Chapter 14

New Ideas in Sparse Matrix Matrix Multiplication

Aydın Buluç[*] *and John Gilbert*[†]

Abstract

Generalized sparse matrix matrix multiplication is a key primitive for many high performance graph algorithms as well as some linear solvers such as multigrid. We present the first parallel algorithms that achieve increasing speedups for an unbounded number of processors. Our algorithms are based on the two-dimensional (2D) block distribution of sparse matrices where serial sections use a novel hypersparse kernel for scalability.

14.1 Introduction

Development and implementation of large-scale parallel graph algorithms pose numerous challenges in terms of scalability and productivity [Lumsdaine et al. 2007, Yoo et al. 2005]. Linear algebra formulations of many graph algorithms already exist in the literature; see [Aho et al. 1974, Maggs & Poltkin 1988, Tarjan 1981].

[*]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).

This work is sponsored by the Department of Energy under award number DE-FG02-04ER25632, in part by MIT Lincoln Laboratory under contract number 7000012980.

Preliminary versions of this paper have appeared in the proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS '08) and the 37th International Conference on Parallel Processing (ICPP' 08).

By exploiting the duality between matrices and graphs, linear algebraic formulations aim to apply the existing knowledge on parallel matrix algorithms to parallel graph algorithms. One of the key linear algebraic primitives for graph algorithms is computing the product of two sparse matrices (SpGEMM) over a semiring. It serves as a building block for many algorithms including graph contraction algorithm [Gilbert et al. 2008], breadth-first search from multiple-source vertices, peer pressure clustering [Shah 2007], recursive formulations of all-pairs shortest paths algorithms [D'Alberto & Nicolau 2007], matching algorithms [Rabin & Vazirani 1989], and cycle detection [Yuster & Zwick 2004], as well as for some other applications such as multigrid interpolation/restriction [Briggs et al. 2000] and parsing context-free languages [Penn 2006].

Most large graphs in applications, such as the WWW graph, finite element meshes, planar graphs, and trees are sparse. In this chapter, 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. Dense matrix multiplication algorithms are inefficient for SpGEMM since they require $O(N^3)$ space and the current fastest dense matrix multiplication algorithm runs in $O(N^{2.38})$ time; see [Coppersmith & Winograd 1987, Seidel 1995]. Furthermore, fast dense matrix multiplication algorithms operate on a ring instead of a semiring, making them unsuitable for many algorithms on general graphs. For example, it is possible to embed the semiring into the ring of integers for the all-pairs shortest paths problem on unweighted and undirected graphs [Seidel 1995], but the same embedding does not work for weighted or directed graphs.

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. Sometimes, we abbreviate and use $nnz(j)$ to denote the number of nonzero elements in the jth column of the matrix in context. Array indices are 1-based throughout this chapter. We use flops($\mathbf{A}\, op\, \mathbf{B}$), pronounced "flops," 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 context, we simply use flops.

The most widely used data structures for sparse matrices are the Compressed Sparse Columns (CSC) and Compressed Sparse Rows (CSR). The previous chapter gives concise descriptions of common SpGEMM algorithms operating both on CSC/CSR and triples. The SpGEMM problem was recently reconsidered in [Yuster & Zwick 2005] over a ring, where the authors used a fast dense matrix multiplication such as arithmetic progression [Coppersmith & Winograd 1987] 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 $\mathbf{C}$ is $\Theta(N^2)$. This assumption rarely holds in reality. Instead, we provide a work-sensitive analysis by expressing the computation complexity of our SpGEMM algorithms in terms of flops.

Practical sparse algorithms have been proposed by different researchers over the years (see, e.g., [Park et al. 1992, Sulatycke & Ghose 1998]) using various data structures. Although they achieve reasonable performance on some classes of matrices, none of these algorithms outperforms the classical sparse matrix matrix multiplication algorithm for general sparse matrices, which was first described by Gustavson [Gustavson 1978] and was used in MATLAB [Gilbert et al. 1992] and CSparse [Davis et al. 2006]. The classical algorithm runs in $O(\text{flops} + nnz + N)$ time.

In Section 14.2, we present a novel algorithm for sequential SpGEMM that is geared toward computing the product of two *hypersparse* matrices. A matrix is hypersparse if the ratio of nonzeros to its dimension is asymptotically 0. The algorithm is used as the sequential building block of our parallel 2D algorithms described in Section 14.3. Our HYPERSPARSE_GEMM algorithm uses a new $O(nnz)$ data structure, called *DCSC* for *doubly compressed sparse columns*, which is explained in Section 14.2.2. The HYPERSPARSE_GEMM is based on the outer product formulation and has time complexity $O(nzc(\mathbf{A}) + nzr(\mathbf{B}) + \text{flops} \cdot \lg ni)$, where $nzc(\mathbf{A})$ is the number of columns of $\mathbf{A}$ that contain at least one nonzero, $nzr(\mathbf{B})$ is the number of rows of $\mathbf{B}$ that contain at least one nonzero, and ni is the number of indices i for which $\mathbf{A}(:, i) \neq \emptyset$ and $\mathbf{B}(i, :) \neq \emptyset$. The overall space complexity of our algorithm is only $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + nnz(\mathbf{C}))$. Notice that the time complexity of our algorithm does not depend on N, and the space complexity does not depend on flops.

Section 14.3 presents parallel algorithms for SpGEMM. We propose novel algorithms based on 2D block decomposition of data in addition to giving the complete description of an existing 1D algorithm. To the best of our knowledge, parallel algorithms using a 2D block decomposition have not earlier been developed for sparse matrix matrix multiplication.

Irony, Toledo, and Tiskin [Irony et al. 2004] proved that 2D dense matrix multiplication algorithms are optimal with respect to the communication volume, making 2D sparse algorithms likely to be more scalable than their one-dimensional (1D) counterparts. In Section 14.4, we show that this intuition is indeed correct by providing a theoretical analysis of the parallel performance of 1D and 2D algorithms.

In Section 14.5, we model the speedup of parallel SpGEMM algorithms using realistic simulations and projections. Our results show that existing 1D algorithms are not scalable to thousands of processors. By contrast, 2D algorithms have the potential for scaling up indefinitely, albeit with decreasing parallel efficiency, which is defined as the ratio of speedup to the number of processors.

14.2 Sequential sparse matrix multiply

We first analyze different formulations of sparse matrix matrix multiplication by using the layered-graph model in Section 14.2.1. This graph theoretical explanation gives insights on the suitability of the outer product formulation for multiplying hypersparse matrices (HYPERSPARSE_GEMM). Section 14.2.2 defines hypersparse matrices and Section 14.2.3 introduces the DCSC data structure that is suitable to

store hypersparse matrices. We present our HYPERSPARSE_GEMM algorithm in Section 14.2.2.

14.2.1 Layered graphs for different formulations of SpGEMM

Matrix multiplication can be organized in many different ways. The inner product formulation that usually serves as the definition of matrix multiplication is well known. Given two matrices $\mathbf{A} \in \mathbb{R}^{M \times K}$ and $\mathbf{B} \in \mathbb{R}^{K \times N}$, each element in the product $\mathbf{C} \in \mathbb{R}^{M \times N}$ is computed by the following formula:

$$\mathbf{C}(i,j) = \sum_{k=1}^{K} \mathbf{A}(i,k)\mathbf{B}(k,j) \tag{14.1}$$

This formulation is rarely useful for multiplying sparse matrices since it requires $\Omega(M \cdot N)$ operations regardless of the sparsity of the operands.

We represent the multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$ as a three-layered graph, following Cohen [Cohen 1998]. The layers have M, K, and N vertices, in that order. The first layer of vertices (U) represents the rows of $\mathbf{A}$, and the third layer of vertices (V) represents the columns of $\mathbf{B}$. The second layer of vertices (W) represents the dimension shared between matrices. Every nonzero $\mathbf{A}(i,l) \neq 0$ in the ith row of $\mathbf{A}$ forms an edge (u_i, w_l) between the first and second layers, and every nonzero in $\mathbf{B}(l,j) \neq 0$ in the jth column of $\mathbf{B}$ forms an edge (w_l, v_j) between the second and third layers.

We perform different operations on the layered graph depending on the way we formulate the multiplication. In all cases, though, the goal is to find pairs of vertices (u_i, v_j) sharing an adjacent vertex $w_k \in W$, and if any pair shares multiple adjacent vertices, to merge their contributions.

Using inner products, we analyze each pair (u_i, v_j) to find the set of vertices in $\widetilde{W_{ij}} \subseteq W = \{w_1, w_2, \ldots, w_k\}$ that are connected to both u_i and v_j in the graph shown in Figure 14.1. The algorithm then accumulates contributions $a_{il} \cdot b_{lj}$ for all $w_l \in \widetilde{W_{ij}}$. The result becomes the value of $\mathbf{C}(i,j)$ in the output. In general, this inner product subgraph is sparse, and a contribution from w_l happens only when both edges a_{il} and b_{lj} exist. However, this sparsity is not exploited using inner products as it needs to examine each (u_i, v_j) pair, even when the set $\widetilde{W_{ij}}$ is empty.

In the outer product formulation, the product is written as the summation of k rank-one matrices:

$$\mathbf{C} = \sum_{k=1}^{K} \mathbf{A}(:,k)\mathbf{B}(k,:) \tag{14.2}$$

A different subgraph results from this formulation as it is the set of vertices W that represent the shared dimension that plays the central role. Note that the edges are traversed in the outward direction from a node $w_i \in W$, as shown in Figure 14.2. For sufficiently sparse matrices, this formulation may run faster because this traversal is performed only for the vertices in W (size K) instead of the inner product traversal that had to be performed for every pair (size MN). The problem

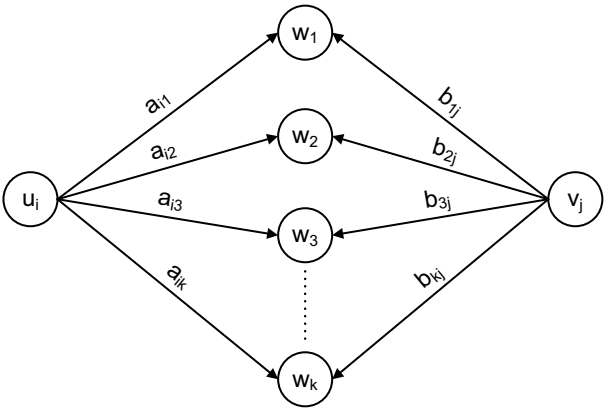

Figure 14.1. Graph representation of the inner product $\mathbf{A}(i,:) \cdot \mathbf{B}(:,j)$.

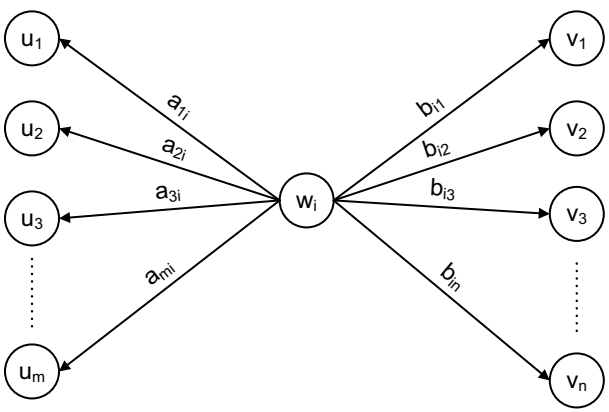

Figure 14.2. Graph representation of the outer product $\mathbf{A}(:,i) \cdot \mathbf{B}(i,:)$.

with outer product traversal is that it is hard to accumulate the intermediate results into the final matrix.

A row-by-row formulation of matrix multiplication performs a traversal starting from each of the vertices in U towards V, as shown in Figure 14.3 for u_i. Each traversal is independent from each other because they generate different rows of $\mathbf{C}$. Finally, a column-by-column formulation creates an isomorphic traversal, in the reverse direction (from V to U).

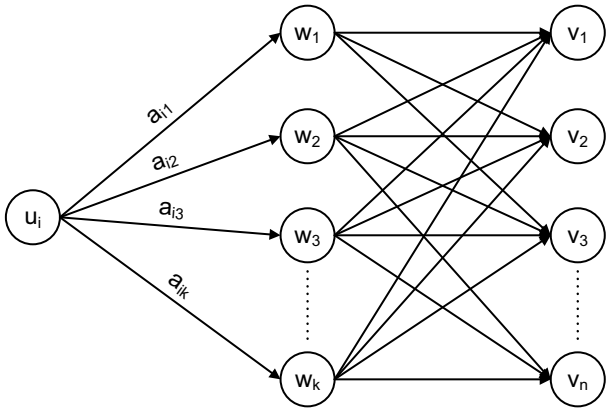

Figure 14.3. Graph representation of the sparse row times matrix product $\mathbf{A}(i,:) \cdot \mathbf{B}$.

14.2.2 Hypersparse matrices

Recall that a matrix is hypersparse if $nnz < N$. Although CSR/CSC is a fairly efficient storage scheme for general sparse matrices having $nnz = \Omega(N)$, it is asymptotically suboptimal for hypersparse matrices. Hypersparse matrices are fairly rare in numerical linear algebra (indeed, a nonsingular square matrix must have $nnz \geq N$), but they occur frequently in computations on graphs, particularly in parallel.

Our main motivation for hypersparse matrices comes from parallel processing. Hypersparse matrices arise after the 2D block data decomposition of ordinary sparse matrices for parallel processing. Consider a sparse matrix with c nonzero elements in each column. After the 2D decomposition of the matrix, each processor locally owns a submatrix with dimensions $(N/\sqrt{p}) \times (N/\sqrt{p})$. Storing each of those submatrices in CSC format takes $O(N\sqrt{p} + nnz)$ space, whereas the amount of space needed to store the whole matrix in CSC format on a single processor is only $O(N + nnz)$. As the number of processors increases, the $N\sqrt{p}$ term dominates the nnz term.

Figure 14.4 shows that the average number of nonzeros in a single column of a submatrix, $nnz(j)$, goes to zero as p increases. Storing a graph using CSC is similar to using adjacency lists. The column pointers array represents the vertices, and the row indices array represents their adjacencies. In that sense, CSC is a vertex-based data structure, making it suitable for 1D (vertex) partitioning of the graph. On the other hand, 2D partitioning is based on edges. Therefore, using CSC with 2D distributed data is forcing a vertex-based representation on edge distributed data. The result is unnecessary replication of column pointers (vertices) on each processor along the processor column.

The inefficiency of CSC leads to a more fundamental problem: any algorithm that uses CSC and scans all the columns is not scalable for hypersparse matrices. Even without any communication at all, such an algorithm cannot scale for $n\sqrt{p} \geq$

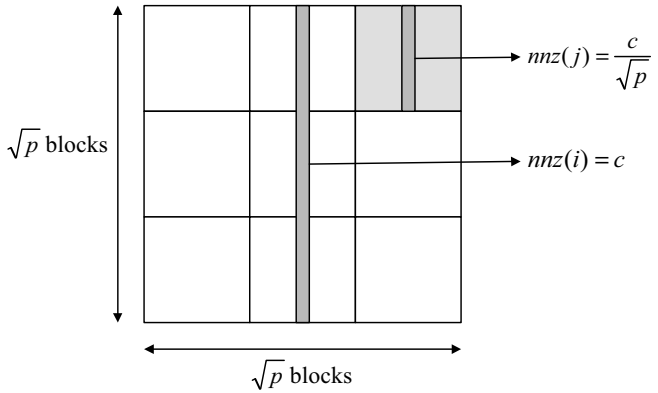

Figure 14.4. 2D sparse matrix decomposition.

$max\{\text{flops}, nnz\}$. SpMV and SpGEMM are algorithms that scan column indices. For these operations, any data structure that depends on the matrix dimension (such as CSR or CSC) is asymptotically too wasteful for submatrices.

14.2.3 DCSC data structure

We use a new data structure for our sequential hypersparse matrix matrix multiplication. This structure, called *DCSC* for doubly compressed sparse columns, has the following properties.

1. It uses $O(nnz)$ storage.
2. It lets the hypersparse algorithm scale with increasing sparsity.
3. It supports fast access to columns of the matrix.

For an example, consider the 9×9 matrix with 4 nonzeros whose triples representation is given in Figure 14.6. Figure 14.5 shows its CSC storage, which includes repetitions and redundancies in the column pointers array (JC). Our new data structure compresses the JC array to avoid repetitions, giving the CP(column pointers) array of DCSC as shown in Figure 14.7. DCSC is essentially a sparse array of sparse columns, whereas CSC is a dense array of sparse columns.

After removing repetitions, CP[i] does no longer refer to the ith column. A new JC array, which is parallel to CP, gives us the column numbers. Although our HYPERSPARSE_GEMM algorithm does not need column indexing, DCSC supports fast column indexing for completeness. Whenever column indexing is needed, we construct an AUX array that contains pointers to nonzero columns (columns that have at least one nonzero element). Each entry in AUX refers to a $\lceil n/\, nzc \rceil$-sized chunk of columns, pointing to the first nonzero column in that chunk (there might be none). The storage requirement of DCSC is $O(nnz)$ since $|\text{NUM}| = |\text{IR}| = nnz$, $|\text{JC}| = nzc$, $|\text{CP}| = nzc + 1$, and $|\text{AUX}| \approx nzc$.

JC	=	1	3	3	3	3	3	3	4	5	5
		↓						↓	↓		
IR	=	6	8					4	2		
NUM	=	0.1	0.2					0.3	0.4		

Figure 14.5. Matrix A in CSC format.

A.1	A.J	A.V
6	1	0.1
8	1	0.2
4	7	0.3
2	8	0.4

Figure 14.6. Matrix A in triples format.

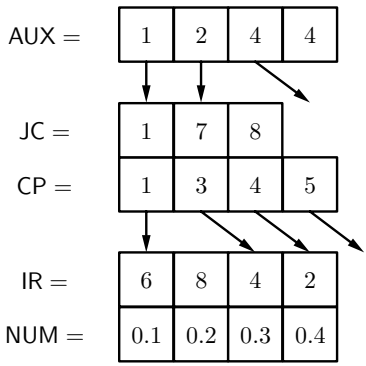

Figure 14.7. Matrix A in DCSC format.

In our implementation, the AUX array is a temporary work array that is constructed on demand, only when an operation requires repetitive use of it. This keeps the storage and copying costs low. The time to construct AUX is only $O(nzc)$, which is subsumed by the cost of multiplication.

14.2.4 A sequential algorithm to multiply hypersparse matrices

The sequential hypersparse algorithm (HYPERSPARSE_GEMM) is based on outer product multiplication. Therefore, it requires fast access to rows of matrix **B**. This could be accomplished by having each input matrix represented in DCSC and also in DCSR (doubly compressed sparse rows), which is the same as the transpose in DCSC. This method, which we described in an early version of this work [Buluç & Gilbert 2008b], doubles the storage but does not change the asymptotic space and time complexities. Here, we describe a more practical version where **B** is transposed as a preprocessing step, at a cost of trans(**B**). The actual cost of transposition is either $O(N + nnz(\mathbf{B}))$ or $O(nnz(\mathbf{B}) \lg nnz(\mathbf{B}))$, depending on the implementation.

The idea behind the HYPERSPARSE_GEMM algorithm is to use the outer product formulation of matrix multiplication efficiently. The first observation about DCSC is that the JC array is already sorted. Therefore, **A**.JC is the sorted indices

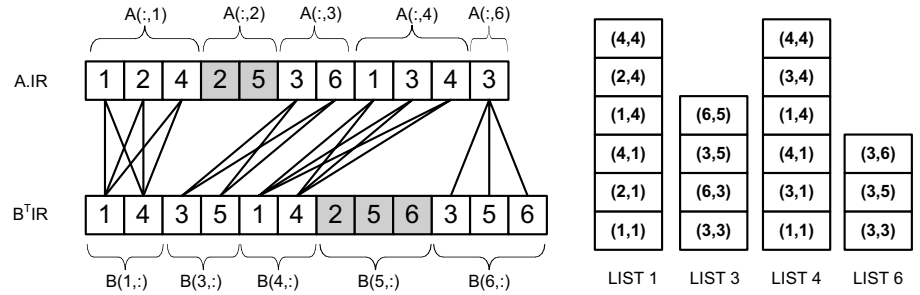

Figure 14.8. Cartesian product and the multiway merging analogy.

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \times & & & \times & & \\ \times & \times & & & & \\ & & \times & \times & & \times \\ \times & & & \times & & \\ & \times & & & & \\ & & \times & & & \end{pmatrix} \end{array}, \quad \mathbf{B} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \times & & & \times & & \\ & & & & & \\ & & \times & & \times & \\ \times & & & \times & & \\ & \times & & & \times & \times \\ & & \times & & \times & \times \end{pmatrix} \end{array}$$

Figure 14.9. Nonzero structures of operands A and B.

of the columns that contain at least one nonzero, and similarly $\mathbf{B^T}.\mathsf{JC}$ is the sorted indices of the rows that contain at least one nonzero. In this formulation, the ith column of $\mathbf{A}$ and the ith row of $\mathbf{B}$ are multiplied to form a rank-one matrix. The naive algorithm does the same procedure for all values of i and gets n different rank-one matrices, adding them to the resulting matrix $\mathbf{C}$ as they become available. Our algorithm has a preprocessing step that finds intersection $\text{Isect} = \mathbf{A}.\mathsf{JC} \cap \mathbf{B^T}.\mathsf{JC}$, which is the set of indices that participate nontrivially in the outer product.

The preprocessing takes $O(nzc(\mathbf{A}) + nzr(\mathbf{B}))$ time as $|\mathbf{A}.\mathsf{JC}| = nzc(\mathbf{A})$ and $|\mathbf{B^T}.\mathsf{JC}| = nzr(\mathbf{B})$. The next phase of our algorithm performs $|\text{Isect}|$ Cartesian products, each of which generates a fictitious list of size $nnz(\mathbf{A}(:,i)) \cdot nnz(\mathbf{B}(i,:))$. The lists can be generated sorted because all the elements within a given column are sorted according to their row indices (i.e., $\mathsf{IR}(\mathsf{JC}(i)) \ldots \mathsf{IR}(\mathsf{JC}(i)+1)$ is a sorted range). The algorithm merges those sorted lists, summing up the intermediate entries having the same (row_id, col_id) index pair, to form the resulting matrix $\mathbf{C}$. Therefore, the second phase of HYPERSPARSE_GEMM is similar to multiway merging [Knuth 1997]. The only difference is that we never explicitly construct the lists; we compute their elements one by one on demand.

Figure 14.8 shows the setup for the matrices from Figure 14.9. Since $\mathbf{A}.\mathsf{JC} = \{1,2,3,4,6\}$ and $\mathbf{B^T}.\mathsf{JC} = \{1,3,4,5,6\}$, $\text{Isect} = \{1,3,4,6\}$ for this product. The algorithm does not touch the shaded elements since they do not contribute to the output.

The merge uses a priority queue (represented as a heap) of size ni, which is the size of Isect, the number of indices i for which $\mathbf{A}(:, i) \neq \emptyset$ and $\mathbf{B}(i, :) \neq \emptyset$. The value in a heap entry is its NUM value and the key is a pair of indices (i, j) in column-major order. The idea is to repeatedly extract the entry with minimum key from the heap and insert another element from the list that the extracted element originally came from. If there are multiple elements in the lists with the same key, then their values are added 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 the right side of Figure 14.8, which are sorted from bottom to top. For further details of multiway merging, consult Knuth [Knuth 1997].

The time complexity of this phase is $O(\text{flops} \cdot \lg ni)$, and the space complexity is $O(nnz(\mathbf{C}) + ni)$. The output is a stack of NUM values in column-major order. The $nnz(\mathbf{C})$ term in the space complexity comes from the output, and the flops term in the time complexity comes from the observation that

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

The final phase of the algorithm constructs the DCSC structure from this column-major ordered stack. This requires $O(nnz(\mathbf{C}))$ time and space.

The overall time complexity of our algorithm is $O(nzc(\mathbf{A}) + nzr(\mathbf{B}) + \text{flops} \cdot \lg ni)$, plus the preprocessing time to transpose matrix $\mathbf{B}$. Note that $nnz(\mathbf{C})$ does not appear in this bound since $nnz(\mathbf{C}) \leq \text{flops}$. We opt to keep the cost of transposition separate because our parallel 2D block SpGEMM will amortize this transposition of each block over $\sqrt{p}$ uses of that block. Therefore, the cost of transposition will be negligible in practice. The space complexity is $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + nnz(\mathbf{C}))$. The time complexity does not depend on N, and the space complexity does not depend on flops.

Algorithm 14.1 gives the pseudocode for the whole algorithm. It uses two subprocedures: CartMult-Insert generates the next element from the ith fictitious list and inserts it to the heap PQ, and Increment-List increments the pointers of the ith fictitious list or deletes the list from the intersection set if it is empty.

To justify the extra logarithmic factor in the flops term, we briefly analyze the complexity of each submatrix multiplication in the parallel 2D block SpGEMM. Our parallel 2D block SpGEMM performs $p\sqrt{p}$ submatrix multiplications since each submatrix of the output is computed using $\mathbf{C}_{ij} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{ik} \mathbf{B}_{kj}$. Therefore, with increasing number of processors and under perfect load balance, flops scales with $1/p\sqrt{p}$, nnz scales with $1/p$, and N scales with $1/\sqrt{p}$. Figure 14.10 shows the trends of these three complexity measures as p increases. The graph shows that the N term becomes the bottleneck after around 50 processors and flops becomes the lower-order term. In contrast to the classical algorithm, our Hypersparse_GEMM algorithm becomes independent of N, by putting the burden on the flops instead.

Algorithm 14.1. Hypersparse matrix multiply.
Pseudocode for hypersparse matrix matrix multiplication algorithm.

$\mathbf{C} : \mathbb{R}^{S(M \times N)} = \text{HYPERSPARSEGEMM}(\mathbf{A} : \mathbb{R}^{S(M \times K)}, \mathbf{B^T} : \mathbb{R}^{S(N \times K)})$

```
1  isect ← INTERSECTION(A.JC, B^T.JC)
2  for j ← 1 to |isect|
3        do CARTMULT-INSERT(A, B^T, PQ, isect, j)
4           INCREMENT-LIST(isect, j)
5  while ISNOTFINISHED(isect)
6        do (key, value) ← EXTRACT-MIN(PQ)
7           (product, i) ← UNPAIR(value)
8           if key ≠ TOP(Q)
9              then ENQUEUE(Q, key, product)
10             else  UPDATETOP(Q, product)
11          if ISNOTEMPTY(isect(i))
12             then CARTMULT-INSERT(A, B^T, PQ, lists, isect, i)
13                  INCREMENT-LIST(isect, i)
14 CONSTRUCT-DCSC(Q)
```

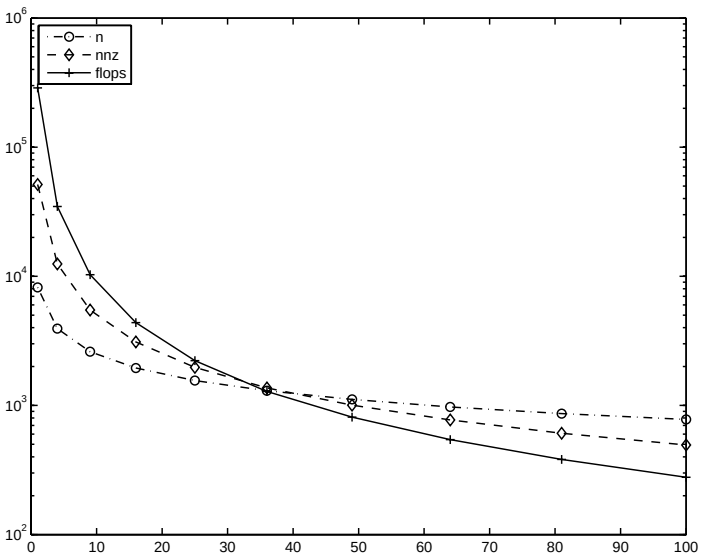

Figure 14.10. Trends of different complexity measures for submatrix multiplications as p increases. The inputs are randomly permuted R-MAT matrices (scale 15 with an average of 8 nonzeros per column) that are successively divided into $(N/\sqrt{p}) \times (N/\sqrt{p})$. The counts are averaged over all submatrix multiplications.

14.3 Parallel algorithms for sparse GEMM

This section describes parallel algorithms for multiplying two sparse matrices in parallel on p processors, which we call PSpGEMM. The design of our algorithms is motivated by distributed memory systems, but we expect them to perform well in shared memory too, as they avoid hot spots and load imbalances by ensuring proper work distribution among processors. Like most message passing algorithms, they can be implemented in the partitioned global address space (PGAS) model as well.

14.3.1 1D decomposition

We assume the data is distributed to processors in block rows, where each processor receives M/p consecutive rows. We write $\mathbf{A}_i = \mathbf{A}(ip : (i+1)p - 1, :)$ to denote the block row owned by the ith processor. To simplify the algorithm description, we use $\mathbf{A}_{ij}$ to denote $\mathbf{A}_i(:, jp : (j+1)p - 1)$, the jth block column of $\mathbf{A}_i$, although block rows are not physically partitioned:

$$\mathbf{A} = \left(\begin{array}{c} \mathbf{A}_1 \\ \hline \dots \\ \hline \mathbf{A}_p \end{array}\right) = \left(\begin{array}{c|c|c} \mathbf{A}_{11} & \dots & \mathbf{A}_{1p} \\ \hline \dots & & \dots \\ \hline \mathbf{A}_{p1} & \dots & \mathbf{A}_{pp} \end{array}\right), \mathbf{B} = \left(\begin{array}{c} \mathbf{B}_1 \\ \hline \dots \\ \hline \mathbf{B}_p \end{array}\right) \tag{14.3}$$

For each processor $P(i)$, the computation is

$$\mathbf{C}_i = \mathbf{C}_i + \mathbf{A}_i\,\mathbf{B} = \mathbf{C}_i = \mathbf{C}_i + \sum_{j=1}^{p} \mathbf{A}_{ij}\,\mathbf{B}_j$$

14.3.2 2D decomposition

Our 2D parallel algorithms, Sparse Cannon and Sparse SUMMA, use the hypersparse algorithm, which has complexity $O(nzc(\mathbf{A}) + nzr(\mathbf{B}) + \text{flops} \cdot \lg ni)$, as shown in Section 14.2.2, for multiplying submatrices. Processors are logically organized on a square $\sqrt{p} \times \sqrt{p}$ mesh, indexed by their row and column indices so that the (i, j)th processor is denoted by $P(i, j)$. Matrices are assigned to processors according to a 2D block decomposition. Each node gets a submatrix of dimensions $(N/\sqrt{p}) \times (N/\sqrt{p})$ in its local memory. For example, $\mathbf{A}$ is partitioned as shown below and $\mathbf{A}_{ij}$ is assigned to processor $P(i, j)$:

$$\mathbf{A} = \left(\begin{array}{c|c|c} \mathbf{A}_{11} & \dots & \mathbf{A}_{1\sqrt{p}} \\ \hline \dots & & \dots \\ \hline \mathbf{A}_{\sqrt{p}1} & \dots & \mathbf{A}_{\sqrt{p}\sqrt{p}} \end{array}\right) \tag{14.4}$$

For each processor $P(i)$, the computation is

$$\mathbf{C}_{ij} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{ik}\,\mathbf{B}_{kj}$$

14.3.3 Sparse 1D algorithm

The row wise SpGEMM forms one row of $\mathbf{C}$ at a time, and each processor may potentially need to access all of $\mathbf{B}$ to form a single row of $\mathbf{C}$. However, only a portion of $\mathbf{B}$ is locally available at any time in parallel algorithms. The algorithm thus performs multiple iterations to fully form one row of $\mathbf{C}$. We use a SPA to accumulate the nonzeros of the current active row of $\mathbf{C}$. Algorithm 14.2 shows the pseudocode of the algorithm. Loads and unloads of SPA, which is not amortized by the number of nonzero arithmetic operations in general, dominate the computational time.

Algorithm 14.2. Matrix matrix multiply.
Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using block row Sparse1D.

$\mathbf{C} : \mathbb{R}^{P(S(N)\times N)} = \text{BLOCK1D-PSPGEMM}(\mathbf{A} : \mathbb{R}^{P(S(N)\times N)}, \mathbf{B} : \mathbb{R}^{P(S(N)\times N)})$

```
1  for all processors P(i) in parallel
2      do INITIALIZE(SPA)
3         for j ← 1 to p
4             do BROADCAST(B_j)
5                for k ← 1 to N/p
6                    do LOAD(SPA, C_i(k,:))
7                       SPA ← SPA + A_ij(k,:) B_j
8                       UNLOAD(SPA, C_i(k,:))
```

14.3.4 Sparse Cannon

Our first 2D algorithm is based on Cannon's algorithm for dense matrices (see [Cannon 1969]). The pseudocode of the algorithm is given in Algorithm 14.5. Sparse Cannon, although elegant, is not our choice of algorithm for the final implementation, as it is hard to generalize to nonsquare grids, nonsquare matrices, and matrices whose dimensions are not perfectly divisible by grid dimensions.

Algorithm 14.3. Circular shift left.
Circularly shift left by s along the processor row.

LEFT-CIRCULAR-SHIFT($\mathbf{Local} : \mathbb{R}^{S(N\times N)}, s$)

```
1  SEND(Local, P(i, (j − s) mod √p))          ▷ This is processor P(i, j)
2  RECEIVE(Temp, P(i, (j + s) mod √p))
3  Local ← Temp
```

Algorithm 14.4. Circular shift up.
Circularly shift up by s along the processor column.

UP-CIRCULAR-SHIFT($\mathbf{Local} : \mathbb{R}^{S(N\times N)}, s$)

```
1  SEND(Local, P((i − s) mod √p, j))          ▷ This is processor P(i, j)
2  RECEIVE(Temp, P((i + s) mod √p, j))
3  Local ← Temp
```

Algorithm 14.5. Cannon matrix multiply.
Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using Sparse Cannon.

$\mathbf{C} : \mathbb{R}^{P(S(N\times N))} = \text{CANNON-PSPGEMM}(\mathbf{A} : \mathbb{R}^{P(S(N\times N))}, \mathbf{B} : \mathbb{R}^{P(S(N\times N))})$

1 **for** all processors $P(i,j)$ in parallel
2 **do** LEFT-CIRCULAR-SHIFT($\mathbf{A}_{ij}, i-1$)
3 UP-CIRCULAR-SHIFT($\mathbf{B}_{ij}, j-1$)
4 **for** all processors P_{ij} in parallel
5 **do for** $k \leftarrow 1$ **to** $\sqrt{p}$
6 **do** $\mathbf{C}_{ij} \leftarrow \mathbf{C}_{ij} + \mathbf{A}_{ij}\,\mathbf{B}_{ij}$
7 LEFT-CIRCULAR-SHIFT($\mathbf{A}_{ij}, 1$)
8 UP-CIRCULAR-SHIFT($\mathbf{B}_{ij}, 1$)

14.3.5 Sparse SUMMA

SUMMA [Van De Geijn & Watts 1997] is a memory efficient, easy to generalize algorithm for parallel dense matrix multiplication. It is the algorithm used in parallel BLAS (see [Chtchelkanova et al. 1997]). As opposed to Cannon's algorithm, it allows a tradeoff to be made between latency cost and memory by varying the degree of blocking. The algorithm, illustrated in Figure 14.11, proceeds in k/b stages. At each stage, $\sqrt{p}$ active row processors broadcast b columns of $\mathbf{A}$ simultaneously along their rows and $\sqrt{p}$ active column processors broadcast b rows of $\mathbf{B}$ simultaneously along their columns.

Sparse SUMMA is our algorithm of choice for our final implementation because it is easy to generalize to nonsquare matrices, matrices whose dimensions are not perfectly divisible by grid dimensions.

14.4 Analysis of parallel algorithms

In this section, we analyze the parallel performance of our algorithms and show that they scale better than existing 1D algorithms in theory. We begin by introducing

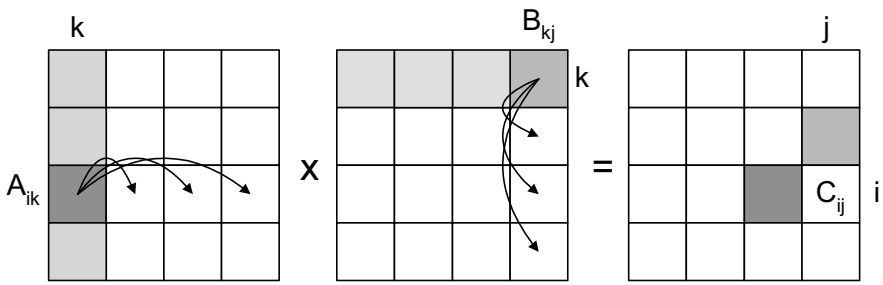

Figure 14.11. Sparse SUMMA execution ($b = N/\sqrt{p}$).

our parameters and model of computation. Then, we present a theoretical analysis showing that 1D decomposition, at least with the current algorithm, is not sufficient for PSpGEMM to scale. Finally, we analyze our 2D algorithms in depth.

In our analysis, the cost of one floating-point operation, along with the cost of cache misses and memory indirections associated with the operation, is denoted by γ, measured in nanoseconds. The latency of sending a message over the communication interconnect is α, and the inverse bandwidth is β, measured in nanoseconds and nanoseconds per word transferred, respectively. The running time of a parallel algorithm on p processors is given by

$$T_p = T_{comm} + T_{comp}$$

where T_{comm} denotes the time spent in communication and T_{comp} is the time spent during local computation phases. T_{comm} includes both the latency (delay) costs and the actual time it takes to transfer the data words over the network. Hence, the cost of transmitting h data words in a communication phase is

$$T_{comm} = \alpha + h\beta$$

The sequential work of SpGEMM, unlike dense GEMM, depends on many parameters. This makes parallel scalability analysis a tough process. Therefore, we restrict our analysis to sparse matrices following the Erdős–Rényi graph model. Consequently, the analysis is probabilistic, exploiting the independent and identical distribution of nonzeros. When we talk about quantities such as nonzeros per subcolumn, we mean the expected number of nonzeros. Our analysis assumes that there are $c > 0$ nonzeros per row/column. The sparsity parameter c, albeit oversimplifying, is useful for analysis purposes since it makes different parameters comparable to each other. For example, if $\mathbf{A}$ and $\mathbf{B}$ both have sparsity c, then $nnz(\mathbf{A}) = cN$ and $\text{flops}(\mathbf{AB}) = c^2N$. It also allows us to decouple the effects of load imbalances from the algorithm analysis because the nonzeros are assumed to be evenly distributed across processors.

The lower bound on sequential SpGEMM is $\Omega(\text{flops}) = \Omega(c^2N)$. This bound is achieved by some row wise and column wise implementations (see [Gilbert et al. 1992 Gustavson 1978]), provided that $c \geq 1$. The row wise implementation of Gustavson that uses CSR is the natural kernel to be used in the 1D algorithm where data is distributed by rows. As shown in the previous chapter, it has an asymptotic complexity of

$$O(N + nnz(\mathbf{A}) + \text{flops}) = O(N + cN + c^2N) = \Theta(c^2N)$$

Therefore, we take the sequential work (W) to be $\gamma c^2 N$ in our analysis.

14.4.1 Scalability of the 1D algorithm

We begin with a theoretical analysis whose conclusion is that 1D decomposition is not sufficient for PSpGEMM to scale. In Block1D_PSpGEMM, each processor sends and receives $p-1$ point-to-point messages of size $nnz(\mathbf{B})/p$. Therefore,

$$T_{comm} = (p-1)\left(\alpha + \beta\,\frac{nnz(\mathbf{B})}{p}\right) = \Theta(p\,\alpha + \beta\,c\,N) \tag{14.5}$$

We previously showed that the BLOCK1D_PSPGEMM algorithm is unscalable with respect to both communication and computation costs (see, for instance, [Buluç & Gilbert 2008a]). In fact, it gets slower as the number of processors grows. The current STAR-P implementation (see [Shah 2007]) bypasses this problem by all-to-all broadcasting nonzeros of the $\mathbf{B}$ matrix, so that the whole $\mathbf{B}$ matrix is essentially assembled at each processor. This avoids the cost of loading and unloading SPA at every stage, but it uses $nnz(\mathbf{B})$ memory at each processor.

14.4.2 Scalability of the 2D algorithms

In this section, we provide an in-depth theoretical analysis of our parallel 2D SpGEMM algorithms and conclude that they scale significantly better than their 1D counterparts. Although our analysis is limited to the Erdős–Rényi model, its conclusions are strong enough to be convincing.

In CANNON_PSPGEMM, each processor sends and receives $\sqrt{p}-1$ point-to-point messages of size $nnz(\mathbf{A})/p$, and $\sqrt{p}-1$ messages of size $nnz(\mathbf{B})/p$. Therefore, the communication cost per processor is

$$T_{comm} = \sqrt{p}\left(2\,\alpha + \beta\left(\frac{nnz(\mathbf{A}) + nnz(\mathbf{B})}{p}\right)\right) = \Theta\left(\alpha\,\sqrt{p} + \frac{\beta\, c\, N}{\sqrt{p}}\right) \qquad (14.6)$$

The average number of nonzeros in a column of a local submatrix $\mathbf{A}_{ij}$ is $c/\sqrt{p}$. Therefore, for a submatrix multiplication $\mathbf{A}_{ik}\mathbf{B}_{kj}$,

$$ni(\mathbf{A}_{ik}, \mathbf{B}_{kj}) = \min\left\{1, \frac{c^2}{p}\right\}\frac{N}{\sqrt{p}} = \min\left\{\frac{N}{\sqrt{p}}, \frac{c^2\, N}{p\sqrt{p}}\right\}$$

$$\text{flops}(\mathbf{A}_{ik}\mathbf{B}_{kj}) = \frac{\text{flops}(\mathbf{AB})}{p\sqrt{p}} = \frac{c^2\, N}{p\sqrt{p}}$$

$$T_{mult} = \sqrt{p}\,\left(2\,\min\left\{1, \frac{c}{\sqrt{p}}\right\}\frac{N}{\sqrt{p}} + \frac{c^2 N}{p\sqrt{p}}\lg\left(\min\left\{\frac{N}{\sqrt{p}}, \frac{c^2 N}{p\sqrt{p}}\right\}\right)\right)$$

The probability of a single column of $\mathbf{A}_{ik}$ (or a single row of $\mathbf{B}_{kj}$) having at least one nonzero is $\min\{1, c/\sqrt{p}\}$, where 1 covers the case $p \le c^2$ and $c/\sqrt{p}$ covers the case $p > c^2$.

The overall cost of additions, using p processors and Brown and Tarjan's $O(m\ \lg n/m)$ algorithm [Brown & Tarjan 1979] for merging two sorted lists of size m and n (for $m < n$), is

$$T_{add} = \sum_{i=1}^{\sqrt{p}}\left(\frac{\text{flops}}{p\sqrt{p}}\ \lg i\right) = \frac{\text{flops}}{p\sqrt{p}}\ \lg\prod_{i=1}^{\sqrt{p}} i = \frac{\text{flops}}{p\sqrt{p}}\ \lg\left(\sqrt{p}\,!\right)$$

Note that we might be slightly overestimating since we assume flops/ $nnz(\mathbf{C}) \approx 1$ for simplicity. From Stirling's approximation and asymptotic analysis, we know

that $\lg(n!) = \Theta(n \lg n)$ [Cormen et al. 2001]. Thus, we get

$$T_{add} = \Theta\left(\frac{\text{flops}}{p\sqrt{p}}\ \sqrt{p}\ \lg\sqrt{p}\right) = \Theta\left(\frac{c^2 N \lg\sqrt{p}}{p}\right)$$

There are two cases to analyze: $p > c^2$ and $p \leq c^2$. Since scalability analysis is concerned with the asymptotic behavior as p increases, we just provide results for the $p > c^2$ case. The total computation cost $T_{comp} = T_{mult} + T_{add}$ is

$$T_{comp} = \gamma\left(\frac{c\,n}{\sqrt{p}} + \frac{c^2 n}{p}\lg\left(\frac{c^2 n}{p\sqrt{p}}\right) + \frac{c^2 n \lg\sqrt{p}}{p}\right) = \gamma\left(\frac{c\,n}{\sqrt{p}} + \frac{c^2 n}{p}\lg\left(\frac{c^2 n}{p}\right)\right) \quad (14.7)$$

In this case, parallel efficiency is

$$E = \frac{W}{p\left(T_{comp} + T_{comm}\right)} = \frac{\gamma c^2 n}{(\gamma+\beta)\,c\,n\,\sqrt{p} + \gamma c^2 n\ \lg\left(\frac{c^2 n}{p}\right) + \alpha\, p\,\sqrt{p}} \quad (14.8)$$

Scalability is not perfect and efficiency deteriorates as p increases due to the first term. Speedup is, however, not bounded, as opposed to the 1D case. In particular, $\lg(c^2 n/p)$ becomes negligible as p increases, and scalability due to latency is achieved when $\gamma c^2 n \propto \alpha\, p\sqrt{p}$, where it is sufficient for n to grow on the order of $p^{1.5}$. The biggest bottleneck for scalability is the first term in the denominator, which scales with $\sqrt{p}$. Consequently, two different scaling regimes are likely to be present: a close to linear scaling regime until the first term starts to dominate the denominator and a $\sqrt{p}$-scaling regime afterwards.

Compared to the 1D algorithms, Sparse Cannon both lowers the degree of unscalability due to bandwidth costs and mitigates the bottleneck of computation. This makes overlapping communication with computation more promising.

Sparse SUMMA, like dense SUMMA, incurs an extra cost over Cannon for using row wise and column wise broadcasts instead of nearest-neighbor communication, which might be modeled as an additional $O(\lg p)$ factor in communication cost. Other than that, the analysis is similar to Sparse Cannon and we omit the details. Using the DCSC data structure, the expected cost of fetching b consecutive columns of a matrix $\mathbf{A}$ is b plus the size (number of nonzeros) of the output [Buluç & Gilbert 2008b]. Therefore, the algorithm asymptotically has the same computation cost for all values of b.

14.5 Performance modeling of parallel algorithms

In this section, we project the estimated speedup of 1D and 2D algorithms in order to evaluate their prospects in practice. We use a quasi-analytical performance model in which we first obtain realistic values for the parameters (γ, β, α) of the algorithm performance, then use them in our projections.

In order to obtain a realistic value for γ, we performed multiple runs on an AMD Opteron 8214 (Santa Rosa) processor using matrices of various dimensions

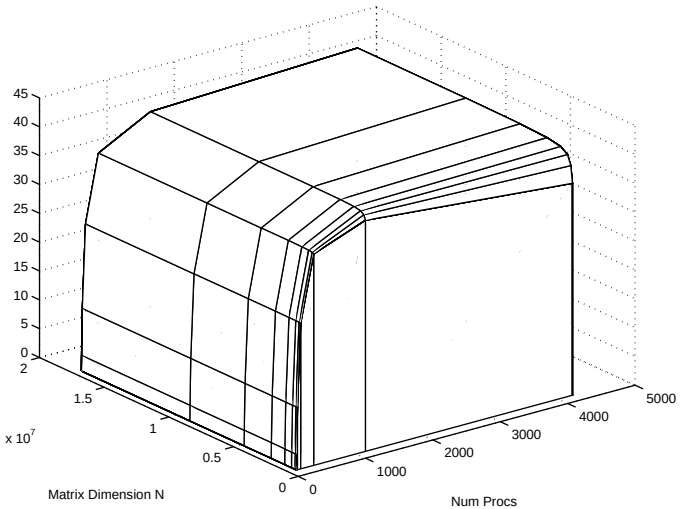

Figure 14.12. Modeled speedup of synchronous sparse 1D algorithm.

and sparsity, estimating the constants using nonlinear regression. One surprising result is the order of magnitude difference in the constants between sequential kernels. The classical algorithm, which is used as the 1D SpGEMM kernel, has $\gamma = 293.6$ nsec, whereas HYPERSPARSE_GEMM, which is used as the 2D kernel, has $\gamma = 19.2$ nsec. We attribute the difference to cache friendliness of the hypersparse algorithm. The interconnect supports $1/\beta = 1$ GB/sec point-to-point bandwidth and a maximum of $\alpha = 2.3$ microseconds latency, both of which are achievable on TACC's Ranger Cluster. The communication parameters ignore network contention.

Figures 14.12 and 14.13 show the modeled speedup of BLOCK1D_PSPGEMM and CANNON_PSPGEMM for matrix dimensions from $N = 2^{17}$ to 2^{24} and number of processors from $p = 1$ to 4096. The inputs are Erdős–Rényi graphs.

We see that BLOCK1D_PSPGEMM's speedup does not go beyond 50x, even on larger matrices. For relatively small matrices, having dimensions $N = 2^{17} - 2^{20}$, it starts slowing down after a thousand processors, where it achieves less than 40x speedup. On the other hand, CANNON_PSPGEMM shows increasing and almost linear speedup for up to 4096 processors even though the slope of the curve is less than one. It is crucial to note that the projections for the 1D algorithm are based on the memory inefficient implementation that performs an all-to-all broadcast of $\mathbf{B}$. This is because the original memory efficient algorithm given in Section 14.3.1 actually slows down as p increases.

It is worth explaining one peculiarity. The modeled speedup turns out to be higher for smaller matrices than for bigger matrices. Remember that communication requirements are on the same order as computational requirements for parallel SpGEMM. Intuitively, the speedup should be independent of the matrix dimension in the absence of load imbalance and network contention, but since we are estimating the speedup with respect to the optimal sequential algorithm, the

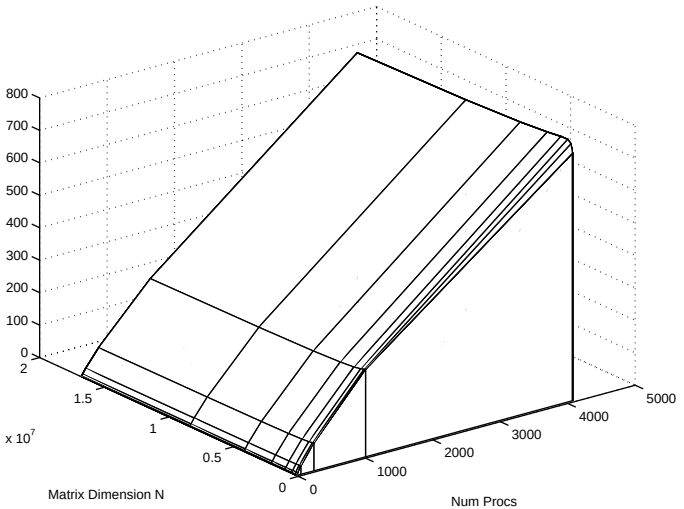

Figure 14.13. Modeled speedup of synchronous Sparse Cannon.

overheads associated with the hypersparse algorithm are bigger for larger matrices. The bigger the matrix dimension, the slower the hypersparse algorithm is with respect to the optimal algorithm, due to the extra logarithmic factor. Therefore, speedup is better for smaller matrices in theory. This is not the case in practice because the peak bandwidth is usually not achieved for small-sized data transfers and load imbalances are severer for smaller matrices.

We also evaluate the effects of overlapping communication with computation. Following Krishnan and Nieplocha [Krishnan & Nieplocha 2004], we define the nonoverlapped percentage of communication as

$$w = 1 - \frac{T_{comp}}{T_{comm}} = \frac{T_{comm} - T_{comp}}{T_{comm}}$$

The speedup of the asynchronous implementation is

$$S = \frac{W}{T_{comp} + w(T_{comm})}$$

Figure 14.14 shows the modeled speedup of asynchronous SpCannon assuming truly one-sided communication. For smaller matrices with dimensions $N = 2^{17}-2^{20}$, speedup is about 25% more than the speedup of the synchronous implementation.

The modeled speedup plots should be interpreted as upper bounds on the speedup that can be achieved on a real system using these algorithms. Achieving these speedups on real systems requires all components to be implemented and working optimally. The conclusion we derive from those plots is that no matter how hard we try, it is impossible to get good speedup with the current 1D algorithms.

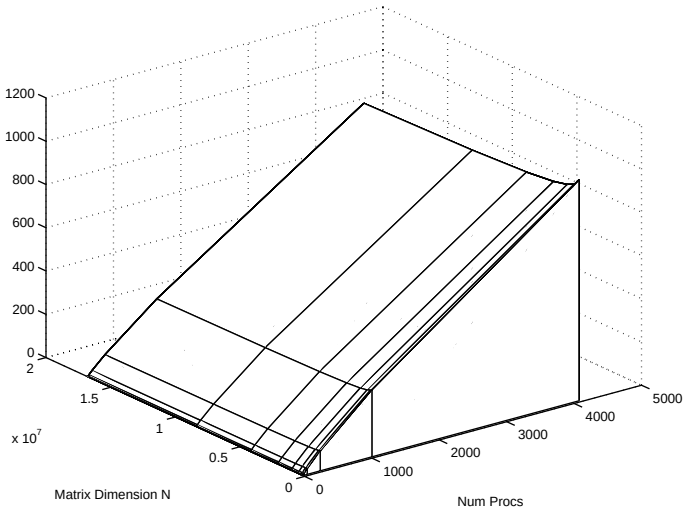

Figure 14.14. Modeled speedup of asynchronous Sparse Cannon.

References

[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.

[Bader et al. 2004] 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.

[Briggs et al. 2000] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A multigrid tutorial, Second Edition*. Philadelphia: SIAM, 2000.

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

[Buluç & Gilbert 2008a] A. Buluç and J.R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *The 37th International Conference on Parallel Processing (ICPP '08)*, 503–510, Portland, Oregon, 2008.

[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)*, 1530–2075, Miami, FL, 2008.

[Cannon 1969] L.E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. Ph.D. thesis, Montana State University, 1969.

[Chtchelkanova et al. 1997] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R.A. van de Geijn. Parallel implementation of BLAS: General techniques for Level 3 BLAS. *Concurrency: Practice and Experience*, 9:837–857, 1997.

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

[Coppersmith & Winograd 1987] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the 19th Annual ACM Conference on Theory of Computing*, 1–6, New York: ACM Press.

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

[D'Alberto & Nicolau 2007] P. D'Alberto and A. Nicolau. R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47:203–213, 2007.

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

[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.

[Gilbert et al. 2008] J.R. Gilbert, S. Reinhardt, and V.B. Shah. A unified framework for numerical and combinatorial computing. *Computing in Science and Engineering*, 10:20–25, 2008.

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

[Irony et al. 2004] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64:1017–1026, 2004.

[Irwin et al. 1997] 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*, 249–256, London: Springer-Verlag, 1997.

[Kleinberg et al. 1979] J.M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web as a graph: Measurements, models, and methods. In *Proceedings of the 5th Annual Conference on Computing and Combinatorics (COCOON '99)*, 1–17, 1999.

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

[Krishnan & Nieplocha 2004] M. Krishnan and J. Nieplocha. SRUMMA: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*, 70, IEEE Computer Society, 2004.

[Leskovec et al. 2005] J. Leskovec, D. Chakrabarti, J.M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005)*, 133–145, 2005.

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

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

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

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

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

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

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

[Shah 2007] V.B. Shah. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity.* Ph.D. thesis, University of California, Santa Barbara, June 2007.

[Sleator & Tarjan 1985] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[Sulatycke & Ghose 1998] P. Sulatycke and K. Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS '98)*, 117–123, IEEE Computer Society, 1998.

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

[Van De Geijn & Watts 1997] R.A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9:255–274, 1997.

[Yoo et al. 2005] 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 *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 25, IEEE Computer Society, 2005.

[Yuster & Zwick 2004] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, 254–260, Philadelphia: SIAM, 2004.

[Yuster & Zwick 2005] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Transactions on Algorithms*, 1:2–13, 2005.