples

We now consider the third option of storing triples in lexicographic order, either in column-major or row-major order. Once again, we focus on the row-oriented scheme in this section. RAM and I/O complexities of key primitives for row-major ordered and CSR are listed in Tables 13.3 and 13.4.

In order to reference a whole row, binary search on the whole matrix, followed by a scan on both directions, is used, as with row ordered triples. As the nonzeros in a row are ordered by column indices, it seems there should be a faster way to access a single element than the method used on row ordered triples. A faster way

Table 13.3. Row-major ordered RAM complexities.
RAM complexities of key primitives on row-major ordered triples and CSR.

	Row-Major Ordered Triples	**CSR**
SpRef	$O(\lg nnz(\mathbf{A}) + \lg nnz(\mathbf{A}(i,:))\{\ \mathbf{A}(i,j)$ $O(\lg nnz(\mathbf{A}) + nnz(\mathbf{A}(i,:))\{\ \mathbf{A}(i,:)$ $O(nnz(\mathbf{A}))\{\ \mathbf{A}(:,j)$	$O(\lg nnz(\mathbf{A}(i,:))\{\ \mathbf{A}(i,j)$ $O(nnz(\mathbf{A}(i,:))\{\ \mathbf{A}(i,:)$ $O(nnz(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(nnz(\mathbf{A})) + O(nnz(\mathbf{B}))$	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$
SpGEMM	$O(nnz(\mathbf{A})+\text{flops})$	$O(nnz(\mathbf{A})+\text{flops})$

Table 13.4. Row-major ordered I/O complexities.
I/O complexities of key primitives on row-major ordered triples and CSR.

	Row-Major Ordered Triples	**CSR**
SpRef	$O(\lg nnz(\mathbf{A}) + search(\mathbf{A}(i,:))\{\ \mathbf{A}(i,j)$ $O(\lg nnz(\mathbf{A}) + scan(\mathbf{A}(i,:))\{\ \mathbf{A}(i,:)$ $O(scan(\mathbf{A}))\{\ \mathbf{A}(:,j)$	$O(search(\mathbf{A}(i,:)))\{\ \mathbf{A}(i,j)$ $O(scan(\mathbf{A}(i,:)))\{\ \mathbf{A}(i,:)$ $O(scan(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$
SpGEMM	$O(\min\{nnz(\mathbf{A}) + \text{flops},$ $scan(\mathbf{A})\ \lg(nnz(\mathbf{B}))+\text{flops}\})$	$O(scan(\mathbf{A}) + \text{flops})$

indeed exists, but ordinary binary search would not do it because the beginning and the end of the ith row is not known in advance. The algorithm has three steps:

1. Spot a triple $(i, j, \mathbf{A}(i, j))$ that belongs to the ith row by doing binary search on the whole matrix.

2. From that triple, perform an unbounded binary search [Manber 1989] on both directions. In an unbounded search, the step length is doubled at each iteration. The search terminates at a given direction when it hits a triple that does not belong to the ith row. Those two triples (one from each direction) become the boundary triples.

3. Perform ordinary binary search within the exclusive range defined by the boundary vertices.

The number of total operations is $O(\lg nnz(\mathbf{A}) + \lg nnz(\mathbf{A}(i,:))$. An example is given in Figure 13.4, where $\mathbf{A}(12, 16)$ is indexed.

While unbounded binary search is the preferred method in the RAM model, simple scanning might be faster in the I/O model. Searching an element in an ordered set of N elements can be achieved with $\Theta(\log_L N)$ cost in the I/O model, using B-trees (see [Bayer & McCreight 1972]). However, using an ordinary array, search incurs $\lg N$ cache misses. This may or may not be less than $scan(N)$. Therefore, we define the cost of searching within an ordered row as follows:

$$search(\mathbf{A}(i,:)) = \min\{\lg nnz(\mathbf{A}(i,:)), scan(\mathbf{A}(i,:))\} \tag{13.9}$$

For column wise referencing as well as for SpAsgn operations, row-major ordered triples format does not provide any improvement over row ordered triples.

In SpMV, the only array that does not show excellent spatial locality is $\mathbf{x}$, since $\mathbf{A}$.I, $\mathbf{A}$.J, $\mathbf{A}$.V, and $\mathbf{y}$ are accessed with mostly consecutive, increasing index

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

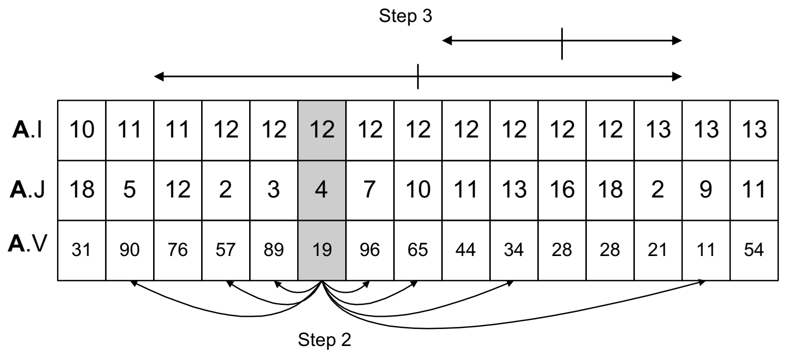

Figure 13.4. Indexing row-major triples.
Element-wise indexing of $\mathbf{A}(12, 16)$ on row-major ordered triples.

values. Accesses to $\mathbf{x}$ are also with increasing indices, which is an improvement over row ordered triples. However, memory strides when accessing $\mathbf{x}$ can still be high, depending on the number of nonzeros in each row and the bandwidth of the matrix. In the worst case, each access to $\mathbf{x}$ might incur a cache miss.

Bender et al. came up with cache-efficient algorithms for SpMV, using the column-major layout, that have an optimal number of cache misses [Bender et al. 2007]. From a high-level view, their method first generates all the intermediate triples of $\mathbf{y}$, possibly with repeating indices. Then, the algorithm sorts those intermediate triples with respect to their row indices, performing additions on the triples with same row index as they occur. I/O optimality of their SpMV algorithm relies on the existence of an I/O optimal sorting algorithm. Their complexity measure assumes a fixed k number of nonzeros per column, leading to I/O complexity of

$$O\left(\mathit{scan}(\mathbf{A}) \log_{Z/L} \frac{N}{\max\{Z, k\}} \right) \tag{13.10}$$

SpAdd is now more efficient even without using any auxiliary data structure. A scanning-based array-merging algorithm is sufficient as long as we do not forget to sum duplicates while merging. Such an implementation has $O(\mathit{nnz}(\mathbf{A}) + \mathit{nnz}(\mathbf{B}))$ RAM complexity and $O(\mathit{scan}(\mathbf{A}) + \mathit{scan}(\mathbf{B}))$ I/O complexity.[*]

Row-major ordered triples allow outer product and row wise SpGEMM implementations at least as efficiently as row ordered triples. Indeed, some finer improvements are possibly by exploiting the more specialized structure. In the case of row wise SpGEMM, a technique called *finger search* [Brodal 2005] can be used to improve the RAM complexity. While enumerating all triples $(i, k, \mathbf{A}) \in \mathbf{A}(i, :)$, they are naturally sorted with increasing k values. Therefore, accesses to $\mathbf{B}(k, :)$ are also with increasing k values. Instead of restarting the binary search from the beginning of $\mathbf{B}$, one can use fingers and only search the yet unexplored subsequence. Note

[*]These bounds are optimal only if $\mathit{nnz}(\mathbf{A}) = \Theta(\mathit{nnz}(\mathbf{B}))$; see [Brown & Tarjan 1979].

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

that finger search uses the unbounded binary search as a subroutine when searching the unexplored subsequence. Row wise SpGEMM using finger search has a RAM complexity of

$$O(\text{flops}) + \sum_{i=1}^{m} O\left(nnz(\mathbf{A}(i,:)) \lg \frac{nnz(\mathbf{B})}{nnz(\mathbf{A}(i,:))}\right) \tag{13.11}$$

which is asymptotically faster than the $O(nnz(\mathbf{A}) \lg(nnz(\mathbf{B})) + \text{flops})$ cost of the same algorithm on row ordered triples.

Outer product SpGEMM can be modified to use only $O(nnz(\mathbf{C}))$ space during execution by using multiway merging [Buluç & Gilbert 2008b]. However, this comes at the price of an extra $\lg k$ factor in the asymptotical RAM complexity where k is the number of indices i for which $\mathbf{A}(:,i) \neq \emptyset$ and $\mathbf{B}(i,:) \neq \emptyset$.

Although both of these refined algorithms are asymptotically slower than the naive outer product method, they might be faster in practice because of the cache effects and difference in constants in the asymptotic complexities. Further research is required in algorithm engineering of SpGEMM to find the best performing algorithm in real life.

13.4 Compressed sparse row/column

The most widely used storage schemes for sparse matrices are compressed sparse column (CSC) and compressed sparse row (CSR). For example, Matlab uses CSC format to store its sparse matrices [Gilbert et al. 1992]. Both are dense collections of sparse arrays. We examine CSR, which is introduced by Gustavson [Gustavson 1972] under the name of sparse row wise representation; CSC is symmetric.

CSR can be seen as a concatenation of sparse row arrays. On the other hand, it is also very close to row ordered triples with an auxiliary index of size $\Theta(N)$. In this section, we assume that nonzeros within each sparse row array are ordered with increasing row indices. This is not a general requirement though. Tim Davis CSparse package [Davis et al. 2006], for example, does not impose any ordering within the sparse arrays.

13.4.1 CSR and adjacency lists

In principle, CSR is almost identical to the adjacency list representation of a directed graph [Tarjan 1972]. In practice, however, it has much less overhead and much better cache efficiency. Instead of storing an array of linked lists as in the adjacency list representation, CSR is composed of three arrays that store whole rows contiguously. The first array (IR) of size $M + 1$ stores the row pointers as explicit integer values, the second array (JC) of size nnz stores the column indices, and the last array (NUM) of size nnz stores the actual numerical values. Observe that column indices stored in the JC array indeed come from concatenating the edge indices of the adjacency lists. Following the sparse matrix/graph duality, it is also meaningful to call the first array the vertex array and the second array the edge

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

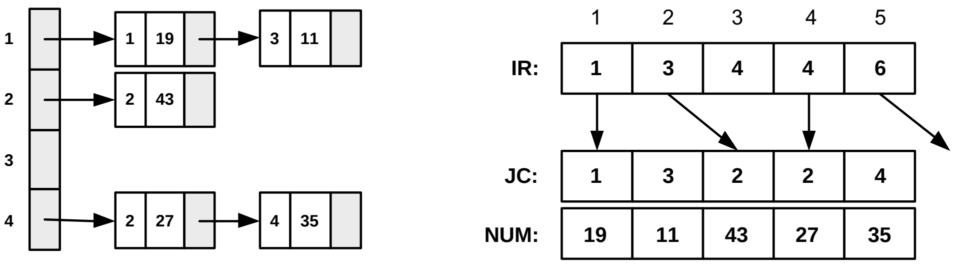

Figure 13.5. CSR format.
Adjacency list (left) and CSR (right) representations of matrix **A**.

array. The vertex array holds the offsets to the edge array, meaning that the nonzeros in the ith row are stored from NUM(IR(i)) to NUM(IR($i+1$) $-$ 1), and their respective positions within that row are stored from JC(IR(i)) to JC(IR($i+1$) $-$ 1). Also note that JC(i) = JC($i+1$) means there are no nonzeros in the ith row.

Figure 13.5 shows the adjacency list and CSR representations of matrix **A**. While the arrows in the adjacency-based representation are actual pointers to memory locations, the arrows in CSR are not. Any data type that supports unsigned integers fulfills the desired purpose of holding the offsets to the edge array.

The efficiency advantage of the CSR data structure compared with the adjacency list can be explained by the memory architecture of modern computers. In order to access all the nonzeros in a given row i, which is equivalent to traversing all the outgoing edges of a given vertex v_i, CSR makes at most $\lceil nnz(\mathbf{A}(i,:))/L \rceil$ cache misses. A similar access to the adjacency list representation incurs $nnz(\mathbf{A}(i,:))$ cache misses in the worst case, worsening as the memory becomes more and more fragmented. In an experiment published in 1998, an array-based representation was found to be 10 times faster to traverse than a linked-list-based representation [Black et al. 1998]. This performance gap is due to the high cost of pointer chasing that happens frequently in linked data structures. The efficiency of CSR comes at a price though: introducing new nonzero elements or deleting a nonzero element is computationally inefficient and best avoided [Gilbert et al. 1992]. Therefore, CSR is best suited for representing static graphs. All the key primitives but SpAsgn work on static graphs.

13.4.2 CSR on key primitives

Contrary to triples storage formats, CSR allows constant-time random access to any row of the matrix. Its ability to enumerate all the elements in the ith row with $O(nnz(\mathbf{A}(i,:))$ RAM complexity and $O(scan(\mathbf{A}(i,:))$ I/O complexity makes it an excellent data structure for row wise SpRef. Element-wise referencing takes, at most, $O(\lg nnz(\mathbf{A}(i,:))$ time in the RAM model as well as the I/O model, using a binary search. Considering column wise referencing, however, CSR does not provide any improvement over the triples format.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

On the other hand, even row wise SpAsgn operations are inefficient if the number of elements in the assigned row changes. In that general case, $O(nnz(\mathbf{B}))$ elements might need to be moved. This is also true for column wise and element-wise SpAsgn as long as not just existing nonzeros are reassigned to new values.

The code in Algorithm 13.10 shows how to perform SpMV when matrix $\mathbf{A}$ is represented in CSR format. This code and SpMV with row-major ordered triples have similar performance characteristics except for a few subtleties. When some rows of $\mathbf{A}$ are all zeros, those rows are effectively skipped in row-major ordered triples, but still need to be examined in CSR. On the other hand, when $M \ll nnz$, CSR has a clear advantage since it needs to examine only one index ($\mathbf{A}.\mathsf{JC}(k)$) per inner loop iteration while row-major ordered triples needs to examine two ($\mathbf{A}.\mathsf{I}(k)$ and $\mathbf{A}.\mathsf{J}(k)$). Thus, CSR may have up to a factor of two difference in the number of cache misses. CSR also has some advantages over CSC when the SpMV primitive is considered (especially in the case of $\mathbf{y} = \mathbf{y} + \mathbf{Ax}$), as experimentally shown by Vuduc [Vuduc 2003].

Algorithm 13.10. CSR matrix vector multiply.
Operation $\mathbf{y} = \mathbf{Ax}$ using CSR.

```
$\mathbf{y} : \mathbb{R}^M$ = CSR-SpMV($\mathbf{A} : \mathbb{R}^{S(M)\times N}, \mathbf{x} : \mathbb{R}^N$)
1  $\mathbf{y} = 0$
2  for $i = 1$ to $M$
3      do for $k = \mathbf{A}.\mathsf{IR}(i)$ to $\mathbf{A}.\mathsf{IR}(i+1) - 1$
4          do $\mathbf{y}(i) = \mathbf{y}(i) + \mathbf{A}.\mathsf{NUM}(k) \cdot \mathbf{x}(\mathbf{A}.\mathsf{JC}(k))$
```

Blocked versions of CSR and CSC try to take advantage of clustered nonzeros in the sparse matrix. While blocked CSR (BCSR) achieves superior performance for SpMV on matrices resulting from finite element meshes [Vuduc 2003], mostly by using loop unrolling and register blocking, it is of little use when the matrix itself does not have its nonzeros clustered. Pinar and Heath [Pinar & Heath 1999] proposed a reordering mechanism to cluster those nonzeros to get dense subblocks. However, it is not clear whether such mechanisms are successful for highly irregular matrices from sparse real-world graphs.

Except for the additional bookkeeping required for getting the row pointers right, SpAdd can be implemented in the same way as is done with row-major ordered triples. Luckily, the extra bookkeeping of row pointers does not affect the asymptotic complexities.

One subtlety overlooked in the SpAdd implementations throughout this chapter is management of the memory required by the resulting matrix $\mathbf{C}$. We implicitly assumed that the data structure holding $\mathbf{C}$ has enough space to accommodate all of its elements. Repeated doubling of memory whenever necessary is one way of addressing this issue. Another conservative way is to reserve $nnz(\mathbf{A}) + nnz(\mathbf{B})$ space for $\mathbf{C}$ at the beginning of the procedure and shrink any unused portion after the computation, right before the procedure returns.

The efficiency of accessing and enumerating rows in CSR makes the row wise SpGEMM formulation, described in Algorithm 13.4, the preferred matrix

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

multiplication formulation. An efficient implementation of the row wise SpGEMM using CSR was first given by Gustavson [Gustavson 1978]. It had a RAM complexity of

$$O(M + N + nnz(\mathbf{A}) + \text{flops}) = O(nnz(\mathbf{A}) + \text{flops}) \qquad (13.12)$$

where the equality follows from Assumption 1. Recent column wise implementations with similar RAM complexities are provided by Davis in his CSparse software [Davis et al. 2006] and by MATLAB [Gilbert et al. 1992]. The algorithm, presented in Algorithm 13.11 uses the SPA described in Section 13.3.2. Once again, the multiple switch technique can be used to avoid the cost of resetting the SPA for every iteration of the outermost loop. As in the case of SpAdd, generally the space required to store $\mathbf{C}$ cannot be determined quickly. Repeated doubling or more sophisticated methods such as Cohen's algorithm [Cohen 1998] may be used. Cohen's algorithm is a randomized iterative algorithm that does $\Theta(1)$ SpMV operations over a semiring to estimate the row and column counts. It can be efficiently implemented even on unordered triples.

Algorithm 13.11. CSR matrix multiply.
Operation $\mathbf{C} = \mathbf{AB}$ using CSR.

```
$\mathbf{C} : \mathbb{R}^{S(M)\times N}$ = CSR-SpGEMM($\mathbf{A} : \mathbb{R}^{S(M)\times K}, \mathbf{B} : \mathbb{R}^{S(K)\times N}$)
1   Set-SPA(SPA)                          ▷ Set $\mathbf{w} = 0$, $\mathbf{b} = 0$ and create empty list LS
2   $\mathbf{C}$.IR(1) ← 0
3   for $i$ ← 1 to $M$
4       do for $k$ ← $\mathbf{A}$.IR($i$) to $\mathbf{A}$.IR($i+1$)
5             do for $j$ ← $\mathbf{B}$.IR($\mathbf{A}$.JC($k$)) to $\mathbf{B}$.IR($\mathbf{A}$.JC($k$) + 1)
6                   do
7                      $value$ ← $\mathbf{A}$.NUM($k$) · $\mathbf{B}$.NUM($j$)
8                      Scatter-SPA(SPA, $value$, $\mathbf{B}$.JC($j$))
9          $nznew$ ← Gather-SPA(SPA, $\mathbf{C}$.NUM, $\mathbf{C}$.JC, $\mathbf{C}$.IR($i$))
10         $\mathbf{C}$.IR($i+1$) ← $\mathbf{C}$.IR($i$) + $nznew$
11         Reset-SPA(SPA)              ▷ Reset $\mathbf{w} = 0$, $\mathbf{b} = 0$ and empty LS
```

The row wise SpGEMM implementation does $O(scan(\mathbf{A}) + \text{flops})$ cache misses in the worst case. Due to the size of the SPA and Assumption 2, the algorithm makes a cache miss for every flop. As long as no cache interference occurs between the nonzeros of $\mathbf{A}$ and the nonzeros of $\mathbf{C}(i,:)$, only $scan(\mathbf{A})$ additional cache misses are made instead of $nnz(\mathbf{A})$.

13.5 Case study: STAR-P

This section summarizes how sparse matrices are represented in STAR-P. It also includes how the key primitives are implemented in this real-world software solution.

13.5.1 Sparse matrices in STAR-P

The current STAR-P implementation includes *dsparse* (distributed sparse) matrices, which are distributed across processors by blocks of rows. This layout makes

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

the CSR data structure a logical choice to store the sparse matrix slice on each processor.

The design of sparse matrix algorithms in Star-P follows the same design principles as in Matlab [Gilbert et al. 1992].

1. Storage required for a sparse matrix should be $O(nnz)$, proportional to the number of nonzero elements.

2. Running time for a sparse matrix algorithm should be $O(\text{flops})$. It should be proportional to the number of floating-point operations required to obtain the result.

The CSR data structure satisfies the requirement for storage as long as $M < nnz$. The second principle is difficult to achieve exactly in practice. Typically, most implementations achieve running time close to $O(\text{flops})$ for commonly used sparse matrix operations. For example, accessing a single element of a sparse matrix should be a constant-time operation. With a CSR data structure, it typically takes time proportional to the logarithm of the length of the row to access a single element. Similarly, insertion of single elements into a CSR data structure generates extensive data movement. Such operations are efficiently performed with the `sparse/find` routines, which work with triples rather than individual elements.

Sparse matrix-dense vector multiplication (SpMV)

The CSR data structure used in Star-P is efficient for multiplying a sparse matrix by a dense vector: $\mathbf{y} = \mathbf{Ax}$. It is efficient for communication and it shows good cache behavior for the sequential part of the computation. Our choice of the CSR data structure was heavily influenced by our desire to have good SpMV performance since it forms the core computational kernel for many iterative methods.

The matrix $\mathbf{A}$ and vector $\mathbf{x}$ are distributed across processors by rows. The submatrix of $\mathbf{A}$ on each processor will need some subset of $\mathbf{x}$ depending upon its sparsity structure. When SpMV is invoked for the first time on a dsparse matrix $\mathbf{A}$, Star-P computes a communication schedule for $\mathbf{A}$ and caches it. When later matvecs are performed using the same $\mathbf{A}$, this communication schedule does not need to be recomputed, thus saving some computing and communication overhead at the cost of extra space required to save the schedule. We experimented with overlapping computation and communication in SpMV. It turns out in many cases that this is less efficient than simply performing the communication first, followed by the computation. As computer architectures evolve, this decision may need to be revisited.

When multiplying from the left, $\mathbf{y} = \mathbf{x}'\mathbf{A}$, the communication is not as efficient. Instead of communicating the required subpieces of the source vector, each processor computes its own destination vector. All partial destination vectors are then summed up into the final destination vector. The communication required is always $O(N)$. The choice of the CSR data structure, while making the communication more efficient when multiplying from the right, makes it more difficult to multiply on the left.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Sparse matrix-sparse matrix multiplication

Star-P stores its matrices in CSR form. Clearly, computing inner products (Algorithm 13.1) is inefficient, since rows of **A** cannot be efficiently accessed without searching. Similarly, in the case of computing outer products (Algorithm 13.2), rows of **B** have to be extracted. The process of accumulating successive rank-one updates is also inefficient, as the structure of the result changes with each successive update. Therefore, Star-P uses the row wise formulation of matrix multiplication described in Section 13.2, in which the computation is set up so that only rows of **A** and **B** are accessed, producing a row of **C** at a time.

The performance of sparse matrix multiplication in parallel depends upon the nonzero structures of **A** and **B**. A well-tuned implementation may use a polyalgorithm. Such a polyalgorithm may use different communication schemes for different matrices. For example, it may be efficient to broadcast the local part of a matrix to all processors, but in other cases, it may be efficient to send only the required rows. On large clusters, it may be efficient to interleave communication and computation. On shared memory architectures, however, most of the time is spent in accumulating updates, rather than in communication. In such cases, it may be more efficient to schedule the communication before the computation.

13.6 Conclusions

In this chapter, we gave a brief survey on sparse matrix infrastructure for doing graph algorithms. We focused on implementation and analysis of key primitives on various sparse matrix data structures. We tried to complement the existing literature in two directions. First, we analyzed sparse matrix indexing and assignment operations. Second, we gave I/O complexity bounds for all operations. Taking I/O complexities into account is key to achieving high performance in modern architectures with multiple levels of cache.

References

[Aggarwal & Vitter 1988] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[Aho et al. 1974] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Boston: Addison–Wesley Longman, 1974.

[Aspnäs et al. 2006] M. Aspnäs, A. Signell, and J. Westerholm. Efficient assembly of sparse matrices using hashing. In B. Kgström, E. Elmroth, J. Dongarra, and J. Wasniewski, eds., *PARA*, volume 4699 of *Lecture Notes in Computer Science*, 900–907. Berlin Heidelberg: Springer, 2006.

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

[Bayer & McCreight 1972] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[Becker 2006] P. Becker. *The C++ Standard Library Extensions: A Tutorial and Reference*. Boston: Addison–Wesley Professional, 2006.

[Bender et al. 2007] 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*, 61–70, New York: Association for Computing Machinery, 2007.

[Black et al. 1998] J.R. Black, C.U. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Algorithm Engineering*, 37–48, 1998.

[Brodal 2005] G.S. Brodal. Finger search trees. In D. Mehta and S. Sahni, eds., *Handbook of Data Structures and Applications*, Chapter 11. Boca Raton, Fla.: CRC Press, 2005.

[Brown & Tarjan 1979] M.R. Brown and R.E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26:211–226, 1979.

[Buluç & Gilbert 2008b] A. Buluç and J.R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, 1–11, Miama, FL, 2008.

[Cohen 1998] E. Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2:307–332, 1998.

[Cormen et al. 2001] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, Second Edition. Cambridge, Mass.: The MIT Press, 2001.

[Davis et al. 2006] T.A. Davis. *Direct Methods for Sparse Linear Systems.* Philadelphia, SIAM, 2006.

[Dietzfelbinger et al. 1994] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer Auf Der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23:738–761, 1994.

[Duff & Reid 1979] I.S. Duff and J.K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, 5:18–35, 1979.

[Feo et al. 2005] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *CF '05: Proceedings of the 2nd Conference on Computing Frontiers*, 28–34, New York: Association for Computing Machinery, 2005.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

[Gilbert et al. 1992] J.R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13:333–356, 1992.

[Gustavson 1972] F.G. Gustavson. Some basic techniques for solving sparse systems of linear equations. In D.J. Rose and R.A. Willoughby, eds., *Sparse Matrices and Their Applications*, 41–52, New York: Plenum Press, 1972.

[Gustavson 1976] F.G. Gustavson. Finding the block lower triangular form of a matrix. In J.R. Bunch and D.J. Rose, eds., *Sparse Matrix Computations*, 275–289. New York: Academic Press, 1976.

[Gustavson 1997] F.G. Gustavson. Efficient algorithm to perform sparse matrix multiplication. *IBM Technical Disclosure Bulletin*, 20:1262–1264, 1977.

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

[Hennessy & Patterson 2006] J.L. Hennessy and D.A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* San Francisco: Morgan Kaufmann Publishers, 2006.

[Kotlyar et al. 1997] V. Kotlyar, K. Pingali, and P. Stodghill. A relational approach to the compilation of sparse matrix programs. In *European Conference on Parallel Processing*, 318–327, 1997.

[Kowarschik & Weiß 2002] M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, eds., *Algorithms for Memory Hierarchies*, 213–232, Berlin Heidelberg: Springer-Verlag, 2002.

[Manber 1989] U. Manber. *Introduction to Algorithms: A Creative Approach.* Boston: Addison–Wesley Longman, 1989.

[Mehlhorn & Näher 1999] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing.* New York: Cambridge University Press, 1999.

[Meyer et al. 2002] U. Meyer, P. Sanders, and J.F. Sibeyn, eds. *Algorithms for Memory Hierarchies, Advanced Lectures (Dagstuhl Research Seminar, March 10–14, 2002)*, volume 2625 of *Lecture Notes in Computer Science.* Berlin Heidelberg: Springer-Verlag, 2003.

[Pagh & Rodler 2004] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[Pinar & Heath 1999] A. Pinar and M.T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, p. 30, New York: Association Association for Computing Machinery, 1999.

[Pissanetsky 1984] S. Pissanetsky. *Sparse Matrix Technology.* London: Academic Press, 1984.

[Rahman & Raman 2000] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *Journal of Experimental Algorithmics*, 5:14, 2000.

[Ross 2007] K.A. Ross. Efficient hash probes on modern processors. In *Proceedings of the IEEE 23rd International Conference on Data Engineering*, 1297–1301. IEEE, 2007.

[Samet 1984] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16:187–260, 1984.

[Tarjan 1972] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[Toledo 1997] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41:711–726, 1997.

[Vuduc 2003] R.W. Vuduc. *Automatic performance tuning of sparse matrix kernels.* PhD thesis, University of California, Berkeley, 2003.

[Wise & Franco 1990] D.S. Wise and J.V. Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing*, 9:282–296, 1990.

Downloaded 09/23/16 to 131.243.223.179. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php