Acknowledgement

We acknowledge funding from Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016, and grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and grant 2010231 from the US-Israel Bi-National Science Foundation.

Communication Optimal Parallel Multiplication of Sparse Random Matrices[*]

Grey Ballard[1], Aydın Buluç[2], James Demmel[1], Laura Grigori[3], Benjamin Lipshitz[1], Oded Schwartz[1], and Sivan Toledo[4]

[1]University of California at Berkeley
[2]Lawrence Berkeley National Laboratory
[3]INRIA Paris - Rocquencourt
[4]Tel-Aviv University

Regular Submission

Abstract

Parallel algorithms for sparse matrix-matrix multiplication typically spend most of their time on inter-processor communication rather than on computation, and hardware trends predict the relative cost of communication will only increase. Thus, sparse matrix multiplication algorithms must minimize communication costs in order to scale to large processor counts.

In this paper, we consider multiplying sparse matrices corresponding to Erdős-Rényi random graphs on distributed-memory parallel machines. We prove a new lower bound on the expected communication cost for a wide class of algorithms. Our analysis of existing algorithms shows that, while some are optimal for a limited range of matrix density and number of processors, none is optimal in general. We obtain two new parallel algorithms and prove that they match the expected communication cost lower bound, and hence they are optimal.

[*]We acknowledge funding from Microsoft (Award #024263) and Intel (Award #024894), and matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle and Samsung, as well as MathWorks. Research is also supported by DOE grants DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-SC0008700, and AC02-05CH11231, and DARPA grant HR0011-12-2-0016, and grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and grant 2010231 from the US-Israel Bi-National Science Foundation.

1 Introduction

Computing the product of two sparse matrices is a fundamental problem in combinatorial and scientific computing. Generalized sparse matrix-matrix multiplication is used as a subroutine in algebraic multigrid [5], graph clustering [27] and contraction [15], quantum chemistry [28], and parsing context-free languages [22]. Large-scale data and computation necessitates the use of parallel computing where communication costs quickly become the bottleneck. Existing parallel algorithms for multiplying sparse matrices perform reasonably well in practice for limited processor counts, but their scaling is impaired by increased communication costs at high concurrency.

Achieving scalability for parallel algorithms for sparse matrix problems is challenging because the computations tend not to have the surface to volume ratio (or potential for data re-use) that is common in dense matrix problems. Further, the performance of sparse algorithms is often highly dependent on the sparsity structure of the input matrices. We show in this paper that existing algorithms for sparse matrix-matrix multiplication are sub-optimal in their communication costs, and we obtain new algorithms which are communication optimal, communicating less than the previous algorithms and matching new lower bounds.

Our lower bounds require two important assumptions: (1) the sparsity of the input matrices is random, corresponding to Erdős-Rényi random graphs (see Definition 2.1), and (2) the algorithm is *sparsity-independent*, where the computation is statically partitioned to processors independent of the sparsity structure of the input matrices (see Definition 2.5). The second assumption applies to nearly all existing algorithms for general sparse matrix-matrix multiplication. While *a priori* knowledge of sparsity structure can certainly reduce communication for many important classes of inputs, dynamically determining and efficiently exploiting the structure of general input matrices is a challenging problem. In fact, a common technique of current library implementations is to randomly permute rows and columns of the input matrices in an attempt to destroy their structure and improve computational load balance [7, 9]. Because the input matrices are random, our analyses are in terms of expected communication costs.

We make three main contributions in this paper.

1. **We prove new communication lower bounds.** While there is a previous lower bound which applies to sparse matrix-matrix multiplication [2], it is too low to be attainable. We use a similar proof technique but devise a tighter lower bound on the communication costs in expectation for random input matrices which is independent of the local memory size of each processor. See Section 3 for details.
2. **We obtain two new communication-optimal algorithms.** Our 3D iterative and recursive algorithms (see Sections 4.3 and 4.4) are adaptations of dense ones [13, 25], though an important distinction is that the sparse algorithms do not require extra local memory to minimize communication. We also optimize an existing algorithm, Sparse SUMMA, to be communication-optimal in some cases.
3. **We provide a unified communication analysis of existing and new algorithms.** See Table 1 for a summary of the expected communication costs of the algorithms applied to random input matrices. See Section 4 for a description of the algorithms and their communication analysis.

There are many extensions of the algorithms and analysis presented in this paper. The new algorithms have not yet been benchmarked and compared against previous algorithms. We plan to extend the performance studies of [9] to include all of the algorithms considered here. Additionally, we hope that our analysis can be extended to many more types of input matrices. We are especially interested in sparsity structures corresponding to applications which are currently bottlenecked by sparse matrix-matrix multiplication, such as the triple product computation within algebraic multigrid. In the case of matrix multiplication, we have shown how to apply ideas from dense algorithms to obtain communication-optimal sparse algorithms. Perhaps similar adaptions can be made for other matrix computations such as direct factorizations.

2 Preliminaries

Throughout the paper, we are interested in the computation $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$. For sparse matrix indexing, we use the 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 $\mathbf{A}$. We use flops to denote the number of nonzero arithmetic operations required when computing the product of matrices $\mathbf{A}$ and $\mathbf{B}$ and $nnz(\cdot)$ to denote the number of nonzeros in a matrix or submatrix.

We consider the case where $\mathbf{A}$ and $\mathbf{B}$ are $n \times n$ ER(d) matrices:

Definition 2.1 *An ER(d) matrix is an adjacency matrix of an Erdős-Rényi graph with parameters n and d/n. That is, an ER(d) matrix is a square matrix of dimension n where each entry is nonzero with probability d/n. We assume $d \ll \sqrt{n}$.*

In this case, the following facts will be useful for our analysis.

Lemma 2.2 *Let $\mathbf{A}$ and $\mathbf{B}$ be $n \times n$ ER(d) matrices. Then*

(a) the expected number of nonzeros in $\mathbf{A}$ and in $\mathbf{B}$ is dn,
(b) the expected number of scalar multiplications in $\mathbf{A} \cdot \mathbf{B}$ is d^2n, and
(c) the expected number of nonzeros in $\mathbf{C}$ is $d^2n(1 - o(1))$.

Proof. Since each entry of $\mathbf{A}$ and $\mathbf{B}$ is nonzero with probability d/n, the expected number of nonzeros in each matrix is $n^2(d/n) = dn$. For each of the possible n^3 scalar multiplications in $\mathbf{A} \cdot \mathbf{B}$, the computation is required only if both corresponding entries of $\mathbf{A}$ and $\mathbf{B}$ are nonzero, which are independent events. Thus the probability that any multiplication is required is d^2/n^2, and the expected number of scalar multiplications is d^2n. Finally, an entry of $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ is zero only if all n possible scalar multiplications corresponding to it are zero. Since the probability that a possible scalar multiplication is zero is $(1 - d^2/n^2)$ and the n possible scalar multiplications corresponding to a single output entry are independent, the probability that an entry of $\mathbf{C}$ is zero is $(1 - d^2/n^2)^n = 1 - d^2/n + O(d^4/n^2)$. Thus the expected number of nonzeros of $\mathbf{C}$ is $n^2(d^2/n - O(d^4/n^2) = d^2n(1 - o(1))$, since we assume $d \ll \sqrt{n}$. □

Definition 2.3 *The* computation cube *of square $n \times n$ matrix multiplication is an $n \times n \times n$ lattice where the voxel at location (i, j, k) corresponds to the scalar multiplication $\mathbf{A}(i, k) \cdot \mathbf{B}(k, j)$. We say a voxel (i, j, k) is* nonzero *if, for given input matrices $\mathbf{A}$ and $\mathbf{B}$, both $\mathbf{A}(i, k)$ and $\mathbf{B}(k, j)$ are nonzero.*

Given a set of voxels V, the projections of the set onto three orthogonal faces corresponds to the input elements of $\mathbf{A}$ and $\mathbf{B}$ necessary to perform the multiplications and the output elements of $\mathbf{C}$ which the products must update. The computation cube and this relationship of voxels to input and output matrix elements is shown in Figure 1. The following lemma relates the volume of V to its projections:

Lemma 2.4 [20] *Let V be a finite set of lattice points in $\mathbf{R}^3$, i.e., points (x, y, z) with integer coordinates. Let V_x be the projection of V in the x-direction, i.e., all points (y, z) such that there exists an x so that $(x, y, z) \in V$. Define V_y and V_z similarly. Let $|\cdot|$ denote the cardinality of a set. Then $|V| \leq \sqrt{|V_x| \cdot |V_y| \cdot |V_z|}$.*

Definition 2.5 *A* sparsity-independent *parallel algorithm for sparse matrix-matrix multiplication is one in which the assignment of entries of the input and output matrices to processors and the assignment of computation voxels to processors is independent of the sparsity pattern of the input (or output) matrices. If an assigned matrix entry is zero, the processor need not store it; if an assigned voxel is zero, the processor need to perform the computation.*

Our lower bound argument in Section 3 will apply to all sparsity-independent algorithms. However, we will analyze a more restricted set of algorithms in Section 4, those that assign contiguous brick-shaped sets of voxels to each processor.

2.1 Communication Model

We use the parallel distributed-memory communication model of Ballard et al. [2]. In this model, every processor has a local memory of size M words which is large enough to store one copy of the output matrix $\mathbf{C}$ distributed across the processors: $M = \Omega(d^2n/P)$. We are interested in the communication that occurs via message passing between processors, and we model the cost of a message of w words as $\alpha + \beta w$, so that α is the cost per message and β is the cost per word. To estimate the running time of a parallel algorithm, we count the cost of communication in terms of number of words W (*bandwidth cost*) and number of messages S (*latency cost*) along the critical path of the algorithm. That is, if two pairs of processors communicate messages of the same size simultaneously, we count that as the cost of one message. We assume a single processor can communicate only one message to one processor at a time. In this model, we do not consider contention or the number of hops a message travels; we assume the network has all-to-all connectivity.

2.2 All-to-all Communication

Several of the algorithms we discuss make use of all-to-all communication. If each processor needs to send b words to each other processor (so each processor needs to send a total of $b(P-1)$ words), the bandwidth lower bound is $W = \Omega(bP)$ and the latency lower bound is $S = \Omega(\log P)$. Each of these bounds is attainable, but it has been shown that they are not simultaneously attainable (see Theorem 2.9 of [6]). Depending on the relative costs of bandwidth and latency, one may wish to use the *point-to-point* algorithm (each processor sends data directly to each other processor) which incurs costs of $W = O(bP)$, $S = O(P)$ or the *bit-fixing* algorithm (each message of b words is sent by the bit-fixing routing algorithm) which incurs costs of $W = O(bP\log P)$, $S = O(\log P)$.

3 Lower Bounds

The general lower bounds for linear algebra [2] apply to our case and give

$$W = \Omega\left(\frac{d^2n}{P\sqrt{M}}\right). \tag{1}$$

This bound is highest when M takes its minimum value d^2n/P, in which case they become $W = \Omega(\sqrt{d^2n/P})$. In this section we improve these lower bounds by a factor of $\sqrt{n}\cdot\max\{1, d/\sqrt{P}\}$. For larger values of M, the lower bound in Equation 1 becomes weaker, whereas our new bound does not, and the improvement factor increases to $\sqrt{M}\cdot\max\{1, \sqrt{P}/d\}$. The previous memory-independent lower bound [4] reduces to the trivial bound $W = \Omega(0)$.

Theorem 3.1 *A sparsity-independent sparse matrix multiplication algorithm with load-balanced input and output has expected communication cost lower bounded by*

$$W = \Omega\left(\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\right\}\right)$$

for $ER(d)$ input matrices on P processors.

Proof. Consider the n^3 voxels that correspond to potential scalar multiplications $\mathbf{A}(i,k)\cdot\mathbf{B}(k,j)$. A sparsity-independent algorithm gives a partitioning of these multiplications among the P processors. Let V be the largest set of voxels assigned to a processor, so $|V| \geq \frac{n^3}{P}$. For each i, j, let ℓ^C_{ij} be the number of values of k such that $(i,j,k) \in V$, see Figure 3. We count how many of the voxels in V correspond to $\ell^C_{ij} < \frac{n}{4}$ and divide into two cases.

Case 1: At least $\frac{n^3}{2P}$ voxels of V correspond to $\ell^C_{ij} < \frac{n}{4}$. Let V' be these voxels, so $|V'| \geq \frac{n^3}{2P}$. We will analyze the communication cost corresponding to the computation of V' and get a bound on the number of products computed by this processor that must be sent to other processors. Since the output is load balanced and the algorithm is sparsity-independent, the processor that computes V' is allowed to store only a particular set of $\frac{n^2}{P}$ entries of $\mathbf{C}$ in the output data layout. Since every voxel in V' corresponds to an $\ell^C_{ij} < \frac{n}{4}$, the $\frac{n^2}{P}$ output elements stored by the processor correspond to at most $\frac{n^3}{4P}$ voxels in V', which is at most half of $|V'|$. All of the nonzero voxels in the remainder of V' contribute to entries of $\mathbf{C}$ that must be sent to another processor. In expectation, this is at least $\frac{d^2 n}{4P}$ nonzero voxels, since each voxel is nonzero with probability $\frac{d^2}{n^2}$. Moreover, from Lemma 2.2, only a small number of the nonzero entries of $\mathbf{C}$ have contributions from more than one voxel, so very few of the values can be summed before being communicated. The expected bandwidth cost is then bounded by $W = \Omega(d^2 n/P)$.

Case 2: Fewer than $\frac{n^3}{2P}$ voxels of V correspond to $\ell^C_{ij} < \frac{n}{4}$. This means that at least $\frac{n^3}{2P}$ voxels of V correspond to $\ell^C_{ij} \geq \frac{n}{4}$. Let V'' be these voxels, so $|V''| \geq \frac{n^3}{2P}$. We will analyze the communication cost corresponding to the computation of V'' and get a lower bound on the amount of input data needed by this processor. For each i, k, let ℓ^A_{ik} be the number of values of j such that $(i, j, k) \in V''$. Similarly, for each j, k, let ℓ^B_{jk} be the number of values of i such that $(i, j, k) \in V''$. Partition V'' into three sets: V_0 is the set of voxels that correspond to $\ell^A_{ik} > \frac{n}{d}$ and $\ell^B_{jk} > \frac{n}{d}$; V_A is the set of voxels that correspond to $\ell^A_{ik} \leq \frac{n}{d}$; and V_B is the set of voxels that correspond to $\ell^B_{jk} \leq \frac{n}{d}$ and $\ell^A_{ik} > \frac{n}{d}$. At least one of these sets has at least $\frac{n^3}{6P}$ voxels, and we divide into three subcases.

Case 2a: $|V_0| \geq \frac{n^3}{6P}$. Let p_A, p_B, and p_C be the sizes of the projections of V_0 onto $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively. Lemma 2.4 implies that $p_A p_B p_C \geq |V_0|^2 = \frac{n^6}{36P^2}$. The assumptions of Case 2 implies $p_C \leq \frac{|V_0|}{n/4}$. Thus $p_A p_B \geq \frac{n^4}{24P}$, or $\max\{p_A, p_B\} \geq \frac{n^2}{\sqrt{24P}}$. Since the situation is symmetric with respect to $\mathbf{A}$ and $\mathbf{B}$, assume without loss of generality that $\mathbf{A}$ has the larger projection, so $p_A \geq \frac{n^2}{\sqrt{24P}}$. Since the density of $\mathbf{A}$ is $\frac{d}{n}$, this means that the expected number of nonzeros in the projection of V_0 onto $\mathbf{A}$ is at least $\frac{dn}{\sqrt{24P}}$. Since each of these nonzeros in $\mathbf{A}$ corresponds to a $\ell^A_{ik} > \frac{n}{d}$, it is needed to compute V_0 with probability at least $1 - (1 - d/n)^{n/d} > 1 - 1/e$. Thus in expectation a constant fraction of the nonzeros of $\mathbf{A}$ in the projection of V_0 are needed. The number of nonzeros the processor holds in the initial data layout is $\frac{dn}{P}$ in expectation, which is asymptotically less than the number needed for the computation. Thus we get a bandwidth lower bound of $W = \Omega(dn/\sqrt{P})$.

Case 2b: $|V_A| \geq \frac{n^3}{6P}$. Each voxel in V_A corresponds to $\ell^A_{ik} \leq \frac{n}{d}$. In this case we are able to bound the re-use of entries of mA to get a lower bound. Count how many entries of $\mathbf{A}$ correspond to each possible value of ℓ^A_{ik}, $1 \leq r \leq \frac{n}{d}$, and call this number N_r. Note that $\sum_{r=1}^{d/n} r \cdot N_r = |V_A|$. Suppose a given entry $\mathbf{A}(i, k)$ corresponds to $\ell^A_{ik} = r$. We can bound the probability that $\mathbf{A}(i, k)$ is needed by the processor to compute V_A as a function $f(r)$. The probability that both $\mathbf{A}(i, k)$ is needed is the probability than $\mathbf{A}(i, k)$ is nonzero and one of the r voxels corresponding to $\mathbf{A}(i, k)$ in V_A is nonzero, so

$$f(r) = \frac{d}{n}\left(1 - \left(1 - \frac{d}{n}\right)^r\right) \geq \frac{rd^2}{2n^2},$$

since $r \leq \frac{n}{d}$. Thus the expected number of nonzeros of $\mathbf{A}$ that are needed by the processor is

$$\sum_{r=1}^{d/n} N_r f(r) \geq \frac{d^2}{2n^2} \sum_{r=1}^{d/n} r \cdot N_r \geq \frac{d^2 n}{12P}.$$

This is asymptotically larger than the number of nonzeros the processor holds at the beginning of the computation, so we get a bandwidth lower bound of $W = \Omega(d^2 n/P)$.

Case 2c: $|V_B| \geq \frac{n^3}{6P}$. The analysis is identical to the previous case, except we look at the number of nonzeros of $\mathbf{B}$ that are required.

Since an algorithm may be in any of these cases, the overall lower bound is the minimum:

$$W = \Omega\left(\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\right\}\right).$$

□

4 Algorithms

In this section we consider algorithms which assign contiguous bricks of voxels to processors. We categorize these algorithms into 1D, 2D, and 3D algorithms, as shown in Figure 2. If we consider the dimensions of the brick of voxels assigned to each processor, 1D algorithms correspond to bricks with two dimensions of length n (and 1 shorter), 2D algorithms correspond to bricks with one dimension of length n (and 2 shorter), and 3D algorithms correspond to bricks with all 3 dimensions shorter than n. Table 1 provides a summary of the communication costs of the sparsity-independent algorithms we consider.

4.1 1D Algorithms

4.1.1 Naive Block Row Algorithm

The naive block row algorithm [8] distributes the input and output matrices to processors in a block row fashion. Then in order for processor i to compute the ith block row, it needs access to the ith block row of $\mathbf{A}$ (which it already owns), and potentially all of $\mathbf{B}$. Thus, we can allow each processor to compute its block row of $\mathbf{C}$ by leaving $\mathbf{A}$ and $\mathbf{C}$ stationary and cyclically shifting block rows of $\mathbf{B}$ around a ring of the processors. This algorithm requires P stages, with each processor communicating with its two neighbors in the ring. The size of each message is the number of nonzeros in a block row of $\mathbf{B}$, which is expected to be dn/P words. Thus, the bandwidth cost of the block row algorithm is dn and the latency cost is P. An analogous block column algorithm works by cyclically shifting block columns of $\mathbf{A}$ with identical communication costs.

4.1.2 Improved Block Row Algorithm

The communication costs of the block row algorithm can be reduced without changing the assignment of matrix entries or voxels to processors [12]. The key idea is for each processor to determine exactly which rows of $\mathbf{B}$ it needs to access in order to perform its computations. For example, if processor i owns the ith block row of $\mathbf{A}$, $\mathbf{A}_i$, and the jth subcolumn of $\mathbf{A}_i$ contains no nonzeros, then processor i doesn't need to access the jth row of $\mathbf{B}$. Further, since the height of a subcolumn is n/P, the probability that the subcolumn is completely empty is

$$Pr\left[nnz(\mathbf{A}_i(:,j)) = 0\right] = \left(1 - \frac{d}{n}\right)^{\frac{n}{P}} \approx 1 - \frac{d}{P},$$

assuming $d < P$. In this case, the expected number of subcolumns of $\mathbf{A}_i$ which have at least one nonzero is dn/P. Since processor i needs to access only those rows of $\mathbf{B}$ which correspond to nonzero subcolumns of $\mathbf{A}_i$, and because the expected number of nonzeros in each row of $\mathbf{B}$ is d, the expected number of nonzeros of $\mathbf{B}$ that processor i needs to access is d^2n/P.

Note that the local memory of each processor must be of size $\Omega\left(d^2n/P\right)$ in order to store the output matrix $\mathbf{C}$. Thus, it is possible for each processor to gather all of their required rows of $\mathbf{B}$ at once. The improved algorithm consists of each processor determining which rows it needs, requesting those rows from the appropriate processors, and then sending and receiving approximately d rows. While this can be implemented in various ways, the bandwidth cost of the algorithm is at least $\Omega\left(d^2n/P\right)$ and if point-to-point communication is used, the latency cost is at least $\Omega(\min\{P, dn/P\})$. The block column algorithm can be improved in the same manner.

4.1.3 Outer Product Algorithm

Another possible 1D algorithm is to partition $\mathbf{A}$ in block columns, and $\mathbf{B}$ in block rows [19]. Without communication, each processor locally generates an $n \times n$ sparse matrix of rank n/P, and processors combine their results to produce the output $\mathbf{C}$. Because each column of $\mathbf{A}$ and row of $\mathbf{B}$ have about d nonzeros, the expected number of nonzeros in the locally computed output is d^2n/P. By deciding the distribution of $\mathbf{C}$ to processors up front, each processor can determine where to send each of its computed nonzeros. The final communication pattern is realized with an all-to-all collective in which each processer sends and receives $O(d^2n/P)$ words. Note that assuming $\mathbf{A}$ and $\mathbf{B}$ are initially distributed to processors in different ways may be unrealistic; however, no matter how they are initially distributed, $\mathbf{A}$ and $\mathbf{B}$ can be transformed to block column and row layouts with all-to-all collectives for a communication cost which is dominated by the final communication phase.

To avoid the all-to-all, it is possible to compute the expected number of blocks of the output which actually contain nonzeros; the best distribution of $\mathbf{C}$ is 2D, in which case the expected number of blocks of $\mathbf{C}$ you need to communicate is $\min\{P, dn/\sqrt{P}\}$. Thus for $P > (dn)^{2/3}$, the outer product algorithm can have $W = O(d^2n/P)$ and $S = O(dn/\sqrt{P})$.

4.2 2D Algorithms

4.2.1 Sparse SUMMA

In the Sparse SUMMA algorithm [8], the brick of voxels assigned to a processor has its longest dimension (of length n) in the k dimension. For each output entry of $\mathbf{C}$ to which it is assigned, the processor computes all the nonzero voxels which contribute to that output entry. The algorithm has a bandwidth cost of $O(dn/\sqrt{P})$ and a latency cost of $O(\sqrt{P})$ [8].

4.2.2 Improved Sparse SUMMA

In order to reduce the latency cost of Sparse SUMMA, each processor can gather all the necessary input data up front. That is, each processor is computing a product of a block row of $\mathbf{A}$ with a block column of $\mathbf{B}$, so if it gathers all the nonzeros in those regions of the input matrices, it can compute its block of $\mathbf{C}$ with no more communication. Since every row of $\mathbf{A}$ and column of $\mathbf{B}$ contain about d nonzeros, and the number of rows of $\mathbf{A}$ and columns of $\mathbf{B}$ in a block is $n/\sqrt{P}$, the number of nonzeros a processor must gather is $O(dn/\sqrt{P})$. If $d > \sqrt{P}$, then the memory requirements for this gather operation do not exceed the memory requirements for storing the block of the output matrix $\mathbf{C}$, which is $\Omega(d^2n/P)$.

The global communication pattern for each processor to gather its necessary data consists of allgather collectives along processor columns and along processor grids. The bandwidth cost of these collectives is $O(dn/\sqrt{P})$, which is the same as the standard algorithm, and the latency cost is reduced to $O(\log P)$. To our knowledge, this improvement has not appeared in the literature before.

We might also consider applying the optimization that improved the 1D block row (or column) algorithm. Processor (i, j) would need to gather the indices of the nonzero subcolumns of $\mathbf{A}_i$ and the nonzero subrows of $\mathbf{B}_j$. This requires receiving $\Omega(dn/\sqrt{P})$ words, and so it cannot reduce the communication cost of Sparse SUMMA.

As in the dense case, there are variants on the sparse SUMMA algorithm that leave one of the input matrices stationary, rather than leaving the output matrix $\mathbf{C}$ stationary [17]. When multiplying ER(d) matrices, stationary input matrix algorithms require more communication that the standard approach because the global data involved in communicating $\mathbf{C}$ is about d^2n, while the global data involved in communicating $\mathbf{A}$ and $\mathbf{B}$ is only dn.

4.3 3D Iterative Algorithm

In this section we present a new 3D iterative algorithm. We start with a dense version of the algorithm and apply a series of improvements in order to match the lower bound.

4.3.1 3D Algorithms for Dense Matrix Multiplication

The term "3D" originates from dense matrix multiplication algorithms [1], where the processors are organized in a 3-dimensional grid, and the computational cube is mapped directly onto the cube of processors. In the simplest case, the processors are arranged in a $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ grid. Let $\mathbf{A}$ be distributed across the $P^{2/3}$ processors along one face of the cube and $\mathbf{B}$ be distributed across the $P^{2/3}$ processors along a second face of the cube. Then each input matrix can be broadcast through the cube in the respective dimensions so that every processor in the cube owns the block of $\mathbf{A}$ and the block of $\mathbf{B}$ it needs to compute its local multiplication. After the computation, the matrix $\mathbf{C}$ can be computed via a reduction in the third dimension of the cube, resulting in the output matrix being distributed across a third face of the cube.

The communication cost of this algorithm is the cost of the two broadcasts and one reduction. The size of the local data in each of these operations is $n^2/P^{2/3}$, and the number of processors involved is $P^{1/3}$, so the total bandwidth cost is $O(n^2/P^{2/3})$ and the total latency cost is $O(\log P)$. These communication costs are less than the costs of 2D algorithms for dense multiplication [1, 11, 26]. However, because the local computation involves matrices of size $n^2/P^{2/3}$, the 3D algorithm requires more local memory that is necessary to store the input and output matrices.

This tradeoff between memory requirements and communication costs can be managed in a continuous way by varying the dimensions of the processor grid (or, equivalently, the dimensions of the bricks of voxels assigned to processors) [21, 24]. Instead of using a cubic $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ processor grid, we can use a $c \times \sqrt{P/c} \times \sqrt{P/c}$ grid, where $1 \leq c \leq \sqrt[3]{P}$ and $c = 1$ reproduces a 2D algorithm. The approach that generalizes Cannon's algorithm [11] is presented as "2.5D-matrix-multiply"[1] as Algorithm 2 by Solomonik and Demmel [24] and the approach that generalizes SUMMA is presented as "2.5D-SUMMA" in Algorithm 1 by Solomonik et al. [25]. Both approaches yields a bandwidth cost of $O(n^2/\sqrt{Pc})$, a latency cost of $O(\sqrt{P/c^3} + \log c)$, and local memory requirements of $O(cn^2/P)$.

4.3.2 Converting to Sparse Case

Naive 3D algorithms for sparse matrix multiplication can be devised directly from the dense versions. As in [24, 25], we assume the data initially resides only on the one of the c layers and gets replicated along the third dimension before the multiplications start. Then, each of those layers executes a partial 2D algorithm (with the partial contribution to $\mathbf{C}$ remaining stationary), in the sense that each layer is responsible for computing $1/c$ of the total computation. Consequently, the number of steps in the main stage of the algorithm becomes $\sqrt{P/c^3}$. The final stage of the algorithm is a reduction step among groups of c processors, executed concurrently by all groups of processors representing a fiber along the third processor dimension.

[1]The origin of the name "2.5D" comes from the fact that the algorithm interpolates between existing 2D and 3D algorithms. We use the term 3D to describe both 3D and 2.5D dense algorithms.

The latency cost is identical to the dense algorithm: $O(\sqrt{P/c^3} + \log c)$. The first term comes from the main stage and the second term comes from the initial replication and final reduction phases. The bandwidth cost can be computed based on the number of nonzeros in each block of $\mathbf{A}$, $\mathbf{B}$, or $\mathbf{C}$ communicated. In the initial replication phase, blocks of $\mathbf{A}$ and $\mathbf{B}$ of dimension $n/\sqrt{P/c} \times n/\sqrt{P/c}$ are broadcast to c different processors for a bandwidth cost of $O(cdn/P)$. In the main stage of the algorithm, the same size blocks are communicated during each of the $\sqrt{P/c^3}$ steps for a total bandwidth cost of $O(dn/\sqrt{Pc})$. The final reduction is significantly different from a dense reduction, resembling more closely a gather operation since the expected number of collisions in partial contributions of $\mathbf{C}$ is very small for $d \ll \sqrt{n}$. Thus, we expect the size of the output to be almost as large as the sum of the sizes of the inputs. The bandwidth cost of the final phase is then $O(cd^2n/P)$.

Thus, the straightforward conversion of the dense 3D algorithm to the sparse case results in the same latency cost and a total bandwidth cost of $O(dn/\sqrt{Pc} + cd^2n/P)$. Further, this algorithm will require extra local memory, because gathering the output matrix onto one layer of processors requires $\Omega(cd^2n/P)$ words of memory, a factor of $\Omega(c)$ times as much as required to store $\mathbf{C}$ across all processors. The extra space required for $\mathbf{C}$ dominates the space required for replication of $\mathbf{A}$ and $\mathbf{B}$.

4.3.3 Removing Input Replication and Assumption on Initial Data Distribution

In developing a more efficient 3D algorithm for the sparse case, our first observation is that we can avoid the first phase of input replication. This replication can also be avoided in the dense case, but it will not affect the asymptotic communication costs.

The dense 2.5D algorithms assume that the input matrices initially reside on one $\sqrt{P/c} \times \sqrt{P/c}$ face of the processor grid, and the first phase of the algorithm involves replicating $\mathbf{A}$ and $\mathbf{B}$ to each of the c layers. One can view the distribution of computation as assigning $1/c^{\text{th}}$ of the outer products of columns of $\mathbf{A}$ with corresponding rows of $\mathbf{B}$ to each of the c layers. In this way, each layer of processors needs only $1/c^{\text{th}}$ of the columns of $\mathbf{A}$ and rows of $\mathbf{B}$ rather than the entire matrices.

In order to redistribute the matrices across c sets of processors in a 2D blocked layout with block size $n/\sqrt{P/c} \times n/\sqrt{P/c}$, blocks of $\sqrt{c} \times \sqrt{c}$ processors can perform all-to-all operations, as shown in Figure 4. The cost of this operation is $W = O(dn/P \log c)$ and $S = O(\log c)$ if the bit-fixing algorithm is used, removing the initial replication cost from Section 4.3.2. This optimization also removes the extra memory requirement for storing copies of $\mathbf{A}$ and $\mathbf{B}$.

We will see in Section 4.3.4 that the output matrix can be returned in the same 2D blocked layout as the input matrices were initially distributed.

4.3.4 Improving Communication of C

Our next observation for the sparse case is that the final reduction phase to compute the output matrix becomes a gather rather than a reduction. This gather operation collects $\mathbf{C}$ onto one layer of processors; in order to balance the output across all processors, we would like to scatter $\mathbf{C}$ back along the third processor dimension. However, performing a gather followed by a scatter is just an inefficient means of performing an all-to-all collective. Thus we should replace the final reduction phase with a final all-to-all phase. This optimization reduces the bandwidth cost of the 3D algorithm to $O(dn/\sqrt{Pc} + d^2n/P)$. Note that the cost of replicating $\mathbf{A}$ and $\mathbf{B}$ in the first phase of the algorithm would no longer always be dominated by the reduction cost of $\mathbf{C}$, as in Section 4.3.2, but the cost of the input all-to-all from Section 4.3.3 is dominated by the output all-to-all. By replacing the reduction phase with an all-to-all, we also remove the memory requirement of $\Omega(cd^2n/P)$.

4.3.5 Improving Communication of $\mathbf{A}$ and $\mathbf{B}$

Furthermore, we can apply the optimization described in Section 4.2.2: to reduce latency costs in the main phase of the 3D algorithm (which itself is a 2D algorithm), processors can collect all the entries of $\mathbf{A}$ and $\mathbf{B}$ they need upfront rather than over several steps. This collective operation consists of groups of $\sqrt{P/c^3}$ processors performing allgather operations (after the initial circular shifts of Cannon's algorithm, for example). Since the data per processor in the allgather operation is $O(cdn/P)$, the bandwidth cost of the main phase remains $O(dn/\sqrt{Pc})$. The latency cost is reduced from $O(\sqrt{P/c^3})$ to $O(\log(\sqrt{P/c^3}))$, yielding a total latency cost (assuming the bit-fixing algorithm is used for the all-to-all) of $O(\log P)$. The local memory requirements increase to $\Omega(dn/\sqrt{Pc})$; when $c \geq P/d^2$, this requirement is no more than the space required to store $\mathbf{C}$.

4.3.6 Optimizing c

If $d > \sqrt{P}$, then $d^2n/P > dn/\sqrt{P}$, and the communication lower bound from Section 3 is $\Omega(dn/\sqrt{P})$. Thus, choosing $c = 1$ eliminates the $d^2n/P \log c$ term, and the 3D algorithm reduces to a 2D algorithm which is communication optimal.

However, in the case $d < \sqrt{P}$, which will become the case in a strong-scaling regime, increasing c can reduce communication. In this case, the lower bound from Section 3 is $\Omega(d^2n/P)$. Depending on the all-to-all algorithm used, increasing c causes slow increases on latency costs and on the $d^2n/P \log c$ bandwidth cost term, but it causes more rapid decrease in the $dn/\sqrt{Pc}$ term. Choosing $c = \Theta(P/d^2)$ balances the two terms in the bandwidth cost, yielding a total bandwidth cost of $O(d^2n/P)$, which attains the lower bound in this case.

In summary, choosing $c = \min\left\{1, P/d^2\right\}$ allows for a communication optimal 3D sparse matrix multiplication algorithm, with a slight tradeoff between bandwidth and latency costs based on the all-to-all algorithm used. Additionally, making this choice of c means that asymptotically no extra memory is needed over the space required to store $\mathbf{C}$.

4.4 3D Recursive Algorithm

We also present a new 3D recursive algorithm which is a parallelization of a sequential recursive algorithm using the techniques of [3, 13]. Although we have assumed that the input matrices are square, the recursive algorithm will use rectangular matrices for subproblems. Assume that P processors are solving a subproblem of size $m \times k \times m$, that is $\mathbf{A}$ is $m \times k$, and $\mathbf{B}$ is $k \times m$, and $\mathbf{C}$ is $m \times m$. We will split into four subproblems, and then solve each subproblem independently on a quarter of the processor. There are two natural ways to split the problem into four equal subproblems that respect the density similarity between $\mathbf{A}$ and $\mathbf{B}$, see Figure 5.

1. Split m in half, creating four subproblems of shape $(m/2) \times k \times (m/2)$. In this case each of the four subproblems needs access to a different part of $\mathbf{C}$, so no communication of $\mathbf{C}$ is needed. However one half of $\mathbf{A}$ and $\mathbf{B}$ is needed for each subproblem, and since each quarter of the processors holds only one quarter of each matrix, it will be necessary to replicate $\mathbf{A}$ and $\mathbf{B}$. This can be done via allgather collectives among disjoint pairs of processors at the cost of $O\left(dmk/(nP)\right)$ words and $O(1)$ messages.
2. Split k in quarters, creating four subproblems of shape $m \times (k/4) \times m$. In this case each of the four subproblems needs access to a different part of $\mathbf{A}$ and $\mathbf{B}$, so with the right data layout, no communication of $\mathbf{A}$ or $\mathbf{B}$ is needed. However each subproblem will compute nonzeros across all of $\mathbf{C}$, so those entries need to be redistributed and combined if necessary. This can be done via all-to-all collective among disjoint sets of 4 processors at a cost of $O(d^2m^2/(nP))$ words and $O(1)$ messages.

At each recursive step, the algorithm chooses whichever split is cheapest in terms of communication cost. Initially, $m = k = n$ so split 1 costs $O(dn/P)$ words and is cheaper than split 2, which costs $O(d^2n/P)$ words. There are two cases to consider.

Case 1: If $P \leq d^2$, the algorithm reaches a single processor before split 1 becomes more expensive than split 2, so only split 1 is used. This case corresponds to a 2D algorithm, and the communication costs are

$$W = \sum_{i=0}^{\log_4 P-1} O\left(\frac{d(n/2^i)n}{P/4^i}\right) = O\left(\frac{dn}{\sqrt{P}}\right), \qquad S = O(\log P).$$

Case 2: If $P > d^2$, split 1 becomes more expensive than split 2 after $\log_2 d$ steps. After $\log_2 d$ steps, the subproblems have dimensions $(n/d) \times n \times (n/d)$ and there are P/d^2 processors working on each subproblem. The first $\log_2 d$ steps are split 1, and the rest are split 2, giving communication costs of

$$W = \sum_{i=0}^{\log_2 d-1} O\left(\frac{d(n/2^i)n}{P/4^i}\right) + \sum_{i=\log_2 d}^{\log_4 P} O\left(\frac{d^2 n}{P}\right) = O\left(\frac{d^2 n}{P}\left\lceil \log \frac{P}{d^2}\right\rceil\right), \qquad S = O(\log P).$$

This case corresponds to a 3D algorithm.

In both cases, the communication costs match the lower bound from Section 3 up to factors of at most $\log P$. Only layouts that are compatible with the recursive structure of the algorithm will allow these communication costs. One simple layout is to have $\mathbf{A}$ is block-column layout, $\mathbf{B}$ in block-row layout. Then $\mathbf{C}$ should have blocks of size $n/d \times n/d$, each distributed on a different $\lceil P/d^2 \rceil$ of the processors.

5 Related Work

The classical serial algorithm of Gustavson [18], which is the algorithm currently implemented in MATLAB [14], does optimal work for the common case of flops $\ll nnz, n$. Yuster and Zwick [29] gave a $O(nnz^{0.7} n^{1.2} + n^{2+o(1)})$ time serial algorithm for multiplying matrices over a ring, which uses Strassen-like fast dense matrix multiplication as a subroutine. Their algorithm is theoretically close to optimal for the case of $nnz(\mathbf{C}) = \Theta(n^2)$, an assumption that does not always hold.

The 1D improved block-row algorithm is due to Challacombe [12], who calls the calculation of required indices of $\mathbf{B}$ the “symbolic” phase. His algorithm uses the allgather collective for the symbolic phase and point-to-point communication for the subsequent numerical phase. Challacombe, however, did not analyze his algorithm’s communication costs. Kruskal et al. [19] gave a parallel algorithm based on outer products, which runs in time $O((\text{flops}/p)\log n/\log(\text{flops}/p))$ on p processors. They use the EREW PRAM model, and hence do not analyze the communication costs of their algorithm.

Sparse 2D SUMMA and its analysis is due to Buluç and Gilbert [8], who also analyzed the 1D naïve block-row algorithm. Their follow-up work showed that SpSUMMA provides good speedup to thousands of cores on various different input types, but its scaling is limited by the communication costs that consume the majority of the time [9]. Recent work by Campagna et al. [10] sketches a parallel algorithm that replicates the inputs (but not the output) to all the processors to avoid later communication. In our model, their algorithm has bandwidth cost $W = O(dn)$.

Grigori et al. [16] gave tight communication lower and upper bounds for Cholesky factorization of sparse matrices corresponding to certain grids. Pietracaprina et al. [23] gave lower bounds on the number of rounds it takes to compute the sparse matrix product in MapReduce. Their lower bound analysis, however, is not parametrized to the density of the inputs and uses the inequality $\text{flops} \leq nnz \cdot \min(nnz, n)$. While it is true that there exist assignment of input matrices for which the inequality is tight, the lower bound does not hold for the majority of input matrix pairs for which the inequality is not tight. By parametrizing the density of inputs, we show that our algorithms are communication optimal over all ER(d) matrices.

References

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575 –582, sep. 1995.

[2] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM. J. Matrix Anal. & Appl*, 32:pp. 866–901, 2011.

[3] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 193–204, New York, NY, USA, 2012. ACM.

[4] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Brief announcement: Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 77–79, New York, NY, USA, 2012. ACM.

[5] 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.

[6] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, pages 298–309, New York, NY, USA, 1994. ACM.

[7] A. Buluç and J. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

[8] A. Buluç and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP'08: Proc. of the Intl. Conf. on Parallel Processing*, pages 503–510, Portland, Oregon, USA, 2008. IEEE Computer Society.

[9] A. Buluç and J. R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal of Scientific Computing (SISC)*, 34(4):170 – 191, 2012.

[10] A. Campagna, K. Kutzkov, and R. Pagh. On parallelizing matrix multiplication by the column-row method. *arXiv preprint arXiv:1210.0461*, 2012.

[11] L. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, Bozeman, MN, 1969.

[12] M. Challacombe. A general parallel sparse-blocked matrix multiply for linear scaling SCF theory. *Computer physics communications*, 128(1-2):93–107, 2000.

[13] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2013.

[14] 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.

[15] J. R. Gilbert, S. Reinhardt, and V. B. Shah. A unified framework for numerical and combinatorial computing. *Computing in Science and Engineering*, 10(2):20–25, 2008.

[16] L Grigori, P.-Y. David, J. Demmel, and S. Peyronnet. Brief announcement: Lower bounds on communication for sparse Cholesky factorization of a model problem. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 79–81, New York, NY, USA, 2010. ACM.

[17] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. A flexible class of parallel matrix multiplication algorithms, 1998.

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

[19] C. P. Kruskal, L. Rudolph, and M. Snir. Techniques for parallel manipulation of sparse matrices. *Theor.*

Comput. Sci., 64(2):135–157, 1989.

[20] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the AMS*, 55:961–962, 1949.

[21] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24:287–297, 1999.

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

[23] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round tradeoffs for mapreduce computations. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 235–244. ACM, 2012.

[24] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Euro-Par'11: Proceedings of the 17th International European Conference on Parallel and Distributed Computing*. Springer, 2011.

[25] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 77. ACM, 2011.

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

[27] S. Van Dongen. Graph clustering via a discrete uncoupling process. *SIAM Journal on Matrix Analysis and Applications*, 30(1):121–141, 2008.

[28] J. VandeVondele, U. Borštnik, and J. Hutter. Linear scaling self-consistent field calculations with millions of atoms in the condensed phase. *Journal of Chemical Theory and Computation*, 8(10):3565–3573, 2012.

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

A Tables and Figures

	Algorithm	Bandwidth cost	Latency cost
	Previous Lower Bound [2]	$\frac{d^2n}{P\sqrt{M}} \leq \sqrt{\frac{d^2n}{P}}$	0
	Lower Bound [here]	$\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\right\}$	1
1D	Naïve Block Row [8]	dn	P
	Improved Block Row* [12]	$\frac{d^2n}{P}$	$\min\{\log P, \frac{dn}{P}\}$
	Outer Product* [19]	$\frac{d^2n}{P}$	$\log P$
2D	SpSUMMA [8]	$\frac{dn}{\sqrt{P}}$	$\sqrt{P}$
	Improved SpSUMMA [here]	$\frac{dn}{\sqrt{P}}$	$\log P \frac{dn}{M\sqrt{P}}$
3D	Iterative* [here]	$\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\right\}$	$\log P$
	Recursive [here]	$\min\left\{\frac{dn}{\sqrt{P}}, \frac{d^2n}{P}\left\lceil\log\frac{P}{d^2}\right\rceil\right\}$	$\log P$

Table 1: Asymptotic expected communication costs of sparsity-independent algorithms. Algorithms marked with an asterisk make use of all-to-all communication. Depending on the algorithm used for the all-to-all, either the bandwidth or latency cost listed is attainable, but not both; see Section 2.2.

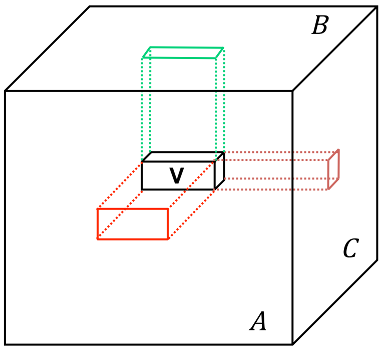

Figure 1: The computation cube for matrix multiplication, with a specified subset of voxels V along with its three projections. Each voxel corresponds to the multiplication of its projection onto $\mathbf{A}$ and $\mathbf{B}$, and contributes to its projection onto $\mathbf{C}$.

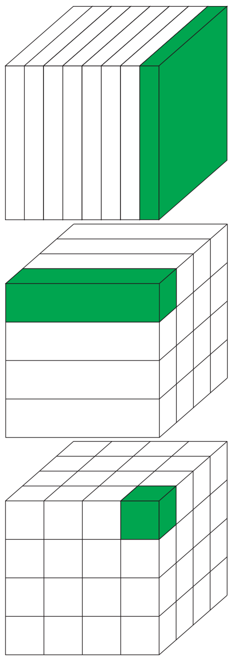

Figure 2: How the cube is partitioned in 1D (top), 2D (middle), and 3D (bottom) algorithms.

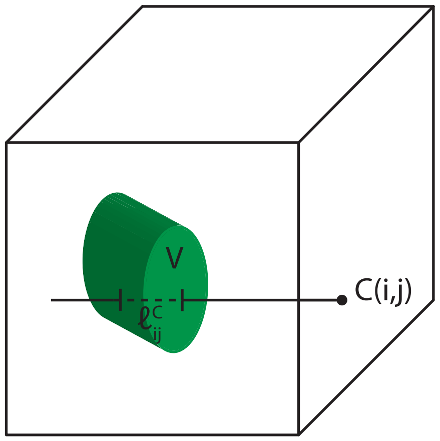

Figure 3: Graphical representation of V and ℓ^C_{ij}.

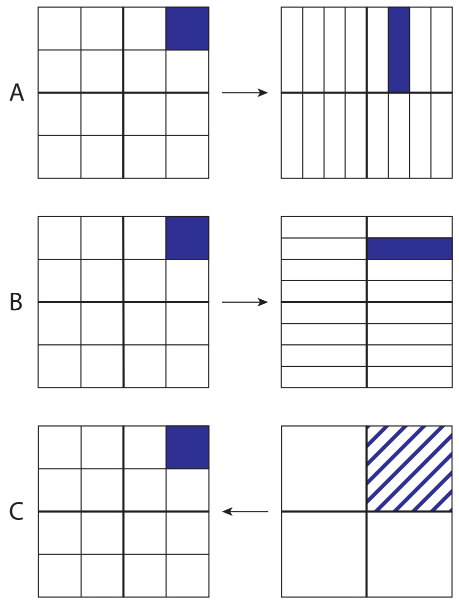

Figure 4: Possible redistribution scheme for input and output matrices for the 3D algorithm with $4 \times 2 \times 2$ processor grid ($c = 4$). The colored regions denote submatrices owned by a particular processor. The input matrices are initially in a 2D block distribution, and redistribution occurs in all-to-all collectives among disjoint sets of 4 processors. Since each of the c layers are 2×2 grids, the intermediate phase consists of allgather collectives among pairs of processors. After local computation, the output matrix is redistributed (and nonzeros combined if necessary) via all-to-all collectives among the same disjoint sets of 4 processors, returning the output matrix also in a 2D block distribution.

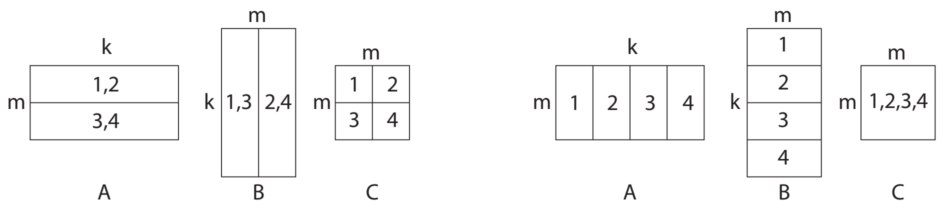

Figure 5: Two ways to split the matrix multiplication into four subproblems, with the parts of each matrix required by each subproblem labelled. On the left is split 1 and on the right is split 2.