Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication *

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

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

Abstract

We identify the challenges that are special to parallel sparse matrix-matrix multiplication (PSpGEMM). We show that sparse algorithms are not as scalable as their dense counterparts, because in general, there are not enough non-trivial arithmetic operations to hide the communication costs as well as the sparsity overheads. We analyze the scalability of 1D and 2D algorithms for PSpGEMM. While the 1D algorithm is a variant of existing implementations, 2D algorithms presented are completely novel. Most of these algorithms are based on the previous research on parallel dense matrix multiplication. We also provide results from preliminary experiments with 2D algorithms.

1 Introduction

We consider the problem of multiplying two sparse matrices in parallel (PSpGEMM). Sparse matrix-matrix multiplication is a building block for many algorithms including graph contraction, breadth-first search from multiple source vertices, peer pressure clustering [21], recursive formulations of all-pairs-shortest-path algorithms [8], multigrid interpolation/restriction [2], and parsing context-free languages [19]. A scalable PSpGEMM algorithm, operating on a semiring, is a key component for building scalable parallel versions of those algorithms.

In this paper, we identify the challenges in the design and implementation of a scalable PSpGEMM algorithm. Section 5 explains those challenges in more detail. We also propose novel algorithms based on 2D block decomposition of data, analyze their computation and communication requirements, compare their scalability, and finally report timings from our initial implementations.

*This research was supported in part by the Department of Energy under award number DE-FG02-04ER25632, in part by MIT/LL under contract number 7000012980, and in part by the National Science Foundation through TeraGrid resources provided by TACC.

To the best of our knowledge, parallel algorithms using a 2D block decomposition have not been developed for sparse matrix-matrix multiplication. Existing 1D algorithms are not scalable to thousands of processors. On the contrary 2D dense matrix multiplication algorithms are proved to be optimal with respect to the communication volume [15], making 2D sparse algorithms likely to be more scalable than their 1D counterparts. We show that this intuition is indeed correct.

Widely used sequential sparse matrix-matrix multiplication kernels, such as the one used in MATLAB [13] and CSparse [9], are not suitable for 2D decomposition as shown before [4]. Therefore, for our 2D algorithms, we use an outer product based kernel, which is scalable with respect to the total amount of work performed.

In Section 6, we compare the scalability of 1D and 2D algorithms for different matrix sizes. In those speed-up projections, we used our real-world estimates of constants instead of relying on the O-notation. We also investigated the effectiveness of overlapping communication with computation. Finally, we implemented and tested a 2D algorithm for PSpGEMM using matrices from models of real world graphs.

2 Terminology

The sparse matrix-matrix multiplication problem (SpGEMM) is to compute $C = AB$, where the input matrices A and B, having dimensions $M \times K$ and $K \times N$ respectively, are both sparse. Although the algorithms operate on rectangular matrices, our analysis assumes square matrices ($M = K = N$) in order not to over-complicate the statement of the results.

In PSpGEMM, we perform multiplication utilizing p processors. 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 . α is the latency of sending a message over the communication interconnect. β is the inverse bandwidth, measured in nanoseconds per word transfered.

The number of nonzero elements in A is denoted by $nnz(A)$. We adopt the colon notation for sparse matrix indexing, where $A(:,i)$ denotes the ith column, $A(i,:)$ denotes the ith row, and $A(i,j)$ denotes the element at the $(i,j)^{\text{th}}$ position of matrix A. *Flops* (flops(AB) or flops in short) is defined as the number of nonzero arithmetic operations required to compute the product AB.

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 with $c > 0$ nonzeros per row/column, uniformly-randomly distributed across the 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 A and B both have sparsity c, then $nnz(A) = cN$ and flops$(AB) = c^2N$. 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. Our analysis is probabilistic, exploiting the uniform random distribution of nonzeros. Therefore, when we talk about quantities such as nonzeros per subcolumn, we mean the expected number of nonzeros.

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 [13, 14], provided that $c \geq 1$. The row-wise implementation of Gustavson is the natural kernel to be used in the 1D algorithm where data is distributed by rows. It has a asymptotic complexity of

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

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

Our sequential SpGEMM kernel used in 2D algorithms requires matrix B to be transposed before the computation, at a cost of $trans(B)$, which can be $O(N + nnz(B))$ or $O(nnz(B)\ \lg nnz(B))$, depending on the implementation. The kernel has total complexity of

$$O(nzc(A) + nzr(B) + \text{flops}\ \lg ni + trans(B)),$$

where $nzc(A)$ is the number of columns of A that contain at least one nonzero, $nzr(B)$ is the number of rows of B that contain at least one nonzero, and ni is the number of indices i for which $A(:,i) \neq \emptyset$ and $B(i,:) \neq \emptyset$.

The data structure used by sequential SpGEMM is DCSC (doubly compressed sparse column). DCSC has the following properties:

1. Its storage is strictly $O(nnz)$
2. It supports a scalable sequential SpGEMM algorithm.
3. It allows fast access to columns of the matrix.

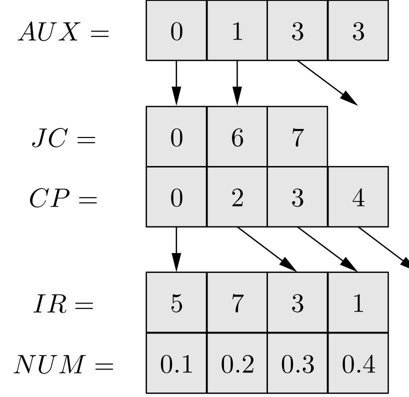

Figure 1. Matrix A in DCSC format

DCSC, therefore, is our data structure of choice for storing submatrices owned locally by each processor in 2D algorithms. Figure 1 shows DCSC representations of a 9-by-9 matrix with 4 nonzeros that has the following triples representation:

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

More details on DCSC and the scalable sequential SpGEMM can be found in our previous work [4].

3 Parallel 1D Algorithm

In 1D algorithms, data can be distributed by rows or columns. We consider distribution by rows; distribution by columns is symmetric. For the analysis, we assume that pairs of adjacent processors can communicate simultaneously without affecting each other. The algorithm used in practice in STAR-P [20] is a variant of this row-wise algorithm where simultaneous point to point communications are replaced with all-to-all broadcasts.

The data is distributed to processors in block rows, where each processor receives M/p consecutive rows. For a matrix A, A_i denotes the block row owned by the ith processor. To simplify the algorithm description, we use A_{ij} to denote $A_i(:, jp : (j+1)p - 1)$, the jth block column of A_i, although block rows are physically not partitioned.

$$A = \begin{pmatrix} A_1 \\ \hline \dots \\ \hline A_p \end{pmatrix} = \left(\begin{array}{c|c|c} A_{11} & \dots & A_{1p} \\ \hline \dots & & \dots \\ \hline A_{p1} & \dots & A_{pp} \end{array}\right), B = \begin{pmatrix} B_1 \\ \hline \dots \\ \hline B_p \end{pmatrix}$$

For each processor P_i, the computation is set up in the following way:

$$C_i = C_i + A_i\, B = C_i = C_i + \sum_{j=1}^{p} A_{ij}\, B_j$$

Each processor sends and receives $p-1$ point-to-point messages of size $nnz(B)/p$. Therefore,

$$T_{comm} = (p-1)\big(\alpha + \beta \frac{nnz(B)}{p}\big) = \Theta(p\alpha + \beta\, c\, N). \quad (1)$$

The row-wise SpGEMM forms one row of C at a time, and they might need to access all of B for that. However, only a portion of B is accessible at any time in parallel algorithms. This requires multiple iterations to fully form one row of C. The nonzero structure of the current active row of C is formed by using an $O(N)$ data structure called the sparse accumulator (SPA) [13]. It is composed of three components: a dense vector that holds the real values for the active row of C, a dense boolean vector that holds the "occupied" flags, and a sparse list that holds the indices of nonzero elements in the active row of C so far. Initialization of the SPA takes $O(N)$ time but loading and unloading can be done in time proportional to the number of nonzeros in the SPA.

```
1  for all processors P_i in parallel do
2      SPA ← Initialize;
3      for j ← 1 to p do
4          Broadcast(B_j);
5          for k ← 1 to N/p do
6              SPA ← Load(C_i(k,:));
7              SPA ← SPA + A_ij(k,:) * B_j;
8              C_i(k,:) ← UnLoad(SPA);
9          end
10     end
11 end
```

Algorithm 1: $\mathbf{C} \leftarrow \mathbf{A} * \mathbf{B}$ **using row Sparse1D**

The pseudocode is given in Algorithm 1. The computational time is dominated by loads and unloads of SPA, which is not amortized by the number of nonzero arithmetic operations in general. The simple running example in Figure 2 shows an extreme case. In this figure, $N = 6$, $p = 3$, and each x denotes a nonzero. It is easy to see that $\text{flops}(AB) = n^2$ due to the first outer product. After the very first execution of the inner loop in Algorithm 1, SPA becomes completely dense for every row $C_i(k,:)$. Those dense SPA's are going to be loaded/unloaded $p-1$ more times during the subsequent iterations of the outer-loop, even though no additional flops are needed.

Since there are N/p rows per processor, and each row leads to p loads/unloads of the corresponding SPA, we have N SPA loads/unloads per processors. Each of those loads takes $O(nnz(\mathsf{SPA})) = N$ time, making up a cost of $O(N^2)$ per processor, when flops per processors where only $O(N^2/p)$.

$$A = \left(\begin{array}{cccccc} \mathrm{x} & 0 & 0 & 0 & 0 & 0 \\ \mathrm{x} & 0 & 0 & 0 & 0 & 0 \\ \hline \mathrm{x} & 0 & 0 & 0 & 0 & 0 \\ \mathrm{x} & 0 & 0 & 0 & 0 & 0 \\ \hline \mathrm{x} & 0 & 0 & 0 & 0 & 0 \\ \mathrm{x} & 0 & 0 & 0 & 0 & 0 \end{array}\right), B = \left(\begin{array}{cccccc} \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Figure 2. Worst case for 1D Algorithm

Analyzing the general case is hard without making some assumptions, even for random matrices. Here, we assume that each nonzero in the output is contributed by a small (and constant) number of flops. In practice, this is true for many sparse matrices, and $\text{flops}/nnz(C) \approx 1$ for reasonably large N.

The cost of SPA load/unload per row is then:

$$\frac{\text{flops}(AB)}{Np} \cdot \sum_{i=1}^{p} i = \frac{c^2 N}{Np} \cdot p \cdot (p+1) = \Theta(c^2 p)$$

Apart from SPA loads/unloads, each processor iterates p times over its rows. As each processor has N/p rows, the cost of computation per processor becomes

$$T_{comp} = \gamma\big(\frac{N}{p}(p + c^2 p)\big) = \gamma\big(N(1+c^2)\big). \quad (2)$$

The efficiency of the algorithm overall is:

$$\begin{aligned} E &= \frac{W}{p\,(T_{comp} + T_{comm})} \\ &= \frac{\gamma c^2 N}{p\,\big(\gamma(N(1+c^2)) + p\,\alpha + \beta\, cN\big)} \end{aligned} \quad (3)$$

We see that the communication to computation ratio of 1D PSpGEMM is higher than its dense counterpart. In the dense case, $O(p\,N^2)$ communication was dominated by the $O(N^3)$ computation in a scaled speedup setting. To keep the efficiency constant in PSpGEMM, work should increase proportionally to the cost of parallel overheads.

We separately consider the factors that limit scalability. Scalability due to latency is achieved when $\gamma c^2 N \propto \alpha\, p^2$, which requires N to grow on the order of p^2. Scalability due to bandwidth costs is achieved when

$$\begin{aligned} \gamma c^2 N &\propto \beta\, c\, p\, N \\ \frac{\alpha c}{\beta} &= \Theta(1) \propto p\,, \end{aligned}$$

which is clearly unscalable with increasing number of processors.

Having shown that 1D algorithm is not scalable in terms of the amount of communication it does, we now analyze its scalability due to computation costs.

$$\gamma c^2 N \propto \gamma\, p\, N(c^2+1)$$
$$\frac{c^2+1}{c^2} = \Theta(1) \propto p\,.$$

We see that there is no way to keep parallel efficiency constant with the 1D algorithm, even when we ignore communication costs. In fact, the algorithm gets slower as the number of processors increase. STAR-P implementation by-passes this problem by all-to-all broadcasting nonzeros of the B matrix, so that the whole B matrix is essentially assembled at each processor.

This type of implementation avoids the cost of loading/unloading SPA at every stage, but it introduces a memory problem. Aggregate memory usage among all processors becomes $nnz(A) + p\, nnz(B) + nnz(C)$, which is unscalable. Furthermore, the whole B matrix is unlikely to fit into the memory of a single node in real applications. There is clearly a need to design better 1D algorithms, that avoid SPA loading/unloading and at the same time do not impose memory problems.

4 Parallel 2D Algorithms

4.1 Sparse Cannon (SpCannon)

Our first 2D algorithm is based on Cannon's algorithm for dense matrices [5]. The p processors are logically organized on a $(\sqrt{p} \times \sqrt{p})$ mesh, indexed by their row and column indices so that the $(i,j)^{\text{th}}$ processor is denoted by P_{ij}. 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, A is partitioned as shown below and A_{ij} is assigned to processor P_{ij}.

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

The pseudocode of the algorithm is given in Algorithm 2. Each processor sends and receives $\sqrt{p}-1$ point-to-point messages of size $nnz(A)/p$, and $\sqrt{p}-1$ messages of size $nnz(B)/p$. Therefore, the communication cost per processor is

$$T_{comm} = \sqrt{p}\,\Big(2\,\alpha + \beta\,\big(\frac{nnz(A)}{p} + \frac{nnz(B)}{p}\big)\Big)$$
$$= 2\,\alpha\,\sqrt{p} + \frac{2\,\beta\,c\,N}{\sqrt{p}} = \Theta(\alpha\,\sqrt{p} + \frac{\beta\,c\,N}{\sqrt{p}}). \quad (4)$$

```
1  for all processors P_ij in parallel do
2  |   LeftCircularShift(A_ij, j);
3  |   UpCircularShift(B_ij, i);
4  end
5  for all processors P_ij in parallel do
6  |   for k ← 1 to √p do
7  |   |   C_ij ← C_ij + A_ij * B_ij ;
8  |   |   LeftCircularShift(A_ij, 1);
9  |   |   UpCircularShift(B_ij, 1);
10 |   end
11 end
12 for all processors P_ij in parallel do
13 |   LeftCircularShift(A_ij, j);
14 |   UpCircularShift(B_ij, i);
15 end
```

Algorithm 2: $\mathbf{C} \leftarrow \mathbf{A} * \mathbf{B}$ **using SpCannon**

The expected number of nonzeros in a given column of a local submatrix A_{ij} is $c/\sqrt{p}$. Therefore, for a submatrix multiplication $A_{ik}B_{kj}$,

$$ni(A_{ik}, B_{kj}) = \min\big\{1, \frac{c^2}{p}\big\}\frac{N}{\sqrt{p}} = \min\{\frac{N}{\sqrt{p}}, \frac{c^2\,N}{p\sqrt{p}}\}$$

$$\text{flops}(A_{ik}B_{kj}) = \frac{\text{flops}(AB)}{p\sqrt{p}} = \frac{c^2\,N}{p\sqrt{p}}$$

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

Overall cost of additions using p processors and Brown and Tarjan's $O(m\ \lg n/m)$ merging algorithm [3] is

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

Note that we might be slightly overestimating, since we assume $\text{flops}/nnz(C) \approx 1$ as we did in the analysis of the 1D algorithm. From Stirling's approximation and asymptotic analysis, we know that $\lg\,(n\,!) = \Theta(n \lg n)$ [7]. Thus, we get:

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

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

$$T_{comp} = \gamma\Big(c + \frac{c^2N}{p}\lg(\frac{c^2N}{p\sqrt{p}}) + \frac{c^2N\lg\sqrt{p}}{p}\Big)$$
$$= \gamma\Big(c + \frac{c^2N}{p}\lg(\frac{c^2N}{p})\Big) \quad (5)$$

In this case, efficiency becomes

$$E = \frac{W}{p\,(T_{comp} + T_{comm})}$$
$$= \frac{\gamma c^2 N}{p\,\Big(\gamma\Big(c + \frac{c^2N}{p}\lg(\frac{c^2N}{p})\Big) + \alpha\sqrt{p} + \frac{\beta\, c\, N}{\sqrt{p}}\Big)}$$
$$= \frac{\gamma c^2 N}{\gamma cp + \gamma c^2 N\,\lg\big(\frac{c^2N}{p}\big) + \alpha\, p\sqrt{p} + \beta\, c\sqrt{p}\, N} \quad (6)$$

Scalability due to computation is much better than the 1D case. As long as N grows linearly with p, efficiency due to computation is constant at $1/\lg(c^2N/p)$. Scalability due to latency is achieved when $\gamma c^2 N \propto \alpha\, p\sqrt{p}$, which requires N to grow on the order of $p^{1.5}$. Bandwidth cost is now the sole bottleneck in scalability. For keeping the efficiency constant, we need

$$\gamma c^2 N \propto \beta\, c\sqrt{p}\, N$$
$$\frac{\gamma c}{\beta} = \Theta(1) \propto \sqrt{p}\,.$$

Although still unscalable, the degree of unscalability is lower than the 1D case. Consequently, 2D algorithms both lower the degree of unscalability due to bandwidth costs and mitigate the bottleneck of computation. This makes overlapping communication with computation becomes more promising.

4.2 Sparse SUMMA (SpSUMMA)

SUMMA [11] is a memory efficient, easy to generalize algorithm for parallel dense matrix multiplication. It is the algorithm used in parallel BLAS [6]. 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 3, proceeds in K/b stages. At each stage, $\sqrt{p}$ active row processors broadcast b columns of A simultaneously along their rows and $\sqrt{p}$ active column processors broadcast b rows of B simultaneously along their columns.

Sparse SUMMA , like dense SUMMA , incurs an extra cost over Cannon for using row-wise and col-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

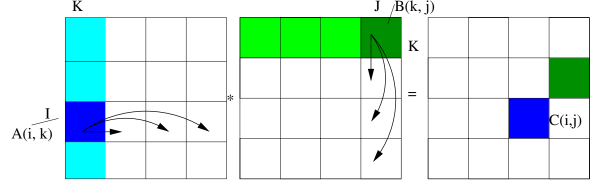

Figure 3. SpSUMMA Execution ($b = N/\sqrt{p}$)

sparse Cannon and we omit the details. Using the DCSC data structure, the expected cost of fetching b consecutive columns of a matrix A is b plus the size (number of nonzeros) of the output [4]. Therefore, the algorithm asymptotically has the same computation cost for all values of b.

5 Challenges

We identify three challenges that are specific to the sparse case. The first one is the load imbalance among processors. Each processor performs a total of W/p work in the dense case where $W = N^3$, but such a balance is hard to achieve in the sparse case. Furthermore, dense GEMM algorithms [5, 11] usually execute in a synchronous manner in s stages, where each processor performs $W/p\,s$ work per stage. For sparse matrices, achieving good load balance per stage is even harder than achieving load balance for the whole computation. Although our analysis assumes constant row/col degree, we have experimented with random permutations for load balancing on more realistic graphs. Random permutations turned out to be reasonably successful except for some class of matrices with full diagonal [4], such as 3D geometric graphs [12]. Asynchronous algorithms might be helpful for mitigating the load balance problem for matrices with a full diagonal.

The second challenge is the addition of submatrices. In the dense case, no extra operations are performed for submatrix additions; the same amount of work would need to be done in the sequential case as well. Contrary to the dense case, $A \leftarrow A+B$ can not be performed in time proportional to the number of nonzeros in B, when both A and B are sparse.

The third challenge is to hide the communication. While this is relatively easier in the dense case due to high computation to communication ratio, it becomes more challenging in the sparse case. Merely increasing the problem size is generally not enough to decrease the communication to computation ratio of sparse algorithms. One sided communication with zero copy RDMA operations previously showed its effectiveness, particularly in the context of dense matrix multiplication [17] and 3D Fourier Transform [1]. In the next section, we estimate the applicability of one-sided communication to sparse Cannon algorithm.

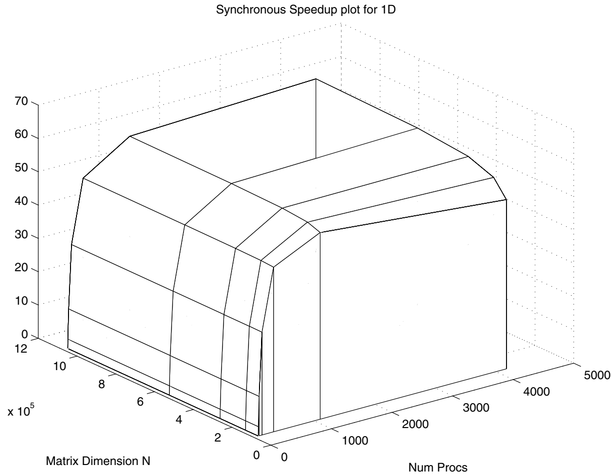

Figure 4. Speedup of Sparse1D

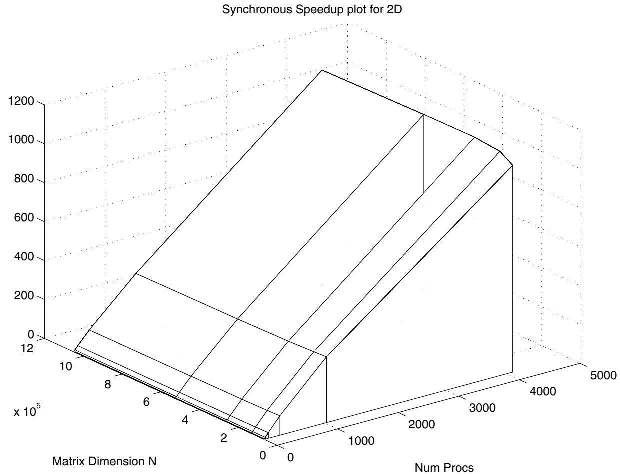

Figure 5. Speedup of SpCannon

6 Experiments and Implementation

In this section, we project the estimated speedup of 1D and 2D algorithms and give timing results from a preliminary version of Sparse SUMMA algorithm that we implemented using MPI.

For realistic evaluation, we first evaluated the constants in the sequential algorithms on a 2.2 Ghz Opteron, in order to be used in projections. We performed multiple runs with different N and c values; estimating the constants using non-linear regression. One surprising result is the order of magnitude difference in the constants between sequential kernels. 1D SpGEMM kernel has $\gamma = 293.6$ nsec, whereas the 2D kernel has $\gamma = 19.2$ nsec. We attribute the difference to cache friendliness of the 2D SpGEMM kernel. We assume an interconnect with 1 GB/sec point-to-point bandwidth, and 1 microsecond latency. TACC's Dell Linux Cluster (Lonestar) is an example system that has the assumed network settings.

Figures 4 and 5 show the estimated speedup of Sparse1D and SpCannon for matrix dimensions from $N = 2^{16}$ to 2^{20} and number of processors from $p = 1$ to 4096. The number of nonzeros per column is 7 throughout the experiments, which is in line with many real world graphs such as the Internet graph [16].

We see that Sparse1D's speedup does not go beyond 60x, even with bigger matrix dimensions. It also starts showing diminishing returns for relatively small matrix sizes. On the contrary, SpCannon shows positive and almost linear speedup for up to 4096 processors, even though the slope of the curve is less than ideal. It is crucial to note that the projections for Sparse1D are based on the complexity of STAR-P 's implementation even though it is shown to be memory inefficient. This is because the original memory efficient algorithm given in Section 3 actually slows down as p increases.

In the second set of projections, we evaluate the effects of overlapping communication with computation. The non-overlapped percentage of the communication [17] is defined as:

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

Since the cost of additions in 2D algorithms is asymptotically different at each stage, we explicitly calculated the amount of non-overlapped communication in stage granularity [1]. Then, the speedup of the asynchronous implementation is given by:

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

Figure 6 shows the estimated speedup of asynchronous SpCannon assuming one-sided communication with RDMA support. For relatively smaller matrices, speedup is about 30% more than the speedup of a synchronous implementation. Interestingly, speedup is higher for smaller matrices (more than 1200x for 4096 processors). This is because almost all communication is overlapped with computation. In this case, as noted in Section 4.1, efficiency is $1/\lg{(c^2N/p)}$. In other words, when communication is fully overlapped, parallel efficiency drops as N increases.

The projected speedup plots should be interpreted as an upper bound on the speedup that can be achieved on a real system using the aforementioned algorithms. That requires all components to be implemented and working optimally. The conclusion we can derive from those plots is that no

[1] This can be considered as a simulation in some sense

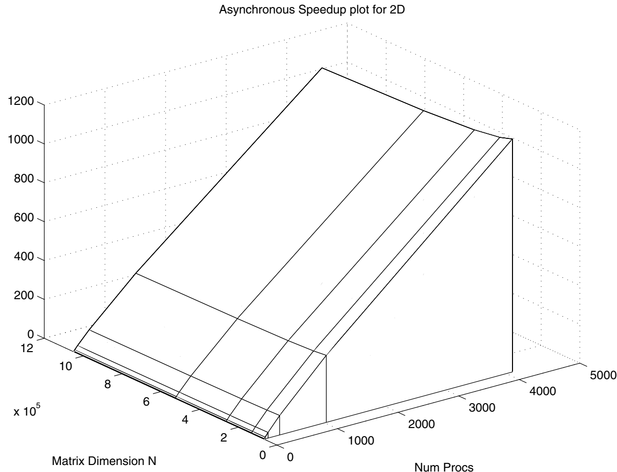

Figure 6. Speedup of SpCannon (Async)

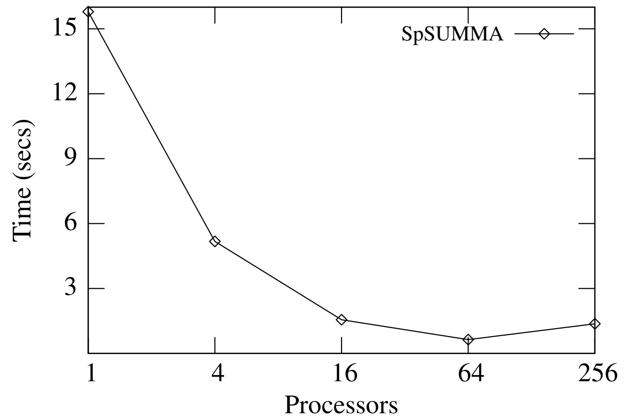

Figure 7. Strong scaling of sparse SUMMA

matter how hard we try, it is impossible to get good speedup with the current 1D algorithms.

Finally, we implemented a real prototype of the Sp-Summa algorithm using C++, with MPI for parallelism. The implementation is synchronous in the sense that overlapping communication with computation is not attempted. We ran our code on the TACC's Dell Linux Cluster (Lonestar), which has 2.66GHz dual-core processors in each node. It has an Infiniband interconnect and we compiled our code using the Intel C++ Compiler version 9.1.

We performed a strong scaling experiment with a matrix dimension of 2^{20}. In our experiments, instead of using random matrices (matrices from Erdős-Rényi random graphs [10]), we used matrices from synthetically generated real-world graphs [18] to achieve results closer to reality. The average number of nonzeros per column is 8 for those synthetically generated graphs. The results are shown in Figure 7.

Our code achieves $10\times$ speedup for 16 processors, and it shows steady speedup until $p = 64$, which are both promising. However, it starts to slow down as the number of processors approach to 256. A couple of factors lead to this behavior:

1. The code currently performs sparse matrix additions by scanning, instead of using an optimal unbalanced merging algorithm. As p increases, the cost of additions starts to become a bottleneck

2. Our input matrices model real worlds graphs with power-law degree distributions. Coupled with our synchronous implementation, this poses a load-balance problem.

3. SpSUMMA has the extra $\lg p$ factor in the communication that also effects the performance as p increases.

Although the results significantly deviate from the ideal speedup curve, they still indicate that ours is a good approach to the problem in practice (one that does not ignore constants so large as to make the algorithm impractical).

7 Conclusions and Future Work

In this paper, we provided full scalability analysis of 1D and 2D algorithms for parallel sparse matrix-matrix multiplication (PSpGEMM). To the best of our knowledge, such an analysis does not exist in the literature. We also proposed the first 2D algorithms for PSpGEMM. We supported our analysis with projections and preliminary experiments.

Both our analysis and experiments show that 2D algorithms are more scalable than their 1D counterparts, both in terms of communication and computation costs. The 2D algorithms proposed in this paper are also amenable to a remote direct memory access (RDMA) implementation to hide the communication costs.

The main future direction of our work is to provide a high-quality implementation of SpSUMMA that scales up to thousands of processors. We are currently developing a non-MPI version that uses the one-sided communication capabilities of GASNET[1] to hide the communication costs. Using an optimal unbalanced merging algorithm is also required for scalability, but our experiments with existing pointer-based merging algorithms show that their constants are quite big to make them practical in an real implementation. Developing a cache-efficient algorithm for unbalanced merging is also one of our priorities.

Lastly, a better 1D algorithm is clearly necessary in order to use 1D algorithms for settings where $p < 100$. Such an algorithm should avoid excessive SPA loading/unloading and should be scalable in terms of memory consumption.

References

[1] C. Bell, D. Bonachea, R. Nishtala, and K. A. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *IPDPS*, Rhodes Island, Greece, 2006. IEEE Computer Society.

[2] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial: second edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[3] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.

[4] A. Buluç and J. R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *IPDPS*, Miami, FL, USA, 2008. IEEE Computer Society.

[5] L. E. Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[6] 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(9):837–857, 1997.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 3, page 55. The MIT Press, second edition, 2001.

[8] 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(2):203–213, 2007.

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

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

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

[12] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM J. Sci. Comput.*, 19(6):2091–2110, 1998.

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

[14] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[15] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[16] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web as a graph: Measurements, models, and methods. In *COCOON*, pages 1–17, 1999.

[17] M. Krishnan and J. Nieplocha. Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *IPDPS*, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

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

[19] G. Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theor. Comput. Sci.*, 354(1):72–81, 2006.

[20] V. Shah and J. R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In *HiPC*, pages 144–155, 2004.

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